# Mitigating the Lying-Endpoint Problem in Virtualized Network Access Frameworks

Ravi Sahita, Uday R. Savagaonkar, Prashant Dewan, and David Durham

Intel Corporation
2111 NE 25th Ave, Hillsboro, OR, USA
{ravi.sahita,uday.r.savagaonkar,prashant.dewan,david.durham}@intel.com
http://www.intel.com

**Abstract.** Malicious root-kits modify the in-memory state of programs executing on an endpoint to hide themselves from security software. Such attacks negatively affect network-based security frameworks that depend on the trustworthiness of endpoint software. In network access control frameworks this issue is called the lying-endpoint problem, where a compromised endpoint spoofs software integrity reports to render the framework untrustworthy. We present a novel architecture called Virtualization-enabled Integrity Services (VIS) to protect the run-time integrity of network-access software in an untrusted environment. We describe the design of a VIS-protected network access stack, and characterize its performance. We show that a network access stack running on an existing operating system can be protected using VIS with less than 5% overhead, even when each network packet causes protection enforcement.

**keywords:** Network Access Framework, Lying Endpoint, Virtualization, Memory Protections.

## 1 Background and Introduction

With increased use of mobile platforms, the effectiveness of perimeter defenses has decreased. Additionally, malicious software has become increasingly stealthier and complex to detect. To counter these threats, a number of network-access control models [1,2] have been proposed. Before allowing an endpoint to connect to a network, these models require it to present its software-integrity information (this is in addition to the existing user authentication). Most integrity-based access models work as shown in Fig. 1. An access-client on the endpoint (a real or virtual machine) requests access to the network, triggering access negotiation with a server. The client and server mutually authenticate each other, and then the access-client sends its integrity posture to the server. This posture information is based on reports from local security services such as firewall, anti-virus, and anti-intrusion. The server uses verification services that match the client security services to ensure that the client posture is acceptable. The server pushes the access decision to the enforcement point, which typically allows, disallows,

A. Clemm, L.Z. Granville, and R. Stadler (Eds.): DSOM 2007, LNCS 4785, pp. 135–146, 2007.

or partially allows the client access. The fundamental weakness of this model is that a compromised lying-endpoint could spoof its posture and gain full access to the network compromising the entire framework.

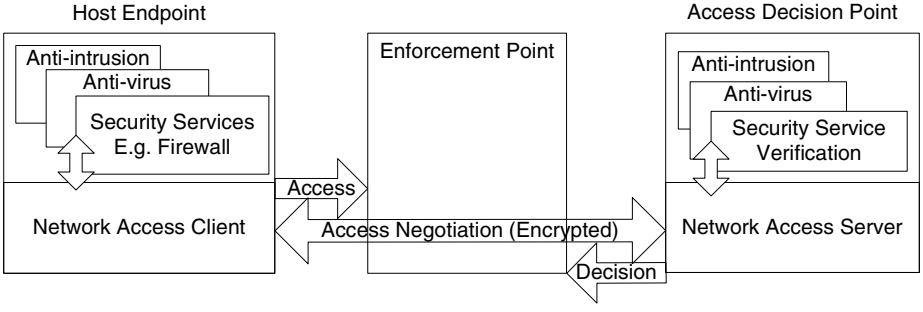


**Fig. 1.** Generalized Network Access Control Model

To this end, we describe a platform service called Virtualization-enabled Integrity Services (VIS) to overlay run-time memory access-control on programs running in an untrusted environment. VIS leverages hardware virtualization to provide memory protections at a page-level granularity. Our approach differs from other virtualization-based techniques [3,4,5] in that, we protect programs within a shared linear address space from peer programs in the same address space, allowing tamper-resistant conversation between a remote server and a VIS-protected access stack, even in un-trusted OS environment. Such trusted communication can then be leveraged to address the lying-endpoint problem. We focus on protection of network access stacks since they are a key attack point [6] for network access frameworks.

The rest of the paper is organized as follows. In Sec. 2, we present related work. Sec. 3 outlines the threat model. Sec. 4 presents the architecture. Sec. 5 presents the application. In Sec. 6, we present threat coverage analysis. In Sec. 7 we evaluate the overhead, followed by conclusion in Sec. 8.

## 2 Related Work

Two fundamental approaches related to our work exist in literature. The first is to verify and load only trusted programs at the highest privilege [7]. Even though this approach can reduce threat exposure, it entails validation of a large body of legacy code against a multitude of run-time vulnerabilities, many of which may not even have been identified. Additionally, any unidentified vulnerability can expose this approach to zero-day attacks. The second approach is to run untrusted code in restricted conditions, and monitor/prevent policy violations. Examples of such include Terra [3], Denali [4], and Tahoma [5], which allow programs to run in isolated Virtual Machines (VMs). Terra leverages properties similar to those provided by Intel® Trusted Execution Technology (TXT) [8]

to create isolated partitions. Denali uses special-purpose VMs crafted for the application. Tahoma uses Dom0 of Xen* to proxy all web queries for browsers running in isolated VMs. However, most untrusted programs are legacy components, and are typically beyond control of the network access stack writers. Thus, in reality, such programs pose significant challenges in terms of making them work in restricted environments. Additionally, running security services in separate VMs is not always possible, since some security services (e.g., intrusion prevention) need some presence on the endpoint to function properly. VIS uses a combination of these two approaches.

A number of other related approaches exist in literature. Engler *et al.* [9] advocate a model where applications manage their own resources with the help of libraries. Witchel *et al.* [10] propose a word-level memory access control matrix. Chen and Morris [11] propose code and data validation for every cache access. Unlike VIS, these approaches require changes to existing hardware and software. Miller *et al.* [12] overlay protection domains on physical memory for capability operating systems that do not support process separation. Bhatkar *et al.* [13] extend address-space randomization techniques to provide probabilistic protections against memory exploits. Arbaugh *et al.* [14] propose a secure bootstrap mechanism similar to TXT. Kiriansky *et al.* [15] use an approach that has similar goals as VIS. They prevent execution of malware by validating code at run-time and putting it in a secure cache. McCune *et al.* [16] propose a minimal code execution model using TXT for running code in isolation.

Our main contributions of this paper are: 1. We describe a novel way to provide inline memory protections to legacy software within a shared address space, without modifying OS infrastructure. 2. We describe a novel method to increase efficiency of protecting software using VIS to mitigate the lying endpoint problem for network access.

## 3 Threat Model

We assume that the attack has compromised the privilege-level separation and installed itself at the highest privilege level in the OS (e.g., see [17]). Consequently, the malware has full access to the OS state. The following types (or combination) of attacks can now be launched to lie about the integrity state of the endpoint.

**Circumvention:** A malicious program can jump into the code of a security service to bypass software checks. For example, a rootkit could prepare a false posture report and jump to the "sign-and-send" function of the network-access client. Examples of such attacks can be found in [6] and [18].

**Tampering:** A malicious program can modify the in-memory contents of a program. For example, a rootkit could modify the conditional branches of a local security service to report a false health report. Examples include Shadow Walker [19] (hooks system handlers and modifies page tables), and tamper of data in transit using function hooking [6].

**Eavesdropping:** A malicious program may read memory containing secrets that are owned by security services. For example, a rootkit could hijack a secure session with the authentication severely simply reading the access-client's memory. Examples of password-stealing malware abound [20].

**Disk-image Modification:** On-disk rootkits, such as Direct Masquerades [21], either modify binaries or completely replace them with malicious binaries.

In Sec. 6 we discuss how each of these threats is mitigated in our proposal.

## 4 Virtualization-Enabled Integrity Services

### 4.1 Software Integrity for Lying-Endpoint Problem

VIS leverages Intel® Virtualization Technology or VT-x [8] to overlay memory protections from the hypervisor onto software running in a VM. In the context of this paper, the network endpoint is running in a VM, and the network-access client and other security services on the client would be protected using VIS. Below we provide a brief overview of hardware-based virtualization, and then describe the VIS architecture in detail. We assume that the hypervisor is a small body of code that is measured on launch thereby reducing the attack surface.

### 4.2 Hardware Virtualization Overview

*Virtualization* refers to the technique of partitioning a machine into Virtual Machines (VMs). Different virtualization techniques have been discussed in literature [4,22,23]. A hypervisor manages VMs by operating at the highest software privilege level (VMX-root mode in VT-x). A control transfer into the hypervisor is called a VMExit and transfer of control to a VM is called a VMEntry. A VM can explicitly force a VMExit by using a VMCall instruction (a hypercall). A Guest OS runs in VMX-non-root mode which ensures that critical OS operations cause a VMExit, which allows the hypervisor to enforce isolation policies. The hypervisor manages launch/shutdown of VMs, memory/device isolation, control register/MSR accesses, interrupts and instruction virtualization.

VIS leverages the hypervisor to control use of physical memory. The hypervisor manages parallel page tables for each VM running on the platform. Each guest OS maintains its own page tables, called the Guest Page Tables (GPTs). The parallel page tables are called the Active Page Tables (APTs) and are used by the processor for address translation. The hypervisor synchronizes APTs with GPTs using a family of algorithms called the Virtual Translation Look-aside Buffer (VTLB) algorithms which emulate the processor TLB. The algorithms leverage VMX-root mode to trap on page-faults and execution of certain instructions such as INVLPG, MOV CR3 that are used by an operating system to manage virtual memory. A discussion of VTLB can be found in [24].

### 4.3 VIS Architecture

VIS comprises of three components–a VIS Registration Module (VRM), an Integrity Measurement Module (IMM), and a Memory Protections Module (MPM).

These modules are isolated from the guest OS. A program (e.g., the access-client) in the guest OS requests protection via a registration hypercall to the VRM. The VRM uses the APT to identify the physical page locations for the program in memory and uses the IMM to verify the integrity of the contents of these pages. If the program passes integrity verification, the VRM uses the MPM to protect the program memory using page-table modifications. Fig. 2 shows these components pictorially.

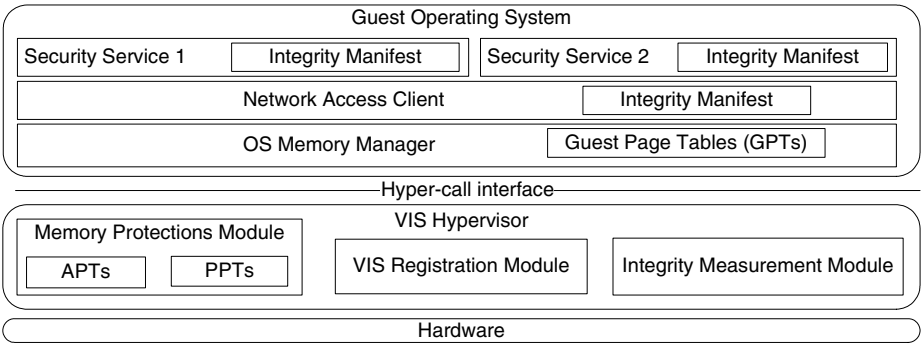


**Fig. 2.** Virtualization-enabled Integrity Services Architecture

**VIS Registration Module:** The VRM is implemented in the hypervisor (see Fig. 2) interacts with programs in the guest OS via hypercall interfaces. It validates the registration requests, coordinates the IMM-MPM interaction, and notifies the program of the registration outcome.

**Integrity Measurement Module:** The IMM is implemented in the VIS hypervisor (see Fig. 2) and has direct access to the guest OS physical memory via page table translations provided by the hypervisor. To facilitate measurement, each program seeking protection is required to provide a vendor-signed reference measurement called an Integrity Manifest or a vendor-signed disk image that can be used to create the Integrity Manifest. The Manifest is cryptographically signed by an entity whose authenticity can be verified by the IMM directly or through a chain of trust. In this respect, the Integrity Manifest is similar to an X.509 certificate. The Manifest contains cryptographic hashes of the program-section's in-memory contents, relocation fix-up information per-section, and exported entry-points per-section. At program load, the OS loader may apply relocation fix-ups to the program image, consequently modifying the in-memory image. The IMM reverses relocation fix-ups using information from the Manifest and with the relocation-reversed copy of the image, can perform meaningful hash verification. Note that, to ensure unrestricted IMM access to the program's memory, VIS requires that the program being protected be pinned in guest OS's physical memory. Most OSes allow such pinning via various application programming interfaces.

**Memory Protections Module:** The MPM is implemented in the hypervisor (see Fig. 2) provides memory protection for each IMM-measured program's code/data sections. As described in Sec. 4.2, the hypervisor maintains shadow APTs corresponding to the OS GPTs. To institute protection on measured physical pages for programs, the MPM creates Protected Page Tables (PPTs) in addition. The MPM populates the PPTs with references to the measured physical pages corresponding to the program's linear address space, and removes references to the program's protected pages from the APTs (or marks them read-only) and flushes the TLBs. All data pages that are mapped to both the APTs and the PPTs are marked execute-disable (XD). Due to this setup, code/data accesses from the APTs to the PPTs or vice versa lead to page-faults that invoke the hypervisor. Depending on the policy defined for the programs, the hypervisor either allows/restricts the access. Fig. 3. shows an example setup for the network access stack from Fig. 2.

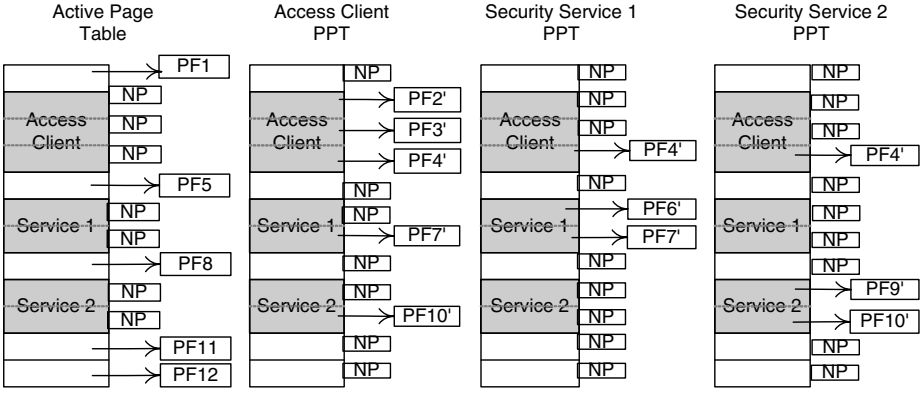


**Fig. 3.** Page Table Setup for a Simple Network Access Stack

## 5 Applying VIS to the Lying-Endpoint Problem

### 5.1 Registration of Access-Client and Security Services

Each access-client and security service program loaded in the guest OS provides the following information in a registration hypercall to the hypervisor: 1. The run-time linear address of the program's integrity manifest. 2. The start and end linear addresses for each program section. 3. The entry points into the programs expressed as offsets into code sections. 4. The type of protection (shared or protected) required for the data sections.

We create a special data section in the access-client and each security service for protected message passing. In the access-client, this section has at least one page for each security service expected. In each security service, this data section must be at least one page long. For example, in Fig. 3, PF 3' and 4' represent the data section pages in the access-client, and PF 7' and 10' represents the

data section in security services 1 and 2 respectively. A protected type will be requested for the data section reserved for communication with the access server.

The MPM verifies that a) the physical memory corresponding to the region is page-aligned and paged-in, b) the physical page is not already protected in another PPT (unless policy-specified), and c) there is no existing alias mapping for that physical frame in any other context. To prevent race conditions, the MPM write-protects the program's pages as program contents are copied for reversing relocation fix-ups. During the measurement phase the MPM does not allow any accesses to occur into the code/data sections. The IMM measures the program as described in Sec. 4.3. If the program passes the IMM check, the MPM creates a new PPT, adds the references for the protected pages into the PPT, and removes them from the APT described in Sec. 4.3. The MPM . To prevent the malware from bypassing registration, the hypervisor disallows any network access until it receives a successful registration and verification of the access-client and a set of security services.

The next phase of registration is to bind the security services to the access-client so that they can communicate securely. A security service does not respond to any access-client request for posture information until it gets a bind success from the hypervisor. Note here that the hypervisor is provisioned with the manifests of the expected security services. After the access-client and the expected security services have been verified, the VRM uses the MPM to associate the security services with the measured access-client. The hypervisor does this by modifying the page table entries corresponding to the reserved data section for the access-client. The hypervisor modifies the page table entries in the access-client PPT to refer to the individual data sections reserved for communication in each of the security service PPTs. This setup is shown in Fig. 3. Page frame pairs 4', 7', and 3', 10' are the protected memory channels from the access-client to security service 1 and security service 2 respectively.

### 5.2 Memory Protected Operation

As noted earlier, the hypervisor grants only restricted network access from the endpoint until it has verified and protected the access-client and the other security services. Additionally, due to the VMExits seen during protected program execution, the hypervisor can also verify that the protected programs are not disabled at run-time. Execution can further be enforced by making the network driver and the memory-mapped network hardware a part of the access stack. In such a setting, a packet must traverse the access stack before it can be sent out onto the network. Alternately, the firewall can cryptographically tag operational data to demonstrate that correct operations were applied to the network packets.

As the packets enter the access stack, execution of the protected sections of the access stack results in a page-fault VMExit. If faulting address corresponds to a valid entry-point into the access stack, the MPM modifies the CR3 register to point to the access-stack's PPT, and resumes execution. Interrupts received during execution of protected stack are handled as follows. On an interrupt, the hypervisor notes down the linear address of the interrupted instruction, switches

CR3 to point to APTs, and jumps into the interrupt service routine (ISR). When the ISR returns, the hypervisor checks the return address against the recorded address, and resumes execution in the protected domain if there is a match.

If the unprotected code attempts to access protected data, the MPM receives a VMExit, and denies/redirects this access. If protected code attempts to access unprotected data, the MPM may map that page to the PPTs with XD permission, and allow the access. Even though such accesses are architecturally allowed, they should be kept to a minimum. Additionally, all posture data exchanged between the security services and access-client should be transferred via the protected data channels setup in the PPT.

The access-client and the security services interact through VIS-enforced entry-points (for code interactions) and protected data channels (for data interactions), and as such, this interaction is tamper-resistant. However, to prevent spoofing, the remote access-server must be able to verify that the access-client is in fact VIS-protected. Such verification can be achieved by requiring the access-client and remote-server maintain a security association using platform-derived keys. These could be stored in the Trusted Platform Module (TPM) on the client, and made available to the hypervisor only if the platform boots with a trusted hypervisor. The hypervisor, in turn, can use these keys to perform crypto operations on behalf of the VIS-protected components. A detailed discussion of a similar model can be found in work by Goldman et *al.* [25].

## 6 Threat Analysis

### 6.1 Threat Mitigation

**Circumvention:** VIS mitigates instruction-flow circumvention by enforcing entry-points into the protected programs. Additionally, protecting the channels between the access-client and services prevents malware from inserting spurious posture data into the protected network-access stack.

**Tampering:** Protected code/data pages belonging to the protected access stack cannot be accessed by external programs, ensuring that malware cannot tamper with the operation of network-access stack.

**Eavesdropping:** Since the protected data of the network-access stack is not mapped into APTs, any access to that data by external entities can be monitored/redirected by VIS. It is responsibility of the security software to ensure that the secrets are not exposed beyond the protected sections.

**Disk-image Modification:** VIS protects a program from tamper after registration. However, on-disk modification may stop the access-client from registering with the hypervisor. However, an unregistered access client would not get access to the platform-derived keys, enabling the access server to detect the attack.

### 6.2 Attack Vectors

VIS does not address hardware attacks from DMA devices, which can largely be addressed by device I/O virtualization [26]. VIS also does not address buffer-overflow

attacks within protected programs. However, a variety of techniques (e.g., XD bit) exist to eliminate such vulnerabilities, and consequently a carefully designed access stack should not suffer from such vulnerabilities. Most commercial OSes force the protected programs to share their stack with rest of system services. However, such an attack can be mitigated by carefully designing the protected program to use private stack/data segment for its critical operations, and verifying the data passed on the stack. Finally if a protected program uses services from an unprotected program, then a malicious program can modify the unprotected program to change the behavior of the protected programs. Similarly, any unprotected data pages accessed by protected programs potentially create an attack vector. VIS cannot inherently protect against such threat vectors. However, for controlled programs such as those in the network-access stack, the interaction with the unprotected programs can be minimized/eliminated. We have a working prototype of network device driver that does not rely on OS services.

## 7 Performance Overhead Analysis

We described how VIS can be used to protect the network-access stack. However, VIS uses VMExits to enforce memory protections, which are extremely CPU-intensive. In this section, we quantify and propose ways to reduce the performance overhead of such protections. Firewalls typically require most frequent protection-domain switching–typically on each packet–and consequently incur the most overhead. Other components of the access stack (e.g., the access-client) typically require much less frequent switching. Hence, we protect a firewall driver that is in the path of the network traffic. Below, we focus only on instruction fetch page faults, as the data-fetch page faults can be eliminated by sharing pages (marked XD) in the PPTs (see Sec. 5.1). We quantified the VIS overhead by measuring the cycles spent by the CPU transferring each byte of network data. A low cycles-per-byte (CPB) indicates an efficient protected program. We derived CPB measurement using NTttcp tests between our prototype (sender) directly connected to an Intel® Xeon™-based receiver. Our test system has an integrated PCIe gigabit Ethernet controller, 1GB of memory, and a VT-x enabled 1.83 GHz Intel® Core™ Solo T1400 processor. We used an internally developed hypervisor and used a Windows* XP* OS. We describe our experiments with software and hardware-assist methods to reduce VIS overhead below.

**Software Assist:** The VMExit cost incurred by an access-stack component can be reduced by mapping the services frequently used by that component into the PPTs. Of course, such services must be mapped read-only in the APTs to protect the access-stack component from code-injection attack. Our firewall frequently called NDIS which results in two VMExits per call. We expanded the scope of the PPT to share NDIS (and other known kernel programs) into the PPT of the firewall service. However, to make NDIS available to other networking programs, it was also mapped to the APTs as a write-protected program. A similar case can

be made for the other security services to reduce the instruction fetch VMExits. As can be seen from Fig. 4, protecting the firewall incurs an overhead of 140 CPB. However, adding other measured components brings the CPB down to less than 90.

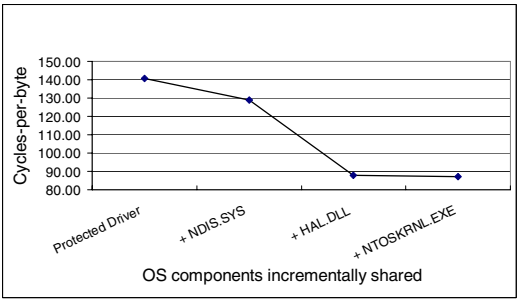


**Fig. 4.** Effect of PPT Expansion on Page-table Transition Cost

**Hardware-assist:** As described in Sec. 4.2, when a VM attempts to modify the value of the CR3 register, it incurs a VMExit. However, VT-x allows the hypervisor to define a "cheat list" of known good CR3 values. If the new CR3 value is included in this list, the VM does not incur a VMExit. We modified our firewall service to leverage this feature. To elaborate, all registered entry-points into the firewall were placed on a read-only page shared between the APTs and the PPTs. The code on this page changes the CR3 value to point to the PPTs, and then calls the protected firewall function. On return from the function, the shared code changes the CR3 to point it back to APTs, and then returns to the caller. The hypervisor is responsible for keeping the CR3 "cheat list" up-to-date with correct addresses of the APTs/PPTs. It should be noted that, to successfully move from APTs to PPTs, the code modifying the CR3 value must be mapped to both the APTs and the PPTs. Consequently, without support from the hypervisor, this feature cannot be attacked by the malware.

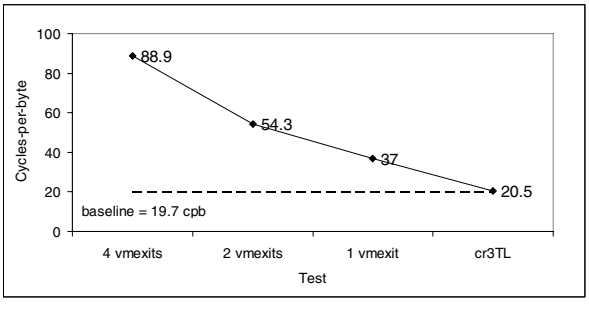


**Fig. 5.** Reducing the Cost of hypervisor Round-trips Using Hardware Assist

Fig. 5 shows the reduction in CPB as the VMExits are reduced. With the use of the cheat list, we experienced no VMExits and recorded a CPB of 20.5 with VIS protections, versus a baseline CPB of 19.7 without any protections (4% overhead). The throughput measured was 715 Mbps (protected) versus 745 Mbps (unprotected). The overhead is due to disabling interrupts at the transition points and the cost of MOV CR3 instructions.

## 8 Conclusion and Future Work

In this paper we demonstrated how virtualization can be used to protect programs operating in a shared linear address space. We used the architecture to protect the network access-client stack to address the lying end-point issue. We implemented the architecture in a hypervisor, and described an OS-independent method to protect software. We outlined software and hardware based techniques to reduce the performance overhead of such protections to negligible levels. For our future work, we are working towards providing case studies of other applications and addressing the attacks not mitigated by VIS.

## References

1. Cisco*: Cisco* Network Admission Control. `http://www.cisco.com/go/nac`
2. Microsoft*: Microsoft* Network Access Protection. `http://www.microsoft.com/nap`
3. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. Special Interest Group on Operating Systems: Operating Systems Review 37, 193–206 (2003)
4. Whitaker, A., Shaw, M., Gribble, S.D.: Scale and performance in the Denali isolation kernel. In: OSDI 2002. Proceedings of the Fifth Symposium on Operating System Design and Implementation, Boston, MA (December 2002)
5. Cox, R.S., Gribble, S.D., Levy, H.M., Hansen, J.G.: A safety-oriented platform for web applications. In: SP 2006. Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06), pp. 350–364. IEEE Computer Society Press, Washington, DC, USA (2006)
6. Thumann, M., Roecher, D.J.: NACATTACK–hacking the cisco* NAC framework. In: Blackhat Europe (2007)
7. Microsoft*: Digital signatures for kernel modules on x64-based systems running windows* vista* (2006), `http://www.microsoft.com/whdc/system/platform/64bit/kmsigning.mspx`
8. Intel Corporation: IA-32 Intel® Architecture Software Developers Manual. `http://www.intel.com/products/processor/manuals/index.htm`
9. Engler, D.R., Kaashoek, M.F., James O'Toole, J.: Exokernel: an operating system architecture for application-level resource management. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pp. 251–266. ACM Press, New York (1995)
10. Witchel, E., Cates, J., Asanović, K.: Mondrian memory protection. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (October 2002)

11. Chen, B., Morris, R.: Certifying program execution with secure processors. In: 9th Workshop on Hot Topics in Operating Systems (2003)
12. Miller, F.W.: Simple memory protection for embedded operating system kernels. In: Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, pp. 299–308. USENIX Association, Berkeley, CA, USA (2002)
13. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: SSYM 2005. Proceedings of the 14th conference on USENIX Security Symposium, pp. 17–17. USENIX Association, Berkeley, CA, USA (2005)
14. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: SP 1997.: Proceedings of the 1997 IEEE Symposium on Security and Privacy, p. 65. IEEE Computer Society Press, Washington, DC, USA (1997)
15. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191–206. USENIX Association, Berkeley, CA, USA (2002)
16. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Seshadri, A.: Minimal TCB code execution. In: IEEE Symposium on Security and Privacy, pp. 267–272. IEEE Computer Society Press, Los Alamitos (2007)
17. Kaslin, K.: Kernel malware: The attack from within. In: Association of anti-Virus Asia Researchers (AVAR) International Conference, New Zealand (2006)
18. Kapoor, A., Sallam, A.: Rootkits part 2: A technical primer (2006), `http://www.mcafee.com/us/local_content/white_papers/wp_rootkits_0407.pdf`
19. Naraine, R.: Shadow walker pushes envelope for stealth rootkits (2005), `http://www.eweek.com/article2/0,1895,1841266,00.asp`
20. Symantec*: Symantec* internet security threat report: Trends for july-dec 2006 (March 2007)
21. Thimbleby, H., Anderson, S., Cairns, P.: A framework for modelling trojans and computer virus infection. The Computer Journal 41(7), 445–458 (1998)
22. Devine, S., Bugnion, E., Rosenblum, M.: Virtualization system including a virtual machine monitor for a computer with a segmented architecture (1998)
23. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 164–177. ACM Press, New York (2003)
24. Smith, J.E., Uhlig, R.: Virtual Machines: Architectures, Implementations, and Applications. In: HOTCHIPS: A Symposium on High Performance Chips (2005)
25. Goldman, K., Perez, R., Sailer, R.: Linking remote attestation to secure tunnel endpoints. Technical Report RC23982, IBM Corporation (June 2006)
26. Hiremane, R.: Intel® Virtualization Technology for Directed I/O (Intel® VT-d). Technology@Intel Magazine 4(10) (May 2007)

*Other names and brands may be claimed as the property of others.