EagleEye: Towards Mandatory Security Monitoring in Virtualized Datacenter Environment

Yu-Sung Wu*, Pei-Keng Sun, Chun-Chi Huang, Sung-Jer Lu, Syu-Fang Lai and Yi-Yung Chen
Department of Computer Science
National Chiao Tung University, Taiwan
hankwu@g2.nctu.edu.tw, bacon.cs99g@nctu.edu.tw, eggwater.cs99g@nctu.edu.tw,
tracyttin.cs00g@nctu.edu.tw, blackxwhite@gmail.com, cheniy.cs97@nctu.edu.tw

Virtualized datacenter (VDC) has become a popular approach to large-scale system consolidation and the enabling technology for infrastructure-as-a-service cloud computing. The consolidation inevitably aggregates the security threats once faced by individual systems towards a VDC, and a VDC operator should remain vigilant of the threats at all times. We envision the need for on-demand mandatory security monitoring of critical guest systems as a means to track and deter security threats that could jeopardize the operation of a VDC. Unfortunately, existing VDC security monitoring mechanisms all require pre-installed guest components to operate. The security monitoring would either be up to the discretion of individual tenants or require costly direct management of guest systems by the VDC operator. We propose the *EagleEye* approach for on-demand mandatory security monitoring in VDC environment, which does not depend on pre-installed guest components. We implement a prototype on-access anti-virus monitor to demonstrate the feasibility of the *EagleEye* approach. We also identify challenges particular to this approach, and provide a set of solutions meant to strengthen future research in this area.

I. Introduction

Virtualization is a generic approach to achieve system-level consolidation in datacenter environments. It brings together systems running diverse applications and transforms a datacenter into a so-called virtualized datacenter (VDC). A VDC naturally inherits all the security threats faced by each of the hosted systems. In addition, the diverse composition of systems in a VDC implies a high likelihood of inconsistent and/or ineffective security policy implementation, which makes it difficult to ascertain if a given security threat has indeed be ruled out per the built-in security isolation mechanism at the VDC infrastructure layer. It is therefore important that a VDC operator should remain vigilant of the security threats at all times and have the ability to apply security monitoring on critical systems in the environment as a means to track and deter the threats that could jeopardize the operation of the VDC.

Security monitoring in VDC environment can be intuitively implemented through deploying security monitors such as anti-virus scanners within each virtual machine (VM) (also referred to as a guest system) hosted by the VDC. However, with thousands or even more number of customized VMs in a VDC [1], it will be a quite expensive process for a VDC operator to deploy and manage security monitors in each of the VMs. In addition, VMs in a large-scale VDC are often managed by individual tenants and not by the datacenter operator. One will have to rely on individual tenants to deploy and manage the security monitors in their respective VMs. Obviously, this approach is problematic since a negligent tenant can inadvertently disable the security monitor, and a malicious tenant may even attempt to tamper with the security monitor.

Motivated by the above difficulties, we propose the *EagleEye* mandatory security monitoring approach for VDC environment. In the approach, security monitors are placed externally to the guest VMs. There is no requirement for installing guest components in the VMs. It requires no attention or cooperation from the VM tenants. The approach also allows automated deployment and management of security monitors in a VDC environment. To demonstrate the feasibility of the proposed approach, we built a prototype on-access malware detection system for guest VMs in a VDC.

Contributions. We propose the *EagleEye* approach for mandatory security monitoring in VDC environment. The approach requires neither modification to guest system nor cooperation from the VM tenants. The *EagleEye* prototype system is the first system to demonstrate the feasibility of on-demand mandatory security monitoring for VDC environment with respect to a real-world security monitoring application. The approach depends on novel techniques for achieving transparent guest system event interception, resolving inconsistent guest states during synchronous security monitoring, bridging the semantic gap across complex black-box guest system models, and reducing the performance overhead of blocking-wait in the synchronous monitoring mode.

II. Background

Due to the aggregation of systems and its abundant computing resources, a VDC is both a conspicuous target for security attacks and a powerful platform for carrying out attacks. Real-world incidents such as Amazon EC2 being leveraged for running Zeus botnet [2] and fueling the attack against Sony PlayStation Network [3] are examples of why security monitoring for VDC environment is an important issue to look at.

The conventional approach for security monitoring in VDC environment is through installing security monitors on each VM as shown in Figure 1 (a). Installing the same set of security monitors on every VM is no doubt a waste of storage space. It can also lead to unwanted resource contention (e.g. the anti-virus storm effect [4]). And, if the VMs are not managed by the VDC operator, it will also require cooperation of the tenants for the deployment and management of the security monitors.

The para-virtualization approach (Figure 1 (b)) aims at consolidating security monitors into a dedicated VM (often referred to as the security VM [5, 6]). Only a tiny guest component (e.g. a light-weight agent program or a driver) is required to be pre-installed in each VM. The guest component can attach hooks in the guest system to intercept system events and leverage the guest APIs to inspect the system states. The guest event and state information will then be forwarded to the external monitor for analysis and attack detection. The para-virtualization approach allows VMs on a hypervisor to share the same security monitor backend thereby avoiding overlapping resource usages. The approach has been adopted in commercial datacenter security solutions such as VMware VMsafe [7], McAfee MOVE [8] , and TrendMicro Deep Security [9].

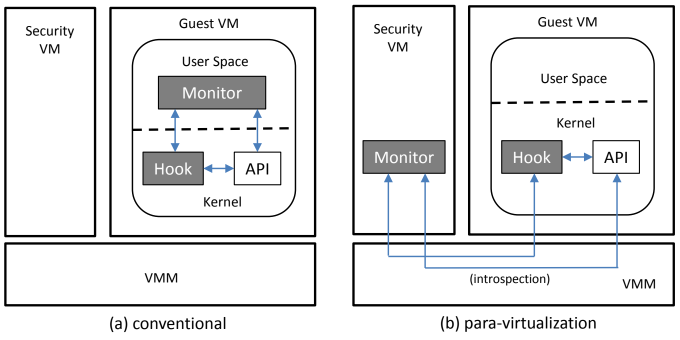


Figure 1. Security Monitoring in VDC

III. The EagleEye Approach

Both the conventional approach and the para-virtualization approach require modification to the guest VMs and, by extension, would depend on the cooperation of VM tenants. For a sizable VDC, both requirements are difficult to achieve. The outcome will be an incomplete security monitoring infrastructure by design.

In view of the difficulties with the existing approaches, we propose the *EagleEye* approach of mandatory security monitoring for VDC environment. We set the following goals in the design of *EagleEye*:

1. Security monitoring should not depend on pre-installed guest components
2. Isolation of security monitor from the guest VMs
3. Applicable to real-world security monitoring applications
4. Synchronous response to security threats

Following the first and the second requirements, the architecture of *EagleEye* is illustrated in Figure 2. The architecture consists of two key components. The first key component is the *EagleEye* hypervisor module (E^2D) for intercepting code execution and memory events and for the introspection of guest memory. The other key component is the *EagleEye* daemon (E^2D) running in the security domain. The daemon exposes an interface to extensible detection engine modules, which implement respective security monitoring logics such as malware detection, network intrusion detection, and etc. On the other hand, E^2D will interact with E^2H to carry out the monitoring of guest events and states. The stealthy hooks (SH) will be installed at runtime, and the idle loops are a optional component used for improving performance (Sec. III.F).

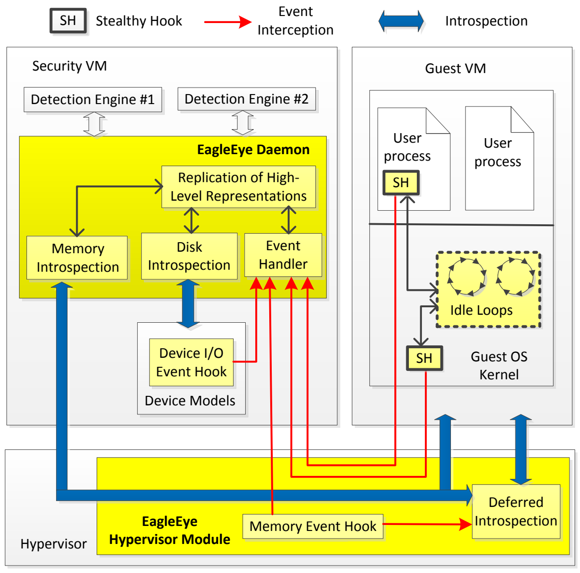


Figure 2. EagleEye Architecture

Following the third requirement, *EagleEye* has to provide the information needed by the detection engines. Security monitoring applications generally operate on the basis of detecting malicious or anomalous patterns in the system states as indications of security threats. The system states can be boiled down to memory state, disk state, and CPU state. *EagleEye* leverages existing VM introspection techniques [6, 10] to extract CPU and memory states of a VM. And, through a modified QEMU device model module, *EagleEye* also has access to a VM's disk content down to the block level on the Xen virtualization platform. However, both the memory data and disk contents can be inconsistent at the time of VM introspection. *EagleEye* employs two novel techniques to address the inconsistency in memory data (Sec. III.D) and the inconsistency in disk content (Sec. III.E) respectively.

For the fourth requirement, *EagleEye* supports synchronous inspection of guest states by allowing execution hooks to be placed in both the guest kernel and the guest user process text segments. The hook does not depend on pre-installed guest components and are hidden from the guest system (Sec. III.C).

A. Threat Model

EagleEye targets the threats posed by the inability to implement mandatory security monitoring for guest VMs in an VDC environment. The threats are real because the VM tenants are not necessarily security-conscious. And, a security-conscious tenant can make mistakes (e.g. forgot to re-enable a monitor driver after a system upgrade). It is also likely that a tenant may be non-cooperating or be malicious.

EagleEye is designed to be operated by datacenter operators for the purpose of monitoring security threats in the guest VMs.

As such, we assume that the datacenter operators are trusted. A malicious datacenter operator can possibly use *EagleEye* to spy on the guest VMs.

EagleEye runs in dom0 and the hypervisor layer. We assume both the management VM (dom0) and the hypervisor are trusted. One may leverage work such as Hypersafe [11] that enforces hypervisor control-flow integrity and work such as CloudVisor [12] that reduces the TCB of a hypervisor to strengthen the assumption.

EagleEye, by itself, does not decide whether a security threat is present or not. The decision is made by the attached detection engines based on the guest system states and events supplied by *EagleEye*. Evasion attacks targeting blind spots of a detection engine will also succeed under *EagleEye*. For instance, if a security threat uses a non-standard system call invocation that the detection engine is unaware of, then that threat will evade the detection under *EagleEye* as well. It is up to the detection engine to set up extra event interception points with *EagleEye* to track the non-standard system call invocation.

B. Event Interception for Synchronous Security Monitoring

Aside from allowing a detection engine to passively inspect the memory and disk states of a VM through VM introspection, *EagleEye* also support synchronous security monitoring by allowing the interception of guest system events, which include block device I/O events, memory access events, and code execution events.

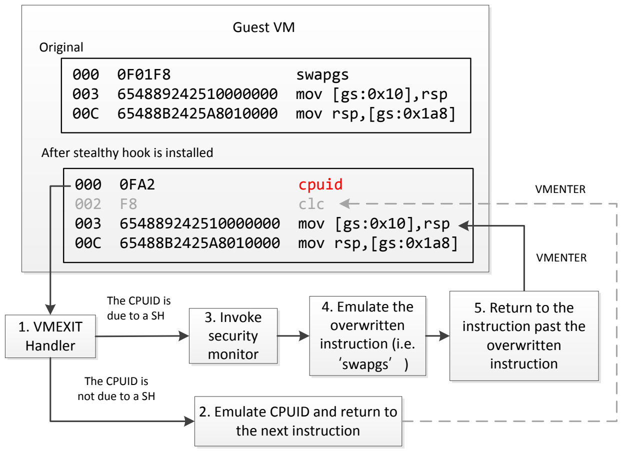
Different from existing approaches [8, 9, 13, 14], event interception in *EagleEye* does not depend on pre-installed guest components (e.g. a PV-driver). For device I/O event, the interception points can be implemented at the device model layer (Sec. IV.B). For memory access event, the interception points can be implemented at the hardware layer (Sec. IV.B). Both can be achieved without relying on a guest component. However, intercepting code execution event is not as straightforward. We need a mechanism to divert code execution in the VM to an external security monitor when a condition on the code execution is met. The most primitive condition is when an instruction at a specific memory address is about to be executed. On top of that, one can further check for specific register values or memory fields to construct a more complex code execution event. Early work on code execution event interception either requires a PV-driver [5] or is restricted to the interception of system calls (e.g. setting MSR.EIP to an invalid virtual address) or is too heavy-weight to be used for security monitoring in production systems (e.g. single-stepping through every guest VM instruction [15]).

In *EagleEye*, we designed a general purpose code execution interception mechanism that allows hooks to be placed at arbitrary code execution points in the guest (Sec. III.C). It is important that the hooks should not interfere with guest functionalities. Ideally, the hooks should be stealthy so that the guest is completely unaware of its existence. Also, the hooks should incur minimal overhead, which turns out to be a difficult challenge because *EagleEye* has no access to the guest kernel scheduler (Sec. III.F).

C. Stealthy Hook

The code execution event interception in *EagleEye* is achieved through the stealthy hook mechanism. Each stealthy hook is essentially a `CPUID` instruction (machine code 0F A2), which occupies only two bytes of memory space. The execution of the `CPUID` instruction in the guest will trigger a VMEXIT event into the hypervisor, which gives *EagleEye* the chance to invoke the corresponding security monitoring process flow. From a practical point of view, the stealthy hook can be placed at any code location in the guest. The only restriction is that the location has to have a contiguous block of at least two bytes in length and the code execution on the block has to be sequential (i.e. there is no jump / branch into the middle of the block).

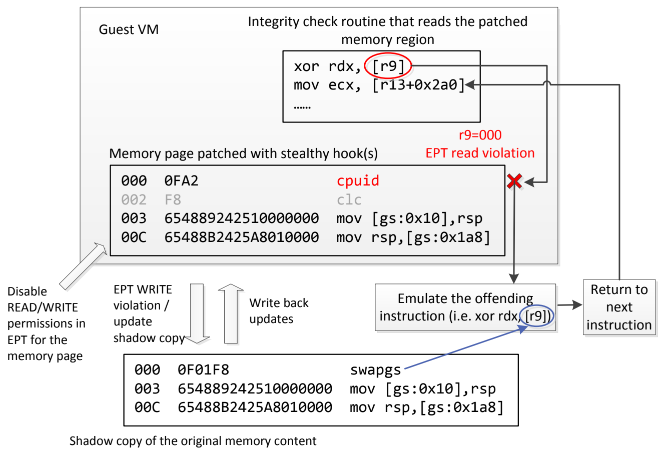
An example of the `CPUID`-based hook is presented in Figure 3. In the example, we place a hook at offset 0x000 right on top of the `SWAPGS` instruction. As `SWAPGS` has a machine code length of three bytes, a residual `CLC` instruction (machine code F8) is left right behind the `CPUID` instruction. When the guest code execution reaches the hook location (i.e. offset 0x000), a VMEXIT event will be triggered due to the execution of the `CPUID` instruction. Control flow will be diverted into the hypervisor (Step 1), where *EagleEye* will check if the `CPUID` is due to a stealthy hook previously installed. If not, it will leave it to the hypervisor to emulate the effect of running the `CPUID` instruction and initiate a VMENTER to return to the next instruction following the `CPUID` in the guest (Step 2). If the `CPUID` is due to a stealthy hook, *EagleEye* will invoke the security monitor (and the corresponding detection engines) (Step 3). Once the security monitor check finishes with a positive acknowledgement, *EagleEye* will then emulate the overwritten instruction for the guest (Step 4) and initiate a VMENTER to return back to the next instruction following the overwritten instruction in the guest (Step 5). If the security monitor rejects the execution (i.e. no positive acknowledgement), there are various ways to abort the code execution. The most extreme way is to kill the whole VM. A more delicate way is to inject error codes into the guest. For instance, we can change the return value of a system call (Sec. IV.C) so that after the VMENTER, the guest will abort the system call as requested by the security monitor.


Guest VM
Original
After stealthy hook is installed
VMENTER
VMENTER
The CPUID is due to a SH
The CPUID is not due to a SH
1. VMEXIT Handler
3. Invoke security monitor
4. Emulate the overwritten instruction (i.e. 'swapgs')
5. Return to the instruction past the overwritten instruction
2. Emulate CPUID and return to the next instruction

```
000  0F01F8              swapgs
003  654889242510000000  mov [gs:0x10],rsp
00C  65488B2425A8010000  mov rsp,[gs:0x1a8]
```

```
000  0FA2                cpuid
002  F8                  clc
003  654889242510000000  mov [gs:0x10],rsp
00C  65488B2425A8010000  mov rsp,[gs:0x1a8]
```

Figure 3. Use of CPUID privileged instruction for stealthy hook


Guest VM
Integrity check routine that reads the patched memory region
r9=000
EPT read violation
Memory page patched with stealthy hook(s)
Disable READ/WRITE permissions in EPT for the memory page
EPT WRITE violation / update shadow copy
Write back updates
Emulate the offending instruction (i.e. xor rdx,[r9])
Return to next instruction
Shadow copy of the original memory content

```
xor rdx, [r9]
mov ecx, [r13+0x2a0]
……
```

```
000  0FA2                cpuid
002  F8                  clc
003  654889242510000000  mov [gs:0x10],rsp
00C  65488B2425A8010000  mov rsp,[gs:0x1a8]
```

```
000  0F01F8              swapgs
003  654889242510000000  mov [gs:0x10],rsp
00C  65488B2425A8010000  mov rsp,[gs:0x1a8]
```

Figure 4. Hiding the hook

The technique of using privileged instructions for diverting control into the hypervisor is more general than techniques based on modifying SSDT or MSR.EIP, which are restricted to system call interception. On the other hand, code execution events can also be intercepted by setting hardware breakpoints at the corresponding hook locations. However, there are a limited of number x86 debug registers for setting hardware breakpoints, and the guest OS may be using some of them as well. Both will limit the number of hooks that be placed in a guest at the same time.

One drawback of the `CPUID`-based approach is that the inserted privilege instructions are not transparent to the guest. This becomes an issue when the guest employs an integrity check routine such as PatchGuard kernel patching protection on x86_64 Windows [16]. PatchGuard periodically checks the integrity of critical kernel structures, system images, and processor MSRs. The inserted `CPUID` instructions will result in a mismatch in the system image checksums and will cause a blue screen of death (BSOD) error. As a result, we have to hide the inserted `CPUID` instruction from guest integrity check, namely making the hooks stealthy. In *EagleEye*, we disable the read / write permissions of the patched memory pages where the hooks are inserted. A shadow copy containing the original memory content prior to the insertion of the hooks is maintained for each of the memory pages as shown in Figure 4. When a guest integrity check routine (e.g. the PatchGuard) attempts to read a patched page, an EPT read violation will be raised. *EagleEye* will intervene and emulate the offending instruction (i.e. the `xor rdx,[r9]` shown in Figure 4). The data which the offending instruction is trying to read will be supplied from the shadow copy of the memory page. On the other hand, if the guest attempts to write to the memory page, an EPT write violation will be raised. *EagleEye* will decode the offending instruction and update the shadow copy correspondingly.

D. Deferred Memory Introspection for Non-present Memory Pages

Right after the interception of a guest system event, a security monitor may need to introspect the guest VM's memory space to gather further details about the event. For instance, the call arguments of `NtCreateFile` including the file handle, access mode, and the file path are essential for establishing the context of a file creation system call event. However, memory introspection may fail if the guest system's page table entries (PTEs) are not properly set. This can be caused by demand paging or memory swapping in the guest system.

We develop a technique called deferred memory introspection to introspect memory regions that are not properly mapped in the guest system's page table yet. The idea is that if the guest will also access the memory region as part of an execution to be monitored, then the introspection can be deferred till the guest has populated the PTEs for the memory region. This is shown in Figure 5, where the failed introspection at Step H1 will be re-attempted at (deferred to) Step H2 after the guest page fault handler fills the PTEs of the memory region.

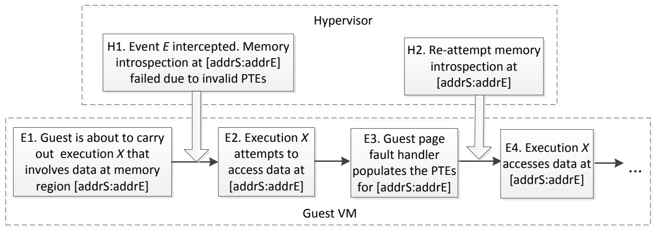

Hypervisor
H1. Event *E* intercepted. Memory introspection at [addrS:addrE] failed due to invalid PTEs
H2. Re-attempt memory introspection at [addrS:addrE]
E1. Guest is about to carry out execution *X* that involves data at memory region [addrS:addrE]
E2. Execution *X* attempts to access data at [addrS:addrE]
E3. Guest page fault handler populates the PTEs for [addrS:addrE]
E4. Execution *X* accesses data at [addrS:addrE]
…
Guest VM

Figure 5. Deferred memory introspection

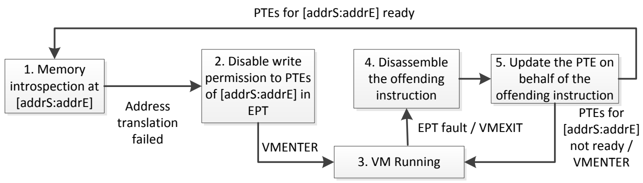

PTEs for [addrS:addrE] ready
1. Memory introspection at [addrS:addrE]
Address translation failed
2. Disable write permission to PTEs of [addrS:addrE] in EPT
VMENTER
3. VM Running
EPT fault / VMEXIT
4. Disassemble the offending instruction
5. Update the PTE on behalf of the offending instruction
PTEs for [addrS:addrE] not ready / VMENTER

Figure 6. Reattempting memory introspection

The time point for the re-attempted memory introspection is determined by intercepting updates to the guest PTEs. This is achieved by disabling write permissions in the EPT for the memory region that holds the corresponding PTEs. As shown in the process flow in Figure 6, when introspection of memory region [addrS:AddrE] fails at Step 1, *EagleEye* will disable write access to the PTEs via setting permission bits in the EPT (Step 2). The guest system will resume execution (Step 3) till the guest system attempts to update the PTEs and trigger a EPT violation / VMEXIT event into the hypervisor. At Step 4, *EagleEye* will disassemble the offending instruction that causes the EPT violation, and at Step 6, *EagleEye* will update the PTE on behalf of the offending instruction. If the PTEs for the memory range [addrS:addrE] are all valid, *EagleEye* will re-attempt the memory introspection again back at Step 1. Otherwise, the guest VM will resume running till the next PTE update (Step 3 → Step 4→ Step 5) occurs.

E. Replication of High-Level Representations

EagleEye can supply a security monitor with the CPU, memory, and disk states of a VM as shown in Figure 2. We use a combination of existing VM introspection techniques and a modified Xen device model to extract the states of a VM. However, most security monitors are designed to operate on a much higher-level representation of the low-level states for

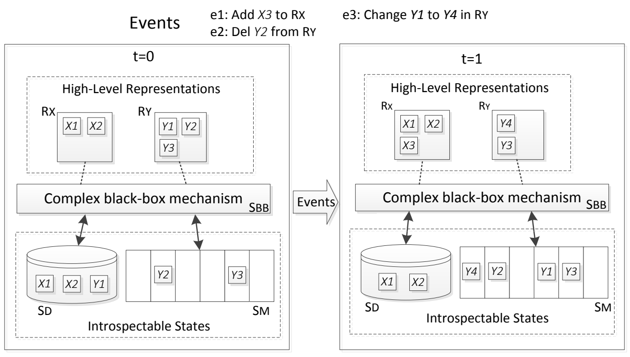


Figure 7. Replication of High-Level Representations

detecting security threats. For instance, an anti-virus scanner may assume that the information about running processes on a system can be acquired in the form of process information records (e.g. EPROCESS [17]) and that the disk content is structured in the forms of files and directories.

Mapping low-level states into high-level representations is hindered by the semantic gap between low-level states and high-level representations. In some cases, the mapping across the semantic gap is readily available, and the high-level representations can be easily attained. For instance, XenAccess [18] relies on guest OS kernel symbols to reconstruct kernel data structures from memory introspection results. The Virtuoso system [19] and Space Traveling [20] can automate the mapping through a training run with the corresponding utility program in an emulator, provided that the low-level data for the high-level representations of interest are readily available in the memory (e.g. be present as part of the guest kernel data in the memory introspection results).

A limitation of existing solutions to the sematic gap problem is that they all assume the low-level data is readily and consistently available from memory introspection. The high-level representations extractable by the state-of-the-art tools [19, 20] are still limited to available information kept in the guest kernel such as getting the list of running processes, getting the list of loaded kernel modules, getting the memory usage statistics, etc. of a guest system. If we look at practical security monitoring applications such as anti-virus scanners and intrusion detection systems, we can see that a lot more high-level representations are required to support their operations. These high-level representations are not limited to in-memory kernel data. And, for synchronous security monitoring applications, many of the high-level representations required are time-sensitive. For instance, an on-access virus scanner may want to check a file for virus before it being accessed. To achieve that, the scanner will have to be able to read the most current content of that file. This is impossible to achieve with existing solutions as the file data will be cached by the guest OS and may not be immediately available from disk introspection. It is equally impossible to introspect the file data from the in-memory disk cache as there is no guarantee the data will stay in there consistently at the time of security monitoring. Finally, many of these mechanisms (e.g. disk caching) are not well-documented and are inherently complex to deal with. It is questionable whether reconstructing their high-level representations from VM introspection results alone can be efficiently and reliably carried out. For the mechanisms are complex, the VM introspection will have to be quite thorough to begin with, and that will also be a problem on the performance if it was to be inlined as part of the security monitoring process.

Instead of pushing the limit of VM introspection further, we pursued a complementary approach to the monitoring of high-level representations of a guest VM. Let $\mathbf{R}_i$ be a high-level representation of guest VM states $\mathbf{S}_{\mathbf{VM}}$:={$s_1,s_2,\ldots,s_N$} that is of interest to some security monitoring application. The high-level representation $\mathbf{R}_i$ also has an associated mapping function $\mathbf{MAP}_i$: $\mathbf{S}_{\mathbf{VM}} \rightarrow \mathbf{R}_i$. In practice, only a subset of VM states $\mathbf{S}_{\mathbf{INTRO}} \subseteq \mathbf{S}_{\mathbf{VM}}$ are introspectable (i.e. can be reliably and efficiently acquired through VM introspection). Existing solutions to the semantic gap problem thus focus on finding the mapping $\overline{\mathrm{MAP}_i}$: $\mathbf{S}_{\mathbf{INTRO}} \rightarrow \mathbf{R}_i$. Note that we require a high-level representation to be always consistent with respect to its specification. This is because inconsistent high-level representations are of little use to security monitoring. We do not want to burden the security monitor developer with situations where the representations presented to the security monitor may contain invalid information in it. On the other hand, the underlying introspectable states $\mathbf{S}_{\mathbf{VM}}$ may be inconsistent, possibly due to introspection at the wrong timing that returns stale or incomplete states. It is up to the designer of $\mathbf{MAP}_i$ or $\overline{\mathrm{MAP}_i}$ to either preclude or tolerate these states in the design of the mapping function.

In *EagleEye*, we bring in the notion of monitoring time into the high-level representations. Thus $\mathbf{R}_i(t)$ is the high-level representation of the VM states $\mathbf{S}_{\mathbf{VM}}(t)$ at time t when a security monitor requests for the high-level representation. We assume that changes to the high-level representation between monitoring time t_1 and t_2 ($t_1 \leqq t_2$) are observable through the interception of corresponding system events $\mathbf{E}_i(t_1,t_2)$ (e.g. system calls), that is $\mathbf{R}_i(t_1) \xRightarrow{\mathbf{Ei}(t1,t2)} \mathbf{R}_i(t_2)$. As such, we can determine $\mathbf{R}_i(t)$ by a one-shot introspection at $R_i(0)$, when the introspection can be reliably and efficiently carried out, and then apply the changes $\mathbf{E}_i(0,t)$ leading to $\mathbf{R}_i(t)$.

As an example of the approach, let us consider the VM illustrated in Figure 7, where states S_D and S_M are introspectable, and state S_{BB} is not introspectable. The high-level representations R_X and R_Y can both be acquired from introspection at t=0. Now at t=1, neither of the representations can be acquired as the introspectable states are not consistent.

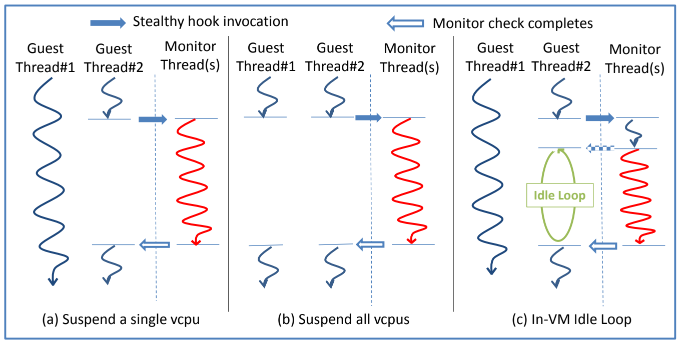


Figure 8. Suspending guest thread

However, the changes to the representations from t=0 to t=1 correspond to the system events e1, e2, and e3. In this case, replicas of R_X and R_Y will be maintained in the *EagleEye* daemon. *EagleEye* will intercept the corresponding system events (i.e. e1, e2, and e3) and apply the changes to the replicas. The replicas can provide a security monitor with a consistent and current view of R_X and R_Y even when the introspectable states are not consistent yet. A concrete application of the approach is presented in Sec. IV.D.

F. In-VM Idle Loop

When a stealthy hook is invoked, the execution of the corresponding guest thread will be diverted into the hypervisor. The guest thread will then be blocked till the event handler acknowledges the invocation. This allows the detection engines to be triggered for security checks in a synchronous manner as shown in Figure 8(a). Since the use of PV driver is not allowed, we cannot leverage existing guest kernel synchronization mechanism to block the guest thread. Instead, the guest thread can be blocked by removing the corresponding VCPU from the hypervisor scheduler runqueue (thread #2 as in Figure 8(a)). However, suspending a VCPU from being scheduled for a prolonged period is problematic for a SMP guest. At the minimum, this may break the scheduling fairness and responsiveness guarantee of the guest scheduler. More seriously, the guest may consider the suspended VCPU as faulty. For instance, we found that Windows guest (i.e. Windows Server 2008) will react drastically with a BSOD when *EagleEye* selectively suspends the VCPUs. On the other hand, Linux (i.e. Fedora Core 14 x86_64 with 2.6 kernel) seems to be insensitive to the selective suspension of VCPUs.

As a compromise, when a guest thread needs to be blocked for a prolonged period pending check by the security monitor, all the VCPUs of the guest VM have to be suspended (Figure 8(b)). This prevents the guest OS from perceiving a VCPU as faulty. However, as none of the guest threads will be running during the period, a noticeable performance drop is expected especially if the guest system is under heavy workload. Besides, suspending VCPUs may disturb interrupt-based guest timekeeping [21].

A third approach is not to suspend any VCPU of the guest VM. Instead of blocking the guest thread, we can just let the guest thread spin in a loop till the detection engines complete the check and the event handler acknowledges the hook invocation as shown in Figure 8 (c). This allows all VCPUs to run continuously and also gives the guest OS scheduler the chance to schedule other guest threads onto the VCPU that is running the loop.

A fixed number of loops are pre-allocated in a special driver that has to be pre-installed in the guest system. Each loop is used to spin a guest thread. If all the loops are occupied, *EagleEye* will fall back to the approach of suspending all VCPUs.

A snippet of the loop code is shown in Figure 9. The entrance to the loop locates at offset 0x5, it first determines the memory address of a 1 byte variable used for signaling the termination of the loop. The memory address will be passed via the RAX register. The code from offset 0x17 to offset 0x25 then keeps checking the variable value and will loop until the value is set to a non-zero value by the *EagleEye* hypervisor module. Note that the In-VM idle loop is optional. The presence of the idle loop only improves performance. The security monitor functionality of *EagleEye* is not affected in the absence of the idle loop.

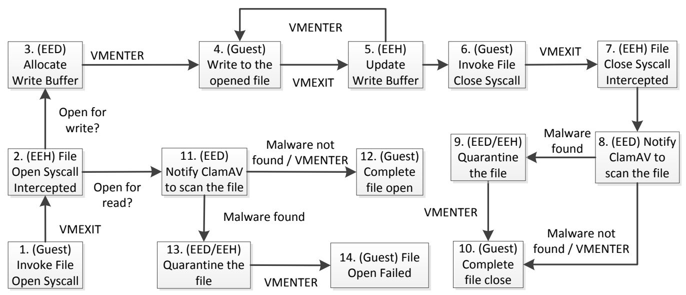
```
00  488B0424          mov rax,[rsp]
04  C3                ret
         Loop_Start:
05  9C                pushfq
06  50                push rax
07  51                push rcx
08  E8F3FFFFFF        call dword 0x0
0D  482500F0FFFF      and rax,0xfffff000
13  4883C000          add rax,byte +0x0
17  488B08            mov rcx,[rax]
1A  4881E1FF000000    and rcx,0xff
21  4883F900          cmp rcx,byte +0x0
25  74F0              jz 0x17
27  59                pop rcx
28  58                pop rax
29  9D                popfq
2A  0FA2              cpuid
```

Figure 9. In-VM Idle Loop

IV. IMPLEMENTATION

To show how security monitoring in *EagleEye* works, we build a prototype system that supports one of the most widely employed security monitoring application in today's computing environment – an on-access malware detection system for VDC environment. *EagleEye* can potentially support other kinds of security monitoring applications such as network intrusion detection, system integrity checking, and many others. The choice is mainly due to its popularity, so the discussion can focus more on the challenges pertaining to the *EagleEye* approach.

The prototype is implemented on Xen hypervisor 4.0.1 (x86_64 architecture). The host OS is Fedora Core 14 with 2.6.32 kernel. The extension to Xen includes 2,312 lines of code in the hypervisor and 1,330 lines of code in the QEMU device model. The current implementation supports Windows guests. Support for other types of guest systems require setting up the appropriate system call interception hooks (Sec. IV.C) and having the corresponding parsers for the memory and disk introspection results as different types of guest systems may use different memory layouts and file systems.


1. (Guest) Invoke File Open Syscall
VMEXIT
2. (EEH) File Open Syscall Intercepted
Open for write?
3. (EED) Allocate Write Buffer
VMENTER
4. (Guest) Write to the opened file
VMEXIT
VMENTER
5. (EEH) Update Write Buffer
6. (Guest) Invoke File Close Syscall
VMEXIT
7. (EEH) File Close Syscall Intercepted
8. (EED) Notify ClamAV to scan the file
Malware found
9. (EED/EEH) Quarantine the file
VMENTER
Malware not found / VMENTER
10. (Guest) Complete file close
Open for read?
11. (EED) Notify ClamAV to scan the file
Malware not found / VMENTER
12. (Guest) Complete file open
Malware found
13. (EED/EEH) Quarantine the file
VMENTER
14. (Guest) File Open Failed
Figure 10. Online Malware Detection Process Flow

A. *Security Monitoring Process Flow: On-Access Malware Detection*

The process flow of on-access malware detection is presented in Figure 10. The idea is to monitor file access activities inside a guest VM and check if the file to be accessed matches the signature of a known malware (the prototype uses ClamAV [22] as the signature matching engine). A key difference between *EagleEye* and existing anti-virus scanners is that all the above operations have to be carried out without guest components or any hooking mechanism [23, 24] provided by the guest OS.

The process flow in Figure 10 starts at Step 1, where a file open system call is invoked inside the guest VM and is immediately intercepted by the *EagleEye* hypervisor module (E^2H) at Step 2 through the stealthy hook mechanism (Sec. IV.C). If the file is opened with write access, subsequent writes to the file will be trapped by E^2H as well. At Step 4~5, E^2H will copy the data written to the file to the corresponding write buffer entry maintained by the *EagleEye* daemon (E^2D). The write buffer is used for dealing with inconsistencies in disk introspection results, which will be discussed in Sec. IV.D. For now, one can just imagine that the write buffer is filled with the data that has been written to a file. At Step 6~8, right after the file is about to be closed, *EagleEye* will invoke ClamAV to scan the file. Upon completion of the scan, the guest system can proceed with the file closing operation if the file is clean. Otherwise, if the file is a malware, *EagleEye* will quarantine it.

The process flow of file open for read follows from Step 2 to Step 11, where ClamAV is invoked to scan the file on the disk through disk introspection. If the opened file was just created moments ago, chances are the content of the file is still available in the write buffer and there will be no need for the disk introspection. If an existing file is opened for both read and write, possibly by different processes, *EagleEye* will merge the disk file content into the write buffer to ensure that the content inspected by ClamAV is consistent and current.

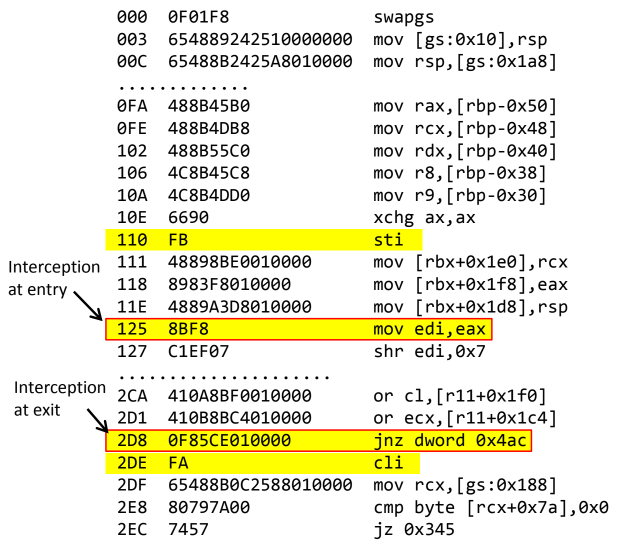
B. *Intercepting I/O and Memory events*

In the current prototype, we assume that a VM uses the storage and network devices emulated by the QEMU device model on Xen. We made modification to the QEMU device model to allow the interception of VM disk access at the block level. Similarly, the network traffic of a VM can be monitored through the QEMU device model. Intercepting VM device I/O events through the device model layer is intrinsically transparent and requires no modification to the guest systems.

EagleEye can intercept guest memory access at the page granularity through manipulating the permission bits in the extended page tables (EPTs) on Intel platform or the nested page tables (NPTs) on AMD platform. The mechanism is backed by hardware and incurs negligible performance overhead. The interception is also intrinsically transparent requires no modification to the guest systems.

C. *Intercepting System call*

System call interception allows a security monitor to mediate critical guest system operations. In the current prototype, we intercept both the entry and exit points of guest system calls. The interception at the exit of a system call is used



```
000  0F01F8              swapgs
003  654889242510000000  mov [gs:0x10],rsp
00C  65488B2425A8010000  mov rsp,[gs:0x1a8]
.............
0FA  488B45B0            mov rax,[rbp-0x50]
0FE  488B4DB8            mov rcx,[rbp-0x48]
102  488B55C0            mov rdx,[rbp-0x40]
106  4C8B45C8            mov r8,[rbp-0x38]
10A  4C8B4DD0            mov r9,[rbp-0x30]
10E  6690                xchg ax,ax
110  FB                  sti
111  48898BE0010000      mov [rbx+0x1e0],rcx
118  8983F8010000        mov [rbx+0x1f8],eax
11E  4889A3D8010000      mov [rbx+0x1d8],rsp
125  8BF8                mov edi,eax
127  C1EF07              shr edi,0x7
.....................
2CA  410A8BF0010000      or cl,[r11+0x1f0]
2D1  410B8BC4010000      or ecx,[r11+0x1c4]
2D8  0F85CE010000        jnz dword 0x4ac
2DE  FA                  cli
2DF  65488B0C2588010000  mov rcx,[gs:0x188]
2E8  80797A00            cmp byte [rcx+0x7a],0x0
2EC  7457                jz 0x345
```

Figure 11. System call interception in *EagleEye*

to acquire the return values of system calls, which may carry information needed for on-access virus scanning such as the file handle to a file opened by the `NtCreateFile` system call. The return values are also used by many system calls to indicate if the requested operations were successful. For instance, we rely on the return values of system calls in the high-level representation replication (Sec. IV.D) to detect successful file system update events so that their effects can be applied to the replicas correspondingly.

The interception of system call is implemented by the stealthy hook mechanism. Intuitively, the interception at the entry point of a system call can be implemented by a stealthy hook at the first instruction of the system call handler (i.e. the `SWAPGS` instruction at offset 0x000 in Figure 11). However, it would not hurt to move the hook a little bit down below as long as it still locates before the dispatching of system call. The reason why we move it beyond the `STI` instruction at offset 0x110 is for the optional In-VM idle loop (Sec. III.F) to be interruptible. This allows the guest OS to be able to schedule other threads onto the VCPU where the calling thread is blocked by *EagleEye* pending security check. For the same reason, instead of placing a hook right on the `SYSRET` instruction at the end for intercepting system call exit, the hook is placed at the location before the `CLI` instruction (at offset 0x2D8 in Figure 11).

D. *Write Buffer for Consistent Filesystem Introspection*

In *EagleEye*, we apply the high-level representation replication for the guest file system. Operating systems nowadays commonly employ disk caching to buffer disk accesses. Data written to a file may not be immediately flushed to the disk for performance reasons. On the other hand, the effect of the writes has to be immediately reflected on the guest file system and be seen by the processes within the guest. Security monitoring cannot assume data from VM disk introspection will reflect the current file system states. On the other hand, it is also quite difficult to use memory introspection on the in-memory disk cache due to the complexity and opacity (assuming closed-source guest kernel) of the mechanism.

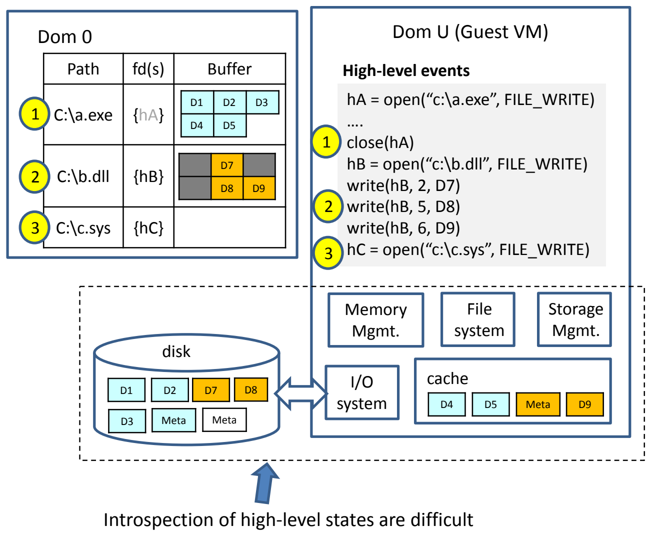


Figure 12. State replication for file writes

EagleEye maintains a per-VM table in the *EagleEye* daemon at Dom 0 for files that are opened for write access in each guest system as shown in Figure 12. Each table entry corresponds to one file. The entry contains the path to the file, its associated file descriptors, the file's memory-mapped views if any, and most importantly, a buffer holding the data written to the file thus far. Essentially, every file write in the guest system will result in an update to the buffer in the corresponding table entry. File open and file write events are captured through intercepting the respective system calls. Memory-mapped file I/O are also tracked by intercepting `NtUnmapViewOfSection`, `NtMapViewOfSection`, ant `NtCreateSection`. When a mapped view is about to be closed or when the memory mapped file is about to be closed, *EagleEye* will synchronize the data from the mapped memory regions to the write buffer. The associated memory-mapped views will then be removed from the write buffer entry.

When a security monitor wants to access the content of a file, *EagleEye* will by default serve the content through disk introspection. For those data blocks that have just been overwritten, the content will be pulled from the write buffer. This ensures that the security monitor always has a consistent view of the current file content as perceived by the guest system. If a write buffer entry has associated memory-mapped views, *EagleEye* will first synchronize data from the associated memory-mapped views to the write buffer before serving the write buffer data to the security monitor.

A write buffer block can be freed when the data hold by the block has been flushed to the disk. The current implementation relies on a background garbage collection process to periodically introspect the disk to determine if the current block data has been reflected on the disk. A table entry can be safely released if all associated descriptors have been closed, and the content in the buffer has all been flushed to the disk. The current implementation may double the space required for disk storage in the worst case. If storage space is of a concern, it is possible to employ further optimization to reduce the space cost. For instance, the virtual disk subsystem (e.g. QEMU-dm) can be hooked to detect guest disk flushing events so the garbage collection process can be triggered timely.

E. *Assumption of System Model*

The implementation depends on knowledge about the entry/exit points of the guest system call dispatcher, the semantics of the guest system calls hooked by *EagleEye* (Table 2), and the guest application binary interface (ABI) formats.

The implementation does not support SR-IOV network adaptors [25], but the mandatory security monitoring of network traffic can be transparently implemented with well-established network intrusion detection appliances.

The quarantine of virus infected files (Step 13 in Figure 10) is implemented by hooks in the QEMU block device emulation layer. It is the only feature in the current *EagleEye* implementation that depends on the QEMU device emulation layer. Should the guest VM have direct access to the block device controller and not use the device emulation layer, the quarantine feature can instead be implemented as stealthy hooks in the guest kernel (requiring further knowledge of the guest system model) or possibly be implemented through extensions of the block device controller hardware.

Table 1. Benchmarks for the experiments

7-zip compression
Compression of 13,791 files (the files are collected from the folders (Windows\SysWOW64, Windows\System32, and Windows\Microsoft.NET) on a freshly installed Windows Server 2008) with a total size of 2.55GB. 7-Zip is configured to use 4 threads for compression.
7-zip decompression
Decompression of the above compressed file. The decompression uses only 1 thread.
Build ClamAV
Build ClamAV 0.97.5 from source code with Visual studio 2010
x264 encoding
Encode a 508mb RMVB file with x264 encoder. The x264 encoder is configured to use 4 threads.

V. EVALUATION

We evaluate *EagleEye* through comparison with the baseline system running no security monitor and with systems running *conventional* in-VM security monitors including Kaspersky Endpoint Security 8 and a home-made in-VM malware scanner InVM_AV. InVM_AV is functionally equivalent to Kaspersky from the experiment point of view, except that it is based on the same ClamAV scan engine as used by *EagleEye*.

For all the experiments, the host machine is equipped with two Intel Xeon E5520 processors (a total of 16 logical cores) and 16 GB memory. The storage consists of 2 SATA HDDs configured in JBOD mode. The host OS is Fedora Core 14 x86_64. Each guest VM is configured with 4 VCPUs and 4GB memory. The guest OS is Windows Server 2008 x86_64 R2.

A. *Overall Performance*

In this experiment, we evaluate the performance of *EagleEye* by looking at the running time of four benchmark programs (Table 1) under seven different security monitor setups: baseline, Kaspersky, Kaspersky*, InVM_AV, InVM_AV*, EE, and EE*.

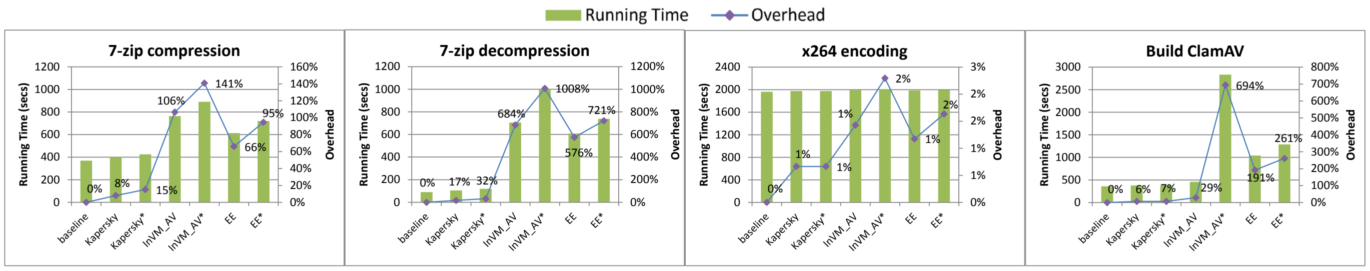


Figure 13. Benchmark running times for different security monitoring setups.

The baseline setup is just a clean VM guest system without any security monitor. The Kaspersky setup is a VM with Kaspersky Endpoint Security 8 installed inside the VM guest. The InVM_AV setup is a VM with the InVM_AV scanner installed inside the VM guest. All the above setups use the vanilla Xen 4.0.1 hypervisor.

The EE setup uses the *EagleEye* prototype system, which includes a modified Xen hypervisor, a modified QEMU device model with *EagleEye* extensions, and the *EagleEye* daemon with the ClamAV detection engine linked as a shared library. The *Eagle* daemon is configured to use 4 threads.

InVM_AV, Kaspersky, and EE only scan files with specific extensions such as .COM, .EXE, .DLL, .SYS, and etc. (there are a total of 59 extensions in the filter). We also create three additional setups that scan all the files, which are Kaspersky*, InVM_AV*, and EE* respectively.

The running times of the four benchmark programs with respect to each security monitor setup is presented in Figure 13. For each setup, the overhead with respect to the baseline setup is also presented. We can see that for the x264 benchmark, which involves few file open and creation activities, the running time with *EagleEye* is comparable to that of Kaspersky or even the baseline. For benchmarks that involve intensive file opens or file creations, notably the 7-zip decompression, the running time from *EagleEye* is substantially longer than that from Kaspersky, with an overhead of 1008%. The poor performance with the 7-zip decompression is due to the high amount of file creation and file write activities. The file writes incur extra overhead from the write buffer mechanism. In comparison, while 7-zip compression also involves lots of file open and file read activities, the overhead is just about 66%.

EagleEye is clearly outrun by Kaspersky in all cases. We believe this is primarily due to various proprietary optimizing heuristics in the Kaspersky detection engine. This can be confirmed by the comparison with our home-made InVM_AV scanner, which is based on the same ClamAV detection engine used by *EagleEye*. In fact, the comparison shows that *EagleEye* approach even outperformed the InVM_AV for 7-zip compression, 7-zip decompression, and x264 encoding. At first sight, the results may seem counter-intuitive, as *EagleEye* requires extra overheads on inter-domain communication and coordination with the hypervisor. One of the reasons here is that the ClamAV engine in the *EagleEye* setup runs externally to the monitored VM and does not need to compete with the benchmark program for the VCPUs allotted to the VM. Another reason is that I/O from within a DomU guest VM is typically slower that from within the Dom0 VM, as can be seen from Figure 14, where the ClamAV offline scan times and the iozone benchmark scores of four different VM setups on Xen including Dom0, PVHVM DomU, HVM DomU Linux. and HVM DomU Windows are compared. The findings actually reveal one additional benefit of the *EagleEye* approach, where security monitoring I/O are performed in the more efficient Dom0,

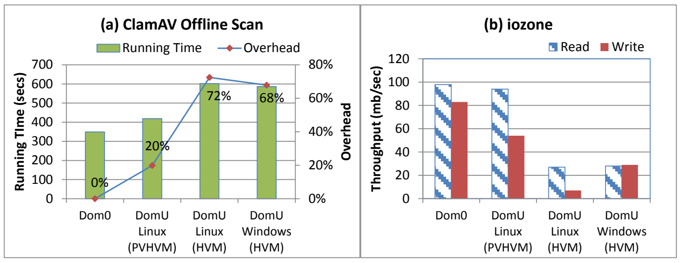


Figure 14. Comparison of Dom 0, PVHVM guest, and HVM guest disk I/O throughputs.

B. Synchronous Monitoring Overhead

Here we conduct an experiment with the build ClamAV benchmark to look at the performance overhead at each stage of security monitoring in *EagleEye*. As the benchmark program is multithreaded and the guest is a SMP VM with 4 VCPUs, the experiment is conducted by adding each monitor stage incrementally and observing the corresponding benchmark running time. We conduct two batches of experiments, one with In-VM idle loop enabled, and one without In-VM idle loop (i.e. suspending all VCPUs of the VM when waiting for detection engine checks).

As shown in Figure 15, adding the stealthy hook (**sih**)

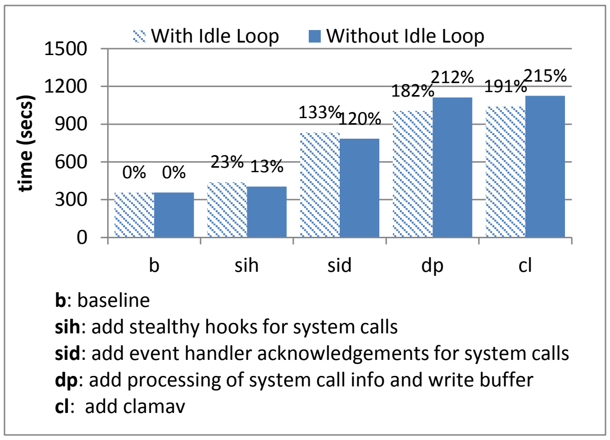


Figure 15. Build ClamAV aggregation time (adding stages incrementally for EE setup)

incurs only slight overhead over the baseline (**b**). The overhead is partly due to the control flow transition to and from the hypervisor. The other portion of the overhead comes from the emulation of instructions in stealthy hooks (Sec. III.C). The overhead with In-VM idle loop enabled is slightly higher because there is a scheduling delay for the spinning loop to detect the termination signal (Figure 9).

A significant amount of overhead occurs when we start passing hook events to the event handler in the daemon (**sid**), where the event handler is set to immediately acknowledge the hook invocation without invoking any detection engine. The overhead includes the time for hypervisor-to-daemon communication via Xen event channel and the time for the daemon-to-hypervisor communication for resuming the VCPUs or terminating the In-VM idle loop. The significant overhead is in part due to the scheduling delay incurred by the interplay of Xen scheduler and Dom0 scheduler. Also, with In-VM idle loop, there is the scheduling delay caused by the DomU scheduler. Another contributing factor to the overhead is due to the high-amount of 2,332,052 stealthy hook invocations in this benchmark.

The effect of In-VM idle loop starts to appear when *EagleEye* is set to process the intercepted system calls (i.e. getting the parameters through introspection / deferred introspection), and when the write buffer is turned on (**dp**). Adding the ClamAV scan engine (**cl**) incurs little overhead, as, with the file extension filter turned on, only a handful of 5,000 files are checked by the scan engine.

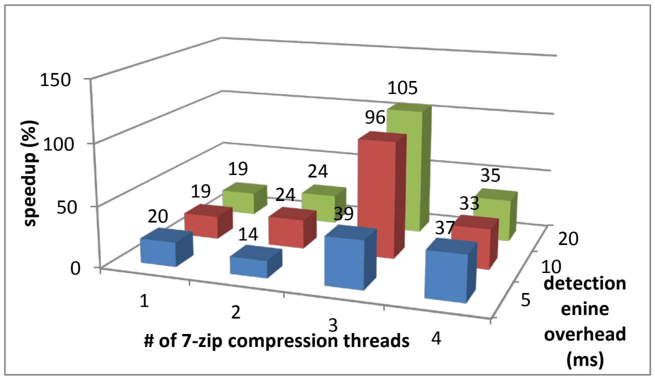


Figure 16. Speedup by In-VM Idle loop

C. *Speedup by In-VM Idle Loop*

We try to explore the factors that might affect the effectiveness of the In-VM idle loop mechanism. The factors considered include the degree of parallelism of the benchmark and the time for the detection engine to complete its check (i.e. the detection engine overhead). We use 7-zip compression as the benchmark and vary the number of compression threads. On the other hand, we replace the ClamAV scan engine with sleep timers of 5, 10, and 20 ms. For each pair of parameters, we measure the running time of the benchmark with and without the In-VM loop mechanism. The speedup ((time without loop) / (time with loop)-1)*100 % is presented in Figure 16. In general, we can see that the speedup goes up with increasing degree of parallelism in the benchmark program (i.e. more number of compression threads). However, with 4 compression threads, the speedup drops back down to about 35%. This is because the VM is provisioned with 4 VCPUs. Both the other background threads on the guest and the spinning idle loops are more likely to compete with the compression threads for VCPUs in this case.

Overall, we can see that the In-VM idle loop mechanism improves the responsiveness of the system when the detection engine overhead is non-negligible. Increasing the degree of parallelism in the benchmark program makes the effect of the In-VM idle loop even more evident till the degree of parallelism reaches the number of VCPUs provisioned. At that moment, the contention among the threads go up significantly and begin to undermine the effect of the In-VM idle loop.

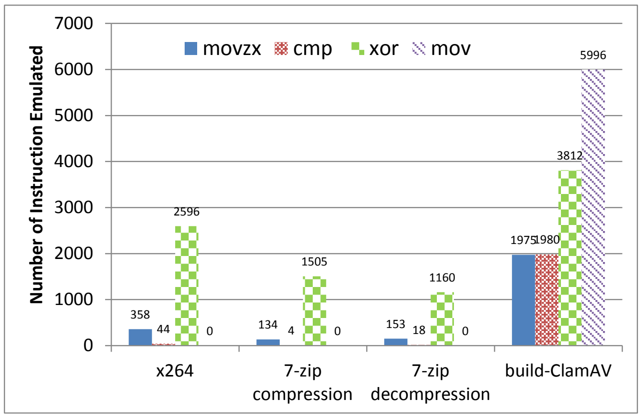


Figure 17. Distribution of Offending Instructions

Table 2. Instruction and System Call Counts of Benchmarks

	7-zip compression	7-zip decompression	build clamav	x264
Instruction Count (billion)	141.6	163.7	6.8	9,752.6
System Call Count	453,975	425,881	404,844	2,116,652
NtOpenFile	4,718	113,273	397,084	2,438
NtCreateFile	28,578	147,830	367,226	1,582
NtWriteFile	21,493	51,803	500,728	24,408
NtClose	41,812	91,088	873,247	61,884
NtCreateSection	910	412	18,340	996
NtMapViewOfSection	2,454	728	33,944	2,500
NtUnmapViewOfSection	563	181	10,354	574

D. *Instructions Emulated for Stealthy Hook*

Figure 17 presents the distribution of instructions emulated by *EagleEye* for cloaking the stealthy hooks in the guest system. The instructions are mostly due to PatchGuard check[16]. Compared with the total number of instructions or the total number of system calls involved in each benchmark program (Table 2), the instruction emulation is actually relatively infrequent. This is consistent with the result in Figure 15, which indicates the overhead incurred by instruction emulation should be negligible. An interesting finding is that the frequency of instruction emulation seems to be affected more by the number of guest system calls than by the number of instructions executed. For instance, the build clamav benchmark has a significantly smaller instruction count than the 7-zip benchmarks, yet their numbers of instruction emulated and also their system call counts are roughly on par with each other.

E. Comparison of Detection Engines: ClamAV vs. Kaspersky

The home-made InVM_AV scanner performs poorly in comparison with Kaspersky. We noticed that Kaspersky was pretty selective (smart) about the locations in a file that it needs to look at for virus patterns, even when the maximum scan settings (most comprehensive scanning) are enabled. In contrast, the ClamAV engine adopted a much more conservative scan heuristic. Specifically, ClamAV will skip MP3, MPEG, RM, OGG, SIP log, PDF image, SQLite journal, and SQLite DB files in its scan (as defined in `libclamav/filetypes_int.h`). Other than those files, ClamAV with the default setting will look into the first 25MB segment of a file for virus patterns.

For Kaspersky, there is no source code for knowing about its heuristics. We thus conduct an experiment to confirm that the inspection by ClamAV is indeed more thorough than that by Kaspersky, which contributes to longer execution time of InVM_AV in the experiments above. Note that for many practical cases, a thorough inspection may not be necessary at all, as long as the virus code is highly unlikely to be activated in those situations. We are not trying to use the experiment to indicate that ClamAV provides better virus protection than Kaspersky.

In Table 3, we can see that both ClamAV and Kaspersky can detect well-formed malware binary (situation 1). However, Kaspersky does not detect virus code appended at the end of a benign PE executable file (situation 2). Similarly, Kaspersky does not detect virus code appended at the end of a JPEG file (situation 3), and neither does it detect virus code appended at the end of a Gzip archive file (situation 4). In comparison, ClamAV can detect the virus codes in all three cases. In situation 5, the virus code is appended at the end of a .rm RealMedia file. Both Kaspersky and ClamAV skip checking the file. If we corrupt the header of the .rm file (situation 6), ClamAV will inspect the file and detect the virus code while Kaspersky still report the file as virus-free.

Table 3. Scan range comparison between ClamAV and Kaspersky

Situation	ClamAV	Kaspersky
1. Well-formed malware binary	○	○
2. Benign PE executable appended by virus code	○	×
3. JPEG image file appended by virus code	○	×
4. Gzip archive file appended by virus code	○	×
5. RealMedia format (.rm) video file appended by virus code	×	×
6. RealMedia format (.rm) video file appended by virus code with corrupted header	○	○

VI. RELATED WORK

The concept of VMM based security monitoring was proposed by Garfinkel et al [6]. Their security monitor can perform integrity check of the guest kernel and programs and can also detect NIC promiscuous mode usage. The semantic gap problem in VM introspection was discussed in XenAccess [10, 18], VMwatcher [26], and Virtuoso [19]. However, none of the above work can be used to deal with semantic gaps caused by complex mechanisms such as disk caching.

Event-driven VMM monitoring was proposed in the system Lares [5]. Lares employs a PV driver in a guest VM to reroute events of interest to an external security application. VMware provides a set of introspection API called VMsafe [7] for security monitoring on VMware platform. The API allows the introspection of guest VM network, CPU, memory, and disk storage states. Event-driven monitoring is supported through a PV driver (i.e. the vShield endpoint driver). VMsafe has been employed in products such as TrendMicro DeepSecurity and McAfee MOVE. Our work is distinct from these works in that our approach does not require PV drivers to hook and reroute the guest events.

Virtualization-based monitoring has also been applied to dynamic malware analysis [15, 27]. The motivation is that hardware assisted virtualization can be leveraged to hide the analyzer. The analysis environment is purposely built and not part of a production system, so issues such as overall system performance and deployment cost are not as relevant as in the realm of online security monitoring. Also, malware analysis system focuses more on extracting the full behavior of a malware. The analysis does not have to be synchronous and responsive. It can assume that complete information about the system and the malware under analysis can be acquired at a later time. On the contrary, security monitoring often has to make monitoring decisions synchronously and immediately based on very limited information at that time point.

Rosenblum et al. [28] first proposed the use of virtualization to separate instruction execution and data access contexts on memory pages. We adopt the same strategy of memory context separation to hide the stealthy hook from guest detection (Sec. III.C). However, our implementation takes advantage of the extended page table virtualization hardware and does not require every guest page fault to be trapped into the hypervisor.

VII. LIMITATIONS

EagleEye provides a set of introspection and event interception primitives to be leveraged for implementing mandatory security monitoring in VDC environment. The scope of monitoring is limited to security violations that will manifest in the guest CPU state, memory state, disk state, or as guest block device I/O, memory access, or code execution events. The effectiveness of monitoring largely depends on the algorithm employed by the corresponding detection engine. *EagleEye*, by itself, does not guarantee the detection of any specifics attack. The fidelity of the introspection and event interception primitives as provided by *EagleEye* depends on certain knowledge of the guest system model (Sec. IV.E). A malicious tenant may use a guest system of which such knowledge is not available to *EagleEye* in advance (e.g. using a non-standard guest system kernel) and thus evade the security monitoring. *EagleEye* can possibly detect the presence of non-standard guest kernels through memory introspection and looking at the patterns of the guest system events but that may not suffice to indicate a guest system being malicious.

EagleEye does not attempt to address denial-of-service (DoS) attacks against the detection engines or the monitoring infrastructure. An adversary may generate high amount of activities in the guest system (e.g. synthesizing a huge number of faked executable binaries) to keep a detection engine busy. If the detection engine operates in asynchronous mode, false negatives due to missed detection are likely to occur. On the other hand, if the detection engine operates in synchronous mode, the DoS attack will not cause false-negatives but the guest system will be frequently blocked pending security monitoring. The impact can be localized to the offending guest by implementing QoS mechanism on the rate of stealthy hook invocation.

VIII. CONCLUSION

We propose the *EagleEye* approach to achieve mandatory security monitoring in virtualized datacenter environment. The approach has been applied to a real-world security monitoring application. The proposed approach requires no modification to a guest VM or attention from the VM tenants, which we believe are the key to achieve mandatory security monitoring in large-scale VDC environments, such as an IaaS cloud.

In *EagleEye*, we come up with the technique of high-level representation replication to address the semantic gap and the inconsistent system state problems. The technique is powerful enough to deal with complex black-box mechanisms such as disk caching. The requirement for synchronous monitoring is supported by the stealthy hook mechanism, which is transparent (to the guest) and scalable. We proposed the deferred introspection technique as an enhancement of memory introspection to deal with inconsistent guest memory states due to on-demand paging or memory swapping. The goals of mandatory security monitoring prevent the use of guest kernel synchronization mechanisms to implement efficient blocking-wait for security monitoring. We come up with the In-VM idle loop mechanism to improve the performance of security monitoring due to the lack of such synchronization mechanisms.

Performance overhead is still an issue of the current *EagleEye* implementation. The strength of *EagleEye* being able to operate without a PV-driver is also its weakness. We hope for the discussion to contribute to potential future research on VDC mandatory security monitoring, or more generally, the mandatory security monitoring of other types of cloud datacenters. *EagleEye* is not yet a perfect solution to mandatory security monitoring for VDC environment. We look forward to the community engaging in dialog that would help mature the technologies.

ACKNOWLEDGMENT

The work was supported by National Science Council in Taiwan under grants 100-2218-E-009-005 and 101-2221-E-009-076. We thank Hong-Wei Li for assisting with the measurement of benchmark program instruction counts.

REFERENCES

[1] H. Liu. (2012, March 13). Amazon data center size. Available: http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/

[2] R. McMillan. (2009). Botnet found in Amazon's EC2 cloud. Available: http://news.techworld.com/security/3208467/botnet-found-in-amazons-ec2-cloud/

[3] F. Rashid. (2011). Sony PSN Hackers Used Amazon EC2 in Attack. Available: http://securitywatch.eweek.com/data_breach/sony_psn_hackers_used_amazon_ec2_in_attack.html

[4] VMware. Antivirus Best Practices for VMware View 5. Available: http://www.vmware.com/files/pdf/VMware-View-AntiVirusPractices-TN-EN.pdf

[5] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in IEEE Symposium on Security and Privacy, 2008, pp. 233-247.

[6] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in NDSS, 2003.

[7] VMware. VMware VMsafe. Available: http://www.vmware.com/technical-resources/security/vmsafe/security_technology.html

[8] McAfee. McAfee MOVE AntiVirus. Available: http://www.mcafee.com/us/products/move-anti-virus.aspx

[9] TrendMicro. Trend Micro Enterprise Security - Changing the Game for Anti-Virus in the Virtual Datacenter. Available: http://www.trendmicro.com/cloud-content/us/pdfs/business/white-papers/wp_vmware-trendmicro-av-virtualization.pdf

[10] B. Payne. (2012, 1/12). LibVMI. Available: http://vmitools.sandia.gov/libvmi.html

[11] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in IEEE Symposium on Security and Privacy, 2010, pp. 380-395.

[12] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in SOSP, Cascais, Portugal, 2011, pp. 203-216.

[13] MSDN. Windows Filtering Platform. Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463267

[14] VMware. VMware vShield Endpoint. Available: http://www.vmware.com/files/pdf/vmware-vshield-endpoint-ds-en.pdf

[15] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in ACM CCS, 2008, pp. 51-62.

[16] skape and Skywing. (2005). Bypassing PatchGuard on Windows x64. Available: http://www.uninformed.org/?v=3&a=3&t=pdf

[17] M. E. Russinovich, D. A. Solomon, and A. Ionescu, Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition ed.: Microsoft Press, June 17, 2009.

[18] B. D. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in ACSAC, 2007, pp. 385-397.

[19] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in IEEE Sympoisum on Security and Privacy, 2011, pp. 297-312.

[20] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in IEEE Symposium on Security and Privacy, 2012, pp. 586-600.

[21] VMware. Timekeeping in VMware Virtual Machines. Available: http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf

[22] Sourcefire. Clam AntiVirus. Available: http://www.clamav.net/lang/en/

[23] Microsoft. scanner Minifilter Sample. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/ff554758(v=vs.85).aspx

[24] Dazuko. A Stackable Filesystem to Allow Online File Access Control. Available: http://dazuko.dnsalias.org/wiki/index.php/Main_Page

[25] Y. Dong, Z. Yu, and G. Rose, "SR-IOV networking in Xen: architecture, design and implementation," in Proceedings of the First conference on I/O virtualization, 2008, pp. 10-10.

[26] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in ACM CCS, 2007, pp. 128-138.

[27] A. M. Nguyen, N. Schear, H. D. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "Mavmm: Lightweight and purpose built vmm for malware analysis," in ACSAC, 2009, pp. 441-450.

[28] N. E. Rosenblum, G. Cooksey, and B. P. Miller, "Virtual machine-provided context sensitive page mappings," in International Conference on Virtual Execution Environments, 2008, pp. 81-90.