Process Out-Grafting: An Efficient "Out-of-VM" Approach for Fine-Grained Process Execution Monitoring

Deepa Srinivasan
NC State University
dsriniv@ncsu.edu

Zhi Wang
NC State University
zhi_wang@ncsu.edu

Xuxian Jiang
NC State University
jiang@cs.ncsu.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

ABSTRACT

Recent rapid malware growth has exposed the limitations of traditional *in-host* malware-defense systems and motivated the development of secure virtualization-based *out-of-VM* solutions. By running vulnerable systems as virtual machines (VMs) and moving security software from inside the VMs to outside, the out-of-VM solutions securely isolate the anti-malware software from the vulnerable system. However, the presence of semantic gap also leads to the compatibility problem in not supporting existing defense software. In this paper, we present *process out-grafting*, an architectural approach to address both isolation and compatibility challenges in out-of-VM approaches for fine-grained process-level execution monitoring. Specifically, by relocating a suspect process from inside a VM to run side-by-side with the out-of-VM security tool, our technique effectively removes the semantic gap and supports existing user-mode process monitoring tools without any modification. Moreover, by forwarding the system calls back to the VM, we can smoothly continue the execution of the out-grafted process without weakening the isolation of the monitoring tool. We have developed a KVM-based prototype and used it to natively support a number of existing tools without any modification. The evaluation results including measurement with benchmark programs show it is effective and practical with a small performance overhead.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection - Invasive Software

General Terms

Security

Keywords

Virtualization, Process Monitoring, Semantic Gap

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

1. INTRODUCTION

Computer malware (e.g., viruses and trojans), in its seemingly infinite evolution of functionality and forms, is one of the largest threat that end users and enterprises are combating daily. A recent report from McAfee [3] shows a rapidly exploding malware growth with new record numbers "accomplished" in the last year. Specifically, as highlighted in the report, "McAfee Labs identified more than 20 million new pieces of malware in 2010," which translates into nearly 55,000 new malware samples discovered every day! Moreover, "of the almost 55 million pieces of malware McAfee Labs has identified and protected against, 36 percent of it was written in 2010!" This alarming trend reveals the disturbing fact that existing malware defenses fail to effectively contain the threat and keep up with the malware growth.

Especially, if we examine traditional anti-malware tools, they are typically deployed within vulnerable systems and could be the first targets once malware infect a computer system. In other words, though these traditional *in-host* tools are valuable in monitoring system behavior or detecting malicious activities (with their native access inside the systems), they are fundamentally limited in their isolation capability to prevent themselves from being infected in the first place. To address that, researchers propose *out-of-VM* approaches [8, 15, 20, 21, 30, 31, 37], which change the malware defense landscape by leveraging recent advances of virtualization to run vulnerable systems as VMs and then moving anti-malware tools from inside the systems to outside. By enlisting help from the underlying virtualization layer (or hypervisor), out-of-VM approaches can effectively overcome the isolation challenge that encumbers traditional in-host approaches. However, by separating anti-malware software from the untrusted systems, they also naturally encounter the well-known semantic gap challenge: these anti-malware software or tools – as out-of-VM entities – need to monitor or semantically infer various in-VM activities.

In the past several years, researchers have actively examined this semantic-gap challenge and implemented a number of introspection-based systems [12, 21] to mitigate it. For example, VMwatcher [21] proposes a guest view casting technique to apply the knowledge of inner guest OS kernel, especially the semantic definition of key kernel data structure and functionality, to bridge the semantic gap. Virtuoso [12] aims to automate the process of extracting introspection-relevant OS kernel information (by monitoring the execution of an in-guest helper program) for the construction of introspection-aware security tools. Though these systems make steady progresses in bridging the semantic gap, the

gap still inevitably leads to a compatibility problem. In particular, none of these introspection-based systems is compatible with existing anti-malware software (including various system/process monitoring tools) that were designed to run within a host. Due to the lack of compatibility, significant effort and advanced mechanisms [12, 21] are still needed to re-engineer these tools and adapt them for different guest OSs. This is especially concerning when fine-grained monitoring of individual processes requires intercepting a wide variety of events (e.g. user-library function calls or system calls) and interpreting them meaningfully (e.g. to determine their arguments). Worse, these introspection-based solutions are sensitive to guest OS versions or variants and to some extent fragile to any change or patch to the guest OS. As a result, this compatibility problem severely limits the effectiveness and the adoption of these out-of-VM approaches.

In this paper, we present *process out-grafting*, an architectural approach that addresses both isolation and compatibility challenges for out-of-VM, fine-grained user-mode process execution monitoring. Similar to prior out-of-VM approaches, out-grafting still confines vulnerable systems as VMs and deploys security tools outside the VMs. However, instead of analyzing the entire VM on all running processes, out-grafting focuses on each individual process for fine-grained execution monitoring. More importantly, our approach is designed to naturally support existing user-mode process monitoring tools (e.g., *strace*, *ltrace*, and *gdb*) outside of monitored VMs on an internal suspect process, without the need of modifying these tools or making them introspection-aware (as required in prior out-of-VM approaches). For simplicity, we use the terms "production VM" and "security VM" respectively to represent the vulnerable VM that contains a suspect process and the analysis VM that hosts the security tool to monitor the suspect process.

To enable process out-grafting, we have developed two key techniques. Specifically, the first technique, *on-demand grafting*, relocates the suspect process on demand from the production VM to security VM (that contains the process monitoring tool as well as its supporting environment). By doing so, grafting effectively brings the suspect process to the monitor for fine-grained monitoring, which leads to at least two important benefits: (1) By co-locating the suspect process to run side-by-side with our monitor, the semantic gap caused by the VM isolation is effectively removed. In fact, from the monitor's perspective, it runs together with the suspect process inside the same system and based on its design can naturally monitor the suspect process without any modification. (2) In addition, the monitor can directly intercept or analyze the process execution even at the granularity of user-level function calls, without requiring hypervisor intervention, which has significant performance gains from existing introspection-based approaches.

To still effectively confine the (relocated) suspect process, our second technique enforces a mode-sensitive split execution of the process, thus the name *split execution*. Specifically, only the user-mode instructions of the suspect process, which is our main focus for fine-grained monitoring, will be allowed to execute in the security VM; all kernel-mode execution that requires the use of OS kernel system services is forwarded back to the production VM. By doing so, we can not only maintain a smooth continued execution of the suspect process after relocation, but ensure its isolation from our monitoring tools. Particularly, from the suspect process' perspective, it is still logically running inside the production VM. In the meantime, as the suspect process physically runs inside the security VM, the monitoring overhead will not be inflicted to the production VM, thus effectively localizing monitoring impact within the security VM.

We have implemented a proof-of-concept prototype on KVM/ Linux (version kvm-2.6.36.1) and tested it to out-graft various processes from different VMs running either Fedora 10 or Ubuntu 9.04. We have evaluated it with a number of different scenarios, including the use of traditional process monitoring tools, i.e., *strace*/*ltrace*/*gdb*, to monitor an out-grafted process from another VM. Note that these fine-grained process monitoring tools cannot be natively supported if the semantic gap is not effectively removed. Moreover, we also show that advanced (hardware-assisted) monitoring tools [26] can be deployed in the security VM to monitor a process in the production VM, while they may be inconvenient or even impossible to run inside the production VM. The performance evaluation with a number of standard benchmark programs shows that our prototype implementation incurs a small performance overhead and the monitoring overhead is largely confined within the security VM, not the production VM.

The rest of the paper is organized as follows: we first present the system design in Section 2. We then describe the implementation and evaluation in Sections 3 and 4, respectively. We discuss possible limitations and explore future improvements in Section 5. Finally, we describe related work in Section 6 and conclude the paper in Section 7.

2. SYSTEM DESIGN

2.1 Goals and Assumptions

Process out-grafting is a virtualization-based approach that advances current out-of-VM approaches for fine-grained process-level execution monitoring. It is introduced to effectively support existing user-mode process-level monitoring tools while removing the inherent semantic gap in out-of-VM approaches. To achieve that, we have three main design goals.

- *Isolation* Process out-grafting should strictly isolate process monitoring tools from the untrusted process. In other words, the untrusted process will be architecturally confined without unnecessarily exposing monitoring tools. This essentially achieves the same isolation guarantee as existing out-of-VM approaches.
- *Compatibility* Our solution should naturally support existing fine-grained process monitoring tools (e.g., *strace* /*ltrace*/*gdb*) without modification. Accordingly, all required semantic information by these tools need to be made available in the security VM.
- *Efficiency* Process out-grafting needs to efficiently support existing process monitoring tools without much additional performance overhead caused by isolation. Due to its process-level granularity, we also need to localize the monitoring overhead to the monitored process, without unnecessarily impacting the production VM as a whole.

In this work, we assume no trust from the suspect process being monitored. An attacker may also introduce malicious software (either user-mode or kernel-mode) to compromise the production VM. However, we assume the presence of a

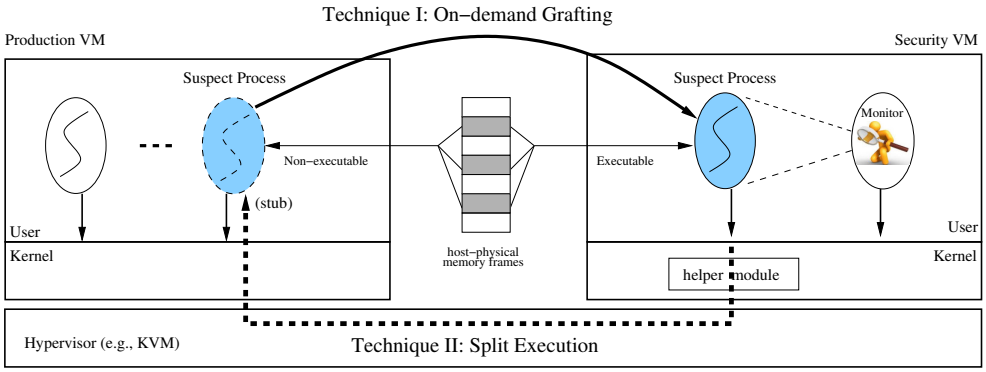

Figure 1: An Overview of Process Out-Grafting

trusted hypervisor [25, 40], which properly enforces the isolation between running VMs. By focusing on fine-grained user-mode process-level monitoring, our system does not address OS kernel monitoring. But a compromised OS kernel should not affect our design goals. Also, we do not attempt to hide the fact that an out-grafted process or malware is being actively monitored. But we do guarantee that the monitoring process itself cannot be disabled by the monitored process.

Figure 1 shows the overview of our system with two key techniques: *on-demand grafting* and *split execution*. Before presenting each technique in detail, we define some terminology used throughout the paper. Our system is designed to work with hardware virtualization extensions available in commodity CPUs, including support for efficient memory virtualization. Leveraging the underlying hardware support, a CPU can enter either host or guest mode. The hypervisor code runs directly in host mode while the VM runs in guest mode. In a virtualized system, there are three different kinds of memory addresses: A *guest-virtual* address is the virtual address observed by a running process inside the guest; a *guest-physical* address is the physical addresses seen by the guest when it runs in the guest mode; a *host-physical* address is the actual machine address seen by the CPU hardware. While executing in guest-mode, a VM never sees the host-physical addresses directly.

We point out that when memory virtualization support is enabled, the CPU utilizes an additional level of page tables managed by the hypervisor to translate from guest-physical to host-physical addresses (i.e., the Nested Page Table (NPT) [7] in AMD CPUs or the Extended Page Table (EPT) [18] in Intel CPUs). Since our current prototype uses an Intel processor, we simply use the term EPT to represent both. With that, the guest OS is free to manage its own guest-physical page frame allocation, with no intervention from the hypervisor. The hypervisor controls only the EPT to allocate host-physical memory, as needed for the guest. While the CPU EPT support significantly improves performance for the running VM [6, 7, 18], such support also poses challenges in our design and some of them will be highlighted below as we describe our system.

2.2 On-demand Grafting

Our first key technique is developed to re-locate a suspect process running in a production VM to a security VM for close inspection. Specifically, it enables efficient, native inspection from existing process-level monitoring tools by avoiding unnecessary hypervisor intervention and eliminating the inherent semantic gap from VM isolation. Relocating a suspect process can be initiated as determined by an administration policy, say a process can be brought under scrutiny either periodically, at random time intervals or using certain event triggers. The monitoring duration can be arbitrary, including the entire lifetime of a process. Regardless of the out-grafting policy, in this paper, we mainly focus on the mechanisms for the out-grafting support.

In order to out-graft a running process, we will need to first accurately locate it (e.g. using the base address of its page table directory). Once it is located, the hypervisor can then redirect or transfer its execution from the production VM into the security VM. In the following, we examine when, what, and how to transfer the suspect process execution across the two VMs.

2.2.1 *When to Out-graft*

To determine the appropriate moment for process execution transfer, we need to ensure it is safe to do so, i.e., the transfer will not corrupt the execution of the out-grafted process and the OS kernel. Particularly, once a process is selected for out-grafting, the hypervisor first pauses the production VM, which is akin to a *VM Exit* event and causes the VM's virtual CPU (VCPU) state to be stored in hypervisor-accessible memory. At this particular time, the to-be-grafted process may be running in either user- or kernel-mode. (If it is not actively running, it is then waiting in the kernel-mode to be selected or dispatched for execution.) If the VCPU state indicates the VCPU is executing the process in user mode, we can immediately start out-grafting the process.

On the other hand, if the VCPU was running in privileged mode (in the context of either the suspect process or another process), we should not start the out-grafting process to avoid leading to any inconsistency. For example, the suspect process may have made a system call to write a large memory buffer to a disk file. If its execution is transferred at this point to another VM, we may somehow immediately resume execution (in the security VM) at the next user-mode instruction following the system call. As we are only transferring the user-mode execution, this will implicitly assume the production VM kernel has already completed servicing the system call, which may not be the case. Therefore, we choose to wait till the process is selected to execute and eventually returns to user-mode. One way the hypervisor could detect this is by monitoring context switches that occur inside the VM. However, in systems that support EPT, the hypervisor is no longer notified of in-VM context switches. Instead, based on the process' page tables, we mark the

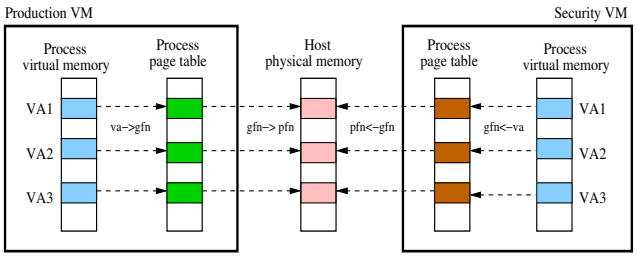

Figure 2: Out-grafted Process Memory Mapping in Production and Security VMs (*va*: virtual address; *gfn*: guest frame number; *pfn*: physical frame number)

corresponding user-level host-physical pages non-executable (NX) in the EPT. When the kernel returns control of the process back to user-mode, it will immediately cause a trap to the hypervisor and thus kick off our out-grafting process.

2.2.2 What to Out-graft

After determining the right moment, we then identify the relevant state that is needed to continue the process execution in the security VM. As our focus is primarily on its user-mode execution, we need to transfer execution states that the user-mode code can directly access (i.e. its code and data). It turns out that we only need to transfer two sets of states associated with the process: the execution context (e.g., register content) and its memory page frames. The hypervisor already identifies the process' register values from the VCPU state (stored in the hypervisor-accessible memory). To identify its memory page frames, we simply walk through the guest OS-maintained page tables (located from the guest *CR3*) to identify the guest-physical page frames for the user-mode memory of the process. For each such page, we then further identify the corresponding host-physical page frame from the EPT. At the same time, we mark NX bit on each user-mode page frame in EPT that belongs to the process. (Although this seems to duplicate the setting from Section 2.2.1, this is required in case the guest OS may allocate new pages right before we start to out-graft.) After that, if the user-level code is executed, inadvertently or maliciously, when the process has been out-grafted for monitoring, the hypervisor will be notified. Note that we mark the NX bit in the EPT, which is protected from the (untrusted) production VM.

We point out that the transferred resources or state do not include those OS kernel-specific states, which the process may access only via system calls (e.g. open file descriptors and active TCP network connections). This is important from at least three different aspects. First, the OS kernel-specific states are the main root cause behind the semantic gap challenge. Without the need of interpreting them, we can effectively remove the gap. Second, keeping these specific states within the production VM is also necessary to ensure smooth continued execution of the out-grafted process in the security VM – as the system call will be redirected back to the production VM. It also allows for later process restoration. Third, it reduces the state volume that needs to be transferred and thus alleviates the out-grafting overhead.

2.2.3 How to Out-graft

Once we identify those states (e.g., the execution state and related memory pages), we then accordingly instantiate them in the security VM. There are two main steps. First, we lock down the guest page table of the out-grafted process in the production VM. Specifically, we mark the page frames that contain the process' page table entries as read-only in the EPT so that any intended or unintended changes to them (e.g. allocating a new page or swapping out an existing page) will be trapped by the hypervisor. This is needed to keep in-sync with the out-grafted process in the security VM. These hardware-related settings are the only interposition we need from the hypervisor, which are completely transparent to, and independent of the monitoring tools (in the security VM). The lock-down of page tables is due to the previously described lack of hypervisor intervention over in-guest page tables. In our system, these settings are temporary and only last for the duration of our monitoring.

Second, we then populate the transferred states in the security VM. For simplicity, we collectively refer to those states as S_{rd}. For this, we prepare a helper kernel module (LKM) running inside the security VM. The hypervisor issues an upcall to the helper module to instantiate S_{rd}. In that case, the helper module retrieves S_{rd} and creates a process context within the security VM for the out-grafted process to execute. At this point, the memory content of the process needs to be transferred from the production VM to the security VM. In a non-EPT supported system, the hypervisor could simply duplicate the page table between the production and security VMs. In the presence of EPT however, we aim to avoid large memory transfers by enabling the memory transfer as follows: The helper module allocates the guest-physical page frames for those virtual addresses that were present in S_{rd} and sends this information to the hypervisor; The hypervisor then simply maps each such page to the host-physical page frame for the corresponding virtual address in the production VM. In other words, this mechanism ensures that a user-level virtual address A of the out-grafted process in the security VM and the user-level virtual address A of the process in the production VM are ultimately mapped to the same host-physical page (as illustrated in Figure 2). After that, the helper also ensures that any system call from the out-grafted process will not be serviced in the security VM. Instead, they will be forwarded back to the production VM and handled by our second key technique (Section 2.3).

When a process is out-grafted for monitoring, its state in the production VM is not destroyed. As mentioned earlier, the production VM still maintains the related kernel state, which not only serves the forwarded system calls but also greatly facilitates the later restoration of the process from the security VM back to the production VM. Meanwhile, because of the separate maintenance of the process page tables inside both VMs, we need to ensure they are kept in-sync. In particular, the production VM may make legitimate changes (e.g. swapping out a page). To reflect these changes back in the security VM, our previous read-only marking on related page tables can timely intercept any changes and then communicate the changes back to the security VM.

With the populated states in a new process inside the security VM, existing process-level monitoring tools such as *strace*, *ltrace*, and *gdb* can naturally access its state or monitor its execution. For example, when the out-grafted process executes system call instructions in the security VM (although they are not actually serviced by the security VM kernel), these can be examined in a semantically-rich manner

(i.e., interpreting the arguments) without any modification to existing tools. Specifically, different from prior out-of-VM approaches, the monitor in our case no longer needs to walk through external page tables to identify the physical addresses for examination. In other words, they can be transparently supported! Finally, in order to support tools that may need to access disk files used by the monitored process, we make the file system that is used by the production VM available in the security VM. We mount this file system as read-only and non-executable. Note that the file system is accessed only by the monitor to access any semantic information. The requests by the out-grafted process to access files are not handled in the security VM, but in the production VM through forwarded system calls.

2.3 Mode-sensitive Split Execution

After selecting and out-grafting a process to the security VM, our second key technique ensures that it can smoothly continue its execution in the security VM, even though the out-grafted process may consider itself still running inside the same production VM. Also, we ensure that the untrusted process cannot tamper with the security VM, including the security VM's kernel and the runtime environment (libraries, log files etc.). We achieve this by splitting the monitored process' execution between the two VMs: all user-mode instructions execute in the security VM while the rest execute in the production VM. In the following, we describe related issues in realizing this mechanism and our solutions.

2.3.1 System Call Redirection

To continue the out-grafted process execution and isolate it from the security VM, there is a need for it to access the kernel-specific resources or states maintained in the production VM. For instance, if the process already opened a file for writing data, after the relocation to the security VM, it must be able to continue writing to it. As the process needs to make system calls to access them, we therefore intercept and forward any system call from the out-grafted process back to the production VM.

To achieve that, there exist two different approaches. The first one is to simply ask hypervisor to intervene and forward the system call (by crafting an interrupt and preparing the appropriate execution context). However, it will unfortunately impact the entire production VM execution. The second one is to have a small piece of stub in place of the out-grafted process. The stub is mainly designed to receive forwarded system calls from the security VM, invoke the same in the production VM, and then return the results back to the security VM. We take the second approach in our design as it can effectively localize the effect within the out-grafted process itself and avoid heavy hypervisor intervention for every forwarded system call.

The placement of the stub code deserves additional consideration. Since the guest page tables are not managed by the hypervisor, it cannot simply allocate a separate guest-physical page for the stub code. As our solution, we choose to temporarily "steal" an existing code page in the process, by saving the original content aside and overlaying it with the stub's code. Recall (from Section 2.2) that the host-physical memory frames corresponding to the process address space are mapped in both VMs. To steal a code page, the corresponding host-physical page frame is replaced with another one that contains the stub code. To protect it from

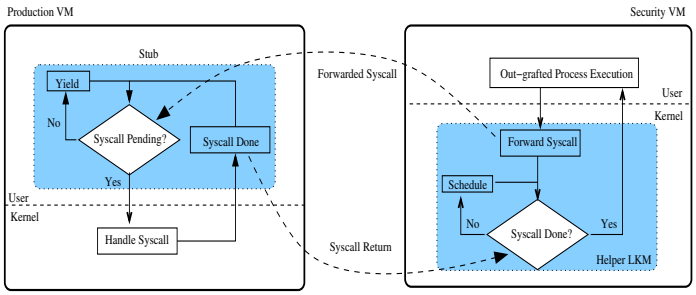

Figure 3: The Interplay Between the Stub (in Production VM) and the Helper LKM (in Security VM)

being tampered by the production VM, we mark it non-writable in the production VM's EPT for the duration of out-grafting.

The stub's main function is to proxy the forwarded system call from the security VM and replay it in the production VM. In order to facilitate direct communication without requiring hypervisor intervention, during the out-grafting phase, we set up a small shared communication buffer accessible to the stub and the helper module in the security VM. Also, note that if a system call argument is a pointer to some content in the process address space, our design ensures that there is no need to copy this data between the production and security VM. Since the process' memory is mapped in both VMs, the production VM kernel can simply access this as it would for any regular process. This eliminates unnecessary memory copy overhead.

An interesting dilemma arises if the stub code needs to use the stack. Specifically, the process' stack is part of its data pages and these pages are mapped faithfully in both VMs. Any user-space instructions in the out-grafted process will use this memory region as the stack. Further, these memory regions may also contain the arguments to be passed down to the production VM kernel for system calls. Hence, the stub code cannot use the original stack pointer as its stack frame, or it will collide with the user-mode execution in the security VM. In our design, we simply avoid using the stack in the stub code execution.

The interplay between the stub code and our helper module is shown in Figure 3. The stub code is self-contained and will directly invoke system calls without relying on any additional library calls. Its size has been kept to the minimum since we are overlaying over existing code memory for this. Also, the stub can handle signals from the production VM kernel and communicate them to the security VM helper module. Specifically, when the production VM invokes a previously registered signal handler, it will cause an exit to the hypervisor, which will be eventually relayed to the helper module to deliver the same signal to the out-grafted process for handling.

2.3.2 Page Fault Forwarding

In addition to forwarding system calls, we also need to forward the related page faults for the out-grafted process. More specifically, at the time when a process' execution is being redirected to the security VM, some of its pages may not be present (e.g., due to demand paging). Hence, when the process is executing in the security VM, if it accesses a non-present page, it will be trapped. Since the security VM has no knowledge of how to correctly allocate and populate

a new page (e.g., in case of a file-mapped page), we forward page faults to the production VM. This also ensures that when the production VM kernel attempts to access a new memory page during process execution, it will be immediately available.

To forward related page faults, out helper module in the security VM registers itself to receive notification from the security VM kernel for any page fault (or protection fault) related to the out-grafted process. When it receives such a notification, it places the virtual address that causes the page fault and a flag to indicate a read-fault (including instruction fetches) or a write-fault, in the communication buffer and notifies the stub in the production VM. To service a page fault, the stub will attempt to either read in a byte from the address or write a byte to the address. This will cause the same page fault in the production VM. After completing this read or write, the stub sends a notification back to our helper module. The stub's write to the memory will be immediately overwritten with the correct value by the out-grafted processes when it re-executes a faulting instruction, thus ensuring correct process execution. Unlike a system call return value, the stub cannot return the guest-physical page allocated for the new address in the production VM. Instead, since we have write-protected the process page tables, when the production VM makes a change to it (i.e. to set the page table entries for the new page), this is intercepted by the hypervisor and our helper module will be notified. After that, it then allows the page fault handling routines in the security VM to continue processing. As previously described, instead of duplicating a memory page, we simply tell the hypervisor to map the same host-physical memory page corresponding to new page in the production VM.

Later on, if we decide not to continue monitoring the out-grafted process, we can place it back in the production VM. In this case, since we have already maintained synchronized page tables between the two VMs, we only need to restore the execution context states back in the production VM. For those memory pages overlaid for the stub use, they need to be properly restored as well. Specifically, if we find the VCPU was executing stub code, which is located in the user mode, we can simply change the VCPU state contents to restore the execution context. Otherwise, we need to detect when the user-mode execution resumes by marking the stub's code page as NX and then restore the VCPU values. In the security VM, any state for the out-grafted process is then simply destroyed (i.e. guest-physical page frames allocated to it are freed and helper module requests the hypervisor to release the mappings in the security VM EPT).

3. IMPLEMENTATION

We have implemented a process out-grafting prototype by extending the open-source KVM [2] hypervisor (version 2.6.36.1). All modifications required to support out-grafting are contained within the KVM module and no changes are required to the host OS. Our implementation increases KVM's 36.5K SLOC (x86 support only) by only 1309 SLOC, since most functionality required for out-grafting (e.g. manipulating a VM's architectural state) is already present in the stock KVM. Our prototyping machine runs Ubuntu 10.04 (Linux kernel 2.6.28) with an Intel Core i7 CPU and hardware virtualization support (including EPT). Though our prototype is developed on KVM, we believe it can be similarly implemented on other hypervisors such as Xen and VMware ESX. Our current prototype supports both 32-bit Fedora 10 and Ubuntu 9.04 as guests (either as the production VM or the security VM). In the rest of this section, we present details about our KVM-based prototype with Intel VT support.

3.1 On-demand Grafting

KVM is implemented as a loadable kernel module (LKM), which once loaded extends the host into a hypervisor. Each KVM-powered guest VM runs as a process on the host OS, which can execute privileged instructions directly on the CPU (in the so-called guest mode based on hardware virtualization support). The VM's virtual devices are emulated by a user-level program called QEMU [4] that has a well-defined *ioctl*-based interface to interact with the KVM module. In our prototype, we extend the interface to define the out-grafting command *graft_process*. This command accepts a valid page table base address (or guest *cr3*) to directly locate the process that needs to be out-grafted. The guest *cr3* could be retrieved either by converting from the process ID or process name via VM introspection, or directly reading from the guest kernel with a loadable kernel module.

Once the *graft_process* command is issued, the KVM module pauses the VM execution, which automatically saves the VM execution context information in the VM control structure (VMCS [19]). KVM provides a number of functions that wrap CPU-specific instructions to read and write to different fields in this VMCS structure. From this structure, we retrieve the current value of the VM's *cr3* register and compare with the argument to the *graft_process*. If they match and the current VCPU is running in user-mode, we then read the VCPU's register values (eax, ebx, esp etc.) and store them in memory. Otherwise, we take a walk through the process page table from the given base address to locate all the user-mode guest-physical page frames (*gfn*) used by the to-be-grafted process. Using the *gfn_to_pfn()* function in KVM, we obtain the host-physical page frames (*pfn*) and set the NX bit in the VM's EPT. As a result, when the VCPU returns back to execute any user-mode instruction in this process, an *EPT violation* occurs and the control is transferred to KVM. At this point, KVM can read the register values from VMCS and save a copy.

After retrieving its execution context (i.e., those register values), we then examine the page table (from the given base address) to determine which virtual addresses in the suspect process have pages allocated or present. That is, it determines the *gfn* for each virtual address and then invokes *gfn_to_pfn()* to determine the corresponding *pfn* in the production VM's EPT. KVM stores this information in a local buffer. It then makes an upcall in the form of a virtual IRQ to the security VM (using the *inject_irq()* function).

In the security VM, we have implemented a helper module which is registered to handle this (virtual) IRQ. The helper module then instantiates a process context for the out-grafted process so that it can continue the execution. Specifically, it first allocates a memory buffer and makes a hypercall to KVM with the address of this memory buffer, so that KVM can copy the state S_{rd} to it. Since spawning a new process is a complex task, our helper module creates a simple "dummy" process in the security VM, which executes in an infinite sleep loop. Upon the IRQ request from KVM, it proceeds to replace the dummy process with the

out-grafted process state. Specifically, our helper module retrieves register values from S_{rd} and instantiates them in the dummy process (using the *pt_regs* structure) After that, we simply destroy any pages (i.e., via *do_unmap()*) allocated to the dummy process and then allocate "new" pages for virtual addresses, as indicated in S_{rd}. Note that we do not actually allocate new host-physical memory pages to accommodate these transferred memory pages. Instead, KVM simply maps (*__direct_map()*) those host-physical memory frames that are used by the out-grafted process to the dummy process in the security VM. After the mapping, the out-grafted process is ready to execute its user-mode instructions in the security VM. An execution monitor (such as *strace*) in this security VM can now intercept the process-related events it is interested in.

3.2 Mode-sensitive Split Execution

After relocating the suspect process to the security VM, any system call made from it will be intercepted by our helper module and forwarded back to the production VM. Specifically, our helper module wraps the exposed system call interface in the security VM to the out-grafted process. For each intercepted system call, we collect the corresponding system call number and its argument values in a data structure (*sc_info*) and save it in the shared communication buffer so the stub code in production VM will pick it up to invoke the actual system call. (As noted in Section 2.3.1, memory contents for pointer arguments are not copied since the user-level memory is present in both VMs).

More specifically, the stub code is created when the process is being out-grafted from the production VM to the security VM. Its main purpose is to proxy the forwarded system calls from the security VM to the production VM. As mentioned in Section 2.3.1, we need to "steal" an existing code page to host the stub code. We have written the stub code in a few lines of assembly with an overall size of 167 bytes. The stub code itself does not make any use of a stack while executing (Section 2.3.1). Similarly, with the help of KVM, we set up a shared communication buffer between the stub code and our helper module. When a system call is to be forwarded to the production VM, our helper module copies the *sc_info* data structure described above to this buffer. It then sets a flag (in the same buffer) to indicate to the stub that a new system call is to be serviced and waits in a loop for this flag to be cleared by the stub. To avoid blocking the entire security VM during this time, it yields from inside the loop.

The stub code checks the flag and then retrieves the *sc_info* values and copies them to the registers in the production VM. It then invokes the requested system call so that the production kernel can service the request. Once the request is complete, the stub places the return value in the buffer and modifies the flag indicating service completion. After that, our helper module in the security VM can now return the same value to the out-grafted process. In addition to forwarding system calls from the out-grafted process, we also need to forward related page faults to the production VM. Naturally, we leverage the above communication channel between the stub code and our helper module. Specifically, when a page fault occurs in the out-grafted process (while running in the security VM), the security VM's page fault handler invokes a callback defined in our helper module, which then forwards the page fault information to the stub. Based on the fault information, the stub either reads or writes a dummy value in the faulting address in the production VM to trigger a page fault of the same nature in the production VM. When the page fault handler in the production VM attempts to update the page tables with a new page table entry (pointing to a new page frame we denote by *gfn_p*), this causes an "EPT violation" and control is transferred to KVM. KVM examines the root cause and saves a copy of the *gfn_p->pfn* mapping while fixing the violation and resuming the production VM. The helper module notifies KVM with the new guest-physical frame it allocated in the security VM (*gfn_s*). KVM then maps *gfn_s* to *pfn* and ensures the same memory content is available in both VMs. After that, the out-grafted process can continue its execution with the new memory page.

3.3 Process Restoration

In our prototype, process out-grafting is initiated through a QEMU command *graft_process*. As mentioned earlier, other mechanisms can also be added to trigger the out-grafting process. An example is an event-based trigger that runs inside the production VM (Section 4).

In our current prototype, we also implemented another QEMU command *restore_process*, which can be invoked to notify KVM (via an *ioctl* interface) to restore the out-grafted process back to the production VM. Similar to the out-grafting procedure, when KVM receives the *restore_process* command, it injects an IRQ to the security VM, which will be received by the helper module. If the module is currently waiting on a forwarded system call's completion, the restoration operation cannot be immediately carried out. Instead, it will wait for the completion of the system call. After that, it fetches register values stored in the process' *pt_regs* data structure and sends this down to KVM with a hypercall. KVM then restores this register state back in the production VM. Due to the similarity with the earlier out-grafting steps, we omit the details here. The key difference however is that, instead of copying values *from* the VCPU fields, we copy values *to* it. For the page tables, as they are kept in-sync between the two VMs, no further actions will be needed. For those process contexts and guest memory pages allocated to the out-grafted process in the security VM, we simply discard them. At this point, the process can seamlessly resume its execution in the production VM.

4. EVALUATION

In this section, we first perform a security analysis on the isolation property from our approach. Then, we present case studies with a number of execution monitoring tools. Finally, we report the performance overhead with several standard benchmarks.

4.1 Security Analysis

Monitor isolation and effectiveness To allow for fine-grained out-of-VM process monitoring, one key goal is to ensure that the process monitoring tool and its supporting environment cannot be tampered with or disabled by the out-grafted process. In the following, we examine possible attacks and show how our system can effectively defend against them. Specifically, one main way that a suspect process can tamper with another process (or the monitor in our case) is through system calls. However, such attack will not work since our system strictly forwards all system calls from

the suspect process back to the production VM. Moreover, the controlled interaction is only allowed from the monitoring process to the suspect process, *not* the other way around. From another perspective, the suspect process may choose to attack the (production VM) OS kernel when it services system calls (for e.g. exploiting a buffer overflow by sending in an invalid argument). Such attack will only impact the production VM and its own execution.

According to our threat model (Section 2), we stress that our system does not attempt to guarantee stealthy monitoring as out-grafted monitoring could be detected by sophisticated malware. But we do enable reliable monitoring in protecting our system from being tampered with by the suspect process. In addition, a strong administrative policy might reduce the time window for such out-grafting detection. For example, out-grafting can be initiated randomly (at any instance in a process' lifetime) and can span arbitrary durations. In such cases, malware would be forced to continuously check for out-grafting which can be costly and expose its presence. Note that the out-grafted process is ultimately serviced by the untrusted production VM, and as such, we cannot guarantee that it accurately services the out-grafted process' system calls, but any inappropriate handling of system calls will not violate the isolation provided by our approach.

Protection of helper components The out-grafting operation itself is initiated and controlled by the hypervisor and cannot be disabled by a malicious production VM kernel. However, there are two helper components in the production VM to support out-grafting: the stub code and shared communication buffer, which may be open to attack. We point out that since the stub code's host-physical page is marked as read-only in the EPT, any malicious attempts to write to it will be trapped by the hypervisor. The stub code's guest virtual-to-physical mapping cannot be altered by the production VM since the page tables of the process are write-protected by the hypervisor. The untrusted kernel in the production VM is responsible for scheduling the stub process as well as saving and restoring its execution context according to the scheduling policy. If it tampers with the execution context states (such as the instruction pointer), then the stub code itself will not execute correctly, which cascadingly affects the execution of the suspect process itself. As mentioned earlier, if the production VM does not properly serve the forwarded system calls or attempts to alter the system call arguments or return incorrect results, such behavior may result in incorrect execution for the out-grafted process, but will not affect the isolation or the integrity of our monitoring process.

4.2 Case Studies

Next, we describe experiments with a number of execution monitoring tools, including the most common ones: *strace*, *ltrace*, *gdb*, as well as an OmniUnpack[26]-based tool (to detect malware unpacking behavior). The common tools are used to demonstrate the effectiveness of our approach in removing the semantic-gap, while the OmniUnpack tool requires special hardware support for the monitoring in the security VM and such support may not be enabled or provided in production VM. As a test process for out-grafting, we chose the *thttpd* web server that uses both disk and network resources and has a performance benchmark tool readily available to automatically exercise it.

4.2.1 Tracing System Calls

In our first experiment, we demonstrate semantically-rich system call tracing. This type of monitoring has been widely applied to detect malicious behavior [13] such as accesses to sensitive resources or dangerous system calls. For this, we install the standard Linux *strace* tool in the security VM. *strace* makes use of the underlying OS facilities to monitor system calls invoked by another running process, which in our case logically runs in another VM. For each intercepted system call, it retrieves and parses the arguments. The results will allow us to know what file was opened by a process, what data is read from it etc.

In prior "out-of-VM" systems, the code to determine system call number and interpret each of its arguments has to be completely re-written. In fact, one of our earlier systems, i.e., VMscope [20], took one of the co-authors more than one month to correctly intercept and parse the arguments of around 300 system calls supported in recent Linux kernels. This task is expected to become even more complicated especially for closed-source OSs.

To better understand the effectiveness, we perform a comparative study. Specifically, we first run *strace* inside the production VM to monitor invoked system calls from *thttp* when it handles an incoming HTTP request. After that, we out-graft *thttp* and run *strace* in the security VM for out-of-VM monitoring when it handles another incoming HTTP request. Our results are shown in Figure 4. From the figure, we can verify that both *strace* runs lead to the same system call patterns in the handling of incoming HTTP requests by accurately capturing system calls invoked by the same *thttpd* process and interpreting each related argument.

4.2.2 Tracing User-level Library Calls

In our second experiment, we show the capability of reusing existing tools for user-level library call tracing. User-level library call tracing is a fine-grained monitoring technique that allows for understanding which library functions are being used by a running process and what are their arguments. It has advantages over system call tracing in collecting semantically-rich information at a higher abstraction level.

As one can envision, the number of library functions available to a program and the type and definition of each argument for such functions can be very large. This can make it complex, expensive, or even impossible to examine such events in a semantically-rich manner using prior "out-of-VM" approaches. Fortunately, in our approach, we can simply re-use an existing tool *ltrace* to intercept and interpret user-level library calls of a running process in one VM from another different VM. In our experiment, we found that *ltrace* extracts process symbol information from the process' binary image on disk. As we mounted the production VM's filesystem read-only in the security VM, *ltrace* works naturally with no changes needed. Our results show that *ltrace* indeed accurately captures and interprets the user-level library calls invoked by the out-grafted *thttpd* process. In a similar setting, we replace *ltrace* with *gdb*, which essentially allows for debugging an in-VM process from outside the VM!

4.2.3 Detecting Malware Unpacking Behavior

Most recent malware apply obfuscation techniques to evade existing malware detection tools. Code packing is one of the popular obfuscation techniques [17]. To detect packed code,

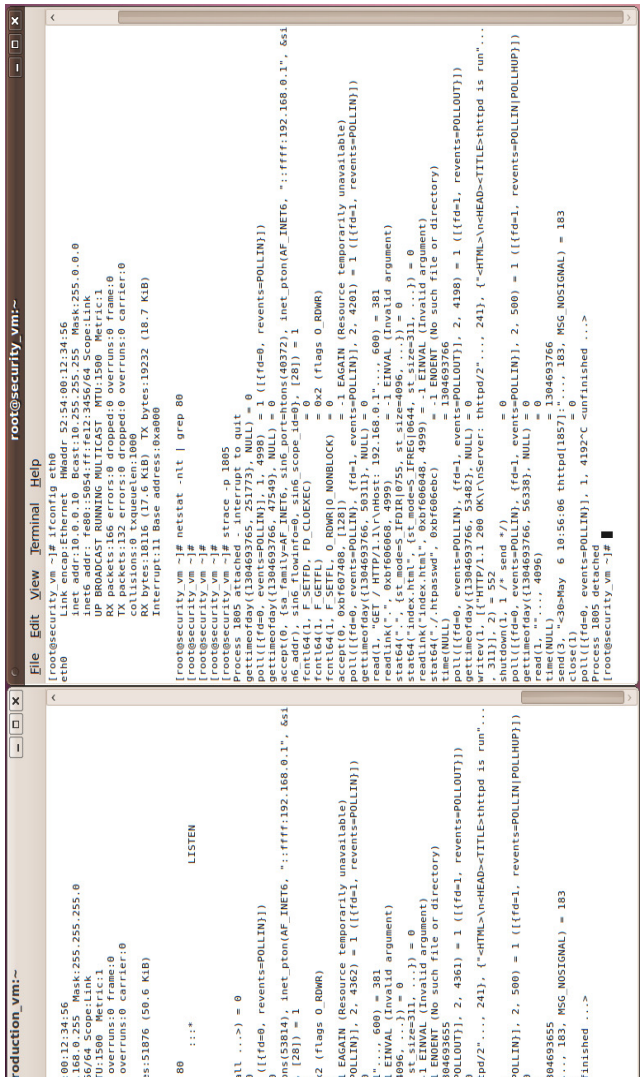

Figure 4: Comparison of strace results of the thttpd server when running inside the VM (on the bottom) and out-grafted to another VM (on the top)

efficient behavioral monitoring techniques such as OmniUnpack [26] have been developed to perform real-time monitoring of a process' behavior by tracking the pages it writes to and then executes from. When the process invokes a "dangerous" system call, OmniUnpack looks up its page list to determine whether any previously written page has been executed from. If so, this indicates packing behavior, at which point a signature-based anti-virus tool can be invoked to check the process' memory for known malware. Since OmniUnpack was developed only for Windows and also is not open source, we wrote a Linux tool that faithfully implements OmniUnpack's algorithm. We stress that if OmniUnpack was previously available for Linux, this porting step would not be necessary. In our Linux porting, we do not need to bridge the semantic gap or be aware of any prior introspection techniques. Instead, we just envision a Linux tool that will be used *in-host*. This experience also demonstrates the benefits from our approach.

In our test, we use the freely available UPX packer [5] to pack the Kaiten bot binary [1]. In this experiment, we also utilize a security-sensitive event trigger that initiates out-grafting when a suspect process invokes the *sys_execve* system call. The trigger is placed such that just before the system call returns to user-mode (to execute the first instruction of the new code), KVM is invoked to out-graft the process' execution to the security VM. Inside the security VM, we run the OmniUnpack tool to keep track of page accesses by the process. Since the system calls invoked by the process are also available for monitoring, OmniUnpack successfully detects the packing behavior.

We highlight several interesting aspects this experiment demonstrates. In the past, packer detection has required a trade-off between tool isolation and performance overhead. Specifically, in-host tools [26, 33], including OmniUnpack can efficiently detect packing behavior, but they are vulnerable to attack. "Out-of-VM" techniques [11] ensure packer detection is isolated, but introduce very high overhead, largely limiting its usability for offline malware analysis, not online monitoring. Using process out-grafting, we are able to effectively move an in-host tool "out-of-VM" without introducing significant overhead while still providing the needed isolation. Another interesting aspect is due to the fact that OmniUnpack requires the NX bit in the guest-page tables. For 32-bit Linux, this bit is available only if PAE support is enabled. In our experiments, we enabled PAE only in the security VM, whereas in the production VM page tables, the NX bit is still not available. Thus, if the monitoring tool requires additional features or support from the underlying OS, even if this support is not present in the production VM, we can take advantage of it in the security VM. Also, we point out that the process out-grafting happens at the very beginning of its execution. When a process begins execution, most of its code pages are not yet mapped in by the OS. As such, this experiment also thoroughly tests the page fault forwarding mechanisms in our system (Section 2.3).

4.3 Performance

To evaluate the performance overhead of our system, we measure two different aspects: the slowdown experienced by the production VM when a process is out-grafted for monitoring as well as the slowdown to the out-grafted process itself. The platform we use is a Dell T1500 system containing an Intel Core i7 processor with 4 cores, running at 2.6 GHz, and 4 GB RAM. The host OS is 32-bit Ubuntu 10.04 (Linux kernel 2.6.32) and the guests run 32-bit Fedora 10 (Linux kernel 2.6.27). Both VMs are configured with 1 virtual CPU each. The production VM is configured with 2047 MB memory and the security VM is configured with 1 GB memory. Table 1 lists the detailed configuration. In all experiments, the two VMs are pinned to run on separate CPU cores in the host (using the Linux *taskset* command).

Production VM overhead First, we measure the slowdown experienced by the VM (i.e. other normal pro-

Table 1: Software Packages in Our Evaluation

Name	Version	Configuration
Host OS	Ubuntu 10.04	Linux-2.6.32
Guest OS	Fedora 10	Linux-2.6.27
SPEC CPU 2006	1.0.1	integer suite
Apache	2.2.10	ab -c 3 -t 60
thttpd	2.25b	ab -c 1 -t 60

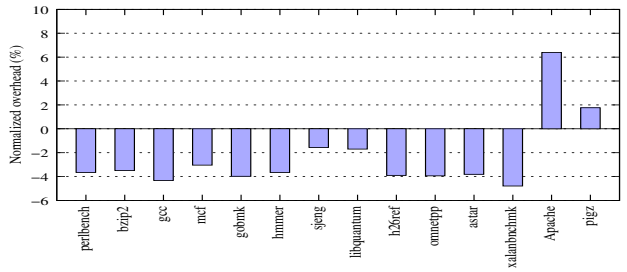

Figure 5: Production VM Slowdown with a Contending Process Out-grafted

cesses running in it) when we out-graft an unrelated process for monitoring. Specifically, we choose a standard CPU benchmark program, i.e., SPEC CPU 2006, and run it twice (1) either with another CPU-intensive process that spins in an infinite loop inside the same VM (2) or with the CPU-intensive process out-grafted to another VM. Our results show benchmarks experience speedups after out-grafting the CPU-intensive process. This is expected as a contending process has been moved to execute in a different VM for monitoring, which is running on a different core. This also confirms the monitoring overhead has been localized inside the security VM, not the production VM. Next, we measure the impact to *Apache* and *pigz* (or parallel *gzip*) when they run together either with *thttp* out-grafted or not. In our experiments, *Apache* and *thttpd* listen on different TCP ports, but their network traffic is handled by the same production VM kernel. Our results are shown in Figure 5. It is interesting to note that when *thttpd* is out-grafted, it is scheduled more often (since both *Apache* and *pigz* dominate it when they run in the same VM). However, the redirected system calls from *thttpd* will not get serviced until the stub is scheduled for execution in the production VM, thus its impact to the dominant processes is still low. Finally, we also measured the time it takes to identify the state for out-grafting during which time the production VM is paused. While this would vary depending on the memory size of a process, we observed an average time of $\sim 250\mu$s in our current experiments. Opportunities still exist to further reduce it (e.g., with lazy updates – Section 5).

Out-grafted process slowdown Second, we measure the slowdown an out-grafted process may experience due to the fact that it is running in a different VM. For this, we first measure slowdown in two out-grafted processes: (1) The first one is a simple file copy command that transfers a *tar* file (421MB) from one directory to another in the production VM, which will result in lots of file-accessing system calls being forwarded. (2) The second is *thttpd*, for which we generate traffic using the *ab* benchmark program. In each set of experiments above, we have rebooted both VMs and physical machine after each run to avoid any caching interference. The average slowdown experienced by them is 35.42% and 7.38%, respectively. Moreover, we run a micro-benchmark program to measure the system call delay experienced by *sys_getpid()*. Our result shows that the average time of invoking *sys_getpid()* is $\sim 11\mu$s. As *sys_getpid()* simply obtains the process ID (after it has been forwarded), its slowdown is approximately equivalent to the system call forwarding latency.

5. DISCUSSION

While our prototype demonstrates promising effectiveness and practicality, it still has several limitations that can be improved. For example, our current prototype only supports out-grafting of a single process. A natural extension will be the support of multiple processes for simultaneous out-grafting. Note it is cumbersome and inefficient to iterate the out-grafting operation for each individual process. From another perspective, simultaneous out-grafting of multiple processes can also lead to the unique scenario where multiple security VMs can be engaged in monitoring different groups of out-grafted processes or different aspects of behavior.

Also, our current way of handling *sys_execve()* can be improved. Specifically, as this system call will completely change the memory layout of the out-grafted process, our current prototype simply chooses to first restore the process back to the production VM and then out-graft again the process immediately after this system call is completed by the production VM. Though this approach can leverage the functionalities we already implemented for *graft_process* and *restore_process*, an integrated solution is still desired. Moreover, our current prototype proactively maps all the (user-mode) memory pages at the very beginning when a process is out-grafted. A better solution will be to only map the currently executing code page to the security VM. For the rest of the pages, they can be lazily mapped when they are being actually accessed. This could further improve our system performance.

One caveat we experienced in our prototype development is related to shared pages. Specifically, most commodity OSs map the same physical pages for common shared library code among different processes. In our system, this means that a single host-physical page can contain code that is used across multiple processes in the production VM. Recall that our system directly maps this host-physical page to the out-grafted process in the security VM. If the monitoring tool modifies such code page (say to install certain code hooks), this could alter other process' behavior in the production VM. Fortunately, this can be resolved in a straightforward manner by co-operating the helper module and KVM to mark all executable code pages for the out-grafted process as read-only in the EPT. By applying the classic copy-on-write technique, if the monitor process (*not* the out-grafted process) attempts to write to this page, a separate copy of the page can be created.

Finally, with the wide adoption of virtualization in data centers, we also envision that different security VMs can be dispatched to each physical machine to inspect running guest VMs and their internal processes (for fine-grained execution monitoring). This is largely feasible as the semantic gap has been effectively bridged to support existing monitoring tools. On the other hand, with our current focus on examining individual suspect process for malicious behavior, we believe other interesting applications and opportunities (e.g., performance monitoring and intelligent parallel job scheduling) remain, which we plan to explore in the future.

6. RELATED WORK

Virtualization has been widely proposed to address various computer system problems, including enhancing the effectiveness and robustness of host-based monitoring tools. Specifically, it has been applied in offline malware analysis [9, 11], honeypot-based malware capture [20], intrusion analysis [23, 24] and malware detection [15, 21]. Among the most notable, Livewire [15] pioneered the concept of placing a monitor "out-of-VM" and applying VM introspection techniques to understand in-VM activities. A number of recent systems address the inherent semantic gap challenge to improve VM introspection for various purposes [8, 10, 21, 30, 31, 37]. For instance, one recent work Virtuoso [12] aims to effectively automate the process of building introspection-based security tools. Another system [10] proposes injecting stealthy agents into a monitored VM to solve the semantic gap problem and enhance out-of-VM tools. Similar to most of these efforts, our approach places security tools out-of-VM. However, our approach mainly differs from them in the way to address the semantic gap challenge. In particular, while prior approaches are sensitive to particular guest kernel versions or patches, our approach *brings* the suspect process to the security tool and allows for native support of existing tools. In other words, by effectively removing the semantic gap, our approach enables *re-use* of existing user-mode process monitoring tools. Further, our approach localizes the impact on the out-grafted process and avoids perturbing the monitored VM as a whole.

From another perspective, one recent system SIM [34] utilizes hardware features to place an "in-VM" monitor in a hypervisor-protected address space. While it is not physically running out-of-VM, SIM still suffers from the semantic-gap and cannot natively support existing monitoring tools. In other words, though the in-VM presence leads to unique performance benefits, there is a need to adapt existing tools to take advantage of the SIM support. Also, the main goal of SIM is to protect "kernel hook"-based monitors. Another recent system Gateway [38] leverages virtualization to detect kernel malware by monitoring kernel APIs invoked by device drivers. In contrast, our focus is for fine-grained process-level execution monitoring (e.g. *ltrace*) that typically requires user-mode interception. Process implanting [22] is another "in-VM" approach, where an "undercover agent" process is dynamically implanted in a target VM for surveillance and repair operations. Contrary to process out-grafting, process implanting relies on the integrity of the target VM's kernel and requires special modification to the program executed by the implanted process.

Process out-grafting requires redirecting the process execution across two different VMs, which bears certain similarities to well-known process migration mechanisms [28, 29, 35, 36]. However, one key difference is that process migration techniques typically move the entire execution states, including kernel-maintained resources while our approach only (temporarily) redirects the user-level execution of a process for secure monitoring. Moreover, most process migration techniques are typically applied for generic purposes such as fault-tolerance and load-balancing [28, 35] and do not consider the isolation challenge for secure monitoring.

We also notice that the idea of system call forwarding has been previously applied in systems to protect a critical application from an untrusted kernel [39]. In this case, a programmer can divide system calls into two sets so that each set will be serviced by either a trusted or untrusted kernel, respectively. In contrast, throughout the out-grafting duration, our system has one single kernel to serve all system calls for the out-grafted process. Moreover, due to the relocated user-level execution, we need to perform mode-sensitive split execution, which leads to an extra but unique need of forwarding page faults. System call forwarding between VMs has also been used [27] to improve the fidelity of runtime environment for better malware behavior monitoring, albeit without isolation guarantees. In contrast, we aim to address compatibility of existing process monitoring tools while ensuring their strong isolation in the context of VM introspection. Further, we dynamically create the memory mapping for the out-grafted process between the production VM and security VM. Also, our on-demand grafting allows for dynamically grafting the execution first and then *restoring* the execution back.

More generally, sandboxing and isolation techniques [14, 16, 32] have been widely researched and applied as effective mechanisms to confine an untrusted process' access to sensitive resources in the host system. Our work is related to them by essentially leveraging the VM isolation provided by the underlying virtualization layer. However, with an out-of-VM approach, process out-grafting can be applied on-demand, which provides certain flexibility in monitoring runtime behavior of suspect processes. Also, our approach is unorthodox when compared with traditional sandboxing and isolation techniques due to its split execution, i.e., the user-mode and kernel-mode execution of an out-grafted process run in two different VMs.

7. CONCLUSION

We have presented the design, implementation and evaluation of process out-grafting, an architectural approach to address isolation and compatibility challenges in out-of-VM approaches for fine-grained process-level execution monitoring. In particular, by effectively relocating a suspect process from a production VM to the security VM for close inspection, process out-grafting effectively removes the semantic gap for native support of existing process monitoring tools. Moreover, by forwarding the system calls from the out-grafted process back to the production VM, it can smoothly continue its execution while still being strictly isolated from the monitoring tool. The evaluation results with a number of performance benchmarks show its effectiveness and practicality.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by the US Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099 and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR and the NSF.

8. REFERENCES

[1] Kaiten. `http://packetstormsecurity.org/irc/kaiten.c`. [last accessed: May 2011].

[2] Kernel Virtual Machine. `http://www.linux-kvm.org`. [last accessed: May 2011].

[3] McAfee Threats Report: Fourth Quarter 2010. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2010.pdf`. [last accessed: May 2011].

[4] QEMU. `http://www.qemu.org`. [last accessed: May 2011].

[5] UPX: The Ultimate Packer for eXecutables. `http://upx.sourceforge.net`. [last accessed: May 2011].

[6] Adams, K., and 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* (2006).

[7] AMD. AMD-V Nested Paging. *AMD White Paper* (2008).

[8] Azab, A. M., Ning, P., Sezer, E. C., and Zhang, X. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Proceedings of the 25th Annual Computer Security Applications Conference* (2009).

[9] Bayer, U., Kruegel, C., and Kirda, E. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research* (2006).

[10] cker Chiueh, T., Conover, M., Lu, M., and Montague, B. Stealthy Deployment and Execution of In-Guest Kernel Agents. In *BlackHat 2009*.

[11] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008).

[12] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy* (2011).

[13] Forrest, S., Hofmeyr, S., and Somayaji, A. The Evolution of System-Call Monitoring. In *Proceedings of the 24th Annual Computer Security Applications Conference* (2008).

[14] Garfinkel, T., Pfaff, B., and Rosenblum, M. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium* (2004).

[15] Garfinkel, T., and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium* (2003).

[16] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium* (1996).

[17] Guo, F., Ferrie, P., and Chiueh, T.-c. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*. (2008).

[18] Intel. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel(R) Technology Journal 10*, 3 (2006).

[19] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual Volume 3: System Programming Guide, Part 1 and Part 2*, (2010).

[20] Jiang, X., and Wang, X. "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection* (2007).

[21] Jiang, X., Wang, X., and Xu, D. Stealthy Malware Detection through VMM-based "Out-of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007).

[22] Gu, Z., Deng, Z., Xu, D., and Jiang, X. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems* (2011).

[23] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (2005).

[24] King, S. T., and Chen, P. M. Backtracking Intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[25] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles* (2009).

[26] Martignoni, L., Christodorescu, M., and Jha, S. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference* (2007).

[27] Martignoni, L., Paleari, R., and Bruschi, D. A Framework for Behavior-Based Malware Analysis in the Cloud. In *Proceedings of the 5th International Conference on Information Systems Security* (2009).

[28] Nuttall, M. A Brief Survey of Systems Providing Process or Object Migration Facilities. *ACM SIGOPS Operating Systems Review 28* (1994).

[29] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The Design and Implementation of Zap: a System for Migrating Computing Environments. *ACM SIGOPS Operating Systems Review 36* (2002).

[30] Payne, B., de Carbone, M., and Lee, W. Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference* (2007).

[31] Payne, B. D., Carbone, M., Sharif, M., and Lee, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy* (2008).

[32] Provos, N. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[33] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (2006).

[34] Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009).

[35] Smith, J. M. A survey of process migration mechanisms. *ACM SIGOPS Operating Systems Review 22* (1988).

[36] Smith, J. M. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. *Berkeley Lab Technical Report* (2002).

[37] Srivastava, A., and Giffin, J. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection* (2008).

[38] Srivastava, A., and Giffin, J. Efficient Monitoring of Untrusted Kernel-mode Execution. In *Proceedings of the 18th Annual Network and Distributed Systems Security Symposium* (2011).

[39] Ta-Min, R., Litty, L., and Lie, D. Splitting Interfaces: Making Trust between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006).

[40] Wang, Z., and Jiang, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010).