SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization

Zhui Deng, Xiangyu Zhang and Dongyan Xu
Department of Computer Science and CERIAS, Purdue University, West Lafayette, IN 47907
{deng14, xyzhang, dxu}@cs.purdue.edu

ABSTRACT

The ability to trap the execution of a binary program at desired instructions is essential in many security scenarios such as malware analysis and attack provenance. However, an increasing percent of both malicious and legitimate programs are equipped with anti-debugging and anti-instrumentation techniques, which render existing debuggers and instrumentation tools inadequate. In this paper, we present SPIDER, a stealthy program instrumentation framework which enables transparent, efficient and flexible instruction-level trapping based on hardware virtualization. SPIDER uses invisible breakpoint, a novel primitive we develop that inherits the efficiency and flexibility of software breakpoint, and utilizes hardware virtualization to hide its side-effects from the guest. We have implemented a prototype of SPIDER on KVM. Our evaluation shows that SPIDER succeeds in remaining transparent against state-of-the-art anti-debugging and anti-instrumentation techniques; the overhead of invisible breakpoint is comparable with traditional hardware breakpoint. We also demonstrate SPIDER's usage in various security applications.

1. INTRODUCTION

In a wide range of security scenarios, researchers need to trap the execution of a binary program, legitimate or malicious, at desired instructions to perform certain actions. For example, in high accuracy attack provenance, instruction-level trapping allows recording of events which are more fine-grained than system calls and library calls. In malware analysis, where malware often includes large number of garbage instructions to hamper analysis, it allows analysts to skip such instructions and focus on the instructions that are related to the behavior of malware.

Debuggers [1, 2, 40] and dynamic instrumentation tools [26, 27, 10, 32, 8, 15] both support efficient instruction-level trapping. As a countermeasure, an increasing percent of malware is equipped with anti-debugging and anti-instrumentation techniques. Such techniques are also commonly used in legitimate software for protection purpose [20]. While they do prevent reverse-engineering and software modification, they also render any security application that relies on instruction-level trapping infeasible at the same time.

Researchers have proposed to build systems that enable transparent trapping to solve the problem. However, existing approaches are insufficient to support transparent, efficient and flexible instruction-level trapping. In-guest approaches [36, 34] could be detected by program running in the same privilege level. Emulation based approaches [6, 33] are not transparent enough due to imperfect emulation. Hardware virtualization based systems [14, 28, 38, 37, 12] provide better transparency. However, none of them supports instruction-level trapping with both flexibility and efficiency. Some of them utilize single-stepping which results in prohibitive performance overhead; others could trap only a certain subset of instructions. More detailed discussion about existing work is presented in Section 2.

In this paper, we present SPIDER, a stealthy program instrumentation and debugging framework built upon hardware virtualization. We propose a novel primitive called *invisible breakpoint* that supports transparent, efficient and flexible trapping of execution at any desired instruction in a program. Invisible breakpoint is an improvement over traditional software breakpoint, with all its side-effects hidden from the guest. SPIDER hides the existence of invisible breakpoint in the guest memory by utilizing the Extended Page Table (EPT) to split the code and data view seen by the guest, and handles invisible breakpoint at the hypervisor level to avoid any unexpected in-guest execution. SPIDER also provides data watchpoint which enables monitoring memory read/write at any address.

We have developed a prototype of SPIDER on KVM [3]. We have evaluated the transparency of SPIDER using software protectors and programs equipped with state-of-the-art anti-debugging and anti-instrumentation techniques. The result shows that SPIDER successfully maintains transparency against all of them. We have also applied SPIDER to the following cases: (1) We improve the applicability and security of an existing attack provenance system [25] by replacing its underlying in-guest instrumentation engine with SPIDER; (2) We demonstrate a threat that involves stealthy introspection on protected software to capture sensitive application data. The performance overhead introduced by SPIDER is less than 6% in our case studies. The quantitative cost of each trap is around 3200 CPU cycles according to our measurement, which is less than a previous work [36] and comparable with a hardware breakpoint.

ACSAC '13 December 09-13, 2013, New Orleans, LA, USA

http://dx.doi.org/10.1145/2523649.2523675

2. RELATED WORK

In this section, we take an in-depth look at existing program debugging, instrumentation and analysis tools and discuss their limitations. We only focus on instruction-level tools as they are most related to SPIDER. We classify them into four categories: *in-guest*, *emulation based*, *hardware virtualization based* and *hybrid*.

In-Guest Approaches. Traditional in-guest debuggers [1, 2, 40] use software and hardware breakpoints to gain control at arbitrary points during the execution of a program. In x86, software breakpoint is implemented by replacing the target instruction with a special 1-byte instruction (*int3*), which triggers a #BP exception upon its execution. Hardware breakpoints are implemented as four debug registers (*DR0-DR3*). Each of these registers holds a target address; a #DB exception is triggered upon instruction execution or data access at the target address. Software breakpoints could be easily detected by code integrity checks as the instruction is modified. Hardware breakpoints are not transparent either. The reason is that they are limited resource such that programs could hold and use all hardware breakpoints exclusively to prevent debuggers from using them.

To solve the transparency issue of traditional breakpoints, researchers proposed to use page-level mechanism to trap execution of arbitrary instruction [36, 34]. The page which contains the target instruction is set to non-present, which will cause a page fault upon execution. In the page fault handler, the page is set to present and the target instruction is executed in single-step mode. Then the page is set back to non-present to enable breakpoint again. There are two limitations with this approach. First, execution of any instruction in the non-present page will cause a page fault, even if there is no breakpoint set on that instruction. This would result in prohibitively high performance overhead. Second, although it is not as straightforward as detecting traditional breakpoints, the modified page table and page fault handler could still be detected by kernel-level programs.

Dynamic binary instrumentation (DBI) frameworks [26, 27, 10, 32, 8, 15] are able to insert instrumentation code at arbitrary points during the execution of a program. The mechanism of DBI frameworks is to relocate and instrument code blocks dynamically and handle control flow transitions between basic blocks. Transparency is an important concern in DBI frameworks. For example, position-independent code makes assumption about relative offsets between instructions and/or data. DBI frameworks may break such assumptions when relocating basic blocks, so they must change some instructions in the program to create an illusion that every address is the same as in a native run. However, despite recent efforts [11, 35] targeting at improving the transparency of DBI frameworks, they are still insufficient. A recent work [30] has also shown that there are a number of ways to detect DBI frameworks. More essentially, the DBI framework itself, along with the relocated and instrumented basic blocks must occupy additional memory in the virtual address space. Programs could scan the virtual address space to detect unsolicited memory consumption and hence the DBI framework.

Emulation Based Approaches. To get rid of in-guest components that are visible to guest programs, researchers have proposed to build program analysis and instrumentation tools [6, 33] using full system emulators such as QEMU [7] and Bochs [24]. Full system emulators create a virtual environment for the guest so it feels like running in a dedicated machine. Instruction-level trapping could be easily implemented as each instruction is emulated. However, attackers have been able to identify various methods [16, 17, 29] to detect emulators by exploiting imperfect emulation of instructions and hardware events (e.g. interrupts and exceptions). Although imperfection that is already known could be fixed, the problem still exists as long as there might be unrevealed imperfections. In fact, it has been proved in [14] that determining whether an emulator achieves perfect emulation is undecidable.

Hardware Virtualization Based Approaches. With recent advances in processor features, researchers propose to leverage hardware virtualization to construct more transparent program analysis and instrumentation tools [14, 28, 38, 37, 12]. Hardware virtualization naturally provides better transparency than emulation by executing all guest instructions natively on processor.

None of the existing hardware virtualization based approaches supports transparent, efficient and flexible trapping of arbitrary instructions during execution of a program. PinOS [12] implements a DBI framework on the Xen [5] hypervisor. As it needs to occupy part of the guest virtual address space, it suffers from the same transparency issue as in-guest DBI frameworks. Ether [14] and MAVMM [28] use single-stepping for instruction-level trapping, which triggers a transition between hypervisor and guest upon execution of every guest instruction. Such transition causes significant performance overhead as it costs hundreds to thousands cycles while an instruction only costs several to tens cycles on average. The mechanism is not flexible either as one is forced to single-step through the whole program even if he is only interested in the states at specific points during execution. Such scenario is often encountered when analyzing obfuscated programs which contain lots of garbage code.

Several recent approaches [38, 37] propose to use x86 processor features to trap specific events for program analysis. In [38], the authors use branch tracing to record all the branches taken by the program during its execution. While the performance is much better than single-stepping, it is still 12 times slower than normal execution. Also, the tool is only able to record all branches. It cannot trap a specific branch, which renders detailed analysis at arbitrary given points during execution impossible. In [37], the authors make use of performance monitoring counters (PMCs) to trap certain types of instructions (e.g. call, ret and conditional branches). However, there are still many other types of instructions (e.g. mov) that could not be trapped this way. Also, the tool does not support trapping instruction at a specific location.

Hybrid Approaches. Researchers have also proposed to use hybrid approaches [22, 39] to take advantage of both the transparency granted by hardware virtualization and the flexibility provided by emulation. In [22], the authors utilize the trace obtained from a transparent reference system (e.g. Ether) to guide the execution of program in an emulator. However, as discussed above, it incurs high performance overhead to obtain execution trace using current hardware virtualization based approaches. V2E [39] takes another approach by emulating only the instructions that can be perfectly emulated. For other instructions in the program, it records the state changes caused by these instructions in a hardware virtualization based system, and then replays the state changes in the emulator. While this method could substantially reduce performance overhead, how to precisely identify the set of instructions that can be perfectly emulated remains a problem.

3. OVERVIEW

The goal of SPIDER is to provide a program debugging and instrumentation framework with flexibility, efficiency, transparency and reliability, which we define as follows:

(R1) **Flexibility:** SPIDER should be able to trap the execution of the target program at any desired instruction and data access at any memory address.

(R2) **Efficiency:** SPIDER should not introduce high performance overhead on the target program.

(R3) **Transparency:** The target program should not be able to detect the existence of SPIDER.

(R4) **Reliability:** The trap should not be bypassed or tampered with by the target program.

An overview of SPIDER is shown in Figure 1. For simplicity, we only show the trapping of instruction execution here. The trapping of data access using data watchpoint (Section 4.5) is much simpler and omitted in the figure. To trap the execution of an instruction, the user provides these inputs to SPIDER: the program address space identifier (CR3 register value in x86), the virtual address to set trap and the function to call on trap. As shown in the figure, SPIDER is mainly implemented inside the Hypervisor. The guest virtual-to-physical mapping monitor component (Section 4.3), which captures guest virtual-to-physical mapping changes, translates the address space identifier and the virtual address into guest physical address and invokes the breakpoint manager to set the trap. The breakpoint manager sets *invisible breakpoint* to trap the execution of the target program.

Invisible breakpoint uses the same triggering mechanism as traditional software breakpoint to inherit its flexibility (R1) and efficiency (R2). However, as discussed in Section 2, traditional software breakpoint is not transparent because: (1) The instructions needs to be modified in order to set breakpoint; (2) The triggering and handling of the breakpoint involves control-flow which is different from natural execution. These side-effects are neutralized in invisible breakpoint to guarantee transparency (R3). Regarding the first side-effect, the breakpoint manager uses EPT to split the code and data views (Section 4.1) of the guest physical page that contains the breakpoint. In the code view, which is used for instruction fetching (shown as the grey path in Figure 1), the instruction is modified to set breakpoint; in the data view, which is used for read/write access (shown as the white path in Figure 1), the instruction is not modified at all, so the guest sees no change to the instruction. To neutralize the second side-effect, when a breakpoint is triggered, the breakpoint manager will capture the event, call the corresponding user-provided function and handle the breakpoint transparently (Section 4.2) so that the control-flow in the guest is the same as a natural execution. The code modification handler (Section 4.4) captures any modification made to the data view and synchronizes with the code view to guarantee transparency (R3); it also makes sure the breakpoint is not maliciously overwritten by the guest to guarantee reliability (R4).

4. DESIGN

4.1 Splitting Code and Data View

SPIDER neutralizes memory side-effects of traditional software breakpoint by splitting the code and data views of guest pages. Several existing techniques could have been used here to split the two views; however, they all have some limitations. For example, one could intercept all read accesses to modified instructions by setting the corresponding pages to not-present, and return original instructions upon read accesses. However, it would introduce significant performance overhead as every instruction fetching or data access in these pages will cause a page fault. A recent work hvmHarvard [19] tries to implement a Harvard architecture on x86 by de-synchronizing the instruction TLB (iTLB) and the data TLB (dTLB). More specifically, it tries to maintain two different virtual-to-physical page mappings in iTLB and dTLB for the code and data view respectively. To prevent the mapping of the code view from being loaded into dTLB, the page table is set to map the data view all the time; the code view is only mapped when an instruction fetching happens, and a single-step is performed in the guest to load the code view into iTLB. Unfortunately, such mechanism could not guarantee the de-synchronization of iTLB and dTLB. As the code view is readable, one could still load the code view into dTLB by executing an instruction that reads from the page that contains it. An attacker could exploit this limitation to read and detect the modified instructions.

SPIDER splits the code and data views of a guest *physical* page by mapping it to two host *physical* pages with *mutually exclusive* attributes. We call such guest physical page with split code and data views a *split page*. The code view of a split page is executable but not readable; the data view is readable but not executable. Both views are set to not writable to handle code modification, which will be discussed in Section 4.4. The mutually exclusive attributes ensure that the guest could neither read from the code view nor execute instruction from the data view of a split page. Traditionally, in x86 there is no way to set a page to *executable but not readable*; however, recent processors introduces a feature that allows one to specify such attribute in EPT entries [21]. Legacy page table still lacks such capability, which is the reason we split physical pages instead of virtual pages.

SPIDER performs on-demand transparent switching between the two views of a split page. For example, let us assume its corresponding EPT entry is currently set to mapping its code view. When a data access happens in the page, since its current view—code view is not readable, an EPT violation will occur. SPIDER will capture the event and adjust the mapping and the attribute in the EPT entry to switch to the data view. It will then resume the guest, and the data access can proceed. Switching from data view to code view works in a similar way.

It seems that SPIDER needs to switch views frequently when instruction fetching and data access in a split page are interleaved, which could result in a lot of EPT violations. However, the problem is greatly mitigated by the separation of iTLB and dTLB in x86. Given a split page, although the corresponding EPT entry could only map one of its views at any given time, the mappings of the two views can exist simultaneously in the iTLB and dTLB, respectively. For example, when SPIDER switches the page from the code view to data view due to a data access, the mapping in the EPT

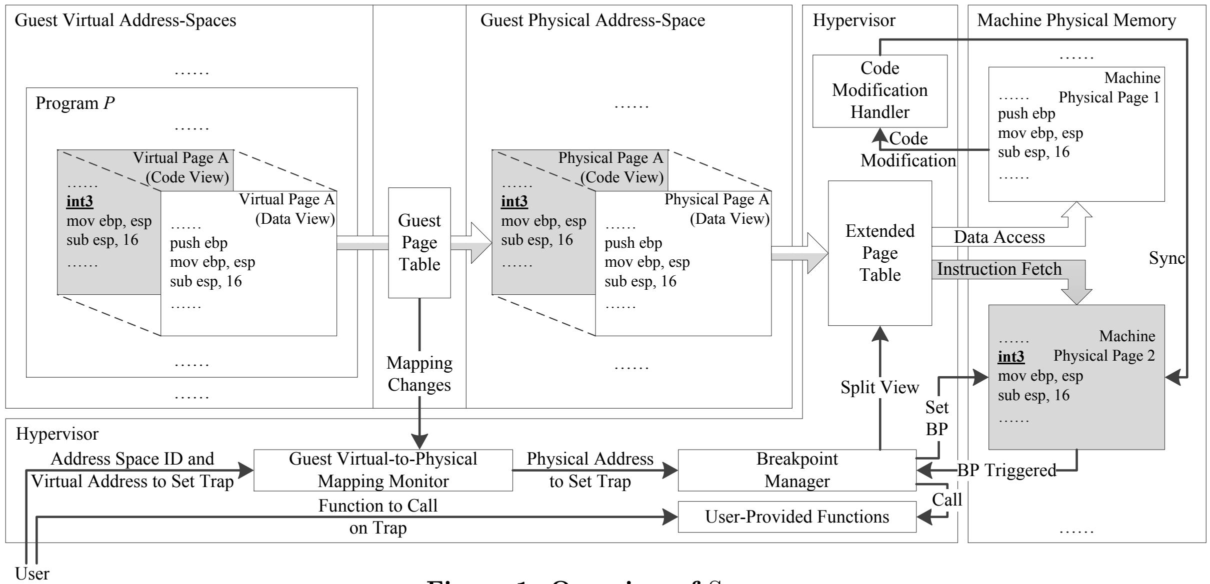


Figure 1: Overview of SPIDER.

is set to mapping its data view. After resuming the guest, the data access will populate the dTLB with the mapping for the data view. However, the mapping for its code view still exists in the iTLB. Further instruction fetching will not cause EPT violation until the mapping is evicted from iTLB.

4.2 Handling Breakpoints

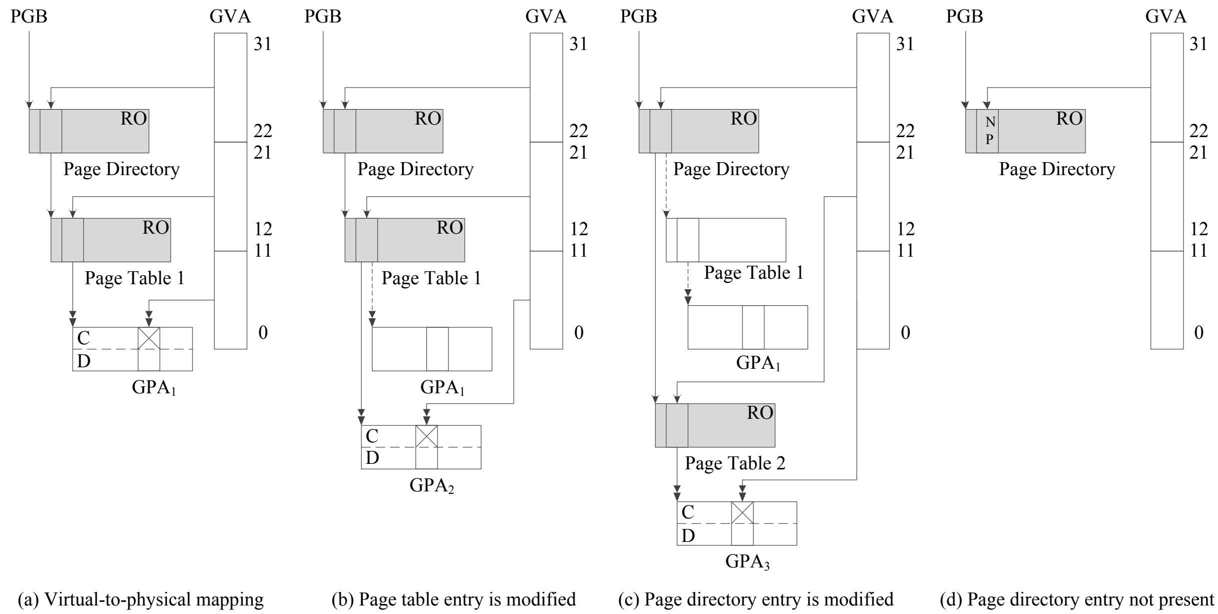
SPIDER hides the #BP exceptions generated by invisible breakpoints and invokes breakpoint handlers at the hypervisor level to neutralize side-effects related to breakpoint handling. SPIDER sets the hypervisor to intercept all #BP exceptions generated by the guest. How to deal with the #BP exceptions depends on their causes: those caused by invisible breakpoints should not be seen by the guest, while those caused by traditional software breakpoints set by the guest should be delivered to the guest transparently.

The breakpoint manager of SPIDER maintains a list which stores the *guest physical* addresses of all invisible breakpoints and their associated handlers that should be called when they are triggered. When SPIDER intercepts a #BP exception, it translates the guest instruction pointer to guest physical address by looking up the guest page table, and compares the address against the list to see whether the triggered breakpoint is an invisible breakpoint or a traditional software breakpoint. If it is a traditional breakpoint, the #BP exception will be re-injected to the guest to let the guest handle the breakpoint on its own. Otherwise, if it is an invisible breakpoint, SPIDER will call its associated handler to handle the breakpoint event. After that, SPIDER will temporarily clear the breakpoint and restore the first byte of instruction which had been replaced. Then it lets the guest single-step through the instruction.

Unlike previous work [14, 28, 19] which enables single-stepping by setting the trap flag in the guest *EFLAGS* register, SPIDER uses the monitor trap flag (MTF) which is a flag specifically designed for single-stepping in hardware virtualization. When MTF is set, the guest will trigger a VM Exit after executing each instruction. The reason why we choose not to use trap flag is that it is visible to the guest. Despite various techniques used in previous work to hide the trap flag, the guest could still see it. For example, if an interrupt is pending right after the guest resumes execution, the processor will invoke the corresponding interrupt handler before single-stepping through the instruction. The *EFLAGS* register is saved onto the stack, and restored after the interrupt handler returns. The interrupt handler could check the *EFLAGS* on the stack to see if the trap flag has been set. Compared with the trap flag, MTF is transparent because it could not be read by the guest. However, using MTF also causes one problem. Consider the same scenario of pending interrupt as above: when using the trap flag, the saving/restoring of the trap flag implicitly avoids single-stepping through the interrupt handler; but when using MTF, the processor will single-step through the interrupt handler before reaching the instruction. SPIDER solves this problem by "retrying": if it finds out that the guest has not executed the instruction after a single-step, it will clear MTF, set the invisible breakpoint again and resume the guest. The invisible breakpoint will be triggered again after the interrupt handler returns. This procedure repeats until the instruction is successfully executed after a single-step, and SPIDER will then clear MTF, set the invisible breakpoint again and resume the execution of the guest.

4.3 Monitoring Virtual-to-Physical Mapping

The invisible breakpoint provides SPIDER the ability to trap the execution of program at arbitrary *guest physical* address. However, since paging is enabled in almost all modern operating systems, the processor usually uses virtual address instead of physical address to reference memory. Hence, it is more desirable to have the ability to trap the execution of program at arbitrary *guest virtual* address in the program's address space. We define the *breakpoint address* where we set a breakpoint using a tuple of the address space identifier and the guest virtual address. In x86, the physical address of the base of the top-level paging structure (stored in CR3 register) serves as the address space identifier, so we write the breakpoint address as $BA = (PGB, GVA)$. If BA is mapped to a guest physical address GPA, we denote it as $BA \rightarrow GPA$. If BA is not mapped to any guest physical address, we denote it as $BA \rightarrow NIL$.



Read-only paging structure is in grey color and marked with "RO". Physical page with split code and data view is represented using rectangle with a dashed line splitting the code ('C') and data ('D') view. The area in the code view with a cross inside represents the breakpoint. Arrow means path that is traversed during address translation. Double arrow indicates memory mapping. Dashed arrow or double arrow means previous path or mapping that is no longer used.

Figure 2: Monitoring guest virtual-to-physical mapping.

To illustrate, assume the user wants to set a breakpoint at $BA_1 = (PGB_1, GVA_1)$. If $BA_1 \to GPA_1$, then we could just set an invisible breakpoint at GPA_1 to solve the problem. However, it is possible that $BA_1 \to NIL$ when we set the breakpoint (e.g., the program has not been loaded). Even if BA_1 is mapped, the mapping could change after the breakpoint is set. If the mapping changes to $BA_1 \to GPA_2$, since there is no breakpoint set at GPA_2, execution of the instruction at BA_1 will not be trapped as expected. Similarly, when BA_1 is no longer mapped to GPA_1, the breakpoint set at GPA_1 will cause problem when another address is mapped to GPA_1. Such virtual-to-physical mapping changes could happen for various reasons. For example, when the guest OS swaps out a virtual page, its corresponding physical page might be used to map another virtual page; when a write access happens in a copy-on-write virtual page, the guest OS will map it to another physical page to perform the writing; kernel-level malware could even modify the guest page table directly to change virtual-to-physical mappings. Hence, Spider must monitor virtual-to-physical mapping changes to handle such scenarios correctly.

Monitoring every change of virtual-to-physical mapping requires heavy-weight techniques such as shadow page table. Fortunately, Spider only needs to monitor the change of virtual-to-physical mapping at each breakpoint address. In x86, the virtual-to-physical mapping is represented using multiple levels of paging structures. The number of levels depends on the operation mode of the processor, for example, whether physical address extension (PAE) or long mode is enabled. Without loss of generality, let us assume that legacy two-level paging structure is being used. As shown in Figure 2(a), given a breakpoint address $BA = (PGB, GVA)$, the processor traverses along a path from the page directory to the page table to translate it to a guest physical address. The only way to change the virtual-to-physical mapping at BA is to modify the paging-structure entries that is traversed during address translation, which is shown as the rectangle area in the page directory and the page table. To capture such modifications, Spider sets these paging structures to read-only (shown as grey) in the EPT. When there is a write access to a paging structure, an EPT violation will be triggered and captured by Spider. Spider will record the current values of paging-structure entries, then temporarily set the paging structure to writable and let the guest single-step through the instruction that performs the write access. After the single-stepping, Spider will read the new values of paging-structure entries and see which ones of them have been modified. After that, Spider will set the paging structure back to read-only to capture future modifications. The action that Spider performs to handle the modification depends on the type of the paging-structure entries that get modified:

Bottom-level paging-structure entries. As shown in Figure 2(b), when the bottom-level paging-structure entry used to translate BA is modified, the mapping changes from $BA \to GPA_1$ to $BA \to GPA_2$. As a result, Spider first removes the invisible breakpoint at GPA_1. Then Spider compares the content of the page that contains GPA_1 and the page that contains GPA_2. If they are exactly the same (which is the case we show in the figure), then it is safe to move the breakpoint to GPA_2. Otherwise, as the code in the page has changed, it is handled in the same way as a breakpoint that might no longer be valid due to code modification (Section 4.4).

It is worth noting that the figure only shows the scenario where the mapping changes from a present one to another present one. The mapping might also changes from not-present to present, or oppositely. If the mapping changes from $BA \to NIL$ to $BA \to GPA$, or from $BA \to GPA$ to $BA \to NIL$, Spider will create/remove invisible breakpoint at GPA, respectively.

Non-bottom-level paging-structure entries. Figure 2(c) shows the scenario when a non-bottom-level paging-structure entry used to translate BA is modified. The virtual-to-physical mapping changes from $BA \rightarrow GPA_1$ to $BA \rightarrow GPA_3$, so SPIDER moves the breakpoint from GPA_1 to GPA_3. In addition to that, the path which the processor traverses along to perform address translation is also modified, so SPIDER also removes the read-only attribute from the paging structures in the previous path (Page Table 1) and sets the paging structures in the new path (Page Table 2) to read-only. For simplicity, we only show the change of one mapping and one path in Figure 2(c). In practice, modification of a non-bottom-level paging-structure entry may affect multiple paths and mappings, each of which will be handled by SPIDER individually.

There is a special case that the path used for address translation is incomplete because a non-bottom-level paging-structure entry is set to non-present, as shown in Figure 2(d). This could happen when setting a breakpoint at a virtual address that is not mapped in the guest, or after a non-bottom-level paging-structure entry is modified. SPIDER sets the paging structures along the path to read-only, including the one that has the non-present entry. Later, when the paging-structure entry changes from non-present to present, the path will extend, and SPIDER will set the paging structures on the extended path to read-only. After the path reaches the bottom-level paging-structure (e.g. as in Figure 2(a)), SPIDER could handle further modifications using standard approaches as mentioned above.

4.4 Handling Code Modification

When the guest tries to modify the content of a split page, the write operation will be performed on its data view. This means that if an instruction is modified, the change will not be reflected in the code view. This could lead to incorrect execution of *self-modifying programs*, and could be utilized by malware to detect the existence of SPIDER. To guarantee transparency, SPIDER must synchronize any change of the data view to the code view.

As mentioned in Section 4.1, SPIDER sets the data view of a split page to read-only in EPT to intercept any writing attempt. When the guest tries to write to the page, an EPT violation will be triggered and captured. SPIDER records the offset of the data OFF that is going to be written in the page. SPIDER also records the length LEN that will be synchronized by matching the instruction's op-code in a pre-built table which stores the maximum data length that could be affected by each type of instruction. Then SPIDER will temporarily set the data view to writable, and let the guest single-step through the instruction that performs the write. After that, it will copy LEN bytes from offset OFF in the data view to the same offset in the code view.

It is worth noting that the breakpoints that have been set in the page may or may not be valid after code modification. For example, if the guest overwrites an instruction with the same instruction, it indicates the guest is trying to overwrite and disable the breakpoint set at that instruction; in that case, the breakpoint is still valid and should be re-set when overwritten. But if the guest overwrites the instruction with a different instruction, re-setting breakpoint at the original place blindly may not make sense. Hence, we allow the user to specify a function which will be invoked when the page that contains the breakpoint is being modified, in which the user could perform proper actions to handle the event, such as re-setting the breakpoint at the same place, or moving it to another location after analyzing the modified code.

4.5 Data Watchpoint

SPIDER allows setting a data watchpoint at a specific *physical* address by adjusting the EPT entry of the guest physical page that contains the memory address to read-only (to trap write access) or execute-only (to trap both read/write access). When the page is accessed, an EPT violation will be triggered and captured by SPIDER. SPIDER will check if a watchpoint has been set on the address that is accessed in the page; if so, it will call the corresponding user-provided watchpoint handler. After that, it will temporarily set the EPT entry to writable and resume the guest to single-step through the instruction that does the memory access. When the guest returns from single-stepping, SPIDER adjusts the EPT entry again to trap future accesses. Like invisible breakpoint, data watchpoint also utilizes the virtual-to-physical mapping monitoring method (Section 4.3) so that it could be used to trap memory access at any *virtual* address.

4.6 Handling Timing Side-Effect

In hardware virtualization, since part of the CPU time is taken by hypervisor and VMEntry/VMExit, a program costs more time to run than in a native environment. Attackers could execute the RDTSC instruction to read the Time Stamp Counter (TSC) which stores the elapsed CPU cycles to detect the discrepancy. To maintain transparency, SPIDER needs to hide the CPU cycles cost by hypervisor (T_h) and VMEntry/VMExit (T_e) from the guest. SPIDER measures T_h by reading the TSC right after each VMExit and right before each VMEntry and calculating the difference. T_e is approximated by profiling a loop of RDTSC instruction in guest. SPIDER sets the *TSC-offset* field in virtual machine control structure (VMCS) to $-(T_h + T_e)$ so the value is subtracted from the TSC seen by the guest [13].

5. IMPLEMENTATION

We have implemented a prototype of SPIDER on the KVM 3.5 hypervisor. The prototype implements the design as described in Section 4 in the kernel module part of KVM (kvm-kmod) to provide the primitive of setting invisible breakpoint at specified virtual address in a process address space. Based on the primitive, it also implements a front-end for SPIDER in the userspace part of KVM (qemu-kvm) to provide features that make debugging and instrumentation more convenient. It is worth noting that SPIDER itself is OS-independent; However, the front-end requires knowledge of the guest OS to perform VMI [18] for some features. Currently, our front-end supports both Windows XP SP2 32-bit and Ubuntu Linux 12.04 32-bit guest. We now discuss the implementation of some features in our front-end.

Kernel Breakpoints. We have to specify an address space when setting an invisible breakpoint. For kernel breakpoints, we could specify the address space of any process as the kernel space is mapped in the same way for any process. We hence choose the address space of a long-lasting process (*init* in Linux and *System* in Windows), so the breakpoint will not be cleared due to process termination.

Monitor Process Creation. In practice, in addition to debugging running programs, it is also desirable to have the ability to get the control of a program at the moment when it is just started. For example, when analyzing malware, users often need to trap the execution at its entry point; if the malware is already running, it would be too late to set breakpoints. To meet such requirement, our front-end monitors process creation events. We set invisible breakpoints at related kernel functions to capture a newly created process and match its name against the one specified by the user. The user could get notified as soon as a process of the target program is created, and perform corresponding actions such as setting an invisible breakpoint at the entry point.

In Windows, a process is created through the *NtCreateProcessEx*[1] system call, which calls the *PspCreateProcess* kernel function to do the actual work. We set a breakpoint at the instruction right after the call to *PspCreateProcess*. When the breakpoint is triggered, we walk through the active process list at *PsActiveProcessHead* to find out the *EPROCESS* of the newly created process. The name is stored in its *ImageFileName* field.

In Linux, there are two system calls *fork* and *clone* that could be used to create a new process. They both call the same function *copy_process* to do the actual work, so we set a breakpoint at the instruction right after the call. When the breakpoint is triggered, the *task_struct* of the newly created task is in the *EAX* register as the return value. As *clone* could also be called to create thread, we need to verify the newly created task is a process by making sure its address space identifier (stored in *task_struct.mm->pgd*) is different from the one of the *current* task. The name is stored in the *task_struct.comm* field.

Monitor Process Termination. When a process terminates, all invisible breakpoints in its address space should be cleared. We set invisible breakpoints at related kernel functions to monitor process termination. When a terminating process is captured, we use its address space identifier to check if it is one of our debuggee targets. If so, we will remove the target and clear all invisible breakpoints in it.

In Windows, we set the breakpoint at the entry of the function *PspProcessDelete*, which handles cleanup when a process terminates. When the breakpoint is triggered, we read the first argument of the function from the stack, which is the *EPROCESS* structure of the process. The address space identifier is in its *Pcb.DirectoryTableBase* field.

In Linux, we set the breakpoint at the entry of the function *do_exit*, which handles the termination of the *current* task. However, the task could be a process or thread. We determine if the task is a process by checking if the *task_struct.pid* field matches the *task_struct.tgid* field. The address space identifier is read from the *task_struct.mm->pgd* field.

The system call *execve* in Linux requires special handling. Although it does not create or terminate a process, it changes the program running in the *current* task. We consider that both process "termination" and "creation" are involved in this procedure: the *current* task which runs the previous program is "terminated", and one that loads the new program is "created". As *execve* calls *do_execve* to do the actual work, we set a breakpoint right before the function call to capture the "terminated" *current* task, and another breakpoint right after the call to capture the "created" one.

[1]Another system call *NtCreateProcess* for process creation is a wrapper of *NtCreateProcessEx*.

6. EVALUATION

In this section, we present the evaluation of SPIDER. The experiments are done on a Thinkpad T510 laptop with Intel Core i7-3720QM 2.6GHz CPU and 8GB RAM. The host OS is Ubuntu Linux 12.10 64-bit. We use Windows XP SP2 32-bit and Ubuntu Linux 12.04 32-bit as the guest OS. We allocate 30GB hard disk and 1GB memory for the guest VM.

6.1 Transparency

Two groups of Windows programs with anti-debugging and anti-instrumentation techniques are used to evaluate the transparency of SPIDER. For comparison, we use SPIDER, two debuggers (OllyDbg and IDA Pro) and two DBI frameworks (DynamoRIO and PIN) to trap the execution of these programs at certain locations. In SPIDER, the trapping is done by setting invisible breakpoints. In the debuggers, we use software or hardware breakpoints. The DBI frameworks insert instrumentations at desired instructions for trapping.

The first group of targets consists of 7 software protectors, which are widely used by both COTS software vendors and malware authors to protect their programs from being analyzed or modified. We apply these software protectors to a system program *hostname.exe* in Window XP SP2. This program reads and displays the host name of the local system; our goal is to trap the execution of its protected versions to get the host name string. We reverse-engineer the original program and find out the address of the host name string is store in the *eax* register when the program runs to the address *0x10011C6*. This also holds in the protected versions, as this program does not contain relocation information and could not be relocated by the protectors. Hence, we set the traps at *0x10011C6* in the protected versions. However, for some of the protectors, we could not set the trap when the program starts, as the instruction at *0x10011C6* is encrypted by the protectors and has not been decrypted at that time. We hence set a data watchpoint at *0x10011C6* to monitor the decryption, and set the trap once the instruction is decrypted.

We turn on all anti-debugging, anti-instrumentation and anti-VM options of the protectors when using them. The only exception is when we use Safengine Shielden, we turn off its anti-VM option. With that option on, we found that the program protected by Safengine Shielden would cease to function even when we run it in vanilla KVM *without* SPIDER; but it runs correctly in BitVisor, which is another hardware virtualization based hypervisor. We hence conclude that the problem is due to the implementation of KVM but not SPIDER.

The second group of targets includes 8 proof-of-concept (POC) samples. Among these programs, eXait [30] aims at detecting DBI frameworks. We randomly select 10 instructions in it for trapping. The rest 7 samples implement the anti-debugging techniques commonly used in malware that is not protected by protectors, according to the statistics in [9]. Since these samples are very small (tens of instructions), we choose to trap every instruction in them.

The result is shown in Table 1. "Pass" indicates the program runs properly and its execution is successfully trapped at the desired location. "Fail" means the program fails to run properly in the environment even without any trap. "Fail HBP" and "Fail SBP" means the program fails to run properly after setting hardware breakpoint or software breakpoint. We can see that OllyDbg and IDA Pro fail at every

Target	Spider	OllyDbg 1.10	IDA Pro 6.1	DynamoRIO 4.0.1-1	PIN 2.12
Software Protectors (Applied to *hostname.exe*)					
Safengine Shielden 2.1.9.0	Pass	Fail	Fail	Fail	Fail
Themida 2.1.2.0	Pass	Fail	Fail	Pass	Pass
PECompact 3.02.1 (w/ead loader)	Pass	Fail	Fail	Pass	Pass
ASProtect 1.5	Pass	Fail	Fail	Pass	Pass
RLPack 1.21	Pass	Fail	Fail	Pass	Pass
Armadillo 9.60	Pass	Fail	Fail	Fail	Pass
tElock 0.98	Pass	Fail	Fail HBP/SBP	Fail	Fail
Anti-debugging & Anti-instrumentation POC Samples					
eXait	Pass	Pass	Pass	Fail	Fail
hardware_bp.exe	Pass	Fail HBP	Fail	Pass	Pass
heapflags.exe	Pass	Fail	Fail	Pass	Pass
instruction_counting.exe	Pass	Fail HBP	Fail HBP	Fail	Fail
ntglobal.exe	Pass	Fail	Fail	Pass	Pass
peb.exe	Pass	Fail	Fail	Pass	Pass
rdtsc.exe	Pass	Fail HBP/SBP	Fail HBP/SBP	Pass	Pass
software_bp.exe	Pass	Fail SBP	Fail SBP	Pass	Pass

Table 1: Transparency evaluation result of Spider and other debuggers/DBI frameworks.

target except eXait; most targets could detect their existence even when no trap is set. DynamoRIO and PIN perform better, but are still detected by 5 and 4 targets, respectively. Compared with them, Spider successfully maintains transparency against all 15 targets; there are 3 targets that could only be transparently trapped by Spider.

We also test Spider against techniques of detecting emulators in [16, 17, 29], which we implement as individual POC programs. We run them in Spider and trap every instruction in these programs as they are very short. As we expected, none of them is able to detect Spider, as Spider is built upon hardware virtualization.

6.2 Case Study I: Attack Provenance

In this case study, we demonstrate the use of Spider to improve the tamper-resistance of an existing attack provenance system BEEP [25]. Traditional attack provenance approaches are based on analysis of system event log with per-process granularity (i.e., each log entry pertains to one process). Such approaches face the problem of dependency explosion when a long running process receives/produces a lot of inputs/outputs during its lifetime as each output is considered causally related to all preceding inputs. To solve this problem, BEEP partitions the execution of a program into individual *units*, with each unit handling an independent input request (e.g., one email or one web request) in one event-handling loop iteration. With such a finer logging granularity, BEEP is able to link each output to the truly related input(s) hence achieving higher attack provenance accuracy.

To capture the entry and exit of each unit, BEEP needs to instrument the target binary program at certain locations. BEEP uses a static binary rewriting tool PEBIL [23] to perform such instrumentation, which has several shortcomings: (1) Attackers could patch the instrumented program at runtime to disable BEEP; (2) The instrumentation needs to modify the code in the program, hence cannot be applied to programs with self-checking and self-protection mechanisms, which widely exist in COTS software to prevent malicious software manipulation. To overcome these problems, we use Spider to replace PEBIL for BEEP's instrumentation. The reliability of Spider (Section 3) guarantees that the instrumentation could not be circumvented or disabled. More importantly, Spider performs instrumentation by setting invisible breakpoints, which are transparent to the target applications.

We evaluate the effectiveness and performance of our approach using 7 Linux[2] binary programs. We first identify the instrumentation points for each program using BEEP. We then set Spider to monitor the creation of processes of these programs. Once a process of a target program is created, we set invisible breakpoints at the instrumentation points in its address space. The original instrumentation routines in BEEP invoke a special system call to log unit-specific events; we modify them to directly log unit events into a file in the host.

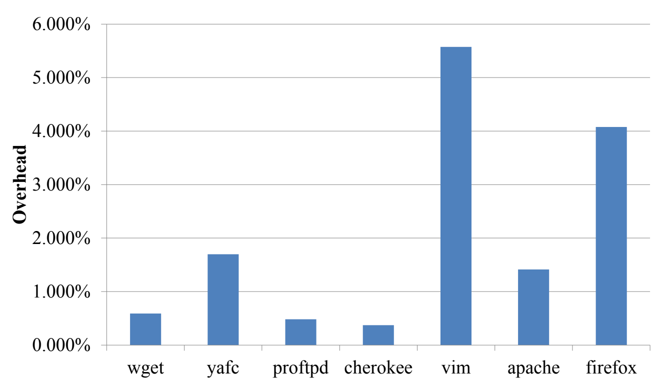


Figure 3: Overhead of using Spider to perform instrumentation for BEEP.

We repeated the case studies in [25] and verified the correctness of attack provenance achieved by our system. We also measure the overhead of our system over the execution of the programs in vanilla KVM. In vanilla KVM we enable Linux audit system but do not perform instrumentation. For *wget* and *yafc*, we run them to download a 1.2MB file from a server 500 times. For *apache* and *cherokee*, we use the *weighttp* to generate 1 million requests with 100 threads and 100 concurrency. For *proftpd*, we use the integration test provided with it. We use the *SunSpider* benchmark for *firefox*. For *vim*, we feed it a script to replace the first letter of each line with 'a' in 50000 text files. All network programs

[2]The prototype of BEEP only supports Linux currently.

except *firefox* are evaluated in a dedicated LAN to rule out the factor of network delay. The result is shown in Figure 3. The overhead is less than 2% except *firefox* and *vim*. The overhead for *firefox* is slightly higher because it has more instrumentation points (24) than other programs (2∼6), which leads to more breakpoint hits. The overhead for *vim* is due to an instrumentation point which gets triggered each time the script processes a line. Users will experience much less overhead when they use *vim* interactively as the instrumentation point is triggered much less frequently.

6.3 Case Study II: Stealthy Introspection

We now demonstrate the use of Spider to reveal a possible threat to two popular Windows instant messaging programs, anonymized as IM_1 and IM_2. The threat involves the acquisition of confidential application data without user awareness. Such data usually have very short lifetime in memory and are encrypted before network transmission. Hence they are deemed difficult/impossible to acquire through memory scanning or network sniffing. We also protect the two applications using the (arguably) strongest protector Safengine Shielden, so that existing debugging/instrumentation techniques cannot be used to analyze them. Now, we show that even with those protections, confidential data could still be "stolen" by using Spider to trap the program at the right instruction. The stealthiness and efficiency of Spider make it possible to perform the attack while the programs are running normally; none of the existing techniques could achieve the same level of user-transparency and efficiency. The realism of the threat is backed by the fact that, an attacker is able to transparently hijack a running OS into a VM on malicious hypervisor (e.g., using BluePill [31]). Once that happens, Spider can be used to stealthily set invisible breakpoints on the target application for confidential data acquisition by the hypervisor. In the following description, such breakpoints are set on the functions and memory locations in bold font.

$\mathbf{IM_1}$. We show the possibility of capturing all communication between a sender and the user. To find the function that handles messages, we search through the functions exported by the libraries of IM_1. We find a function named **SaveMsg**[3] in KernelUtil.dll and set an invisible breakpoint at the entry of that function. As expected, the function is called every time a message is received; we also find out one of its parameters is the ID of the sender. However, the message text is not directly present in the argument list, which implies that it might be part of a data structure rooted at one of the arguments. We further speculate that a message may need to be decoded either inside **SaveMsg** or through some other related function. We find a function named **GetMsgAbstract** in the list of exported functions. The name suggests that it may need to decode a message. We set a breakpoint at its entry and another one at its return. We observe that the message text is in fact decoded as its return value. We also find out that at the entry of **GetMsgAbstract** that the value of one of its parameters is always the same as one of the parameters of **SaveMsg**, which might both point to the same opaque structure that contains the message text. Therefore, we log all messages at **GetMsgAbstract** return and associate them to individual senders by matching the parameters of **GetMsgAbstract** and **SaveMsg**. As such, we are able to identify all messages from individual senders.

$\mathbf{IM_2}$. We show the possibility of capturing user login credentials in IM_2. We first find the functions that read the username and password. As a native Win32 application, we suspect it uses the **GetWindowTextW** Windows API function to retrieve the text from the controls in the login dialog. We set a breakpoint at the entry of that function and log all its invocations. After we rule out unrelated invocations by checking if the retrieved text matches a login credential, we find out the invocations at **0x449dbd** and **0x437a23** are for retrieving username and password, respectively. The remaining problem is to find out if the captured login credential is valid. As an error message will be displayed upon failed login, we set a breakpoint at the **MessageBoxW** function. From the call stack we could read the functions on the path of failed login. We set breakpoints on these functions too. We then do a successful login to see if it shares the same path. We find that both successful and failed logins will execute to the function at **0x48591c**, and then the path deviates. Successful login will execute to the branch of **0x485bcd**, while failed login leads to another branch. Therefore, we log the content acquired by **GetWindowTextW** when it is invoked at **0x449dbd** and **0x437a23**, and then we use the call stack path to prune those belonging to failed logins.

We verified that the confidential data (messages or login credentials) is correctly and completely acquired through stealthy introspection, without any slow-down of program execution.

6.4 Performance Overhead

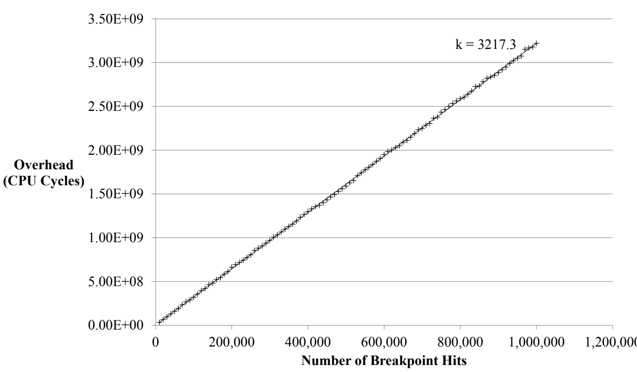


Figure 4: Relation between the overhead of Spider and the number of breakpoint hits.

We have already presented the empirical overhead of Spider in our case studies in Section 6.2. In this experiment, we further study the overhead of Spider. We build a micro benchmark program that executes a loop for a given number of times. In each loop iteration, the program increments a variable 1000 times. The program executes the *RDTSC* instruction to read the CPU cycle counter before and after the loop, and calculate the difference which is the number of CPU cycles cost by the loop. We compile the program with Visual Studio 2010 in Windows.

We run the program using the parameter from 10^4 to 10^6 iterations, with a step of 10^4. The program is executed in both vanilla KVM and Spider; In Spider, we set an invisible breakpoint at the first instruction of the loop. We obtain

[3]Note that the binary of IM_1 does not contain symbolic information. We simply inspect the export table.

the number of CPU cycles cost by the loop in vanilla KVM and SPIDER, and the difference is the overhead, as shown in Figure 4. From the figure, we could see that the overhead is linear to the number of breakpoint hits. A single invisible breakpoint hit costs around 3217 CPU cycles. A large part of the overhead is due to the transitions between host and guest during breakpoint handling. A round-trip transition costs about 1200 cycles (measured using kvm-unit-test). This is the cost we have to pay to maximize stealthiness: To prevent any in-guest side effect, the breakpoint handler must run outside the guest VM, which means the transition is inevitable. Nevertheless, the overhead of our invisible breakpoint is still less than the breakpoint in an existing work [36] and comparable with in-guest hardware breakpoint. Considering that the cost of VMExit/VMEntry is decreasing over the years [4], the overhead of our approach is likely to be less in future processors.

We also measure the overhead of other components in SPIDER, including the cost of splitting code and data views and monitoring the guest virtual-to-physical mapping. We exclude the overhead of breakpoint hits by setting "fake" breakpoints, which use the original instruction as the breakpoint instruction instead of *int3*. The target program we use is gzip 1.2.4. We run the program in both vanilla KVM and SPIDER to compress a 98.7MB file and measure the execution time. In SPIDER, we set a breakpoint at one instruction in each page of the code section to make sure all code pages are split. The run in vanilla KVM costs 4171ms, while the run in SPIDER costs 4192ms. The overhead is less than 1% which confirms that the number of breakpoint hits is the dominant factor of overhead.

7. CONCLUSION

In this paper, we present SPIDER, a stealthy binary program instrumentation and debugging framework. SPIDER uses invisible breakpoint, a novel primitive to trap execution of program at any desired instruction efficiently. Our evaluation shows SPIDER is transparent against various anti-debugging and anti-instrumentation techniques. We have applied SPIDER in two security application scenarios, demonstrating its transparency, efficiency and flexibility.

Acknowledgements

This research has been supported in part by DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA.

8. REFERENCES

[1] Gdb. `http://www.gnu.org/software/gdb/`.

[2] Ida pro. `http://www.hex-rays.com/idapro/`.

[3] Kvm. `http://www.linux-kvm.org/`.

[4] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX ATC'12*.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SOSP'03*.

[6] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *EICAR'06*.

[7] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC'05*.

[8] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE'06*.

[9] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Blackhat USA'12.

[10] D. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004.

[11] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *VEE'12*.

[12] P. P. Bungale and C.-K. Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *VEE'07*.

[13] Z. Deng, D. Xu, X. Zhang, and X. Jiang. Introlib: Efficient and transparent library call introspection for malware forensics. In *DFRWS'12*.

[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS'08*.

[15] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *ASPLOS'12*.

[16] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, 2006.

[17] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[18] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS'03*.

[19] M. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh. Transparent protection of commodity os kernels using hardware virtualization. In *SecureComm'10*.

[20] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *RAID'08*.

[21] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3C.

[22] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *VMSec'09*.

[23] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS'10*.

[24] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996.

[25] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS'13*.

[26] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*.

[27] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation.

[28] A. Nguyen, N. Schear, H. Jung, A. Godiyal, S. King, and H. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *ACSAC'09*.

[29] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *ISC'07*.

[30] N. Riva and F. Falcón. Dynamic binary instrumentation frameworks: I know you're there spying on me. REcon'12.

[31] J. Rutkowska. Subverting vista kernel for fun and profit. Blackhat USA'06.

[32] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO'03*.

[33] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *ICISS'08*.

[34] A. Vasudevan. Re-inforced stealth breakpoints. In *CRiSIS'09*.

[35] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *IEEE S&P'06*.

[36] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *ACSAC'05*.

[37] S. Vogl and C. Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *EuroSec'12*.

[38] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan. Down to the bare metal: Using processor features for binary analysis. In *ACSAC'12*.

[39] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *VEE'12*.

[40] O. Yuschuk. Ollydbg. `http://www.ollydbg.de/`.