HBSP: A Lightweight Hardware Virtualization Based Framework for Transparent Software Protection in Commodity Operating Systems

Miao Yu, Peijie Yu, Shang Gao, Qian Lin, Min Zhu, Zhengwei Qi

School of Software, Shanghai Jiao Tong University
{superymk,yupiwang,chillygs,linqian,zhumin,qizhwei}@sjtu.edu.cn

***Abstract*—Commodity operating systems are usually large and complex, leading host-based security tools often provide inadequate protection against malware because execution environment for software is untrusted. As a result, most software currently uses various ways to defend malware attacks. However, these approaches not only raise the complexity of the software but also fail to offer an engrained security solution. The focal point in the software protection battle is how to protect effectively versus how to conceal the protector from untrusted OSes. This paper describes a lightweight, transparent and flexible architecture framework called HBSP (Hypervisor Based Software Protector) for software protection. HBSP, which is based on hardware virtualization extension technology such as Intel VT, and by taking advantage of Memory-Hiding strategy, resides completely outside of the target OS environment. Our security analysis and the performance experiment results demonstrate that HBSP effectively protects applications running on unmodified Windows XP, while the total overhead is only 0.25% in average.**

***Index Terms*—Hardware Virtualization, Lightweight Transparent Software Protection, Commodity Operating Systems, Memory-Hiding, HBSP**

I. Introduction

Nowadays, commodity operating systems are deployed every corner in the house, office, and government, managing various commercial software on them. Unfortunately, most of the OSes cannot provide adequate security and software protection due to the design and hardware limitation. Therefore, nearly all the commercial software needs to implement its own additional protection module, which raises the total cost on software development.

Current approaches of software protection can be separated into two categories [1]. One is to implement both static and dynamic code validation through the insertion of objects into the generated executable, such as watermarking and software birthmark [2] [3]. The other, more radical, method is to protect software with hardware supporting [15]. The target program is divided into various parts which run in an encrypted form on secure coprocessors.

Nevertheless, the risk of attacking the target software still exists for the following reasons: first, hardware architecture protection is limited. Hardware architecture defines that the code running in the privileged mode owns system-wide access to the resources, while the code in the user mode can only access a limit range [16]. So once the malicious code or analyzing tools are running in the privileged mode, no more powerful mode can be used to restrict it.

Second, bugs and debugging functions in the OSes are inevitable. The kernel and the 3rd party device drivers of the OS contain millions of lines of code [6]. Meanwhile, nearly all the commodity OSes provide tools to observe other processes' address space and attach a thread to each process for debug use. In such cases, there is no way to stop someone who intends to crack commercial software.

The growing popularity of hardware virtualization motivates our new solution for software protection. Previous efforts address to retrofit a trusted execution environment on commodity system by separating malicious code and system kernel in isolated VMs [9] [10] [11] , or using active monitor to handle security sensitive behaviors [12] [14]. However, these approaches pose a substantial barrier to adoption as their low performance or critical requirement of modification to application code or system kernel.

Our work represents the following contributions:

- A lightweight framework with least temporal and spatial overhead of the communication between the guest OS and hypervisor as our experiment results revealed. The hypervisor implemented on which requires no code modification to the existing OS.
- Implementation of a transparent memory-protecting mechanism which takes advantage of hardware virtualization and Memory-Hiding technology offering protection to hypervisor by memory remapping, thus makes it feasible to conceal the hypervisor in a private memory region.
- Description of the flexibility and extensibility of HBSP which offers a rich set of interfaces for configuration as well as being compatible with other hardware virtualization platform.

The following section presents the design goals of HBSP. In section 3, we explain the framework implementation and the challenges when introducing in Memory-Hiding. Section 4 describes an example to protect software with the help of

HBSP framework in detail. Section 5 shows how we apply our implementation to a default installation of the Windows XP and evaluates the overall performance. Section 6 reviews related work. Finally we will conclude the ideas in section 7.

II. Design Goals

Hardware Enabled Virtualization (HEV) technology can be adopted as a way of software defense [7] [8] [14]. Our motivation is to offer another line of software protection even on an untrusted environment while keeping external and transparent to existing software, as well as being adopted easily. As a result, we build the lightweight HBSP with the following advantages:

Install/Uninstall on the fly The key idea of HBSP is to load the host OS into a VM at runtime. By install/uninstalling hypervisor on the fly, it is much easier to use and debug the hypervisor, while not affecting the legacy OS and application's integrity, as well as the user experience.

Flexible Configuration In order to support various ways to protect software, HBSP supplies a rich set of interfaces to configure Virtual Machine Control Structure (VMCS) and memory strategy. Hypervisor can be configured at both the compile time and the runtime. However, it's not advised to keep all sensitive content in hypervisor for performance reasons.

Support for other HEV technology Currently HBSP supports Intel VT technology. Other technologies such as AMD SVM are also largely adopted in the market. To minimize the cost when refactoring the hypervisor to support these platforms, we build a layer to hide the hardware details. Hence, extending hypervisor platform support will not affect other components.

III. System Implementation

It is challenging to implement the HBSP running on unmodified commodity OSes. In this section, we firstly describe the HBSP architecture, and then introduce its control flow and how it controls the guest OS. The analysis and implementation of Memory-Hiding technology will be discussed at the end of this section.

A. HBSP Architecture

Figure 1 presents the HBSP architecture. Tailored from BluePill Project [4] and migrated on to the x86 platform to expand its usage scope, HBSP is divided into three layers: Platform Related Layer, HBSP Interfaces and 3rd Hypervisor Layer.

Platform Related Layer This layer is used to mask implementation differences among HEV technologies (e.g. HEV-related instructions, checking platform, VMCS configurations, etc) and provide unique interfaces to upper layers. In Platform Related Layer, HBSP requires implementation of HEV technology supporting the following routines, listed in Table 1 (Since HBSP is tailored from BluePill Project,

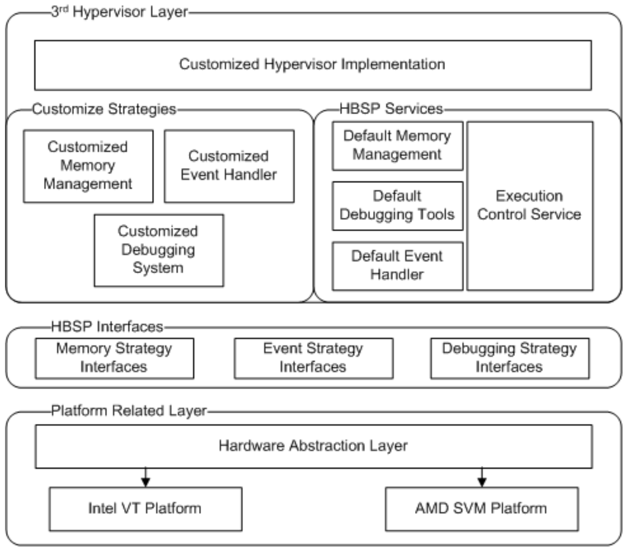
3rd Hypervisor Layer
Customized Hypervisor Implementation
Customize Strategies
Customized Memory Management
Customized Event Handler
Customized Debugging System
HBSP Services
Default Memory Management
Default Debugging Tools
Default Event Handler
Execution Control Service
HBSP Interfaces
Memory Strategy Interfaces
Event Strategy Interfaces
Debugging Strategy Interfaces
Platform Related Layer
Hardware Abstraction Layer
Intel VT Platform
AMD SVM Platform

Fig. 1. **HBSP Architecture** HBSP consists of three layers. User defined hypervisor will be placed in the top layer, with the help of the HBSP Services and the HBSP Interfaces.

TABLE I
The Required Routines in Platform Related Layer.

ArchIsHvmImplemented()	This procedure checks if a hardware platform supports HEV technology.
ArchInitialize()	This procedure takes the responsibility of initializing the hypervisor and guest machine, it can also enter the VMM if needed.
ArchVirtualize()	This routine is responsible for starting the guest machine. As a result, the original OS is put into the virtual machine and continue executing instructions with no sense to the underlying hypervisor.
ArchDispatchEvent()	This function dispatches events to the proper handler. It is invoked under the following condition: the hypervisor is using the Default Event Handler supplied in HBSP, and a #VMEXIT event is occurred in virtual machine.

we keep most of the procedure names and some function implementations).

HBSP Interfaces HBSP exports a set of interfaces called *Strategy* to meet the requirement on the framework's behavior and resource management from hypervisor developer. For instance, with respect to Memory Strategy Interfaces, in order to allocate memory for future use, the Platform Related Layer calls proper functions declared in this set of interfaces. If Memory Hiding Strategy is used as the definition of the current memory strategy, HBSP always uses its private page table to allocate memory for the caller. Besides, by default, the Default Memory Strategy delegates all the memory management tasks to Windows system.

3rd Hypervisor Layer This layer is concentrated on implementing customized hypervisor logic. Building on top of HBSP Interfaces, it supplies services and default HBSP Interfaces implementation to accelerate constructing a hypervisor.

By default, the build-in HBSP Services mainly include:

- Default Memory Management HBSP supports two memory management strategies by default. The Default Memory Strategy uses the Windows kernel API to manage the hypervisor's code and data memory. Thus, any allocation and deallocation to this address space can be detected by other processes. In contrast, the Memory-Hiding Strategy is dedicated to keep the hypervisor out of the view of guest OS; see Section 3.3 for details.
- Default Event Handler This is used to register a callback function with the indicated #VMEXIT Reason. For the performance optimization, each #VMEXIT Reason is linked to an independent event handler chain, cutting down the total time spending in the hypervisor via limiting the length of the chain.

Currently, HBSP is designed to support only one guest machine at the moment, though a typical hypervisor can support more virtual machines as a nature. Regardless the amount of guest VMs, HBSP offers a new approach in protecting software. It brings in a valuable layer of protection, and requires no change to the hardware and the operating system.

B. HBSP Control Flow

Since HBSP is based on HEV technology, it plays a role as guest machine controller between the guest machines and the physical hardware. Considering a single guest machine running on the top layer, at any time, the whole system can be in only one context among the following three types: guest application (Ring 3), guest kernel (Ring 0/1) and hypervisor (VMM). Both the hypervisor interested events and the ones which should not be handled in the guest machine will be transferred to hypervisor and activated. Later, hypervisor transfers control back to the guest machine explicitly after it finishes handling the events.

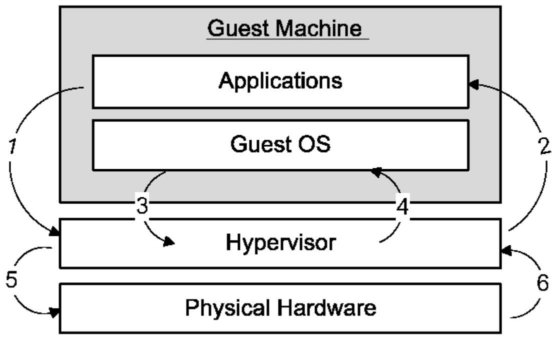

Fig. 2. **Basic State Transition Diagram**

As shown in Figure 2, both guest user mode instructions and guest kernel mode instructions are capable to trigger hardware to generate #VMEXIT events and then transfer control to the hypervisor. For example, when a guest application executes the CPUID instruction, a #VMEXIT event is generated and hardware activates the hypervisor automatically to handle the event (Transition 1). The hypervisor examines the stored guest machine state to determine how to handle it properly, then uses VMX instructions to force hardware to resume the execution of guest machine and transfer the control back to the guest machine (Transition 2).

Guest kernel has more chances to trap into the hypervisor. It is possible the case that a guest machine always triggers #VMEXIT event once to read MSR registers if and only if the hypervisor is configured to monitoring RDMSR instructions with the help of HBSP. Then the hypervisor does the same handling process as that with application (Transition 3, 4).

HBSP also supports intercepting the guest machine on accessing physical resources such as I/O related instructions. A hypervisor mediating between guest machine and physical hardware can always perform additional operations on the I/O access (Transition 5, 6) before resuming to the guest machine. With such approach, a hypervisor can cheat the malicious kernel and software to protect applications.

C. Memory-Hiding Technology

A hypervisor is vulnerable if it can be accessed from the guest OS. To improve the approach of hypervisor based software protection, Memory-Hiding technology is applied to conceal the hypervisor completely. In this section, we firstly describe the conventional transparency limitation, then analyze the implementation of Memory-Hiding technology and show its effectiveness after the hypervisor is turned on.

A typical commodity OS needs to build and manage process page tables for address translation. Consequently, the mapping from the hypervisor's virtual address (VA) to real physical address (PA_{real}) is created as P(VA, PA_{real}) in the system page table, as shown in Figure 3. Being accessible from the guest OS, a hypervisor can be easily invalidated in the face of a malicious kernel.

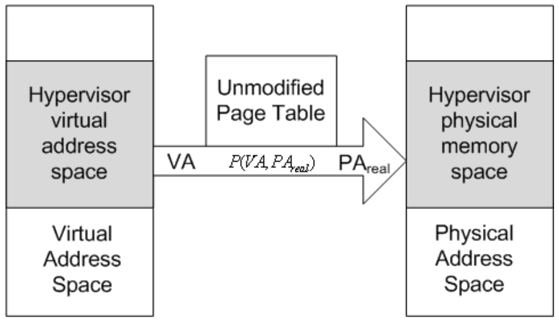

Fig. 3. **Translation without Memory-Hiding.** The space of hypervisor can be accessed even if the hypervisor itself is running under the guest machine.

Memory-Hiding technology patches the indicated PTE in the guest OS's page table. It clones the current page table for private usage, and then changes the mapping from P(VA, PA_{real}) in the hypervisor own address space to P'(VA, PA_{spare}), where PA_{spare} refers to the physical address of a special spare page. This strategy makes all access to the hypervisor be swept out from the patched page table. As shown in Figure 4, when the execution context switches to hypervisor, the private page table takes effect and the hypervisor can reference itself. Once leaving the hypervisor context, the patched page table with mapping of P'(VA, PA_{spare}) will be enabled automatically by hardware, making the hypervisor obscure again.

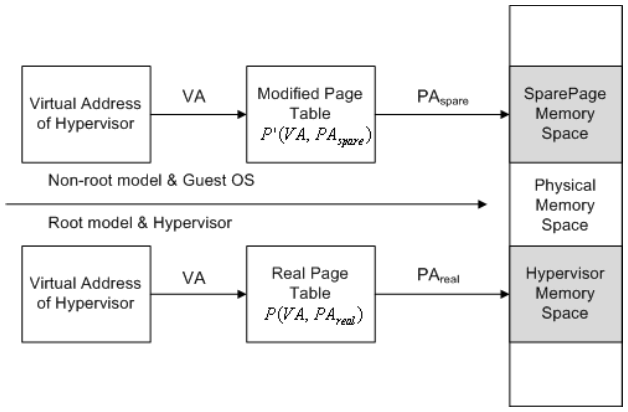

Fig. 4. **Translation Procedure with Memory-Hiding.**

We make no attempt to let the Memory-Hiding technology support PAE mode in current system, though it is a promotional functionality and not hard to implement.

IV. Case Study: Protecting Software with HBSP

In order to verify the efficiency and effectiveness of HBSP, we develop a simple hypervisor called *SNProtector* to store the application's serial key validation algorithm and registration state. Our protection goal is: even the source code of the application field is publicly known, the application still remains protected as long as the SNProtector hypervisor is active.

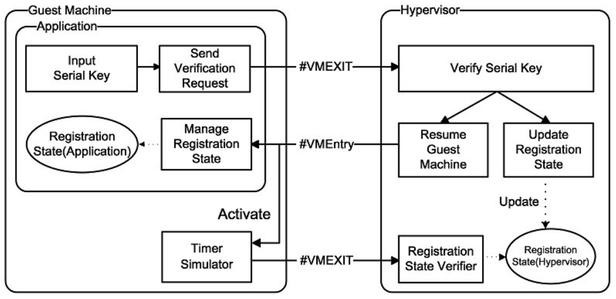
Guest Machine
Application
Input Serial Key
Send Verification Request
#VMEXIT
Hypervisor
Verify Serial Key
Registration State(Application)
Manage Registration State
#VMEntry
Resume Guest Machine
Update Registration State
Update
Activate
Timer Simulator
#VMEXIT
Registration State Verifier
Registration State(Hypervisor)

Fig. 5. **SNProtector Design and Usage Model.**

Figure 5 illustrates the design and usage model of SNProtector. To fulfill the protection goal, it is important to use Memory-Hiding Strategy to conceal its code and data segment and render guest OS imperceptible. Thus, it is hard for an attacker to find out and lock the registration state in the hypervisor space in that the hidden pages are never referenced in the guest OS's virtual space.

```
main:
  // if reqire unload hypervisor
  // Reveal hypervisor then exit
  if( reqRevealHypervisor ) {
      RevealHypervisor();
      exit;
  }

  ReadIn(&UserName,&SerialNumber);

  // Hide hypervisor
  // Pass the reg info into hypervisor
  HideHypervisor();
  bRegState = VerifySN(&UserName,
    &SerialNumber);

  // I am Cracker!!!
  // bRegState = TRUE;

  // Output proper info
  if( bRegState )
      RegSuccessful();
  else
      RegFailure();

  // To simulate a real commercial software
  while( bRegState ) {
      printf("WorkWork!\n");
      Sleep(2000);
  }

  exit;
```

Fig. 6. **The Protected Software Implementation**

It's also vital to realize that the conventional way of protecting software using serial key is weak because it merely verifies the serial key for only once. To address this problem, SNProtector sets up a timer simulator, which will be triggered to verify the registration state in the hypervisor field after a fixed amount of CR3 switches. A better choice of adopting VMX Preemption Timer is omitted here due to the limited hardware support temporarily. Nevertheless, the design reliably renders attackers impossible to find out which instruction exactly causes the timer trap into the SNProtector hypervisor and intercept it on a multi-tasking operating system like Windows.

Figure 6 demonstrates a typical implementation of protected program. Uncommenting the line *"bRegState = TRUE;"* will crack and lock the registration state in the application field, while not affecting that in the hypervisor field. Thus, the tamper behavior is detected immediately as long as the SNProtector hypervisor is running in the background. So our approach is effective even in the condition that the source code of application is public.

Taking performance optimization into account, SNProtector also stores the registration state in the application field to reduces the clock cycles at runtime as the application doesn't need to query the registration state initiatively by means of triggering traps. Although it costs tens of hundreds of cycles

TABLE II
Microbenchmarks. Clock cycles of execution CPUID instruction before and after installing SNProtector.

	Before Loading SNProtector	After Loading SNProtector
Execution Cycle	218	2573

to handle a VMM trap, the overall performance overhead when applying the protection is limited.

V. Experiments and Results

All experiments were conducted on a desktop computer with an Intel Core2 Duo E6320 processor and 2GB RAM. SNProtector is installed on both the processor cores. Windows XP SP3 is selected as the guest operating system of the testbed.

Microbenchmarks Table 2 highlights the results of microbenchmarks that measure the overhead of intercepting instruction execution by SNProtector. In this experiment, the benchmarks exhibited low performance on executing intercepted instructions on guest machine, nearly 11 times more cycles needed to handle the interception and relevant events after loading SNProtector. The reason is that trapping to hypervisor introduces in overhead due to access VMCS region, so does invoking the proper callback function.

Application Benchmarks Although the microbenchmarks show an unacceptable result, the performance impact on the real application is imperceptible. Scaling the performance by measuring the program run time, Figure 7 and 8 present results from the SPEC CPU2006 integer suite and float point suite, illustrating that running the hypervisor only brings in a little overhead. Even the worst individual benchmark suffers less than 0.9% performance overhead.

The web server experiment, measuring the throughput bytes per second (Bps), used the default configuration of APACHE 2.2.11 win32 version. A test website is created with 10 random files, the size of which varies from 1KB to 8MB. The http_load tool was set up on a remote host to generate requests for fetching all of these files with 100 concurrent connections. The client and server were connected by a 100Mbps switch. The overhead of running the SNProtector is 0.55%. Merged with the overall SPEC benchmarks, the total overhead to the guest machine is 0.25% in average.

VI. Related Work

HEV technology enables transparent intercepting guest OS exceptions and interrupts. One example is Overshadow [5], which also can be implemented by HBSP. With the help of additional layer of address translation, it provides different views of the sensitive application's address space according to the current context. Without affecting the existing OS and legacy protected application, even the hardware, kernel and other programs can only get the encrypted content from the protected application's virtual space. Considering only instruction interception at the moment, the performance of SNProtector is much higher than Overshadow comparatively.

Many previous systems have attempted to provide a higher-assurance execution environment by means of building separate VMs. Proxos [9] runs its protected applications in trusted VMs, with the ability of using the resource in the untrusted VM. Similarly, iKernel [10] improves the commodity operating system's reliability and security by isolating the buggy and malicious device drivers running on separate virtual machine. To establish a configurable trust between applications and operating systems, partition system call interface or separation of privilege [11] is utilized to enable secure level code running in different VMs. Hypervisor based monitoring on malicious behaviors [12] [14] is used to handle certain security sensitive instructions. Focusing on the similar design goal, our approach excels at higher performance and less affect to legacy OS kernel on account of the optimized design. Proxos [9] requires code modification to both application and the kernel. While Igor Burdonov's work [11], Ether [12] and Lares [14] pose a high performance penalty and iKernel [10] hasn't range its experiment statistics on temporal overhead test.

Intel's Trusted Execution Technology [13] is another hardware-based software security approach, providing isolated protected execution environment, which offers no privilege to unauthorized software even to observe. Furthermore, it provides protected input and storage channel to ensure the data security. This approach can be regarded as a complement of our Memory-Hiding technology, albeit only available on some special hardware.

VII. Conclusion

In this paper, we have presented the architecture and design of HBSP, which can be used to implement external hypervisor running transparently under the guest OS and intercepting indicated guest machine actions without modifying existing OS and hardware, even software.

Memory-Hiding is entitled by patching the page table of guest OS while holding a real one in the hypervisor. This causes some anti-debug tools and hypervisor detect tools invalid in the face of HBSP. Thus it constructs a safer execution environment for customized hypervisor layer software protector.

As a prototype implementation, we build a simple serial key protector using HBSP. A series of experiments on Windows OS have proven that our approach introduces little overhead to the existed environment. We believe that HBSP is a practical approach to protect software in commodity operating systems.

Acknowledgment

This work is supported by National Natural Science Foundation of China (Grant No.60773093 and 60873209), the Key Program for Basic Research of Shanghai (Grant No.09JC1407900), Microsoft Research Asia Young Teacher

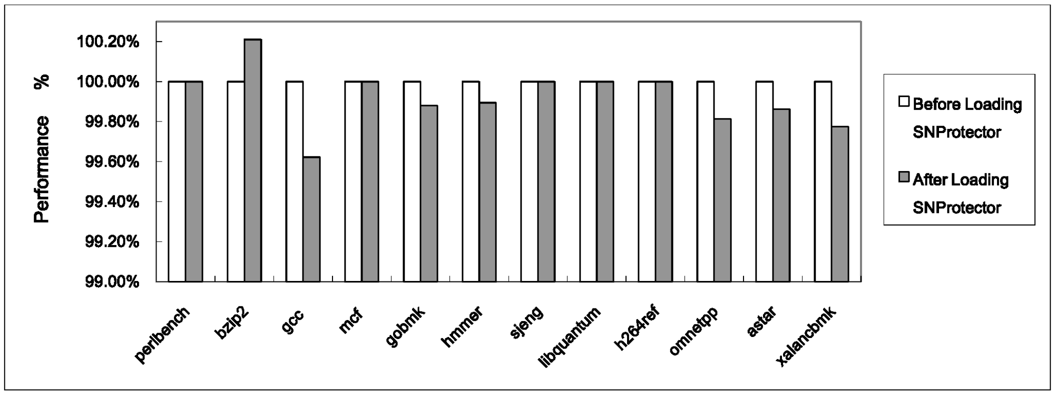

Fig. 7. **SPEC CINT 2006 Benchmarks.**

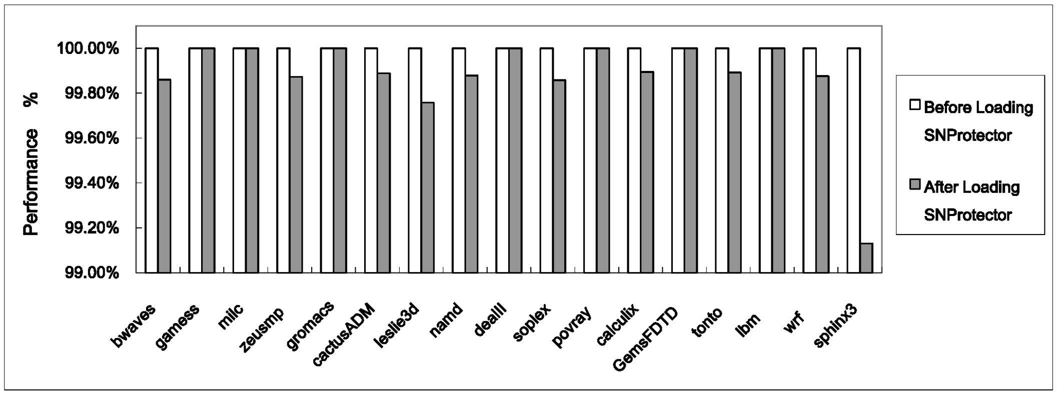

Fig. 8. **SPEC CFP 2006 Benchmarks.**

Funding. We would like to thank the anonymous reviewers for their thoughtful comments and suggestions.

References

[1] Zambreno, J.; Honbo, D.; Choudhary, A.; Simha, R.; Narahari, B.. *High-Performance Software Protection Using Reconfigurable Architectures.* Proceedings of the IEEE Volume 94, Issue 2, Feb. 2006 Page(s):419 - 431.

[2] Xiaoming Zhou; Xingming Sun; Guang Sun; Ying Yang; *A Combined Static and Dynamic Software Birthmark Based on Component Dependence Graph.* Intelligent Information Hiding and Multimedia Signal Processing, 2008. IIHMSP '08 International Conference on 15-17 Aug. 2008 Page(s):1416 - 1421.

[3] Heewan Park; Hyun-il Lim; Seokwoo Choi; Taisook Han. *A Static Java Birthmark Based on Operand Stack Behaviors.* Information Security and Assurance, 2008. ISA 2008. International Conference on 24-26 April 2008 Page(s):133 - 136.

[4] Invisible Things Lab. *BluePill Project.* http://www.bluepillproject.org/stuff/nbp-0.32-public.zip

[5] Xiaoxin Chen; Tal Garfinkel; E. Christopher Lewis; Pratap Subrahmanyam; Carl A. Waldspurger; Dan Boneh; Jeffrey Dwoskin; Dan R.K. Ports. *Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems.* In ASPLOS'08, March 2008.

[6] Nick L. Petroni; PMichael Hicks. *Automated detection of persistent kernel control-flow attacks.* Proceedings of the 14th Computer and communications security, Oct. 2007, Alexandria, Virginia, USA.

[7] Carl A. Waldspurger. *Memory resource management in VMware ESX server.* In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, pages 181-194, December 2002.

[8] M. Nelson; B.H. Lim; G. Hutchins. *Fast Transparent Migration for Virtual Machines.* In Proceedings of the USENIX Annual Technical Conference, pages 391-394, April 2005.

[9] R. Ta-Min; L. Litty; D. Lie. *Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable.* In Proceedings of the Seventh Symposium on Operating Systems Design and Implementation, pages 279-292, November 2006.

[10] PLin Tan; Ellick M. Chan; PReza Farivar; Nevedita Mallick; Jeffrey C. Carlyle; Francis M. David; Roy H. Campbell. *iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support.* In Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing, September 2007.

[11] Igor Burdonov; Alexander Kosachev; Pavel Iakovenko. *Virtualization-based separation of privilege: working with sensitive data in untrusted environment.* In Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems, Mar. 2009.

[12] Artem Dinaburg; Paul Royal; Monirul Sharif; Wenke Lee. *Ether: malware analysis via hardware virtualization extensions.* In Proceedings of the 15th ACM conference on Computer and communications security, Oct. 2008.

[13] Intel. *Intel Trusted Execution Technology Architecture Overview.* September 2006.

[14] Payne, B.D.; Carbone, M.; Sharif, M.; Wenke Lee; *Lares: An Architecture for Secure Active Monitoring Using Virtualization* Security and Privacy, 2008. SP 2008. IEEE Symposium on 18-22 May 2008 Page(s):233 - 247.

[15] Jun Yang; Lan Gao; Youtao Zhang. *Improving Memory Encryption Performance in Secure Processors.* IEEE Trans. Computers (TC) 54(5):630-640 (2005).

[16] Russinovich M.E.; Solomon, D.A. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000.* Microsoft Press (2005).