Virtualization-based separation of privilege: working with sensitive data in untrusted environment

I. BURDONOV, A. KOSACHEV, AND P. IAKOVENKO
Institute for System Programming, Moscow, Russia

Contemporary commodity operating systems are too big and do not inspire trust in their security and reliability. Still they are used for processing sensitive data due to vast amount of legacy software and good support for virtually all hardware devices. Common approaches used to ensure sensitive data protection are either too strict or not reliable.

In this article we propose virtualization-based approach for preventing sensitive data leaks from a computer running untrusted commodity OS without sacrificing public network connectivity, computer usability and performance. It's based on separating privileges between two virtual machines: public VM that has unlimited network access and private (isolated) VM that is used for processing sensitive data. Virtual machine monitor uses public VM to provide transparent access to public resources for selected trusted applications running inside private VM on a system call level.

Proposed security architecture allows using one and the same untrusted OS on both virtual machines without need to encrypt any data. However is poses a challenge of enforcing dynamic protection over trusted applications running in potentially compromised OS. We investigate this problem and provide our solution for it.

Categories and Subject Descriptors: D.4.6 [Operating Systems]: Security and Protection; K.6.5 [Management of Computing and Information Systems]: Security and Protection
General Terms: Design, Security
Additional Key Words and Phrases: security architecture, data leak prevention, memory protection, disaggregation, virtualization, virtual machine monitor, VMM, hypervisor

1. INTRODUCTION

Computer and internet technologies constantly evolve their spread into various parts of human life improving accessibility and efficiency of services. The downside of online banking, shopping, e-government and other similar services is that personal data and other private information are exposed to the computer used to access or provide the service. This poses the question of how much we can trust potentially compromised system software in preserving confidential data privacy and integrity. Commodity operating systems are rather big and not sufficiently reliable and secure due to monolithic kernel architecture [Tanenbaum et al. 2006] meaning that exploiting one kernel vulnerability may provide malware with unrestricted access to system resources.

Computer physical isolation from public network (Internet) or even complete network disconnection does help to protect sensitive data leaks but such drastic approach is not always possible and usually not convenient. Typically controlled network access is achieved with the help of personal (application) firewalls that may be used to block network access for all but limited set of trusted applications. However firewalls operate on the same hardware privilege level as OS kernel and malware infecting kernel may bypass them [Tereshkin 2006].

Reliability of privacy-preserving system may be improved if implementing it on a higher privilege level than OS kernel. Thanks to virtualization technology such solution may be realized for legacy OS. Hardware virtualization support in modern x86 processors allows realization of relatively tiny hypervisor (eg. Xen [Barham et al. 2003] size is about 200 KLOC) comparing to the kernel size of contemporary consumer OS. This makes hypervisor a perfect place for security systems since removing kernel from application trusted computing base (TCB) and substituting it with hypervisor would dramatically reduce TCB size.

We present novel approach for protecting privacy of confidential data based on the separation of privileges between two virtual machines (VM): *private* and *public*, - both running commodity untrusted OS. All sensitive data and applications that work with it are located inside private VM. OS in private VM has unrestricted access to these data but cannot transmit it outside because private VM is physically isolated from public network, in fact it may not have network interface at all. In turn public VM has unlimited access to public resources and may be connected to Internet. However OS in public VM may not access sensitive data located in private VM due to isolation of virtual machines enforced by hypervisor. Figure 1 provides architectural view on our approach.

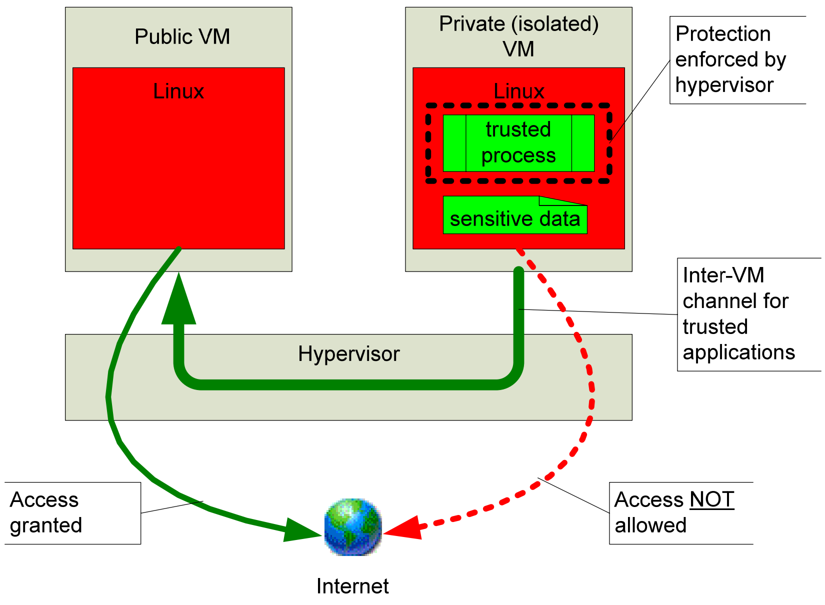

Figure 1. Security architecture based on privilege separation between two VM

Hypervisor maintains *secure channel* between virtual machines and grants access to it to selected applications running in private VM. This channel provides trusted applications a way to access public (Internet) resources which normally are not visible from inside private VM. Hypervisor intercepts system calls executed by trusted processes in private VM and forwards calls related to public resources for servicing to public VM. All other system calls including process and memory management are serviced locally in private VM. The channel may be implemented by sharing memory pages between virtual machines (eg. using Xen grant tables) and restricting writes to these pages from untrusted contexts in private VM, although the particular implementation of such channel is outside the scope of this article.

The advantage of our approach is that sensitive data may be stored in private VM in *non-encrypted* form and processed by legacy software without performance loss caused by on-the-fly encryption. However the only obstacle that blocks kernel-level malware from communicating stolen sensitive data to remote computer is lack of access to inter-VM channel that is the only path to outer world. Access to the channel is granted to trusted processes only so it’s vital to protect execution of trusted process from attacks performed by kernel malware.

In this article we describe hypervisor-based mechanism for protecting process execution in untrusted environment. It watches over process memory and registers and detects attempts to execute malicious code in the context of guarded process before even the first byte of the code will be executed. The presented mechanism is based on the hardware virtualization extensions available in modern x86 family processors. We assume that private VM runs under Linux, however we believe that our approach may be adapted for Windows.

2. GUARDED PROGRAM EXECUTION

Hypervisor realizes protection mechanisms basing on guarded program execution model. In this model program P is considered as an active entity that transforms its states. State S is a snapshot of process virtual memory (containing code, stack, heap, etc) and registers. Program execution is interrupted regularly transferring control to OS kernel to serve interrupt or other system event from the finite set of events E so the overall program execution is comprised of ordered set of non-interruptible slices of execution. The number of slices N varies between program executions due to asynchrony of external interrupts. Program state at the interruption point is called *output state* $S^{(O)}$ while program state at the moment when it resumes execution is called *input state* $S^{(I)}$. Then particular execution of program *P* may be described by a mapping $S_{\mathrm{k}}^{(\mathrm{I})} \to (S_k^{(O)}, E_k), 0 \le k \le \mathrm{N}$ where *k* denotes the index of execution slice. The initial input state $S_0^{(I)}$ is constructed by OS program loader while final output state $S_N^{(O)}$ corresponds to the state in which process calls *exit()* system call or in which process receives signal terminating its execution.

OS kernel behavior may be described by mapping $(S_{\mathrm{k}}^{(\mathrm{O})}, E_k) \to S_{k+1}^{(I)}$ meaning that kernel services event E_k and constructs new input state for the interrupted program. Assuming interrupt is serviced properly relations between current output state $S_k^{(O)}$ and new input state $S_{k+1}^{(I)}$ produced by kernel are well defined and typically depend on the event kind only. The pair of output state and event is used to construct constraint on the next input state. For example if E_k is caused by *read()* system call then differences between $S_k^{(O)}$ and $S_{k+1}^{(I)}$ are allowed only within the provided buffer and contents of *eax* register. Some events like registration of signal handler that adds new valid return (from interrupt) point influence constraints for all subsequent input states. Constraint for the initial input state is constructed from executable file contents basing on loaded segments and starting execution address and externally provided addresses and sizes of heap and stack.

Hypervisor guards process integrity by intercepting interrupt event E_k, analyzing program output state $S_k^{(O)}$ and constructing constraint C_{k+1} for the next input state $S_{k+1}^{(I)}$. Constraints are stored in the hypervisor private memory area. Whenever OS kernel finishes servicing interrupt and transfers control back to the program, hypervisor intercepts this event and validates actual program state against the constraint. Failing this check means that process memory or registers (eg. return address) have been corrupted while it was de-scheduled and program may not be considered as trusted any longer. Hypervisor rejects all subsequent attempts to access public resources (through the channel) made from the context of this process.

The implementation of guarded program execution model poses several challenges. First, hypervisor should be able to intercept all events in the VM that interrupt program execution and intercept control transfers from OS kernel to process after interrupt has been serviced. Second, it should provide efficient mechanism for evaluating and validating constraints. Finally, it should have means of identifying current context inside VM in which public resource is accessed. In next sections we will describe our solution for all these tasks and provide efficient approach for lazy evaluation and validation of constraints.

3. INTERCEPTING EVENTS INSIDE VIRTUAL MACHINE

Three groups of events are of interest to the hypervisor: process interruptions, control transfers from kernel to process and memory accesses both to particular virtual and physical pages. Hypervisor uses hardware virtualization to intercept and process these events.

Process execution is interrupted whenever event that requires attention from OS happens inside VM. It may be hardware or software interrupt, exception or system call. All of these events including system calls that are executed using software interrupts may be trapped by properly filling bit mask describing intercepted events in virtual machine control block (VMCB). VMCB describes VM state and must be provided to CPU upon starting or resuming VM execution. Event is intercepted before process enters kernel mode so hypervisor may immediately take snapshot of memory and registers contents. Hypervisor restricts using fast system call mechanism by intercepting and emulating cpuid, wrmsr and rdmsr instructions.

Hypervisor maps VM physical memory addresses to actual machine addresses using virtual TLB algorithm [Intel 2008]. There exist two copies of pages tables in computer memory – guest OS page tables and *shadow* page tables maintained by hypervisor. Guest OS freely modifies its page tables but attempt to load them into *cr3* register is intercepted by hypervisor and it loads shadow page tables instead. Whenever hypervisor intercepts page fault exception it decides whether deliver it to OS (if there is no valid translation in OS page tables) or update translation in shadow tables.

Hypervisor uses shadow tables to solve several distinct tasks: trap access to specific virtual page, physical page and control transfer from kernel to user code. Trapping memory accesses is required for memory integrity preserving algorithm described below. Whenever process is interrupted hypervisor clears U/S and R/W flags in page directory entries of shadow tables and sets fresh new TLB tag (ASID in AMD's terminology) in VMCB structure. In contrast hypervisor employs one and the same TLB tag for VM execution in user mode. Such way of using TLB tags provides selective flushing of TLB entries that correspond to kernel accesses. This assures trapping kernel memory accesses to every page while minimizing performance impact caused by memory guarding mechanism. Clearing R/W flag is another optimization that helps trapping memory write operations only. U/S flag is cleared to trap execution return to user mode.

4. CONTEXT IDENTIFICATION

The way hypervisor identifies current context depends on whether intercepted event causes or is caused by switching current privilege level (CPL). This leaves us with three cases: CPL switches to user level, CPL switches to kernel level and CPL stays at kernel level.

If CPL switches to user level then hypervisor identifies context in which intercepted event has happened and binds it to certain process inside VM basing on the overall state of VM resources namely virtual memory and registers (particularly *eip* register). Whenever process is scheduled for execution hypervisor iterates through available constraints on input states and validates them against current VM state until it finds a match. If match is found then hypervisor associates currently scheduled process with the trusted process that owns matched constraint and stores trusted context identifier in own memory. Mismatching all constraints means that scheduled process may not be trusted.

Stored context identifier is used to distinguish events happened in trusted context (interruption of trusted process) from untrusted ones. Whenever event happens that switches CPL to kernel level hypervisor queries this identifier and decides on context trustworthiness. Finally event that keeps CPL on kernel level by definition happens in untrusted context.

Hypervisor implements Linux-specific optimization that speedups context identification. When process is scheduled it validates only registers' values and virtual page containing currently executing instruction. Checking other virtual pages is deferred until that pages are accessed by the process. Such reduced check may result in locating several matching constraints. To overcome this issue hypervisor lookups Linux kernel *task_struct* structure, extracts process identifier and uses it as a key into set of constraints. Additionally such help from Linux kernel saves hypervisor from iterating through all available constraints.

Certainly process identifier provided by malicious kernel may not be trusted however hypervisor uses it only as a hint to lookup constraint. If kernel intentionally provides wrong identifier then it will result either in immediate or deferred constraint mismatch. Deferred mismatch may happen if state of scheduled malicious process equals the state of trusted process (whose identifier OS uses to cheat hypervisor) in the part that is validated during reduced check. In this case memory corruption will be detected later on access to page containing malicious code. In any way guarded memory execution mechanism will not be compromised.

5. MEMORY INTEGRITY

Hypervisor preserves memory integrity by detecting illegal changes made to process working memory areas while it was de-scheduled. Working areas are comprised of virtual memory pages that are accessed during program execution. In the majority of cases contents of a virtual page in some input state must be exactly the same as in previous output state; we will describe below cases that legally violate this rule. Constraints that are enforced at initial virtual page access depend on memory area kind. Contents of a page containing code and static data are validated at first access against hash codes calculated beforehand on a separate trusted computer. We require that executable file doesn't contain relocatable symbols so its contents are not patched by the loader. In contrast hypervisor allows heap and stack virtual pages to have arbitrary contents at first access. Although initial stack contents (command line arguments, environment variables, auxiliary vectors) are prepared by the loader which is not trusted we assume that guarded process follows safety programming rules and checks validity of the input data.

Hypervisor maintains two data structures for efficient memory integrity checking: memory integrity table $MIT : VA \rightarrow (HASH,PA)$ and access protection table $APT : PA \rightarrow (ID,VA)$. MIT represents process view on its virtual memory. Whenever process accesses guarded virtual page it must find it in the same form at it left it during previous access. This rule may be fulfilled by assuring any of following two conditions. Either translation of virtual page VA points to the same physical page PA while there were no external writes to PA since last access. Or VA page contains expected contents validated by evaluating page hash and comparing it with stored HASH value. Initially MIT table is filled with hash entries for code and static data pages. Hashing of pages modified during process execution is deferred until kernel decides to swap out the page. Lazy hashing is implemented with the help of APT table which stores modified guarded pages. Whenever hypervisor detects a write operation on physical page PA that is present in APT table it lookups identifier ID of the owning process and updates hash value for the virtual page VA in the corresponding MIT table. Figure 2 shows flowchart for the memory integrity preserving algorithm. State ERROR corresponds to the detection of memory corruption.

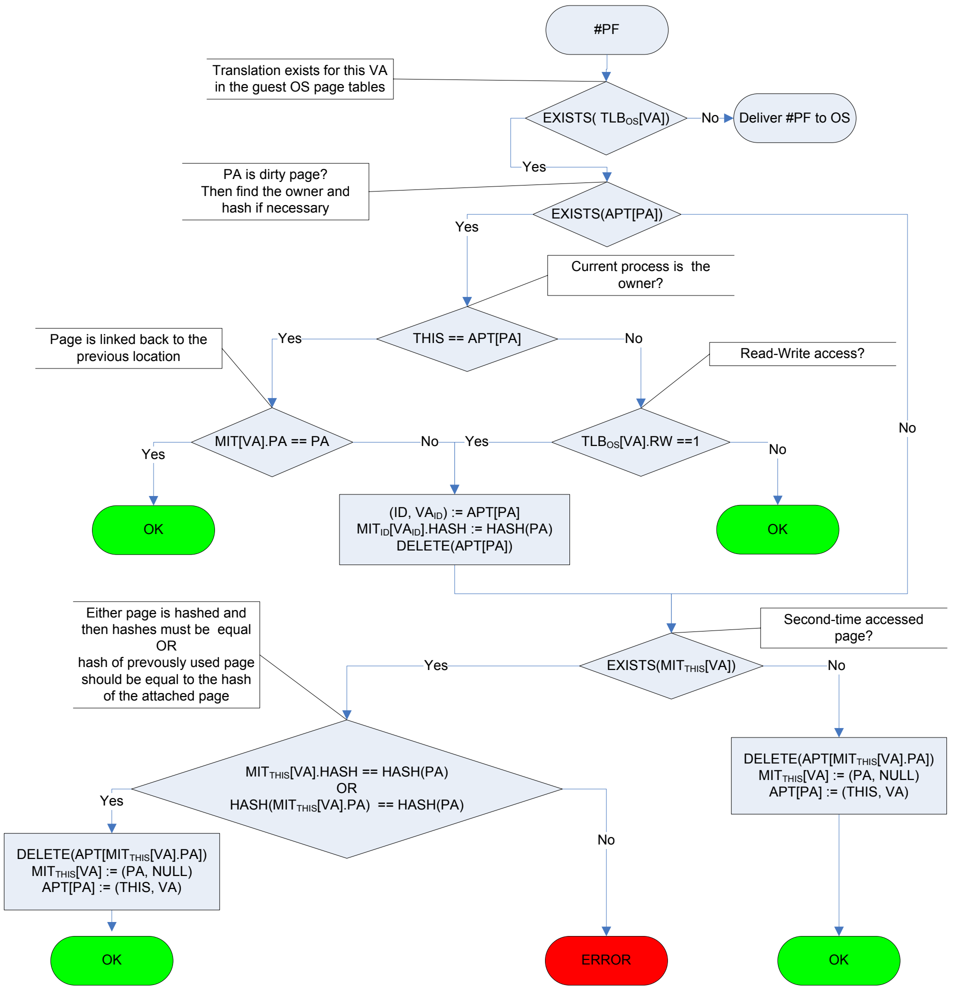
#PF
Translation exists for this VA in the guest OS page tables
EXISTS(TLB_OS[VA])
No
Deliver #PF to OS
Yes
PA is dirty page? Then find the owner and hash if necessary
EXISTS(APT[PA])
Yes
Current process is the owner?
Page is linked back to the previous location
Yes
THIS == APT[PA]
No
Read-Write access?
No
MIT[VA].PA == PA
No
Yes
TLB_OS[VA].RW ==1
No
Yes
OK
(ID, VA_ID) := APT[PA]
MIT_ID[VA_ID].HASH := HASH(PA)
DELETE(APT[PA])
OK
Either page is hashed and then hashes must be equal OR hash of prevously used page should be equal to the hash of the attached page
Second-time accessed page?
Yes
EXISTS(MIT_THIS[VA])
No
MIT_THIS[VA].HASH == HASH(PA) OR HASH(MIT_THIS[VA].PA) == HASH(PA)
DELETE(APT[MIT_THIS[VA].PA])
MIT_THIS[VA] := (PA, NULL)
APT[PA] := (THIS, VA)
Yes
No
DELETE(APT[MIT_THIS[VA].PA])
MIT_THIS[VA] := (PA, NULL)
APT[PA] := (THIS, VA)
OK
ERROR
OK

Figure 2. Flowchart for memory integrity preserving mechanism.

The described algorithm protects memory integrity at page granularity. This is too strict for the cases when kernel legally writes to the user space buffers (eg. passed in *read()* system call) or when kernel invokes process-registered signal handlers which implies saving signal frame on user-space stack. To work around such issues hypervisor provides support for byte-granularity memory protection. Instead of hashing page contents virtual pages that require fine-granular memory protection are guarded by saving exact copies of byte sequences that should stay unchanged within the page. For *read()* system call these are the data blocks between page start and buffer start and between buffer end and page end.

6. RELATED WORKS

Protecting process execution in the commodity potentially compromised OS has been studied in [Chen et al. 2008] and [Yang and Shin 2008]. Both systems maintain two views for trusted process virtual memory: opened and encrypted. Depending on the context in which memory is accessed they present page contents either in original (for the owner) or encrypted (for the kernel) form.

Chen et al. [2008] and Yang and Shin [2008] also provide mechanisms for reliable guest OS process identification and control transfer. Other approaches include tracking changes of hardware virtual address translations [Jones et al. 2006] or accessing guest OS kernel structures that are known to be present at fixed addresses [Onoue et al. 2008]

The idea of improving reliability and security by disaggregating resources between separate address spaces is a keynote of microkernel architecture and it gained new wave of attention with virtualization technology spread. Murray et al. [2008] report on reducing TCB size in Xen by moving domain bootstrap code from Dom0 to separate DomB domain and providing RPC-based communication between them. LeVasseur et al. [2004] restrict driver access to kernel address space by moving driver execution into separate virtual machines. Ta-Min et al. [2006] present Proxos system that isolates application from OS on system call level by executing application code in separate VM which is controlled by trusted OS.

Recent efforts in preventing execution of malicious code include Manitou [Litty and Lie 2006] and NICKLE [Riley et al. 2008] systems. Manitou restricts executing code from unauthenticated memory pages. Before executing instruction from a page it evaluates its hash and compares it against known hashes for this page. Page is considered corrupted if no match is found. NICKLE prevents execution of malicious code in kernel mode by copying authenticated kernel code to shadow memory located inside hypervisor memory area. Every guest OS kernel instruction fetch is transparently redirected to shadow memory which contains authenticated code only.

7. CONCLUSION

We have presented novel approach for preventing leaks of sensitive data based on virtualization technology. Our approach differs from the existing solutions in the reduced set of requirements. We allow computer to be connected to Internet and run commodity untrusted OS while sensitive data may stay unencrypted both on persistent storage and in memory. This allows using legacy applications without performance loss imposed by on-the-fly memory encryption. The implementation challenge posed by this approach is to protect memory and control transfer integrity of the trusted process granted with access to public network. OS controls process execution and may inject malicious code into its address space that will transmit sensitive data to remote computer. We have thoroughly described our solution to this problem optimized for Linux and x86 hardware virtualization.

8. REFERENCES

[1] TANENBAUM, A., HERDER, J., AND BOS, H. 2006. Can we make operating systems reliable and secure? *Computer* 39, 44-51.

[2] TERESHKIN, A. 2006. Rootkits: Attacking Personal Firewalls. In *Proceedings of the Black Hat USA 2006 Conference*, Las Vegas, USA, July 2006.

[3] BARHAM, P., DRAGOVIC B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, USA, 164-177.

[4] INTEL. 2008. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3B: System Programming Guide, Part 2. 358-363.

[5] CHEN, X., GARFINKEL, T., CHRISTOPHER LEWIS, E., SUBRAHMANYAM P., WALDSPURGER C., BONEH, D., DWOSKIN, J., AND PORTS, D. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Seattle, WA, USA, 2-13.

[6] YANG, J., AND SHIN, K. 2008. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. Seattle, WA, USA, 71-80.

[7] MURRAY, D., MILOS, G., AND HAND, S. 2008. Improving Xen Security through Disaggregation. In Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. Seattle, WA, USA, 151-160.

[8] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. 2004. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, San Francisco, CA, USA, 2-2.

[9] TA-MIN, R., LITTY, L., AND LIE D. 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In Proceedings of the 7th symposium on Operating systems design and implementation. Seattle, WA, USA, 279-292.

[10] LITTY, L., AND LIE, D. 2006. Manitou: A Layer Below Approach to Fighting Malware. In Proceedings of the 1st workshop on Architectural and system support for improving software dependability. San Jose, CA, USA, 6-11.

[11] RILEY, R., JIANG, X., AND XU, D. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, Cambridge, MA, USA, 1-20.

[12] JONES, S., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. 2006. Antfarm: Tracking Processes in a Virtual Machine Environment. In Proceedings of the annual conference on USENIX '06 Annual Technical Conference. Boston, MA, USA, 1-1.

[13] ONOUE, K., OYAMA, Y., AND Yonezawa, A. 2008. Control of System Calls from Outside of Virtual Machines. Proceedings of the 2008 ACM symposium on Applied computing. Fortaleza, Ceara, Brazil, 2116-1221.

[14] JIANG, X., WANG, X., AND XU, D. 2007. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In Proceedings of the 14th ACM conference on Computer and communications security. Alexandria, Virginia, USA, 128-138.