Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine

Nguyen Anh Quynh, Kuniyasu Suzaki
National Institute of Advanced Industrial Science and Technology, Japan.
{*nguyen.anhquynh,k.suzaki*}*@aist.go.jp*

Abstract

This paper presents *Xenprobes*, a lightweight framework to probe the guest kernels of Xen Virtual Machine. Xenprobes is useful for various purposes such as as monitoring real-time status of production systems, analyzing performance bottlenecks, logging specific events or tracing problems of Xen-based guest kernel. Compared to other kernel probe solutions, Xenprobes introduces some unique advantages. To name a few: First, our framework puts the the breakpoint handlers in user-space, so it is significantly easier to develop and debug. Second, Xenprobes allows to probe multiple guests at the same time. Last but not least, Xenprobes supports all kind of Operating Systems supported by Xen.

1 Introduction

Testing and debugging Operating System (OS) kernel is a hard and tired job of kernel developers. An easy and intuitive solution like inserting debug code (such as *printk()* function in Linux) into the kernel source, then recompile it and reboot the system is widely used. Nevertheless, this technique has a major drawbacks: It is very time-consuming and slow, especially because compiling kernel might take no less than 30 minutes on old machines. Moreover, in general commercial OS-es do not provide the source code for us to modify and recompile in the first place.

One solution to the problem is to use kernel debugger [16] [10]. However, while kernel debuggers allow developers to inspect kernel at run-time, a debugger is not always desirable because it requires user interactivity. In case the testing and debugging must be done automatically, for example to monitor system status in a long time, this approach is not up to the task.

Hence the advent of dynamical *probing* technique, which allows the developers - at run-time - to specify the actions when corresponding events happen in the kernel. Basically this technology dynanmically inserts a breakpoint into a running kernel, without having to modify the kernel source. The place of the breakpoint, usually specified by its address in kernel address space, is called *probe-point*. Each *probe* associates a probe-point with a *probe handler*. A probe handler runs as extension to the system breakpoint interrupt handler and usually has little or no dependence on system facilities. Probes are able to be inserted in the almost everywhere in the kernel, like interrupt-time, task-time, disabled, inter-context switch and SMP-enabled code paths, etc...

In Linux world, such a probe framework is *Kprobes*. Kprobes was merged into Linux kernel from version 2.6.9, and provides a lightweight interface for kernel modules to inject probes and register corresponding probe handlers. Kprobes are intended to be used in test and development environments. During test, faults may be injected or simulated by the probing module. In development, debugging code, like a *printk()* function, may be easily inserted without having to recompile the kernel.

However, solutions like Kprobes have some major shortcomings. Here are the most notable drawbacks:

1. Probe handlers come in the shape of kernel code, specifically in kernel module. The problem is that comparing to programming in user-space, programming in kernel context is limited by many restrictions, such as allocating and accessing resource, short on available library, no floating-point math, etc... Programming for OS kernel is considered very complicated and tricky, thus usually requires highly experienced developers, because a broken code can make the whole system in-stable.

   That is a reason why programming in user-space is always encouraged over kernel-space. In fact, there is a basic principle within kernel developer community: Whatever can be done in user-space, never do it in kernel-space unless absolutely necessarily. Various attempts from different open source

projects trying to submit their work to Linux kernel are rejected, and asked to re-architecture their code to work in user-space instead.

2. Kprobes cannot put the probes in some places in the kernel, such as in the code that implements Kprobes and functions like *do_page_fault()* and *notifier_call_chain()* [8].

3. Kprobes makes no attempt to prevent probe handlers from stepping on each other – for example it is not a good idea to probe *printk()* and then call *printk()* from inside the probe handler. If a probe handler hits a probe, then the second probe's handlers will not run in that instance, and the *kprobe.nmissed* member of the second probe will be incremented [8].

4. Kprobes is designed and works for Linux only, thus the probing code made for Kprobes cannot be easily reused for other OS-es.

Recently virtual machine (VM) technology has emerged as one of the hottest topics in computer research. The principle of VM technology is to allow the creation of many virtual hosts running at the same time on the same physical machine, each running an instance of an OS. Obviously VM software such as Xen Virtual Machine [5] [14] can help to reduce both hardware and maintenance costs for organizations that need to use various machines for different services. Especially Xen even offers a convenient method of debugging OS kernel, so it is possible to debug an OS like debugging an user-process [1].

Taking the advantage of Xen technology, this paper proposes a framework named *Xenprobes* for probing Xen-based guest OS kernel. Similar to Kprobes, Xenprobes allows developers to dynamically inject probes into guest VMs at run-time. However, Xenprobes is able to address the above-mentioned problems of Kprobes:

1. Xenprobes' handlers completely work in user-space, therefore significantly easier than Kprobes to develop the handlers, even with less experienced programmers. This includes the benefit of using any library available in user-space.

2. Xenprobes can put the probes at any place in probed VMs without worrying about conflict.

3. Because the Xenprobes handlers run in user-space instead of in the kernel of the probed VM, we eliminate the problem of stepping on other handlers like in Kprobes' case.

4. Xenprobes is OS-independent, and can provide service for any OS supported by Xen. In addition, Xenprobes is designed to support probing multiple VMs at the same time.

The rest of this paper is organized as followings. Section 2 briefly covers some background of Xen Virtual Machine and Xen debugging technique. Section 3 presents the architecture and implementation of Xenprobes framework, while section 4 discusses some issues that can be raised when using the framework. Section 5 evaluates the performance overhead of our framework in Linux guest kernels. Section 6 summaries the related works, and compare their advantage as well as drawbacks with our approach. Finally we conclude the paper in section 7.

2 Background on Xen Virtual Machine

Our framework Xenprobes is based on Xen, and exploits the debugging architecture of Xen to inject software breakpoints into probed VM. In this part, we will take a brief look at Xen technology. After that we discuss the kernel debugging architecture for Xen VM version 3.0.3, the latest version as of this writing, which is used by our Xenprobes.

2.1 Xen Virtual Machine

Xen is an open source virtual machine monitor initially developed by the University of Cambridge Computer Laboratory and now promoted by various industrial players like Intel, AMD, IBM, HP, RedHat, Novel and by the open source community. Xen can be used to partition a machine to support the concurrent execution of multiple operating systems (OS). Xen is outstanding because the performance overhead introduced by virtualization is negligible: the slowdown is around 3% only ([4]). Various practices take the advantages offered by Xen, such as server consolidation, co-located hosting facilities, distributed services and application mobility, as well as testing and debugging software.

Basically, Xen is a thin layer of software running on top the bare hardware. This layer is either called hypervisor or virtual machine monitor. Its main job is to provide a virtual machine abstraction to the above OSes. Running on top of Xen, VM is called Xen domain, or domain in short. A privileged special domain named *Domain0* (or *Dom0* in short) always runs. Dom0 controls other domains (called *User Domain*, or *DomU* in short), including jobs like start, shutdown, reboot, save, restore and migrate them between physical machines. Especially, Dom0 is able to map and access to memory of other DomUs at run-time.

Initially, Xen only supports the *para-virtualization* technique, in which Xen exposed a hardware architecture called *xen* to the VMs, and the OS-es running on Xen must be modified to work with *xen*.

Recently, realizing the potential of virtualization, Intel and AMD launch special CPUs that support virtualization at lowest level [7] [2]. Xen takes the advantage of these processors to provide the *full-virtualization* technique, in which all OS-es can run on Xen without any modification.

2.2 Exceptions Handling in Xen

Xen handles exceptions differently with para-virtualization and full-virtualization.

- **Para-virtualization**: In Xen, to manage VMs and the physical hardware, the hypervisor layer runs at the highest privilege level (ring 0 in the case of x86 architecture). To provide a strong isolation between VMs as well as between VMs and the hypervisor, all the VMs are modified run at lower level (ring 1 in the case of x86 architecture). So are the interrupt handlers of VMs: While normally the interrupt handlers are registered in the interrupt descriptor table (IDT), Xen does not allow VMs to install their handlers themselves because of the security reasons: it cannot give VMs the direct access to the below hardware. Instead, VM kernels are modified at source code, so the hypervisor captures the interrupts instead of letting the VMs handle them.

  Specifically, in the asynchronous interrupt case, also called exception and generated when the system executes the *INTO*, *INT1*, *INT3*, *BOUND* instructions or caused by page faults: these exceptions are processed in the hypervisor layer first instead in the VM's kernel. To register handlers, a VM's kernel is modified to call the hypercall named *HYPERVISOR_set_trap_table* to setup the exception handlers. The handlers are functions initialized at machine boot time, and managed by hypervisor layer.

- **Full-virtualization**: Virtualization-enable processors such as Intel-VT and AMD-V support full virtualization by adding new privileged ring, which is at a higher privilege level than ring 0. Xen runs this privilege, and lets the OS-es run in ring 0. In ring 0, the OS-es run as normally without being aware that it is managed by the hypervisor run below. Xen virtualizes the processor for VMs on it by intercepting special instructions and exceptions. In case of debugging related instructions *INT1* and *INT3*, these interrupts are intercepted as privilege ring transitions, then virtualized exceptions are injected into related VMs.

  For more detail, readers are encouraged to read the paper [1].

2.3 Debugging Support Architecture in Xen

Similarly to exceptions, there is a little difference in the way Xen supports debugging in para-virtualization and full-virtualization.

- **Para-virtualization**: In x86 architecture, *INT3* is a breakpoint instruction which is used for debugging purpose [1]. Whenever this instruction is hit, the control is passed to the exception handler of *INT3* in kernel space. In Xen, the sequence of handling the *INT3* exceptions is as in the following steps:

  - When the VM hits the breakpoint instruction, the exception *#BP* is raised.
  - The system makes a hypervisor switch to give control to the *INT3* handler staying in the hypervisor layer.
  - The *INT3* handler in hypervisor checks if VM is in kernel mode. If that is not the case, Xen returns the control to VM.
  - If the exception comes from VM's kernel, Xen pauses the VM for inspection.

  In fact, the Xen debugger works by exploiting the mentioned feature: When the debugger server running in Dom0 detects that the concerned domain is paused, it comes to inspect the VM's kernel, then resume it after it finishes the job [1].

  Besides *INT3*, *INT1* is another special interrupt made for debugging. This interrupt sends the processor into the single-step mode, in which after each construction, the handler of *INT1* is called. To make this happen, we only need to enable the trap flag *TF* of the *FLAGS* register. The processor switches to normal mode if the *TF* flag is turned off. And similarly to the case of *INT3*, when the system is in single-step mode, after each instruction the control is changed to the INT1 handler at hypervisor layer. The sequence of handling *INT1* is same as in *INT3*'s case.

- **Full-virtualization**: The way Xen handles *INT1* and *INT3* in full-virtualization is quite straightforward: when the processor hits *INT3*, a *#BP* exception is raised, resulting in a privilege transition from to the hypervisor. The hypervisor then simply pauses the VM for inspection.

[1] Breakpoint instruction is an one-byte opcode with the value of *0xCC* on x86 platform.

The processing of *INT1* is similar. For more detail, please see [1].

3 Xenprobes Architecture and Implementation

Xenprobes works by modifying the kernel code of probed VMs at run-time, in which it replaces the instruction at the probe-point with a breakpoint instruction. Xenprobes provides a very lightweight and simple framework, so the developers can easily employ it and writes his probing code in user-space of Dom0.

In this section, we present the framework, then go into detail of the implementation of Xenprobes.

3.1 Xenprobes Framework

In the design of Xenprobes, we take advantage of the Xen debugging infrastructure: We notice that if we put a breakpoint into a domain memory, whenever the breakpoint is hit in execution, the control is given to the hypervisor, and the domain is paused for inspection. So if we put the breakpoint handlers in user-space of Dom0 and somehow inform Dom0 about the breakpoint event, Dom0 can run the corresponding handler and let the handler does everything in user-space.

3.1.1 Xenprobes Types

Xenprobes allows the developers to dynamically inject breakpoints into any place in a Xen VM, and collect debugging and performance information non-disruptively. We can trap at any kernel code address, specifying handler functions to be invoked when the probe-point is hit.

We define a Xen probe (or probe in short from now on) as a set of handlers placed on a certain instruction address of a specific VM. For convenience, the instruction address is set in *virtual address* of that VM, so the developer can take the address from the kernel symbol file accompanied the VM kernel binary.

Currently, Xenprobes supports two types of probes, *XProbe* - or *XP* in short - and *XrProbe* - or *XrP* in short. An XP can be placed on any instruction in the kernel. An XrP is put in the entry of a specified function, and fired when entering and leaving the function. XrP can be used to collect information such as parameters and returned value of kernel functions.

Each XP comes with a pair of functions called *pre-handler* and *post-handler*. When a breakpoint of a particular VM is hit, the corresponding pre-handler is executed just before the execution of the probed instruction. The post-handler is executed right after the execution of the probed instruction.

An XrP is also accompanied by two handlers: a *entry-handler* and a *return-handler*. The entry-handler is executed when the probe is first hit, usually when the execution enters the probed function. This is the approriate time to gather the parameters of function. When the probed function returns, the return-handler is fired, and the returned value of the function can be collected.

The handlers can do other things such as check and modify registers, inspect and change the memory content of the probed VM. In addition, because Xenprobes handlers operating in user-space context of Dom0, it is possible to use use any library available in user-space, as with any other user-space application. In contrary, Kprobes handlers run in kernel context instead, thus cannot have such flexibility.

3.1.2 Xenprobes Framework

The Xenprobes framework has been designed in such a way that tools for debugging, tracing and logging could be built by extending it. The framework provides an user-space library, so the developers can use it to write their probing applications.

In the design, Xenprobes supports multiple hardware architectures and hides all the internal complexity in order to give developers a very simple interface to work with. The framework's interface is described in C programming language with some newly-defined data types and seven functions as in Figure 1. To employ the framework, developers simply include a C prototype header file named *xenprobes.h*, and compile their code with *xenprobes* library.

Below is a brief summary of Xenprobes interface.

- *xenprobes_handle_t*: Xenprobes defines a new data type to manage probes, named *xenprobes_handle_t*. Both XP and XrP can be referred to using a variable of this data type.

- *xenprobes_handler_t*: The breakpoint handler can be defined using a new function pointer type *xenprobes_handler_t*. The handler is a function with two arguments: a probe handle of *xenprobes_handle_t* type and a pointer to the virtual cpu structure of *cpu_user_regs*. [2]

- *register_xenprobe()*: An XP must be registered using this function. *register_xenprobe()* requires four arguments: The first is a domain id, which indicates which Xen VM we want to probe. The second argument is the address where we want to inject the breakpoint. For convenience, this is the virtual address in the VM. The third and fourth argument are

[2] *cpu_user_regs* is a data type defined by Xen and can be referred to by including a C header file *xenctrl.h* from Xen's *libxc* library.

pre-handler and post-handler, which are called before and after the original instruction at the probepoint is executed. One and only one of these last two arguments can be *NULL*, and can be used if we want to ignore the pre- or post- event.

This function returns a handle which will be used as the first argument of the probe handlers when they are called. *XENPROBES_HANDLE_ERROR* [3] value is returned in case there is a problem when registering the breakpoint.

- *unregister_xenprobe()*: An XP can be unregistered using this function. *unregister_xenprobe()* takes only one parameter, which is a handle returned when that XP was registered.

- *register_xenretprobe()*: An XrP must be registered using this function. *register_xenretprobe()* is quite similar to the *register_xenprobe()*, but the address argument specifies the entry address of the function we want to probe. The *entry_handler* argument specifies the entry-handler function executed right before the breakpoint instruction is hit, and the *return_entry* argument specifies the return-handler function executed right before the probed function returns. The last argument *maxactive* indicates how many instances of the specified function can be probed simultaneously.

  Without the XrP, if we need to inspect the return point of a function, we may have to put multiple XPs to cover multiple code paths. But with we use XrP, we only need to put a single probe at the entry of the function, and it automatically fires whenever that function returns, regardless of how it exits.

- *unregister_xenretprobe()*: Similar to XP's case, an XrP must be unregistered using this function, which takes an XrP handle as the only argument.

- *xenprobes_loop()*: When all the probes are successfully registered, we can start probing the VMs with this function. *xenprobes_loop()* sends us into an infinite loop, in which Xenprobes waits for the debugging events, that indicates that a particular VM is waiting for probing, and executes the corresponding handlers. This infinite loop only quits if the *xenprobes_stop()* function below is called.

- *xenprobes_enable()*: The probe handlers can call *xenprobes_enable()* function to turn on or turn off another probe. This function takes two no arguments: the first is corresponding handle, and the second argument indicates if we want to enable (if *active* is *1*) or disable (if *active* is *0*) the probe.

- *xenprobes_stop()*: The probe handlers can call this function to stop probing. *xenprobes_stop()* takes no argument, and will immediately get *xenprobes_loop()* out of its loop.

[3] *XENPROBES_HANDLE_ERROR* is a constant value defined in *xenprobes.h*

```
typedef unsigned long xenprobes_handle_t;

typedef int (*xenprobe_handler_t)(
    xenprobes_handle_t,
    struct cpu_user_regs *);

xenprobes_handle_t register_xenprobe(
    domid_t domid,
    unsigned long address,
    xenprobe_handler_t pre_handler,
    xenprobe_handler_t post_handler);

int unregister_xenprobe(
    xenprobes_handle_t handle);

xenprobes_handle_t register_xenretprobe(
    domid_t domid,
    unsigned long address,
    xenprobe_handler_t entry_handler,
    xenprobe_handler_t returned_handler,
    int maxactive);

int unregister_xenretprobe(
    xenprobes_handle_t handle);

int xenprobes_loop(void);

int xenprobes_enable(
        xenprobes_handle_t handle,
        int active);

void xenprobes_stop(void);
```

Figure 1: Xenprobes framework.

We demonstrate the usage of our framework in two simple examples in the Figure 2 and Figure 3. Figure 2 gives some hints on how to use an XP to monitor a Linux VM running on Xen. The XP is registered to watch the *sys_open()* function, so it can notify us each time the *open* system-call is executed. For brevity, the sample assumes that the probed VM has domain id of *1*. We get the virtual addresses of the *sys_open()* function, which is available with the *sys_open* symbol at the address *0xc01511d0* from the symbol file *System.map* accompanying the binary kernel of the Linux VM. The pre-handler of the XP,

function *xp_open()*, prints out each time the *open* system-call is executed, for example when we *read* a file, and quits the loop after *10* times by calling *xenprobes_stop()* function. Finally, it removes the XP after the loop with *unregister_xenprobe()*.

Sample in figure 2 hints us how to probe with an XrP. The XrP is registered to watch the *sys_unlink()* function (at the address *0xc0161780*, corresponding to symbol *sys_unlink* in *System.map* file.) of domain *1*, so it can notify us the function parameters and returned value each time the *unlink* system-call is executed. The entry-handler *xrp_entry_unlink()* prints out the address of the *path-name* parameter of the system-call, which is retrieved from the second integer above the stack pointer *ESP*. The return-handler *xrp_return_unlink* prints out the returned value of the system-call, retrieved from the *EAX* register. This returned value indicates the result when when a file is *removed*. We want to probe the *unlink* system-call at most *8* times at a time. The probe also quits the loop after *5* times by calling *xenprobes_stop()* function. Finally, it unregisters the XrP after the loop.

These samples must be compiled with the Xenprobes library and run in Dom0.

```
...
static int xp_open(
    xenprobes_handle_t handle,
    struct cpu_user_regs *regs)
{
   static int count=0;
   count++;
   printf("sys_open: %d\n", count);
   if (count == 10)
       /*quit probe looping*/
       xenprobes_stop();
   return 0;
}
...
xenprobes_handle_t h;

h = register_xenprobe(
   1,          /*domain id*/
   0xc01511d0,/*sys_open address*/
   xp_open,  /*XP handler*/
   NULL);    /*No post-handler*/

xenprobes_loop(); /*Enter the loop*/
unregister_xenprobe(h);
...
```

Figure 2: A simple example on how to use XP.

```
...
static int xrp_entry_unlink(
    xenprobes_handle_t handle,
    struct cpu_user_regs *regs)
{
   unsigned long *stack;
   static int count=0;
   count++;
   stack = &regs->esp;
   /*stack[0] = returned address*/
   printf("sys_unlink: path-name @%x\n",
          (unsigned int)stack[1]);
   if (count == 5)
       /*quit probe looping*/
       xenprobes_stop();
   return 0;
}

static int xrp_return_unlink(
    xenprobes_handle_t handle,
    struct cpu_user_regs *regs)
{
   static int count=0;
   count++;
   /*get the returned value in EAX*/
   printf("sys_unlink returned: %d\n",
          regs->eax);
   if (count == 5)
       /*quit probe looping*/
       xenprobes_stop();
   return 0;
}
...
xenprobes_handle_t h;

h = register_xenretprobe(
   1,          /*domain id*/
   0xc0161780, /*sys_unlink addr*/
   xrp_entry_unlink, /*entry-handler*/
   xrp_return_unlink,/*return-handler*/
   8);  /*At most 8 probes handled*/

xenprobes_loop();  /* Enter the loop */
unregister_xenretprobe(h);
...
```

Figure 3: A simple example on how to use XrP to retrieve function parameters and returned value.

3.2 Xenprobes Implementation

Xenprobes heavily depends on specific features of processor architecture and uses different mechanisms depending on the architecture on which it is being executed. At the moment, Xenprobes is available on i386 and x86_64 architectures.

Xenprobes is provided in a shaped of an user-space library named *xenprobes* in Dom0. Totally the code is around less than 4000 lines of C source code, in which the architecture-dependent code is around 1000 lines.

3.2.1 Performance Challenges

One of the first challenges when we implemented Xenprobes is the performance penalty problem: every time a breakpoint is hit leads to several hyper-switches: first is a switch from the probed VM kernel to the hypervisor; second is a switch from hypervisor to Dom0 to have Xenprobes handled the breakpoint event; and finally the control is given back to the probed VM. These switchings can cause a lot of negative impact to to the overall performance of the probed VM.

When we first investigated the problem, we thought it was a good idea to employ the same tactic of the current Xen kernel debugger to monitor VM, because the Xen debugger also exploits breakpoint mechanism to inspect VM's kernel at run-time ([1]). However, this approach has a disadvantage that can badly affect the system performance: the debugger in Dom0 detects the debugging event by periodically polling VM's status to see if it is paused [4], which is the evidence that the breakpoint was hit. By default the checking interval is 10 million nanoseconds, which means breakpoints cannot be processed immediately if they come between the checking time. For debugging purpose, that is not a major concern because performance is not a priority. But for our target, that is unfortunately unacceptable because the whole process slows down significantly.

To address the problem, we decide not to adopt the mentioned polling tactic of the Xen kernel debugger. Instead we exploit a special feature of the debugging architecture in Xen: no matter whether the VM is para-virtualization or full-virtualization, whenever the hypervisor gets debugging control given to it from its VMs [5], the hypervisor sends an event to Dom0 to notify any potential debugger running there. While the standard debugger does not use this feature, we do employ it, and have Xenprobes handled the debugging event. To do that, Xenprobes only needs to put the protected VM into the debugging mode [6], and binds to the virtual interrupt *VIRQ_DEBUGGER*, which is dedicated for debugging event, to get notified by the hypervisor. Thanks to this strategy, Xenprobes is instantly aware when probed VMs hits the breakpoints, therefore does not need to poll VMs for the paused status. Some experiements demonstrate that our approach significantly improves the overall performance.

3.2.2 Access VM's Kernel Memory

In Xenprobes architecture, we need to read and write to VM's kernel memory, for example to read the original instruction at probe-point and overwrite it with the breakpoint. In order to access to a specific virtual address of VM, we must first translate it into physical address. Currently Xen support several kinds of architecture: *x86_32*, *x86_32p* and *x86_64*, and each of these platforms has different schemes of paging memory. Hence Xenprobes must detect the underlying hardware, and then translates the virtual memory accordingly by traversing the page table tree.

To traverse the page table tree, it is imperative to know the physical address of the page directory. In Xen, we can have the virtual control register *cr3* of each virtual CPU of VM by getting corresponding CPU context via Xen function *xc_vcpu_getcontext()* [17]. Besides, as Xen supports several architectures such as *x86*, *PAE* and *x86_64* (thus different page-table formats), Xenprobes must handle the page-table accordingly to convert the virtual address to physical address.

After that, Xenprobes accesses memory of VM by mapping the physical address with the function named *xc_map_foreign_range()* [17]. Then it goes on reading or writing to the mapped memory [7].

To ensure integrity, each read or write access to the VM requires pausing it, and we need to resume it back after finishing.

3.2.3 Out-of-line Execution Area

Regarding the technique of handling the breakpoints, Xenprobes adopts the same solution proposed by Kprobes [8] [11], with some modifications. Basically Xenprobes replaces the instruction at the probe-point with a breakpoint. The original instruction at that point is copied to a separate area, which is executed when Xenprobes handles the breakpoint event. We call this area "*Out-of-line Execution Area*" or *OEA* in short.

Regarding the size of each OEA: besides storing the original instruction, each OEA must also have enough

[4] This can be done thanks to the Xen's *libxc* function *xc_waitdomain()*.

[5] This means either the processor hits the breakpoint, or it is put into the single-step mode

[6] This can be done with a domain control hypercall, with the special command *XEN_DOMCTL_setdebugging*.

[7] This depends on the mapped access is *PROT_READ* (read) or *PROT_WRITE* (write).

space for one relative jump instruction, which is used to jump back to the instruction next to the original instruction. Because the instruction size varies on different architectures, we move the code handling OEA into the architecture layer of Xenprobes.

This approach raises a question: how to have the OEA for each probe? Kprobes solves this issue simply by allocating an area of memory as OEA for each registered probe, and frees the OEA when unregistering the probe. In principle, a new OEA is assigned on demand whenever there is a need for a new probe.

However, this technique cannot be employed in our case: the probes are registered from user-space of Dom0 instead of from inside the corresponding kernel as with Kprobes. If we need to allocate an OEA for a new probe, there is no clean way to ask the probed VM to allocate the new chunk of memory inside its kernel for us.

We solve the problem by a simple method: We preallocate a fixed area of memory from inside the probed VM to store the OEAs for upcoming probes. We split the area into contiguous, non-overlap chunks of memory, and each chunk can be used as one OEA. When a new probe is registered, a free chunk will be given to that probe and the probe will use it as OEA. Xenprobes manages all the chunks from Dom0 with a bitmap structure, which indicates which chunk is in-use, which chunk is still free, thus can be allocated. Whenever a probe is unregistered, its associated OEA chunk is recovered for other demands.

To allocate the area of memory for OEAs, each VM that wants to support Xenprobes must be loaded with a kernel module. We provides such a module for Linux VM, named *xenprobesU*. This module is very simple: the only job of it is to allocate a configurable size of memory. The virtual address of this area and its size are then sent to Dom0 using the *XenBus* [17] interface . These values will be picked up from the *Xenstore* [17] in Dom0, and Xenprobes can then determine how many OEAs are available for each probed VM. More discussions on this issue are delayed to section 4.

3.2.4 Probes Registration

There are some differences on the way Xenprobes processes registration for XP and XrP.

- **XP Registration**: Registering an XP probe leads to allocate a dynamic memory in user-space of Dom0 for a new probe. All the probes are managed in a hash list, and the new probe is added to the list. Similarly to Kprobes, Xenprobes supports multiple probes, called *aggregate probes* by Kprobes, at the same probe-point. That allows the developers to register more than one probe at the same instruction address. All the probes have a list of aggregate probes named *aggregate list*, which is *NULL* normally. When a new probe needs to register at the same address, Xenprobes puts it in the *aggregate list* of the first probe, and execute the handlers of all the probes in the list, one by one and in the order, when the breakpoint at that address is hit at execution.

  After allocating memory for the new probe, the control is given to the architecture dependent code, in which the probe is prepared according to specific characteristics of the architecture. Xenprobes gets one free OEA for the probe. Then the instruction at the probe-point is copied to the OEA. Note that OEA stays inside the probed VM, so this steps requires one read and one write access to the VM's memory: the instruction is firstly read from the probe-point, then it is immediately written to the OEA.

  To speed up the procedure of breakpoint handling, we employ the *booster* technique proposed by Kprobes started from Linux kernel 2.6.17 [1]. The instruction at the probe-point is first checked to see if it is *boostable* [8]. The boostable instruction makes the probe boostable, and allow us to execute the instruction as if in inline case. The trick is to use a relative jump instruction to come back to the instruction next to the probe-point. This technique allows us to skip the single-step mode, thus significantly improve the performance.

  Any failure in allocating memory or preparing the probe return the error *XENPROBES_HANDLE_ERROR*. The system variable *errno* will be set to indicate what went wrong.

  An XP probe can be unregistered after finishing probing. The framework function *unregister_xenprobe()* removes the probe from the hash list of probes, then frees the OEA for later usage, and recovers the original instruction at the probe-point. After that, execution hit this point will not raise Xenprobes handler any more.

- **XrP Registration**: XrP actually builds on top XP to avoid duplicating code. When registering an XrP with the *register_xenretprobe()* function, Xenprobes puts a *entry-XP* at the entry of the probed function, and uses the entry-handler argument as the pre-handler of the entry-XP. Note that this entry-XP does not have the post-handler (in fact its post-handler has the *NULL* value). When the probed

[8] Boostable instructions are all instructions not belong to the set of *unboostable* instructions like relative jumps, relative calls or instructions that has hardware side-effect [11].

function is executed and this entry-XP is hit, Xenprobes saves a copy of the function's return address, and replaces it with the address of a *trampoline-XrP* in the probed VM.

The trampoline-XrP routine is actually a piece of code provided by the *xenprobesU* module, in which its only job is to execute a breakpoint (*INT3* instruction in i386 case). The address of this trampoline-XrP is also sent to Xenprobes via XenBus/Xenstore at initialization time together with the information about OEA memory.

The *maxactive* parameter specifies how many instances of the function can be probed at the same time, and *register_xenretprobe()* will preallocate enough memory to save the return address of the function. For example, if the function is non-recursive and is called with a spinlock held in the kernel, *maxactive* can get the value of 1. If the function is non-recursive and can never relinquish the CPU (like via a semaphore or preemption), we can set this parameter to the number of virtual cpus that the probed VM has.

One problem Xenprobes must handle is that when a VM is shutdown, all the probes as well as OEA memory is gone. Regarding this issue, Xenprobes watches for the shutdown VM by registering the built-in Xen watch *@releaseDomain* [17]. When this watch is fired, which indicates that the VM is not existent anymore, Xenprobes removes all the probes of the related VM.

3.2.5 Handling Probes

After probe registration step, the probing process starts by executing *xenprobes_loop()*. Firstly, this function registers to get notified by the hypervisor on debugging event by binding to the virtual interrupt *VIRQ_DEBUGGER*. Then it switches all the probed VMs to debugging mode and enters an infinite loop. In this loop, we listens for the debugging events sent from Xen. When a breakpoint is hit inside the kernel of a probed VM, the hypervisor pauses it, and sends a debugging event to notify Dom0. As the xenprobes_loop() registers to get the event, it comes up to handle the breakpoint. The procedure at this step is also different for XP and XrP, as followings.

- **XP Handling**: The XP is processed in the following sequences:

  (1) Find the XP probe related to this debugging event. After that we get the registers of the probed VM [9], which is later given to the probe handlers as a function parameter. If the processor is handling single-step mode, go to the step (3). Otherwise, we are handling breakpoint, so we execute the *pre-handler* of the probe. In case this is an aggregate probe, execute all the pre-handlers got from the aggregate list.

  (2) Point the instruction register [10] of the probed VM to the head of the OEA of the probe. As the OEA saves the original instruction at the probe-point, this will execute the original instruction. Then if the instruction is boostable, and there is no post-handler, go to the step (4). Otherwise, put the probed VM into the single-step mode [11]. The modified VM context which includes the new register value is updated [12]. After that, the probed VM is resumed.

  (3) The instruction in the OEA is executed. As the processor is in single-step mode, it traps back to the hypervisor, and another debugging event is sent to *xenprobes_loop()*. At this point, we execute the *post-handler* of the probe. After that, we fix the instruction pointer to point it to the instruction next to the original instruction at the probe-point. Then we resume the probed VM, and let it run normally, and wait for the the next breakpoint event (Do not go to the next step, as it handles the boostable-only instruction).

  (4) The instruction is boostable, so we just run the original instruction by pointing the instruction pointer to OEA, update the VM's context and resume the probed VM. The loop repeats with the next debugging events.

  Actually there is a hidden procedure in the step (3) above: We must prepare for the boost in the first ever single-step time in case the instruction is boostable. We use information on the first probe hit to determine the location to put the jump instruction after the instruction at the head of OEA, as presented in [1]. So even if the instruction is boostable, we will execute it in single-step once in the first ever time the breakpoint is hit, but from the second time we can skip it and just run the OEA directly.

  In conclusion, an XP causes at least one hyperswitch to Dom0. Ideally, probed instruction is boostable and there is no post-handler, Xenprobes

[9] Registers of a VM can be retrieved with the *libxc* function xc_vcpu_getcontext().

[10] On i386, that is the EIP register.

[11] On i386, this can be done by enabling the trap flag (TF) of the FLAGS register.

[12] Xen VM context can be set with the *libxc* function *xc_vcpu_setcontext()*.

only has to switch out once to execute the pre-hancler in Dom0. In the worse case, when the above condition is not satisfied, we have to single-step the probed instruction in OEA, thus need the second switch after single-step. Of course in term of performance, that causes negative impact to the VM.

- **XrP Handling**: Note that because XrP employs XP in its design, the handling procedure of its XP is similar to the above descriptions. However, there are few differences when it comes to execute its handlers.

  – When the probed function is executed, its entry-XP is hit. We switch out to Dom0 and execute its pre-handler, which is in fact the entry-handler of the registered XrP. After that, Xenprobes saves the return address of the probed function in a structure for the corresponding XrP. Remember that when registering the XrP, we must specify how many instances of the function can be probed simultaneously with the argument *maxactive*. However, if this limit is reached, Xenprobes does not save the return address, but simply increases the missed number in the XrP structure for the developers to investigate.

    After the above steps, Xenprobes overwrites the return address of the probed function with the address of the trampoline-XrP, and resumes the VM.

  – When the probed function returns, the trampoline-XrP is called. This code executes a breakpoint instruction as explained above. This action causes another switch to Dom0, and lets Xenprobes execute the return-handler of the XrP. After that, Xenprobes overwrites the return address of the probed function, and point the instruction pointer to this address. Finally, Xenprobes resumes the VM, which continues to execute at the return address of the probed function. The loop repeats and Xenprobes continues to wait for the next probes.

In conclusion, an XrP causes at least two hyperswitches to Dom0. In case the entry instruction is boostable (which is mostly the case), we have the first switch when the probed function is hit at entry time. At the end, when the function returns we have to switch out once more to execute the return-handler.

In the worse case, when the entry instruction is not boostable, we have to suffer two more hyperswitch for single-step: one is for the entry probe, another is for the return probe. However, as the first few instructions of the function prologue never have *unboostable* instructions [13], it is very unlikely that we have such a problem [11].

At any moment, a probe handler can enable or disable another probe with the *xenprobes_enable()* function. This operation is done by overwriting the probe-point with the breakpoint instruction (in case we want to activate the probe) or the original instruction (in case we want to deactivate the probe).

For both XP and XrP handlers, it is possible to quit the probing loop anytime by calling the framework function *xenprobes_stop()*.

4 Discussion

While the Xenprobes framework is very simple and easy to use, there are some doubts about how much memory is enough for OEA, which place we should put the breakpoints and how to properly access the kernel objects of the probed VM from our handlers. This section is dedicated to discuss these issues.

4.1 OEA Memory Allocation

In our approach, we preallocate an area of memory in the probed VM and use it as OEAs for Xprobes. This can be done thanks to the kernel module *xenprobesU* loaded inside the VM. Regarding the size of this *configurable* area, the developer must anticipate how many probes he wishes to have at the same time. For example, assume that we never use more than *100* probes at once. Each OEA must be able to store one machine instruction and one relative jump instruction, as explained above. In i386, the maximum size of one instruction is *16* bytes, and a relative jump instruction has the fixed size of *5* bytes. Therefore one OEA should have the size of *21* bytes at least. Since we need to have *100* probes, the total size of memory for all the OEAs is *(21 * 100) = 2100* bytes. So in this case, *xenprobesU* preallocates one page of memory, which is *4096* bytes on i386, for OEA at initializing, and that is more than enough for our purpose.

An issue might be raised here: this approach does not allow us extend the memory once we reach the limit of probes. We solve the problem by let Xenprobes notify the probed VM once it sees that the memory is going to run out soon, so *xenprobesU* can allocate a new area of memory for it to use. The notifications regarding new

[13] Instructions such as relative jump, call, software interrupts or that cause hardware side-effects are all *unboostable*.

memory area for more OEAs between Xenprobes and *xenprobesU* are done via XenBus/Xenstore interface.

4.2 Probe Address

Regarding the breakpoints, one of the major concerns is that how can we know exactly where we must put the breakpoints into the probed VM kernel? An intuitive answer for this question is to rely on the kernel source, and and we can decide to put the breakpoints at the addresses corresponding to related lines of source code. Clearly this is a convenient way, because we can inspect the code and see where is the best place to intercept the system flow. So if we know the address in the memory of related lines of code, we can put the breakpoints there. But then, we have another question: how to determine the address of related lines of code?

Fortunately, this problem can be solved quite easily thanks to debugging information coming with kernel binary. In fact, we can exploit a feature made for kernel debugger: If the kernel is compiled with debug option, the kernel binary stores detail information in *DWARF* format about the kernel-types, kernel variables and, most importantly to our purpose, the kernel address of every source code line [6]. As a result, we only need to compile VM's kernel with debug option on, and analyze the kernel binary to get the kernel addresses of the source code lines we want to insert the breakpoints to. Note that this option only generates a big debugged kernel binary file besides the normal kernel binary, and this debugged kernel saves all the information valuable for debugging process. We can still use the normal kernel binary, thus the above requirement does not affect our system at all.

Another choice is to reverse the kernel binary with debugging data, using a tool such as *objdump (1)*. Option *-d* of *objdump* disassembles the machine instructions of the kernel binary, and inform us the virtual address of each instruction, together with the corresponding line of kernel source code. We can investigate the output and easily choose where is the most approriate place to put the probes.

4.3 Accessing VM's Objects

Usually when writing the breakpoint handlers, we want to access to the kernel of the probed VM to inspect its internal status and collect desired information. Here we have a key challenge: how to bridge the semantic gap between the raw memory and kernel objects. To do that, we must be able to have a good knowledge about the OS structure of the VM, so we can have the exact addresses and structures of its kernel objects.

* **Object's address**: In the case of Linux, each global defined object in the kernel is located at a certain memory address, and kept unchanged during its life-time [14]. We can find the address of Linux kernel objects via the kernel symbol file *System.map* coming with the kernel binary.

* **Object structure**: Knowing only the object address is far from enough. For example, in Linux if we want to get the list of kernel modules, first we must retrieve the address of the first kernel module, the global variable *modules*. But then to get the next kernel module pointed by a field named *list.next* in the *module* structure, we must know the relative address of this field in the structure. This job is not trivial, as the *module* structure depends on kernel compiled option, and it might also change between kernel versions [15].

  To extract data about kernel-types, we leverage part of code of LKCD project [15]. LKCD is an open source tool to save and analyze the Linux kernel dump. LKCD can parse the dump thanks to an internal library *libklib*, which extract all the information it needs from the DWARF data in the kernel binary as well as from the kernel symbol file. This library parses the kernel symbols and extracts kernel-types from debugged kernel binary, then caches the data in the memory for its tool named *lcrash* to use. Besides, *libklib* also interprets *lcrash* command, and serves as a disassemble engine for various hardware platforms. Because of these reasons, *libklib* is a very big and complicated code, thus cannot be employed as it is. Another problem is that *libklib* is designed to analyze kernel dump, but not to cope with hostile data. So if somehow the attacker modifies the kernel structure in malicious way, *libklib* might crash.

  In our experiment, we only reused part of *libklib*, in which we only keeps the code that extracts and parses kernel-type information from kernel binary. The library is also hardened to resist potential attacks. Finally, our kernel parse code is around only 14000 lines of C source code, which is about 30% size of the original *libklib*. We plan to include this work into Xenprobes library, so it can be available for all the programs that use our framework.

5 Evaluation

This section presents the performance evaluation of Xenprobes framework comparing with the native speed. The

[14] Note that Linux kernel memory is never swapped out.

[15] Linux kernel never tries to keep compatible between different versions. The Linux kernel developers argue that backward compatibility might block its continuous innovation.

evaluation is done on a para-virtualization Linux VM, the most stable OS platform supported by Xen we have as of this writing. Each benchmark is done in 10 times, and we get the average numbers as the final result.

The configuration of the Xen VMs in the benchmarks are as below:
Dom0: Memory: 384MB RAM, CPU: AthlonXP 2500, IDE HDD: 40GB.
DomU: Memory: 128MB RAM, file-backed swap partition: 512MB, file-backed root partition: 2GB

All the VMs in the tests run Linux Ubuntu distribution (version Breezy Badger).

5.1 Microbenchmark

In the first benchmark, we want to measure how much overhead an XP and XrP can cause to a probed VM. To do that we employ the popular microbenchmark *lmbench* [12]. We inject four probes of XP and XrP into the VM, one at a time, and in the following system-calls: *getppid*, *read*, *write* and *open*. Specifically, we put the handlers at the entry of the *sys_getppid()*, *sys_read()*, *sys_write()*, *sys_open()* functions, respectively. In order to measure the overhead exactly, we use *null* handlers, which are functions doing nothing in the body. We carry out three evaluations: The first, named *Native*, runs the benchmark on the native VM without any probe. The second, named *XP*, registeres the XP with only a null pre-handler. The third evaluation, named *XrP*, registers null entry-handler and return-handler.

Note that since we place the handlers at the entry of these functions, they are always put at *boostable* instruction, so the handling process never suffer a sing-step mode.

The benchmark is done with the commands "*lat_syscall null*", "*lat_syscall read*", "*lat_syscall write*" and "*lat_syscall open*" to measure the overhead on the system-calls *getppid*, *read*, *write* and *open*, respectively. These commands will tell us the latency of these system-calls. Table 1 shows the result of the benchmarks - all the numbers are in microseconds. Next to each number is the overhead compared to the native test, in number of *times*.

	Native	XrP	XP
null	0.2664	107.6731 (404.17)	48.1009 (180.55)
read	0.4732	129.1951 (273.02)	49.6081 (104.83)
write	0.4162	108.8627 (261.56)	49.6027 (119.19)
open	4.0706	117.8936 (28.96)	59.7527 (14.67)

Table 1: Microbenchmark Xenprobes with null handler for XP and XrP.

The benchmarks show us that injecting probes into VM causes quite a big overhead. The *null* benchmark causes the highest penalty for both XrP and XP (*404.17* and *180.55* times, respectively) because *getppid* is a rather simple system-call, thus the main overhead generated is from the switches between the VM, hypervisor and Dom0 when the breakpoint is hit. Meanwhile, the *open* system-call is the most complicated function of all, so the contribute of the penalty by our probes to the overall latency is much more decreased: it is only *28.96* and *14.67* times slower, respectively for XrP and XP.

The notable observation is that in all benchmark, the XrP causes around more than twice overhead compared to the XP. The reason is pretty clear: XrP with two handlers always switches out from its VM twice more than XP with only a pre-handler. In addition, XrP must take time to read and write to the return address of the probed function when the entry function is executed, and when the function returns. These job also causes significant time, as accessing the VM's memory is quite an expensive operation.

5.2 Macrobenchmark

Besides the microbenchmark, we also evaluate Xenprobes in a more reality case with a classical benchmark: decompressing the Linux kernel source. The reason we take this benchmark because Linux kernel contains a lot of data, and the decompress process creates a great number of files and directories. For example unzipping the kernel 2.6.17 generates more than *27000* files and directories, including temporary data. This time we install probes into three system-calls: *mkdir*, *chmod*, *open*. These sytem-calls are triggered when *making directory*, *chmod-ing directories and files*, and *opening files*. These exercises are done quite a lot during unziping kernel: *1201*, *1201* and *19584* times respectively for *mkdir*, *chmod* and *open* [16]. Similarly to the micro benchmark above, this time we also put three XPs (with only null pre-handlers) and three XrPs (with null entry-handlers and return-handlers) , respectively, into the entries of these system-calls.

The benchmark decompresses the Linux kernel 2.6.17 with the command "*time tar xjvf linux-2.6.17.tar.bz2*". Table 2 shows the time to complete the benchmark - all the numbers are in seconds:

	Native	XrP	XP
real	76.781	106.743	81.631
user	44.870	47.360	45.750
sys	5.260	18.380	8.400

Table 2: Macrobenchmark XP and XrP with *mkdir-chmod-open* system-calls and null handlers.

[16] From these numbers, we can safely say that if a new directory is created, it is then immediately *chmod*.

We can see that while the microbenchmark suggests that the probed VM causes very high overhead, in reality the impact is not that much: XP evaluation causes only around *6.31%* penalty, and XrP evaluation causes around *39.02%* overhead. Again, this benchmark shows that probing with XP is significantly faster than XrP. Another observation is that the system mainly suffers in kernel execution, but not in user-space.

In another attempt to measure the impact when more probes are used and put at more performance critical places, we run another test. This time, along with three probes in the above benchmark, we put two more probes (XPs and XrPs, respectively) into the *read* and *write* system-calls. These system-calls are executed in a great number of times when the kernel is unzipped: *78011* times and *139981* times for *read* and *write*, respectively.

The kernel unzip benchmark gives us the result below, in Table 3. All the numbers are in seconds.

	Native	**XrP**	**XP**
real	76.781	165.187	94.572
user	44.870	45.050	44.930
sys	5.260	28.800	16.000

Table 3: Macrobenchmark XP and XrP with *read-write-mkdir-chmod-open* system-calls and null handlers.

Again, we can confirm that even with probes placed at critical execution path in kernel, the performance penalty is not too high. Especially, this benchmark shows the major improvements of XP against XrP: *23.17%* overhead compares with *115.14%* overhead.

Our conclusion is that Xenprobes can be employed to inspect VM's status at run-time without causing too much overhead.

6 Related Works

Our work is strongly inspired by Kprobes, a probing framework available in Linux kernel. Kprobes is widely used for kernel tracing [13] and performance evaluation.

When using Xenprobes for the same job on Xen VMs, our framework has some advantages over Kprobes as mentioned in section 1. Besides, Xenprobes brings several benefits as followings.

- When programming Xenprobes handler, we do not need to worry about page-fault problem, as we work in user-space of Dom0. Kprobes handlers must not cause other exceptions such as page-fault. Though Kprobes allows to specify an user-define page-fault handler to handle the issue, the specified page-fault handler cannot always solve the problem. The Kprobes developers are still working to make the Kprobes fault handling more robust.
- Xenprobes has no problem of reentry as with Kprobes [8].

However, Xenprobes suffers a drawback comparing with Kprobes: Kprobes works in Linux and can interact with the real hardware, thus can be used to monitor and debug all kind of hardware devices. Because Xen VMs cannot work with real hardware, but use the virtual hardware provided by hypervisor, Xenprobes cannot be used to debug arbitrary device drivers. Actually this is a fundamental issue of Xen, rather than Xenprobes.

Another notable difference with Kprobes is that besides "normal" probe, Kprobes supports two other types: Jprobes and Function-return probes [11]. Jprobes is mainly used to gather the probed function parameters, and Function-return probes is used to get the returned value of probed function. Actually from our observation, it is quite common for developers to employ both of these two type of probes at the same time. Therefore, we strongly believe that it is better to combine them, and Xenprobes framework realizes our idea: it is possible to collect both function parameters and returned value with only one XrP probe.

Technically, XrP is inspired by an old version of function-return probes of Kprobes: Until Linux kernel version 2.6.16, Kprobes also employs the "trampoline" technique with two Kprobes to handle the function at return time. But from version 2.6.17, Kprobes uses a new technique, in which the probe for the trampoline is eliminated by some assembly code that saves and recovers the return address. The reason we adopt the old technique of Kprobes for XrP is that because we need two switches to Dom0: first is to execute the entry-handler, and second is to execute the return-handler.

Xenprobes exploits the debugging architecture introduced by Xen. Xen also has built-in support for debugging VM kernel at run-time with gdb [1]. However, as we discussed in the first section, debugging tool does not allow automatic monitoring and probing VM. That is the gap our framework tries to fill in.

Our work shares some ideas with the work of K.Arigos et.al in [3]: their paper also proposes to put breakpoints into Xen VMs to get notified when interested events occur. However, the way we handle debugging events is quite different from [3]: their proposal pushes the breakpoint handlers into the hypervisor layer, and loads the handling policy from Dom0 to hypervisor via an add-in hypercall. Their idea is to let the hypervisor capture the breakpoint events and analyze them there. In order to do that, the authors made quite a big modification to the hypervisor layer (around 2700 lines of code), which they also mentioned as belong to the *Trusted Computing Base* (TCB), the critical and core component required to enforce the system security. We would argue that it is not

desired to make such a major change to such an important component, because it makes the whole system less stable, less secure as well as increase the maintenance cost [17]. In our solution, Xenprobes takes the advantage of the Xen debugging technique, thus makes absolutely no modification to the hypervisor.

Another disadvantage of putting all breakpoint handlers inside the hypervisor is that every time we wish to modify the probe handlers, we have to to alter then recompile the hypervisor, and the system must be rebooted for the change to take effect. Meanwhile, Xenprobes puts all the handlers in user-space, which makes them very easy and convenient to work with. All the development are done in Dom0, and requires no recompilation or reboot whatsoever to the hypervisor.

Last but not least, we go further in providing a framework for injecting breakpoints into the VM, so it can be employed by other projects, not only for security purpose. We are going to publish the code of Xenprobes under open source license (GPL) for everybody to use.

7 Conclusions

This paper describes Xenprobes, a novel framework that allows developers to probe Xen VM's kernel in a more convenient way. In designing Xenprobes, we exploit the infrastructure available for Xen debugging architecture, so the framework offers several interesting benefits in a very simple and easy-to-use interface. Most importantly, Xenprobes allows developers to program their probe handlers in user-space, and it is possible to probe multiple VM at the same time. Because our approach is independent of OS, all the OS-es are supported, even closed source ones such as Microsoft Windows.

Regarding the performance penalty caused by Xenprobes, we believe that the impact is acceptable and can be used in production systems.

Xenprobes internal has quite many things in common with Kprobes, so we believe that it can be easily adopted by the developers who are currently familiar with Kprobes.

We are going to release Xenprobes under the open source GPL license, with the hope that it can attract more interests and become useful for many people.

While the technique to probe guest machine in this paper is specifically described on Xen environment, there is no reason why it does not work on other kind of virtual machine. We are working to have a similar framework on *KVM* [9], a virtual machine technology based on virtualization-enable processor [18].

[17] Xen layer can change anytime, and actually always under active development as of this writing.

[18] KVM has been merged into Linux kernel since version 2.6.20.

References

[1] A.KAMBLE, N., NAKAJIMA, J., AND K.MALLICK, A. Evolution in kernel debugging using hardware virtualization with Xen. In *Proceedings of the 2006 Ottawa Linux Symposium* (Ottawa, Canada, July 2006).

[2] AMD CORP. AMD64 Architecture Programmer's Manual Volume 2: System Programming. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf`, 2005.

[3] ASRIGO, K., LITTY, L., , AND LIE, D. Virtual machine-based honeypot monitoring. In *Proceedings of the 2nd international conference on Virtual Execution Environments* (New York, NY, USA, June 2006), ACM Press.

[4] CLARK, B., DESHANE, T., DOW, E., EVANCHIK, S., FINLAYSON, M., HERNE, J., AND MATTHEWS, J. N. Xen and the art of repeated research. In *Proceedings of the Usenix annual technical conference, Freenix track.* (July 2004), pp. 135–144.

[5] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[6] DWARF WORKGROUP. DWARF Debugging Format Standard. `http://dwarf.freestandards.org/Home.php`, January 2006.

[7] INTEL CORP. Intel Virtualization Technology. `http://www.intel.com/technology/virtualization/index.htm`, 2006.

[8] JIM KENISTON AND PRASANNA PANCHAMUKHI. Kprobes Documentation. `linux-kernel/Documentation/kprobes.txt`, October 2006.

[9] KVM PROJECT. KVM: Kernel based Virtual Machine. `http://kvm.qumranet.com`, 2006.

[10] LINSYSSOFT TECHNOLOGIES LTD. KGDB: Linux kernel source level debugger. `http://kgdb.linsyssoft.com/`, 2006.

[11] MAVINAKAYANAHALLI, A., PANCHAMUKHI, P., AND KENISTON, J. Probing the Guts of Kprobes. In *Proceedings of The Linux Symposium 2006* (July 2006).

[12] MCVOY, L., AND STAELIN, C. LMbench - Tools for Performance Analysis. `http://lmbench.sf.net`, August 2004.

[13] PANCHAMUKHI, P. Kernel debugging with Kprobes. `http://www-128.ibm.com/developerworks/library/l-kprobes.html`, 2004.

[14] PRATT, I., FRASER, K., HAND, S., LIMPACH, C., WARFIELD, A., MAGENHEIMER, D., NAKAJIMA, J., AND MALLICK, A. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium* (Ottawa, Canada, July 2005).

[15] SGI INC. LKCD - Linux Kernel Crash Dump. `http://lkcd.sf.net`, April 2006.

[16] SILICON GRAPHIC INC. KDB: Built-in kernel debugger. `http://oss.sgi.com/projects/kdb/`, 2006.

[17] XEN PROJECT. Xen interface manual. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html`, August 2006.