2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine

Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS, Purdue University, West Lafayette, IN, USA, 47907-2107
{*gu16, bsaltafo, xyzhang, dxu*}*@cs.purdue.edu*

***Abstract*—Kernel minimization has already been established as a practical approach to reducing the trusted computing base. Existing solutions have largely focused on whole-system profiling – generating a globally minimum kernel image that is being shared by all applications. However, since different applications use only part of the kernel's code base, the minimized kernel still includes an unnecessarily large attack surface. Furthermore, once the static minimized kernel is generated, it is not flexible enough to adapt to an altered execution environment (e.g., new workload). FACE-CHANGE is a virtualization-based system to facilitate dynamic switching at runtime among multiple minimized kernels, each customized for an individual application. Based on precedent profiling results, FACE-CHANGE transparently presents a customized kernel view for each application to confine its reachability of kernel code. In the event that the application exceeds this boundary, FACE-CHANGE is able to recover the missing code and backtrace its attack/exception provenance to analyze the anomalous behavior.**

***Keywords*-Attack Surface Minimization; Attack Provenance; Virtualization;**

I. INTRODUCTION

Modern operating systems strive to shrink the size of the trusted computing base (TCB) to ease code verification and minimize trust assumptions. For a general-purpose operating system (OS) like Linux, kernel minimization has already been established as a practical approach to reducing attack surface. But existing approaches [1]–[4] have a number of problems:

Coarse-Grained Profiling: In order to eliminate unnecessary code from the kernel, one must identify the kernel code that is required to support the multiple applications within a system. The conventional approach is to generate typical workloads and measure all active kernel code in a training session. Profiling is performed on the whole system and does not distinguish between the requirements of different applications [1]. This approach is well suited for generating a customized kernel for a static, special-purpose system (e.g., an appliance or embedded system). But for a general-purpose operating system supporting a variety of applications, whole-system profiling unnecessarily enlarges the kernel attack surface of the system.

In practice, we observe that kernel code executed under different application contexts varies drastically. Our experiments show that two distinct applications may share as little as 33.6% of their executed kernel code – thus system-wide kernel minimization would over-approximate both applications' kernel requirements. For example, the kernel functionality needed by task manager *top* is to read statistics data from the memory-based *proc* file system and write to the tty device. In sharp contrast, the *Apache* web server primarily requires network I/O services from the kernel. If we profile a system running *top* and *Apache* simultaneously, we will expose the kernel's networking code to *top* simply because *Apache* is in the same environment. Further, assume *top* is the target of a malicious attack, the compromised *top* may be implanted with a parasite network server as a backdoor without violating the minimized kernel's constraint.

Flexibility to Adapt to Runtime Changes: The output of traditional kernel minimization approaches is a static kernel image customized for a specific workload. However, it is nearly impossible to cover all execution paths within an application's code to trigger every possible kernel request. Even when leveraging automatic test case generation techniques [5]–[7], profiling may still suffer from the path coverage problem for large programs. Insufficient profiling may lead to an underestimation of the kernel code required to support some application(s) at runtime. Further, the required kernel code may change when running a new application that was not profiled before or when the workload of an existing application suddenly changes. If this newly requested kernel code is not included in the customized image, the violation may crash the application or even panic the kernel.

To address these problems of whole-system-based kernel minimization, we have developed FACE-CHANGE, a virtualization-based system to support *dynamic switching among multiple minimized kernels, each for an individual application*. Throughout this paper, we use the term *kernel view* to refer to the in-memory kernel code presented to an individual application. In conventional kernels, all concurrently running user-level processes share the same kernel view containing the entire kernel code section, which we refer to as a *full kernel view*. FACE-CHANGE aims to present each process with a different, customized kernel view, which is prepared individually in advance by profiling the application's needs. Any unnecessary kernel code is eliminated to minimize the attack surface accessible to this specific application. At runtime, FACE-CHANGE identifies

978-1-4799-2233-8/14 $31.00 © 2014 IEEE
DOI 10.1109/DSN.2014.52

the current process context and dynamically switches to its customized kernel view.

To support applications that were not previously profiled, we are able to profile them in independent (off-line) sessions to generate their kernel views. We then load the kernel view for a new application dynamically without interrupting the system's execution. This removes the burden of re-compiling and/or installing a new customized kernel upon the addition of a new application.

Furthermore, we include a kernel code recovery mechanism for the event that an application tries to reach code outside of the boundary of its kernel view. This may be due to incomplete profiling (e.g., interrupt handler's code with no attachment to any process or some workload not completely exercised) or malicious tampering (e.g., some injected logic requests new/different kernel features). We are able to recover the missing code and backtrack its provenance to identify the anomalous execution paths. Such capability can be leveraged by administrators to analyze the attack patterns of both user-level and kernel-level malware.

This paper makes the following contributions:

- A quantitative study of per-application kernel requirements in a multi-programming system.
- A virtualization-based dynamic kernel view switching technique. FACE-CHANGE is transparent to the guest virtual machine (VM) and requires no patching or recompilation of the guest OS kernel.
- A kernel code recovery mechanism to recover requested but missing code and backtrack the provenance of such an anomaly/exception.

The rest of this paper is organized as follows. Section II presents the motivation, goals and assumptions of FACE-CHANGE. Section III provides the detailed design of FACE-CHANGE. Section IV gives case studies on the effectiveness of FACE-CHANGE on user/kernel malware attacks and evaluates its performance. Section V discusses limitations and future work. Section VI describes related work and we conclude in Section VII.

II. SYSTEM OVERVIEW

In this section, we introduce a quantitative method to measure the kernel code requirements of a specific application. We then use these measurements to evaluate the similarity of kernel code requirements between applications. The result of this quantitative study motivates the development of FACE-CHANGE. Finally, we present the goals and assumptions of our design.

A. Motivation

Each application, including both the base program and any libraries loaded into the user address space, interacts with the OS through system calls to request services (e.g., manipulating files, spawning threads, IPC, etc.). The set of system calls utilized by an application varies substantially across different application types and workloads, and intuitively, different system calls will reach different parts of the kernel's code. Further, different values passed as parameters to the same system calls may lead to totally different execution paths within the kernel. For example, because of Linux's *virtual file system* (vfs) interface, a *read* system call for disk-based files in *ext4-fs* and memory-based files in *procfs* will be dispatched to entirely different portions of the kernel's code.

To accurately measure a target application's kernel code requirements, we monitor the system execution at the basic block level. We briefly describe the profiling tool here and will present the detailed design in Section III-A. We record any executed basic blocks which satisfy the following two criteria:

1) The basic block belongs to the kernel, i.e., its memory address is in kernel space.
2) The basic block is executed in the target application's context.

After merging any adjacent blocks, we get a range list $\mathrm{K}_{[\mathrm{app}]}$ for a target application (denoted by subscript $[\mathrm{app}]$) in the form:

$$\mathrm{K}_{[\mathrm{app}]} = \{([\mathrm{B}_1, \mathrm{E}_1], \mathrm{T}_1), \cdots, ([\mathrm{B}_\mathrm{i}, \mathrm{E}_\mathrm{i}], \mathrm{T}_\mathrm{i})\}$$

B_i and E_i denote the beginning and end addresses for the i-th in-memory code segment. T_i indicates the type for this memory segment, where T_i can be either "base kernel" or the name of a kernel module. For kernel modules, we record addresses relative to the module's base address because a module's loading addresses may change at runtime.

We introduce three definitions for comparing two distinct application's kernel code requirements:

1) $\mathrm{K}_{[\mathrm{app1}]} \cap \mathrm{K}_{[\mathrm{app2}]}$
   The intersection of two range lists outputs the overlapping address ranges between them. The result is still a range list.
2) $\mathrm{LEN}(\mathrm{K}_{[\mathrm{app}]})$
   The LEN of a range list outputs the number of elements in this list.
3) $\mathrm{SIZE}(\mathrm{K}_{[\mathrm{app}]}) = \sum_{\mathrm{i} \in [1, \mathrm{LEN}(\mathrm{K}_{[\mathrm{app}]})]} (\mathrm{E}_\mathrm{i} - \mathrm{B}_\mathrm{i})$
   The SIZE of a range list outputs the size of kernel code in this range list.

We use Equation (1) below to define the similarity index *S* between $\mathrm{K}_{[\mathrm{app1}]}$ and $\mathrm{K}_{[\mathrm{app2}]}$:

$$\mathrm{S} = \frac{\mathrm{SIZE}(\mathrm{K}_{[\mathrm{app1}]} \cap \mathrm{K}_{[\mathrm{app2}]})}{\mathrm{MAX}(\mathrm{SIZE}(\mathrm{K}_{[\mathrm{app1}]}), \mathrm{SIZE}(\mathrm{K}_{[\mathrm{app2}]}))} \quad (1)$$

A similarity index *S* indicates the proportion of the overlapping of kernel code required between two applications. Besides common system call execution paths, the overlapping kernel code also consists of functionality needed by every application, e.g., process scheduler and interrupt handling code. Through the profiling of well-known Linux applications, we find that similarity indices range from

33.6% for applications that are orthogonal in type (such as *top* vs. *Firefox*) to 86.5% for similar applications (such as *Apache* vs. *vsftpd*). Table I (Section IV) shows the similarity indices for all profiled applications. These measurements support our earlier hypothesis that kernel code execution paths vary substantially across different application types. This also indicates that application-specific kernel views can minimize the kernel attack surface far beyond that of system-wide kernel minimization.

B. Goals and Assumptions

We state the goals for our system in four aspects: strictness, robustness, transparency and flexibility.

Strictness: The kernel view generated for a specific application should only contain the kernel code that is necessary for the correct execution of this application under a normal usage scenario. We should eliminate all other excessive code from the kernel view to avoid enlarging the kernel's attack surface. If an application reaches kernel code that does not belong to its kernel view, we should record the access in detail for later analysis.

Robustness: If an application is running under the same workload and same usage scenario as during profiling, the behavior of this application running with a customized kernel view should be no different than with a full kernel view. If the application accesses any kernel code that is not included in the customized kernel view, we should recover the missing code and record this violation silently without being detected by the application.

Transparency: There is no need to change any code in the applications or operating system. The hypervisor controls all FACE-CHANGE operations, which remain transparent to the guest VM.

Flexibility: Administrators can dynamically load, unload, and switch the kernel view for a specific application at any time. This should neither jeopardize the functionality of the currently running application nor the system as a whole.

We assume that, when we generate customized kernel views in the profiling phase, the environment, including both the applications and the kernel, should not be tampered with by malware.

III. DESIGN AND IMPLEMENTATION

In this section, we give a detailed description of the overall design of FACE-CHANGE, highlight the challenges we face and the solutions we propose. Then we discuss the detailed implementation of our prototype system.

We divide the whole system into two phases in chronological order: the *profiling phase* and the *runtime phase*. The profiling phase monitors a target program's execution and, based on the active kernel code in this process' context, generates a configuration file describing the application's customized kernel view. In the runtime phase, FACE-CHANGE builds a new customized kernel view based on each application's configuration file and forces the process to use this customized kernel view whenever the guest OS schedules it.

Figure 1 shows a high-level example of these two phases. Assume we want to profile *Process 1* in the profiling phase. When the kernel schedules *Process 1* to run, we start to record all the kernel code executed in its context. When *Process 1* is scheduled out, we pause the recording until the process is re-scheduled. This procedure also applies to *Processes 2* and *3*. At last we generate three configuration files for the kernel views of these three processes respectively. In the runtime phase, we load each customized kernel view for the corresponding process. For example, *Process 1* can only access *[Process 1] kernel view* when it is running.

A. Profiling Phase

1) Design of the Profiler: We implemented our profiler as a component of the *QEMU* [8] 1.6.0 full system emulator. This enables the profiler to track an application's execution at the granularity of a basic block, and we use virtual machine introspection (VMI) techniques to track context switches within the guest OS. When the guest OS schedules the target application, the profiler records any address ranges of kernel code executed in this process' context. For code within a kernel module, we record addresses relative to the module's base address. Once the application has been sufficiently profiled, the profiler exports all recorded kernel code segments to a kernel view configuration file.

2) Test Suite Selection: For each application to be profiled, the user should choose a test suite to simulate the expected real-world workload for this application. For instance, when profiling a server application, the user may deploy it in the real environment to handle requests, or for an interactive application, one may simulate the I/O operations of a typical user. To give a specific example, when profiling a *mysql* server, we set up a RUBiS[1] [9] server and used its own simulated client to generate workloads for the *mysql* database.

It is difficult to ensure that all code paths through an application are executed during profiling, and thus it is possible that at runtime the application may access some kernel code missed by the profiling phase. One alternative to a test suite driven profiler is to use symbolic execution to generate high-coverage test cases, but this approach may not scale to large applications. To address this problem, we employ a *kernel code recovery mechanism* in the runtime phase to recover any missing kernel code. We explain this mechanism in detail in Section III-B3.

3) Interrupt Context: In modern OS kernels, hardware triggered asynchronous interrupts can happen at any time, and thus interrupt handler code is not attached to any single process' context. We choose to include the interrupt

[1] RUBiS is an ebay-like auction service that heavily uses *mysql*.

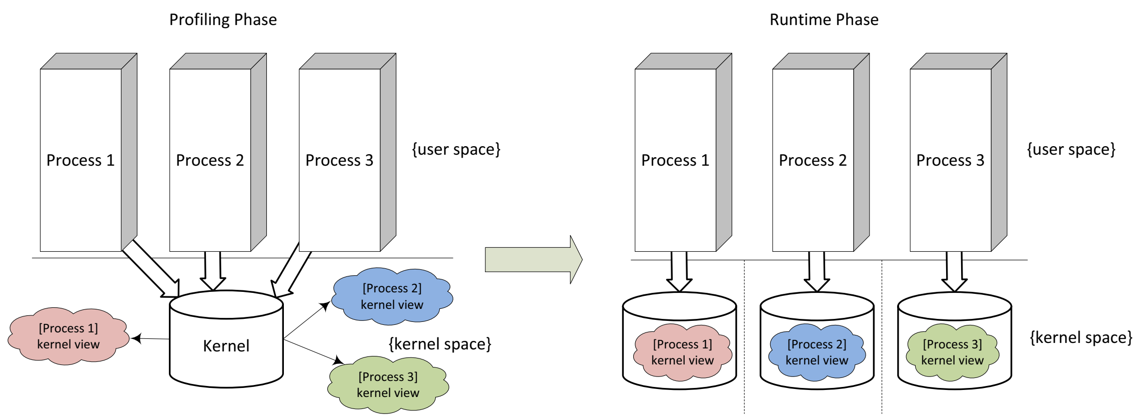

Figure 1: Overview of FACE-CHANGE

handler's code in every application's kernel view to avoid having to repeatedly recover this code at runtime. Our profiler leverages *QEMU* to identify the occurrence of an interrupt. If this interrupt is not a software interrupt (such as system call), we can infer that the system has entered interrupt context. At this point, we record all kernel code addresses accessed in the interrupt's context for use in all applications' customized kernel view.

B. Runtime Phase

We describe the general design of the runtime phase in Algorithm 1 and discuss some interesting features below in detail.

1) Kernel View Initialization: When loading a new kernel view configuration, FACE-CHANGE allocates memory pages for both the base kernel code and any kernel modules' code and fills them with undefined instruction (UD2) "*0xf 0xb*" (UD2 will raise an invalid opcode exception when executed). FACE-CHANGE then loads the kernel code specified in the kernel view configuration into it's appropriate locations in the new pages. Recall that during profiling, we track the kernel control flow at the basic block level. However, rather than loading individual basic blocks, we slightly relax the condition to load the entire kernel function which contains the valid basic blocks. The rationales for this relaxation are: (1) The adjacent code within the same kernel function is more likely to be accessed at runtime. Thus, we can reduce the frequency of kernel code recovery by loading the whole kernel function. (2) UD2 is a 2-byte instruction. If an address range in the kernel view configuration starts from an odd-numbered address, only the first byte of UD2 will be in the kernel view, and the processor may misinterpret the fragmented UD2 as a different instruction. Loading entire kernel functions avoids this problem because the boundaries of kernel functions are aligned on powers-of-two[2].

To identify function boundaries, we search for a function header signature backwards and forwards from the

Algorithm 1 Kernel View Switching/Kernel Code Recovery

```
Input:  modulelist ← kernel module list
        context_switch_addr ← Address of context_switch function
        resume_userspace_addr ← Address of resume_userspace function
        full_kernel_view_index ← Index of full kernel view
- - - - - - - - Kernel Code Recovery - - - - - - - - - -
procedure BACK_TRACE(rip, rbp)
    iter_rbp := rbp
    prev_rip := rip
    while IS_VALID(prev_rip) do
        DUMP_BACKTRACE(prev_rip)
        prev_rip := READ_PREV_RIP(iter_rbp)
        prev_rbp := READ_PREV_RBP(iter_rbp)
        if PREV_RIP = '0B 0F' then
            RECOVER_BACKTRACE(prev_rip)
        iter_rbp := prev_rbp
procedure HANDLE_INVALID_OPCODE(vcpu)
    BACK_TRACE(vcpu.rip, vcpu.rbp)
    mem_page := GET_MEMORY_PAGE(vcpu.rip)
    start_addr := SEARCH_BACKWARDS(vcpu.rip)
    end_addr := SEARCH_FORWARDS(vcpu.rip)
    FETCH_FILL_CODE(page, start_addr, end_addr)

- - - - - - - - Kernel View Switching - - - - - - - - - -
procedure SWITCH_BASE_KERNEL(index)
    kernel_range := GET_KERNEL_RANGE()
    LOAD_KERNEL_VIEW_EPT(kernel_range, index)
procedure SWITCH_KERNEL_MODULES(index)
    for all mod in modulelist do
        module_range := GET_MODULE_RANGE(mod)
        LOAD_MODULE_VIEW_EPT(module_range, index)
procedure SWITCH_KERNEL_VIEW(index)
    SWITCH_BASE_KERNEL(index)
    SWITCH_KERNEL_MODULES(index)
procedure HANDLE_KERNEL_VIEW_TRAP(vcpu)
    if vcpu.rip = context_switch_addr then
        procinfo := READ_PROC_INFO(vcpu)
        index := KERNEL_VIEW_SELECTOR(procinfo)
        if index = full_kernel_view_index then
            CLEAR_RESUME_USERSPACE_TRAP()
            SWITCH_KERNEL_VIEW(index)
        else
            ENABLE_RESUME_SPACE_TRAP()
            lastindex := index
    else if vcpu.rip = resume_userspace_addr then
        CLEAR_RESUME_USERSPACE_TRAP()
        SWITCH_KERNEL_VIEW(lastindex)
```

[2]Linux kernel is by default compiled with -O2 that contains optimization flag -falign-functions

basic blocks marked in the kernel view configuration. For example, a common function header signature in the x86 Linux kernel is "*push ebp; mov ebp, esp*"(binary opcodes "*0x55 0x89 0xe5*"). There is a possibility that one kernel function may cross two memory pages and further, one single instruction may split across pages. In this case, we continue searching from the head of the next page or the tail of the previous page to locate the complete kernel function.

After all of the kernel view's code is identified and loaded into the new pages, FACE-CHANGE redirects any kernel code access made by this application to the customized kernel view. We implement our FACE-CHANGE runtime component within a *KVM* hypervisor (i.e., kvm-kmod-3.6 and qemu-kvm-1.2.0) and leverage Extended Page Tables (EPT) to manipulate kernel code mappings. When using EPT, the guest VM maintains its own page table to translate guest virtual addresses to guest physical addresses. The hypervisor then uses EPT to transparently map the guest physical addresses to host physical addresses. During guest OS context switches, FACE-CHANGE changes the page table entries in the EPT to direct any kernel code accesses to the customized kernel view for the application (instead of the original kernel's code). This procedure is explained in the Section III-B2.

Again, FACE-CHANGE must take care when handling kernel modules' code in a customized kernel view. Recall that kernel modules are dynamically loaded at runtime in the kernel's heap, and thus, during the profiling phase, we record these addresses relative to the module's base address. Before we load modules' code into a kernel view, we traverse the kernel's module list to identify the loading addresses for any modules marked in the kernel view configuration. Then we load the valid kernel code in the code pages for the kernel modules.

2) Kernel View Switching: Figure 2 illustrates each step of the kernel view switching procedure. In step 1, the guest OS chooses a process to run and prepares to context switch to the new process. In step 2, using VMI, we intercept this context switch and determine which customized kernel view is needed for the new application. In step 3A and 3B, we modify the pointers to the page directory (level 2 in the EPT) corresponding to the base kernel code and all kernel modules' code respectively. Because kernel modules' code pages are scattered in the kernel heap, we reuse any entries in the page directory that point to kernel data and only modify the entries pointing to the modules' code.

We also develop a set of optimizations to improve performance. Through our experimentation, we find that switching kernel views immediately at context switches may cause the application to miss interrupts, and thus jeopardize I/O performance. We choose instead to switch kernel views when the code resumes user space execution after the context switch. This will still satisfy the strictness goal (minimize the attack surface) and mitigate the performance degradation caused by missed interrupts. We also check whether the previous process and the next process use the same kernel view, and if so, we can avoid one additional kernel view switch.

3) Kernel Code Recovery: There are two situations where FACE-CHANGE may need to recover missing kernel code:

(i) *An incomplete kernel view generated during profiling:* Testing in a controlled runtime environment without introducing any attacks, we find that the majority of the benign kernel recoveries are triggered due to missing code for handling interrupts. For example, *KVM* provides a para-virtualized clock device to the guest VM. This *KVM* specific code cannot be included in the kernel view during the profiling in *QEMU*. Thus, at runtime, FACE-CHANGE needs to recover the missing kernel functions shown below in chronological order:
$kvm\_clock\_get\_cycles \rightarrow kvm\_clock\_read$
$\rightarrow pvclock\_clocksource\_read \rightarrow native\_read\_tsc$
In addition, interrupt handling code is not bound to any process and can be triggered by hardware interrupts at any time. In the profiling phase, we may not observe all possible interrupts for this application. Before missing code recovery, we inspect the current call stack to determine whether the current execution is in interrupt context (through backtracking the current function traces). Thereafter we recover the missing kernel code to correctly handle those interrupts.
All other benign kernel code recoveries due to incomplete profiling of the application's execution paths are recorded as a reference for the administrator to ameliorate the profiling test suite.

(ii) *Anomalous execution caused by malicious attacks:* User level malware may hijack a normal process to execute shellcode which requests kernel services that are not in the customized kernel view. Additionally, kernel level rootkits can detour the kernel's execution path to their payload's malicious code, and obviously, this malicious payload will not be in any application's kernel view. FACE-CHANGE is designed to report the suspicious execution traces, but still recover the kernel code in this case. In order to track the provenance of the attack, we not only record any recovered functions, but also *backtrack the anomalous execution's call stack* to find the origin of the invocation chain for later analysis.

As we mentioned in Section III-B1, we fill any kernel code space that is not in the kernel view with UD2 "*0xf 0xb*". When executed, UD2 raises an invalid opcode exception which causes a trap to the hypervisor. We illustrate this as step 4 *invalid opcode trap* in Figure 2. After intercepting the trap, we check the faulting address and try to fetch the missing kernel function from the original kernel code pages (step 5 in Figure 2).

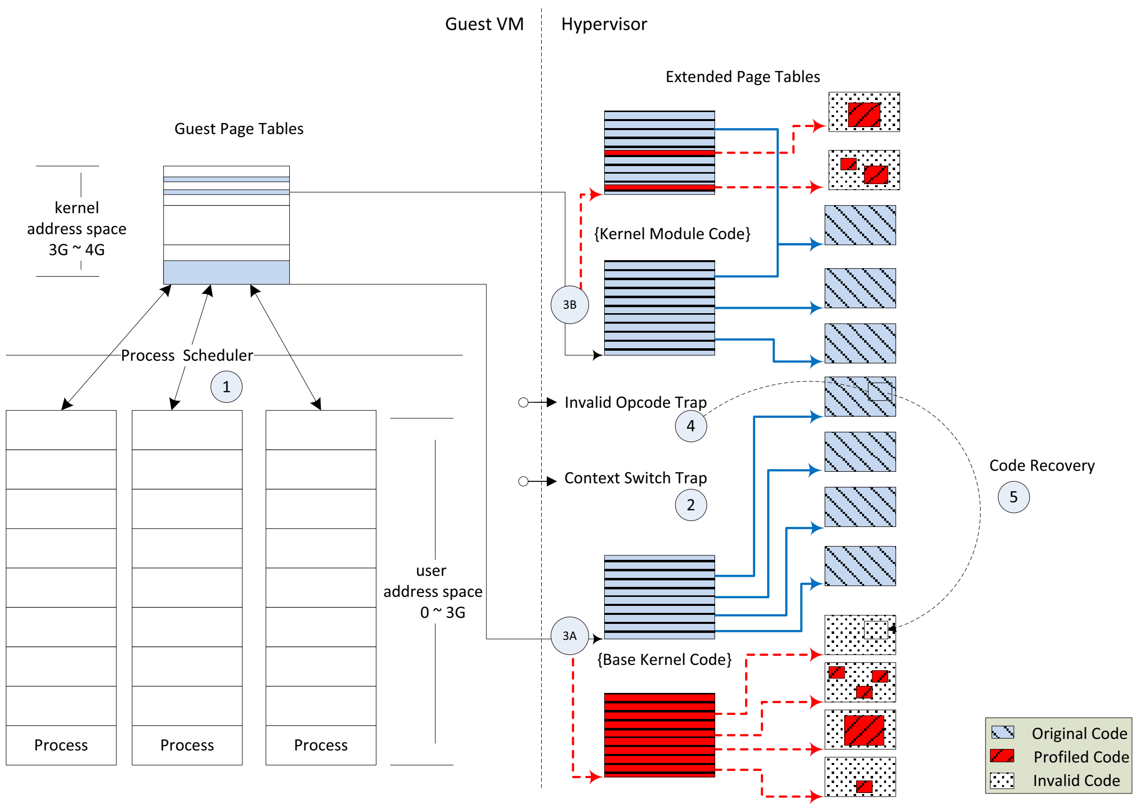

Figure 2: The Procedure of Dynamic Kernel View Switching and Kernel Code Recovery

During our implementation of the kernel code recovery mechanism, we fixed an interesting cross-view bug in FACE-CHANGE that is worth mentioning here. If no customized kernel view is enabled for a specific process, it will have a full kernel view. When executing this process, its kernel execution may be interrupted or the process may voluntarily give up the CPU. If we enable a customized kernel view for that process at this time and the process is re-scheduled by the kernel, some functions in the process' execution stack may not be in the new kernel view. We give an example of this situation in Figure 3.

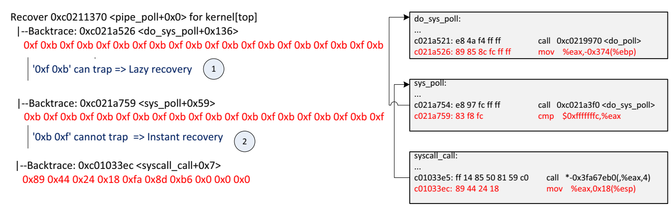

Figure 3: Cross-View Kernel Code Recovery

In this case, the process was re-scheduled while executing *pipe_poll* at address 0xc0211370. The invocation chain in the stack is as follows:

$syscall\_call \rightarrow sys\_poll \rightarrow do\_sys\_poll$

We find that because *sys_poll* and *do_sys_poll* are not in the new customized kernel view, their code regions are filled with UD2 (shown in red). If we recover *pipe_poll* and return to its caller (*do_sys_poll*), the process will execute undefined instructions. For *do_sys_poll* this is not a problem because the return address (0xc021a526) is an even number. Execution will return to the "*0xf 0xb*" opcode (UD2), causing an invalid opcode trap, and we can recover *do_sys_poll* as normal. We call this as a *lazy recovery*. But for *sys_poll*, the return address (0xc021a759) is an odd number, and the opcode starting at address 0xc021a759 is "*0xb 0xf*." This opcode will be misinterpreted by the processor and not cause a trap. Our solution is, during code recovery, to backtrack the stack and recover any caller whose return-target opcode starts with "*0xb 0xf*" in the new kernel view. We call this *instant recovery*. In this example, when we recover the code for *pipe_poll*, we recover *sys_poll* instantly.

4) Disable Customized Kernel View: We can load/unload customized kernel views dynamically at runtime to satisfy our flexibility goal. When we disable a kernel view, FACE-CHANGE de-allocates all memory pages for that kernel view and switches the EPT back to a full kernel view without interrupting the running application. This enables us to adapt to an altered environment smoothly by "hot-plugging" kernel views.

IV. EVALUATION

In this section, we present the evaluation of our system in two aspects: security and performance. For the security evaluation, we first use the similarity index to measure the similarities of kernel views among applications. Then we demonstrate the effectiveness of our system to track attack provenance of both user-level malware and kernel-level rootkits. For the performance evaluation, we measure the overall system performance with FACE-CHANGE enabled and the I/O performance for an *Apache* web server with a minimized kernel view. The hardware configuration of our testing platform is a Lenovo Ideapad U410 with Intel® Core™ i7 3.10GHz and 8GB memory. We run FACE-CHANGE on Linux Mint 13 x86_64 (Linux kernel version 3.5.0). We test our prototype with a guest VM using Ubuntu 10.04 (Linux kernel version 2.6.32) i386 LTS release[3], further since FACE-CHANGE requires minimal domain knowledge, it will be convenient to extend our current system to support more Linux kernel versions with only minor changes to the implementation. The guest VM's memory is 2GB and it uses bridged networking.

A. Security Evaluation

1) Kernel View Variation among Applications: We use the similarity index defined in Section II to measure the difference of kernel views among 12 well-known Linux applications from different categories. For example, *Apache* and *vsftpd* are server applications that handle network requests. *Firefox* and *gvim* are interactive applications that respond to user input. We present the profiling results as a square matrix in Table I. The main diagonal(↘) of the matrix is marked with gray cells. Each cell on the main diagonal presents the size of the kernel view for this specific application (e.g., *vsftpd* executes 341KB kernel code in the profiling phase). We compare the kernel code address ranges between every two applications to get the overlapping size. All entries above the main diagonal represent the overlapping size between two applications' kernel views (e.g., *tcpdump* and *Firefox* have 218KB overlapping kernel code). Entries below the main diagonal represent the similarity index calculated using Equation (1) in Section II. The similarity index demonstrates the similarity of kernel attack surface between different applications. For applications of different types, lower percentages are better as this ensures a distinct minimized kernel in both cases, and for similar applications high percentages are expected since both require similar kernel services. As Table I shows, the similarity indices range from 33.6% for dissimilar applications to 86.5% for applications with common kernel requirements. This proves our intuition that if two applications are from different categories they have relatively low similarity index and leverage different parts of the kernel.

2) Attack Detection and Provenance: Because the kernel attack surface for each individual application is reduced

[3]We use Ubuntu 10.04 because the kernel rootkit samples we use in the evaluation do not support newer Linux kernel yet.

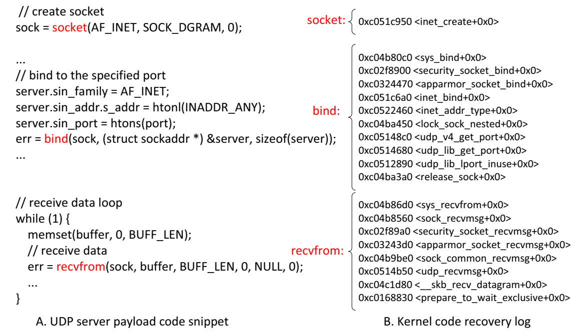

Figure 4: Attack Pattern of Injectso's Payload

according to the profiling results, we can reveal malicious attack patterns whenever a process goes beyond the boundary of its kernel view. Further, we backtrack the requested kernel code to identify the exact attack provenance.

This result is a step further than traditional system-wide kernel minimization techniques [1]–[4] because FACE-CHANGE is able to detect anomalous execution based on an individual application's kernel view. To demonstrate that FACE-CHANGE can reveal attack evidences that may go unnoticed under traditional system-wide minimization techniques, we also create a "union" kernel view (the union of all kernel views from the applications we have profiled) as the system-wide minimized kernel. System-wide minimization may fail to detect an attack if the attack utilizes kernel code required by *any application in the system*. FACE-CHANGE greatly reduces this "blind spot" because it is able to detect kernel execution anomalies *specific to a single application*.

In this paper, we evaluate the effectiveness of attack detection with 13 user-level malware (8 of them use online runtime infection and 5 use offline binary infection) and 3 kernel-level rootkits. This data is presented in Table II. We highlight four of these attack case studies in detail.

Case Study I – Injectso: Injectso [10] is a well-known hot-patching tool used to modify the behavior of a running process by injecting a dynamic shared object into its address space. It detours the current instruction pointer to *__libc_dlopen_mode* and builds a fake stack to invoke the shared object's code. The shellcode's payload is a UDP server, and the target program is *top*. Obviously, the kernel view for *top* does not contain any kernel code needed to run a UDP server (even if the kernel views of other co-existing applications do), and thus Injectso's payload triggered the kernel code recovery mechanism.

From the kernel code recovery log, we can precisely identify the anomalous execution caused by Injectso in the *top* process. In Figure 4, we present the UDP server payload's code and the corresponding kernel code recovery log. The UDP server will create a socket, bind to an address/port, and receive data using the C library calls *socket*, *bind* and *recvfrom* respectively. It is straightforward to identify which library functions correspond to the recovered kernel

	firefox	totem	gvim	apache	vsftpd	top	tcpdump	mysqld	bash	sshd	gzip	eog
firefox	443KB	275KB	251KB	302KB	284KB	149KB	218KB	305KB	221KB	316KB	213KB	286KB
totem	62.1%	286KB	239KB	210KB	217KB	140KB	166KB	228KB	196KB	220KB	174KB	257KB
gvim	56.7%	83.6%	262KB	206KB	206KB	142KB	160KB	220KB	190KB	211KB	166KB	247KB
apache	68.2%	62.7%	61.5%	335KB	284KB	141KB	210KB	265KB	203KB	292KB	200KB	215KB
vsftpd	67.9%	63.6%	60.5%	83.5%	341KB	145KB	208KB	272KB	205KB	293KB	206KB	222KB
top	33.6%	49.2%	54.2%	42.2%	42.7%	167KB	135KB	138KB	147KB	153KB	121KB	143KB
tcpdump	49.2%	58.0%	61.1%	62.6%	61.0%	57.6%	234KB	203KB	165KB	216KB	169KB	168KB
mysqld	68.7%	68.1%	65.4%	78.9%	79.8%	41.1%	60.5%	336KB	186KB	260KB	212KB	230KB
bash	50.0%	68.7%	72.6%	60.6%	60.1%	60.8%	68.3%	55.5%	242KB	223KB	158KB	215KB
sshd	71.3%	58.4%	55.9%	77.5%	77.7%	40.5%	57.3%	68.9%	59.0%	378KB	216KB	233KB
gzip	48.1%	60.9%	63.4%	59.6%	60.4%	49.5%	69.0%	63.2%	64.6%	57.1%	245KB	177KB
eog	64.6%	86.5%	83.2%	64.2%	65.2%	48.1%	56.5%	68.7%	72.4%	61.7%	59.7%	297KB

Table I: Similarity Matrix for Applications' Kernel Views

Name	Infection Method	Payload	Note
Injectso	Online infection: Shared object injection	UDP server	Case study I
Cymothoa v1	Online infection: Fork process	Bind /bin/sh to TCP port and fork shell	Recover *sys_fork* and TCP server
Cymothoa v2	Online infection: Clone thread	Bind /bin/sh to TCP port and fork shell	Recover *sys_clone* and TCP server
Cymothoa v3	Online infection: Settimer parasite	Remote file sniffer	Recover *sys_settimer* and signal handler
Cymothoa v4	Online infection: Signal/Alarm parasite	Single process backdoor	Case study II
Hotpatch	Online infection: Library injection	File writing of injecting timestamp	Recover injection and file writing procedure
Xlibtrace	Online infection: $LD_PRELOAD linker	Tracking function invocation	Recover tty procedures on terminal
Hijacker	Online infection: Global offset table poisoning	Redirection of library function	Recover the procedure of hijacking
Infelf v1	Offline binary infection	Remote shell server	Recover remote shell socket operations
Infelf v2	Offline binary infection	Register dumping	Case study III
Arches	Offline binary infection	Register dumping	Recover register dumping operations on terminal
Elf-infector	Offline binary infection	Register dumping	Same as above
ERESI	Offline binary infection	UDP server	Recover creation of udp server
KBeast	Kernel rootkit	File/Process hiding, keystroke sniffer	Case study IV
Sebek	Kernel rootkit	Confidential data collection	Recover kernel code in sebek module
Adore-ng	Kernel rootkit	File/Process hiding	Recover kernel code in adore-ng module

Table II: Results of Security Evaluation Against a Spectrum of User/Kernel Malware

code sections (e.g., *bind* executes a kernel code path from *sys_bind* to *release_sock* [4] in chronological order).

We test the system again and apply the "union" kernel view, which includes both *top* and some network applications (such as *Firefox* and *Apache*) – to represent a system-wide minimization technique. These network applications require the same kernel networking code as the UDP server payload, and thus this case results in no UDP related kernel functions being recovered. Due to the enlarged attack surface of the system-wide minimized kernel, this attack would achieve its goal with the available kernel code and thus go undetected.

Case Study II – Cymothoa: Cymothoa [11] is a shellcode injection framework that uses different infection methods and payload types. The parasite executable coexists with the host process stealthily while the host process continues to work properly. We test all four working parasites introduced in the article "Single Process Parasite" [12] in Phrack issue 68 and successfully reveal all four attack behaviors. The parasite uses the *sys_fork* and *sys_clone* system calls to create a child process/thread to execute its payload. Later variants are more stealthy, utilizing *settimer* and *signal* to schedule the shellcode inside the host process. Here, we give a detailed description of the most stealthy (variant 4) parasite's control flow. This variant creates a backdoor parasite living within another process (*bash* is the target program in this case). First the shellcode registers a signal handler for the *SIGALRM* signal. Then it opens a nonblocking I/O socket, binds it to a specific port, and sets the *SIGALRM* timer. When the *SIGALRM* signal is handled, the parasite accepts any connection on the socket and launches a remote shell. The parent then sets the timer again and resumes execution of the host process.

Again, the kernel code executed by the shellcode's actions, e.g., setting the signal handler, creating the TCP server, and setting the alarm clock are recorded in the kernel recovery log. This reveals both the infection method and payload behaviors of the stealthy parasite. Also, like before, existing kernel minimization techniques may fail to detect this attack entirely because other applications will likely add these kernel regions into the union-based minimized kernel.

Case Study III – Infelf: In addition to runtime infection malware, we also apply our techniques to detect compromised applications. Infelf [13] is an offline binary infection tool that is able to implant trojan code into an existing binary program. It splits trojan code into multiple instruction blocks, inserts them into free alignment areas between functions, and concatenates their execution path with jump instructions. We use this tool to implant a hardware register printing function into the *gvim* binary and redirect *gvim's* entry function to this shellcode. During *gvim's* startup, FACE-CHANGE recovers numerous TTY kernel functions

[4] Symbols of kernel functions are not necessary for backtracking. We use them here for clear demonstration.

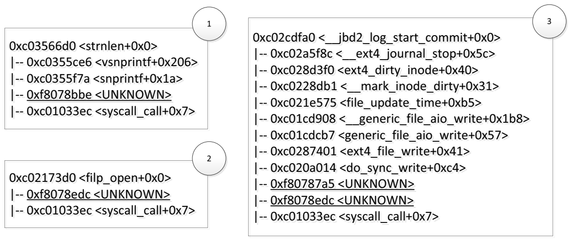

Figure 5: Attack Pattern of KBeast Rootkit

which are not included in *gvim's* kernel view. Again, in this case, a whole-system kernel minimization technique would be unable to detect this attack on a system containing both *gvim* and terminal applications that require the kernel's TTY functions (such as *tcpdump* or *bash*).

Case Study IV – KBeast Rootkit: In addition to user-level attacks, our system is also able to detect rootkit attacks at the kernel level. Because rootkit attacks originate from shellcode in kernel space, the interpretation of kernel recovery logs is different from user-level attacks. Kernel-level attacks aim to hide their malicious behavior by detouring the kernel's control flow during execution of certain kernel routines (e.g., listing kernel modules, network connections, etc.). Again, we assume that no rootkit is present during the initial profiling phase, and so no rootkit code can be included in the kernel view configuration files. When FACE-CHANGE allocates a new kernel view, if a rootkit has already been installed in the runtime system's kernel, the rootkit's code will not be loaded into the new view and will be filled with UD2 by default. If the application later triggers FACE-CHANGE'S code recovery, the log will allow us to clearly see where the hijack took place. A more complicated scenario that FACE-CHANGE can detect is a rootkit which is installed while FACE-CHANGE is enforcing an application's kernel view. In this scenario, the rootkit will be detected in the same way as user-level malware: by the kernel functionality that it requests to perform its malicious functionalities. Again, this code will be recovered and we can backtrace recovered kernel code to reveal the anomalous execution.

We use the KBeast [14] rootkit as an example to show this process in detail. KBeast is a new rootkit that inherits many features from traditional Linux kernel rootkits (e.g., file/process/socket/module hiding, keystroke sniffer) and it supports recent kernel versions. We use the kernel view for the *bash* program to detect the existence of KBeast. All the keystrokes typed in bash are processed by the keyboard event handler. KBeast is able to intercept and read the keystrokes and store this data into a hidden file, and it will hide its existence by removing itself from the kernel module list. In Figure 5, by backtracking the recovered kernel functions, we find code addresses with an *UNKNOWN* tag. This indicates that these memory addresses are not in any identified memory regions. We also find that KBeast's code hijacks the entries of some system calls and invokes *strnlen* to check the length of the keystroke buffer, *filp_open* to open the hidden file, and *do_sync_write* to write the keystroke data into this file.

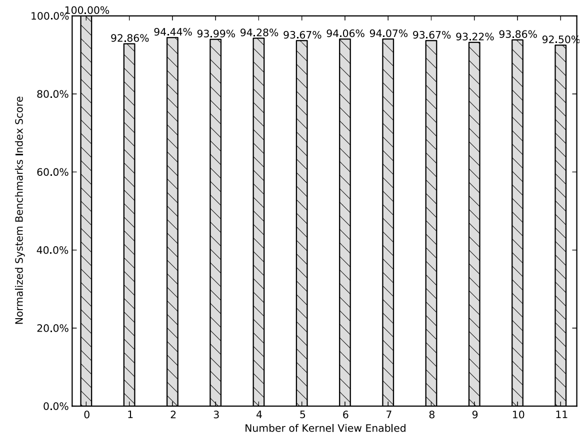

Figure 6: Normalized System Performance Results from UnixBench

B. Performance Evaluation

1) System Performance: We use the UnixBench benchmark suite to measure and evaluate system performance. Specifically, we take three different measurements:

(i) We run UnixBench without enabling FACE-CHANGE to get a baseline result.
(ii) We enable FACE-CHANGE, load one kernel view (*Apache*), and run the benchmark. This tests whole-system performance overhead after enabling our system.
(iii) Next, we launch the applications[5] from Table I and load their kernel views one at a time. After each kernel view is loaded, we rerun the benchmark. This measures any performance influence on the whole system after loading multiple kernel views.

In Figure 6, we normalize the performance scores of the UnixBench (higher performance score indicates better performance) based on the baseline score from step 1. The X axis represents the number of kernel views we enabled simultaneously. We find that, compared to the baseline result, enabling our system incurs 5%∼7% performance overhead on the whole system. Adding multiple kernel views incurs trivial impact on the system performance. We find that the only performance degradation occurs during the subtest *Pipe-based Context Switching* of UnixBench. This is not surprising because FACE-CHANGE triggers additional traps for each context switch. We could largely minimize the performance overhead with optimization of the context switch handler's code.

[5] We exclude *gzip* here because it is not a long running application (i.e. it is difficult to ensure it executes during the entire benchmarking measurement).

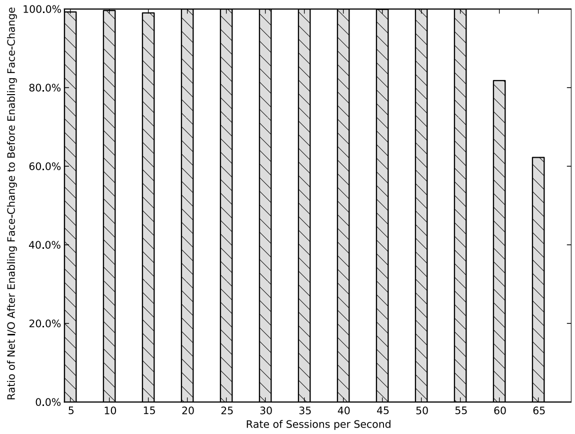

Figure 7: I/O Performance Results for *Apache* Web Server

2) I/O Performance for Apache*:* In addition to measuring overall system performance, we also evaluate FACE-CHANGE's influence on application's I/O performance. Specifically, we use *httperf* to compare *Apache*'s performance before and after enabling FACE-CHANGE. In this test, we increase the request rate from 5 to 60 requests per second (100 connections in total) to test the I/O performance. We present the ratio of the I/O throughput after enabling FACE-CHANGE to before in Figure 7. From Figure 7, we find that I/O throughput will not be affected below the threshold rate of 55 reqs/second but may begin to degrade afterwards. This indicates that our system has no influence on the network throughput before the CPU becomes a bottleneck. The reason is that the bursts of network traffic cause frequent kernel view switching in a short period of time. One solution is to measure the rate of requests for an expected workload of the server before enabling FACE-CHANGE. If the rate is below the threshold rate, the application's I/O throughput should be unaffected by FACE-CHANGE. If the rate is far over the threshold rate, FACE-CHANGE may require a more powerful CPU to handle any traffic peaks in the network without slow-down.

V. DISCUSSION

In this section, we discuss the limitations of our current approach and propose some potential directions for future work.

A. Malicious Attack within the Application-specific Minimized Kernel Attack Surface

Our approach aims to minimize the kernel attack surface for each specific application. If a malicious attack breaks the boundary of the kernel view generated in the profiling phase, we can detect and report the violations. Compared to system-wide minimization techniques, FACE-CHANGE enforces stricter constraints on kernel code visibility. It is still possible, however, that the kernel code used by the malicious attack is within the subset of the application's kernel view. For example, suppose a web server is compromised and a parasite command-and-control(C&C) server is installed. If this C&C server uses only kernel functionalities that are within the kernel view of the host web server, FACE-CHANGE does not need to recover any missing kernel code and it would be impossible for us to detect its existence in this case. This problem may require a deeper understanding and finer-grained profiling of the semantic behaviors of each application. In addition to recording an application's kernel usage in the profiling phase, we also need to profile the application's behavior, specifically its interactions with the kernel. Thereby we can classify the malicious behavior during the runtime phase if it violates the application's known behaviors.

B. Non-persistent/DKOM Kernel Rootkit

Non-persistent kernel rootkits perform a one-time attack on the kernel and attempt to remove any traces of the incident. If such an attack happens before enabling FACE-CHANGE, then we have already missed the opportunity to capture the attack.

For DKOM rootkits [15], which only manipulate kernel data, FACE-CHANGE is unable to identify the attack because it only monitors anomalies in kernel code execution. In order to detect this kind of attack, we could integrate some existing works [16], [17] into our system to check the kernel's data integrity. We leave this effort as future work.

C. Multiple-vCPU Support for Guest VM

Our current prototype supports guest VMs with a single vCPU. In order to support multiple vCPUs per guest VM, FACE-CHANGE will need to identify context switches on every vCPU. Each process has its own page table and is pinned to one CPU during execution, likewise each vCPU has its own EPT maintained by the hypervisor. Like before, FACE-CHANGE should manipulate each vCPU's EPT to perform per-vCPU kernel view switching. Extending FACE-CHANGE to support multiple vCPUs per guest VM is our future work.

VI. RELATED WORK

This work was inspired by two broad categories of related works: kernel minimization and sandboxing. In this section, we describe some representative works from each category in detail.

A. Kernel Minimization

Earlier research on kernel minimization was not specifically security oriented. The primary goal of these works was to shrink the kernel's in-memory size to adapt to the limited hardware resources of embedded systems. Lee et al. [2] used a call graph approach to eliminate redundant code from the Linux kernel. Chanet et al. [4] applied link-time compaction and specialization techniques to reduce the kernel memory. He et al. [3] reduced the memory footprint by keeping infrequently executed code on disk and loading it on demand.

Recent research has focused on minimizing the OS kernel to reduce the attack surface exposed to applications. Kurmus et al. [1] proposed a kernel reduction approach which automatically generates kernel build configurations based on profiling results of expected workloads. DRIP [18] is an offline approach to purify trojaned kernel drivers via binary rewriting. It leverages a functional test suite to profile a driver and reserve the minimal required set of kernel function invocations.

Compared to previous kernel minimization works, FACE-CHANGE dynamically presents a customized kernel view to each individual application to minimize the kernel's exposed attack surface. In addition, our system is more flexible and can adapt to changes in the execution environment and support new applications without rebooting the system.

B. Sandboxing

Sandboxing is a general security mechanism that provides a secure execution environment for running untrusted code.

One category of sandboxing works is to constrain the untrusted code's capabilities via predefined security policies. Janus [19] is a filtering approach to perform system call interposition based on the predefined policy. Ostia [20] proposed a delegating architecture to virtualize the system call interface and provides a user level sandbox to control the access of resources. Capsicum [21] extends the Unix API to allow an application to perform self-compartmentalization, i.e., confining itself in a sandbox that only allows essential capabilities. Seccomp [22] is a sandboxing mechanism implemented in the Linux kernel to constrain the system call interface of process. If the process attempts to issue the system call that is not allowed, it will be terminated by the kernel. SELinux [23] is a security module in the Linux kernel that enforces mandatory access-control policies on applications. Similar to SELinux, AppArmor [24] restricts the capabilities of a program through binding a security profile. TxBox [25] is based on system transactions to speculatively execute an untrusted application and recover from harmful effects. Process Firewalls [26] is a kernel-base protection mechanism to avoid resource access attacks through examining the internal state of a process and enforcing invariants on each system call.

Another category of sandboxing approaches is to enforce access control through recompilation, binary rewriting and instrumentation: PittSFIeld [27] extends software fault isolation [28] (SFI) to x86. It checks unsafe memory writes and constrains jump targets to aligned addresses. Vx32 [29] is a sandbox that confines the system calls and data accesses of guest plugins without kernel modification. NaCl [30] leverages SFI to provide a constrained execution environment for the native binary code of browser-based application. TRuE [31] replaces the standard loader with a security-hardened loader and leverages SFI to run untrusted code. Program shepherding [32] enforces security policies by monitoring control flow transfers during the execution of a program.

In the virtualization/emulation environment, a full system is considered to be confined in a sandbox and the protection is provided at hypervisor level: Secvisor [33] ensures that only approved code can be executed in kernel mode to protect the kernel against code injection attacks. NICKLE [34] enforces that only authorized kernel code can be fetched for execution in kernel space. To guarantee the integrity of kernel hooks, HookSafe [35] relocates hooks to a page-aligned memory space and regulates accesses to them via page-level protection. HUKO [36] is a hypervisor-based approach to enforce mandatory access control policies on untrusted kernel extensions. Gateway [37] isolates kernel drivers in a different address space from the base kernel and monitors their kernel API invocations.

FACE-CHANGE can also be considered a type of sandboxing approach. The difference from these previous works is that we sandbox each individual application by constraining its reachability of kernel code. We also enforce our approach at the hypervisor level to be transparent to the guest system.

VII. CONCLUSION

We make a key observation that the kernel code required by applications of different types varies tremendously. Thus, generating a single system-wide minimized kernel will enlarge the attack surface for all applications involved. We develop FACE-CHANGE, a virtualization-based system to facilitate dynamic kernel view switching among individual applications executed in a VM. FACE-CHANGE transparently presents a customized kernel view to each application to confine its reachability of kernel code and switch this view upon context switches. In the event that a process breaks its kernel view boundary, FACE-CHANGE is able to recover the missing kernel code and backtrack this anomaly via analysis of the execution history. Our evaluation demonstrates the drastic difference in the size of kernel views of multiple applications, the effectiveness of FACE-CHANGE in revealing the attack patterns of both user and kernel attacks, and the potential of enabling FACE-CHANGE for production VMs.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments. This research has been supported in part by the AFOSR under award FA9550-10-1-0099, DARPA under Contract 12011593, and NSF under award 0855141. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors above.

REFERENCES

[1] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring," in *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[2] C. tai Lee, J. min Lin, Z. wei Hong, and W. tsong Lee, "An application-oriented linux kernel customization for embedded systems," *Journal of Information Science and Engineering*, 1995.

[3] H. He, S. K. Debray, and G. R. Andrews, "The revenge of the overlay: automatic compaction of os kernel code via on-demand code loading," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007.

[4] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "System-wide compaction and specialization of the linux kernel," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[5] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[6] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[7] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[8] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[9] "RUBiS," http://rubis.ow2.org/.

[10] "injectso: Modifying and spying on running processes under linux," http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt.

[11] "Cymothoa - Stealth backdooring tool," http://cymothoa.sourceforge.net/.

[12] "Single Process Parasite," http://www.phrack.org/issues.html?issue=68&id=9#article.

[13] "Injected Evil(executable files infection)," http://z0mbie.host.sk/infelf.html.

[14] "kbeast-v1," http://core.ipsecs.com/rootkit/kernel-rootkit/kbeast-v1/.

[15] J. Butler, "DKOM (Direct Kernel Object Manipulation)."

[16] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, 2010.

[17] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, 2009.

[18] Z. Gu, W. N. Sumner, Z. Deng, X. Zhang, and D. Xu, "Drip: A framework for purifying trojaned kernel drivers," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, 2013.

[19] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[20] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A delegating architecture for secure system call interposition." in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[21] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix." in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[22] "Seccomp and sandboxing," http://lwn.net/Articles/332974/.

[23] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[24] M. Bauer, "Paranoid penguin: An introduction to novell apparmor," *Linux Journal*, 2006.

[25] S. Jana, D. Porter, and V. Shmatikov, "Txbox: Building secure, efficient sandboxes with system transactions," in *Security and Privacy (SP), 2011 IEEE Symposium on*, 2011.

[26] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Process firewalls: Protecting processes during resource access," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[27] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[29] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008.

[30] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*, 2009.

[31] M. Payer, T. Hartmann, and T. R. Gross, "Safe loading-a foundation for secure execution of untrusted programs," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.

[32] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[33] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[34] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[35] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[36] X. Xiong, D. Tian, and P. Liu, "Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[37] A. Srivastava and J. Giffin, "Efficient Monitoring of Untrusted Kernel-Mode Execution," in *Proceedings of the 18th Annual Network and Distributed Systems Security Symposium*, 2011.