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 Jiaotong 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 fail to provide adequate protection from malware. The execution environment for software is untrusted. As a result, most software currently uses various ways to defend malware attack. However, those 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 OS. 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 based on SPEC 2006 demonstrate that HBSP effectively protects applications running on unmodified Windows XP, while the total overhead to existing application is only 0.29%.**

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

I. Introduction

Nowadays, commercial operating systems are deployed every corner in the home, office, and government, manage various commercial software on them. Unfortunately, most of the operating systems cannot provide adequate security and software protection due to the design and hardware limitation. Consequently, though it will raise the total cost on software development, nearly all the commercial software needs to implement its own additional protection module.

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] which utilize inherent characteristics of a program to identify it. Another more radical method is to protect software with hardware supporting [19]. The target program is divided into various parts which can be run in an encrypted form on secure coprocessors. Smart card can be viewed as one of the types which could store sensitive computations and data but offer no direct I/O to users [4].

However, 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 [20]. So once the malicious code or analyzing tools are running in the privileged mode, no more powerful mode can be used to stop it.

Second, there are always bugs and debugging functions in the commodity operating systems. The kernel and the 3rd party device drivers of a commodity operating system contain millions of lines of code [7]. Meanwhile nearly all the commercial operating systems provide tools to see other processes' address space and attach a thread to each process for debug use. As a result, there is no way to stop someone who intends to crack commercial software.

Finally, current platform cannot verify the sender of a message. When verifying a serial key of software via Internet connections, the software cannot identify whether the received message comes from the target server. Even, the remote machine cannot distinguish if a verification request comes from the real application or not.

The growing popularity of hardware virtualization motivates our new solution of software protection. Previous efforts address to retrofit a trusted execution environment on commodity system by separating malicious code and system kernel in isolated VMs [12, 13, 14], or using active monitor to handle security sensitive behaviors [15, 17]. However, these approaches pose a substantial barrier to adoption as their low performance or critical requirement of modification to application code or system kernel.

In this paper, we demonstrate a *transparent, external* approach to software protection. Unlike priors, our design requires no code modification of the existing operating systems or adding additional hardware. Even the control flow between the application and the OS kernel remains unchanged. HBSP makes use of Intel VT to create a transparent execution environment instead of employing in-guest components or partial/full system virtualization, for none of these approaches satisfies all transparency requirements. By using hypervisor, applications as well as existing host OS can be put into the

virtual machine transparently on the fly and vice versa when the hypervisor is unloaded. Along with Intel VT, memory-hiding technology makes it feasible for hypervisor to conceal itself in a private memory region. Consequently, the hypervisor is able to run transparently under the commercial operating systems while keeping the ability of monitoring the target application's execution and state.

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.
- Implementation of a transparent memory-protecting mechanism which takes advantage of hardware virtualization and Memory-Hiding technology offering protection to hypervisor by memory remapping.
- Description of the flexibility and extensibility of HBSP Framework 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 framework. 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, as well as how it can prevent cracking by dynamic analysis. 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 [10, 11, 17]. Consequently, HBSP is built to offer another line of software protection even on an untrusted environment. This paper will start with describing its features, and then the limitations not studied yet.

Our motivation is to introduce a new approach in protecting software that could be external and transparent to existing software, as well as being adopted easily. As a result, we build the lightweight HBSP Framework with the following advantages:

Install/Uninstall on the fly Hardware Enabled Virtualization technology advices to start the hypervisor before constructing and powering on each virtual machine. HBSP Framework inverts such process and loads the host OS into a virtual machine. By install/uninstalling hypervisor on the fly, it is much easier to use and debug the hypervisor, while not affecting any bit of the legacy OS and application, as well as the user experience.

Flexible Configuration In order to support various ways to protect software, for example, using *Hardware-Virtual-Machine (HVM)* protecting the application data and OS kernel , HBSP supplies a rich set of interfaces to configure VMCS (Virtual Machine Control Structure) structure and memory strategy. When comes to the high level it offers interfaces to initialize/start/shutdown the hypervisor. However, it's not advised to keep all sensitive data or code 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 parts, as well as the protected software.

III. System Implementation

It's challenging to implement the HBSP running on unmodified commodity operating systems. 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

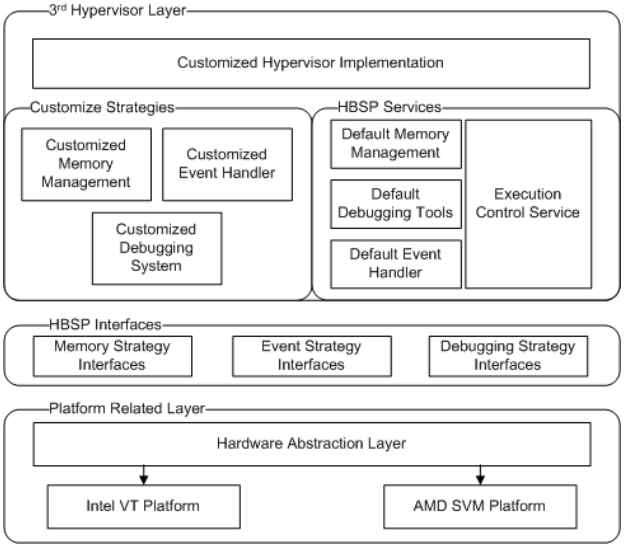

Fig. 1. **HBSP Architecture** HBSP consists of 3 layers. User defined hypervisor will be placed in the top layer, with the help of the HBSP Services and the HBSP Interfaces.

HBSP uses a hierarchical structure, providing a low level abstraction of the HEV technology with a selected set of interfaces. HBSP leverages this mechanism by enabling hypervisor developers to customize the strategy and logic to protect software. Tailored from BluePill Project [5], HBSP doesn't include the anti anti-VMM detection module and runs on the x86 platform to expand its usage scope.

Figure 1 presents the HBSP architecture. It is divided into three layers: Platform Related Layer, HBSP Interfaces and 3rd Hypervisor Layer. In this section we describe all the layers and the challenges to implement them in detail. The demonstration of making use of HBSP will be shown in Section 4.

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.

Platform Related Layer All the platform related implementations are held within this layer. Intel VT and AMD SVM are the most common implementations of HEV technology on x86/x64 platform, whereas there exist differences between them (e.g. HEV-related instructions, checking platform, virtual-machine control structures, etc). Therefore, HBSP requires such a layer being responsible for hiding the differences and provide an HEV technology abstraction to the upper layers.

In Platform Related Layer, HBSP requires any implementation of HEV technology supporting the following routines (as we mentioned before, HBSP is tailored from BluePill Project, so we keep most of the procedure names and some function implementations), listed in Table 1.

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 framework always uses its private page table to allocate memory for the caller. While adopting the Default Memory Strategy as the current memory strategy leads to delegating all the memory management jobs to the 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, such as services for configuring VMCS. Modifying hypervisor configurations and implementing necessary HBSP interfaces are required when substituting the default strategies. 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 uses memory-hiding technology and always preserves a private page table. The contents of the whole hypervisor's address space are hidden from guest OS; see Section 3.3 for details.
- Default Event Handler Default Event Handler is used to register a callback function with the indicated #VMEXIT Reason. When an #VMEXIT event happens, hardware automatically records the #VMEXIT Reason in the VMCS before executing the first instruction in hypervisor. For the performance optimization, each #VMEXIT Reason is linked to an independent event handler chain in HBSP. It cuts down the total time spending in the hypervisor for limiting the length of the chain.

Currently, HBSP framework is designed to support only 1 guest machine at the moment, though a typical hypervisor can support more virtual machines as a nature. Regardless, HBSP framework 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. Consider 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: 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 activate it. Later hypervisor transfers control back to the guest machine explicitly after it finishes handling the events.

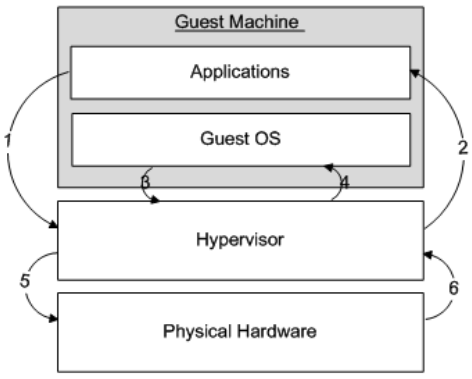

Fig. 2. **Basic State Transition Diagram**

Figure 2 presents the basic state transition diagram. Both guest user mode instructions and guest kernel mode instructions have the capability to cause the hardware generating a #VMEXIT event then transferring control to the hypervisor. For example, when a guest application executing the CPUID instruction, a hardware #VMEXIT event is generated and hardware activates the hypervisor automatically to handle the event (Transition 1). The hypervisor examines the #VMEXIT

reasons as well as the stored guest machine state to determine how to handle it properly. Hypervisor then uses VMX instructions to force the physical hardware to resume the guest machine's execution and transfer the control back to the guest machine (Transition 2).

Guest kernel has more chances to trap into the hypervisor, for example, a guest machine always causes #VMEXIT event once reading 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).

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

C. Memory-Hiding Technology

A hypervisor is vulnerable if 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. With this technical, applications and even the guest OS are unable to access its real content. In this section, we firstly describe the transparency limitation without memory-hiding, and then analyze its implementation. Finally, we will show the effectiveness after the hypervisor is turned on.

A typical commercial OS needs to build and manage process page tables for address translation. Nevertheless, this limits the hypervisor transparency to implement the feature of installing itself on the fly, in that all data and the hypervisor code is loaded by Guest OS. Consequently, the mapping from the hypervisor's virtual address (VA) to real physical address (PA_{real}) is created as P(VA, PA_{real}). Being accessible from the guest OS, a hypervisor can be easily modified and unloaded in the face of a malicious kernel. Figure 3 shows the limitation under this situation.

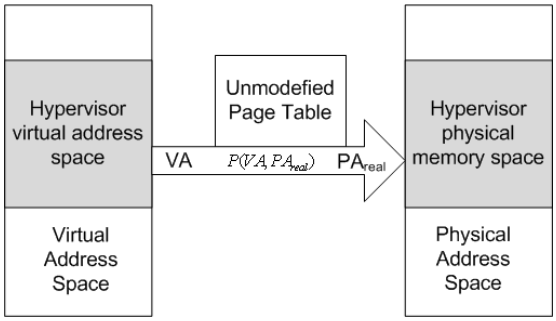

Fig. 3. **Translation without Memory-Hiding.** The space of hypervisor can be accessed even if the hypervisor itself is running under the guest machine.

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's. This strategy makes all access to the hypervisor be swept out from the patched page table. When the execution context switches to hypervisor, the private page table takes effect and the hypervisor can reference itself. As shown in figure 4, 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.

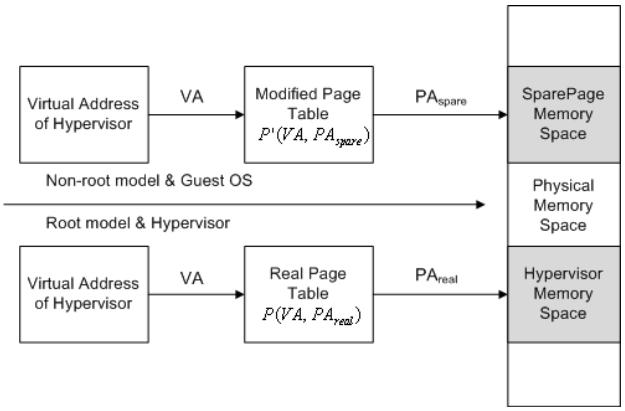

Fig. 4. **Translation Procedure with Memory-Hiding.**

At the moment, we make no attempt to let the memory-hiding technology support PAE mode, though it's a desirable function and not hard to implement. We have not tried to support this in the current system.

IV. Case Study: Protecting Software with HBSP

In order to verify the efficient and effectiveness of HBSP, we develop a simple hypervisor called *SNProtector* to store the application's serial key validation algorithm and registration state. As a consequence, it protects the target software from cracked by dynamic analysis. Our protection goal is: even the source code of the application side is publicly known, the application still remains protected as long as the SNProtector hypervisor is running actively.

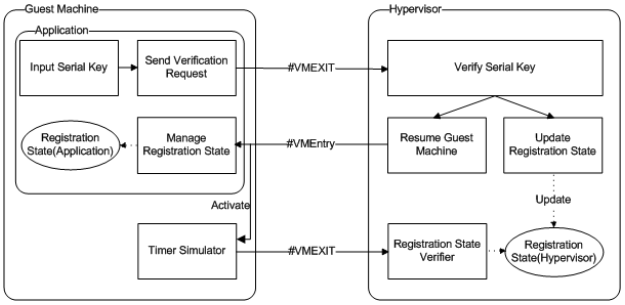

Fig. 5. **SNProtector Design and Usage Model.**

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

	Before Loading SNProtector	After Loading SNProtector
Execution Cycle	268	2660

Figure 5 shows 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, as well as making use of hypercalls causing #VMEXIT event to ensure that the guest machine state is stored before continuing executing any instruction in guest machine or hypervisor. 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.

It's also vital to realize that the conventional way of protecting software using serial key is vulnerable 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 side after a fixed amount of CR3 switches. A better choice of adopting VMX Preemption Timer here is omitted due to the limited hardware support currently. 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-task 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, but not the hypervisor field. Consequently, the cracking 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 application's source code is public.

Taking performance optimization into account, SNProtector also stores the registration state in the application field. This 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 to handle a VMM trap, the overall performance overhead when applying the protection is limited.

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

// Copy Registration Information
// into memory/register
memcpy(passin.UserName,argv[1],4);
memcpy(passin.SerialNumber,argv[2],4);

// Hide Hypervisor and send Hypercall
// pass the registration
// information into hypervisor
__try {
    HideHypervisor();
    bRegState = VerifySN(
        (ULONG)(*((PULONG)passin.UserName)),
        (ULONG)(*((PULONG)passin.SerialNumber))
    );
} __except (EXCEPTION_EXECUTE_HANDLER) {
    printf ("Exception!");
    return 0;
}

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

// Output proper information in the client side.
if( bRegState ) {
    RegSuccessful();
} else {
    RegFailure();
}

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

return 0;
```

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

V. Experiments and Results

All experiments were conducted on a desktop computer with a 1.83GHz Intel Core2 Duo 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 9 times more cycles needed to handle the interception 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 Though the microbenchmarks show an unacceptable result, the performance impact on the real application is imperceptible. 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.

In these tables, program run time is measured to scale the performance. When the individual benchmarks are considered, only GCC has a little higher overhead. This derives from GCC's relatively high system call rates, thus accessing CR3 register more often than other programs, which always causes

trapping into hypervisor.

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%. Combined with the overall SPEC benchmarks, as shown in Figure 9, the total overhead to the guest machine is 0.29% on average.

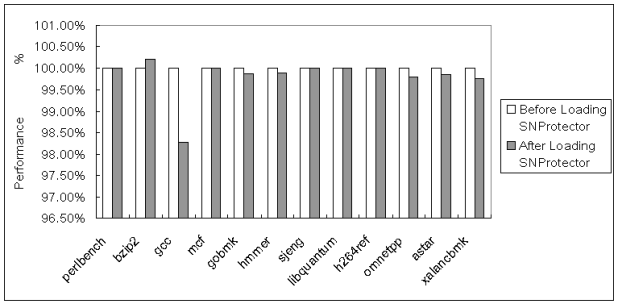

Fig. 7. **SPEC CINT 2006 Benchmarks.**

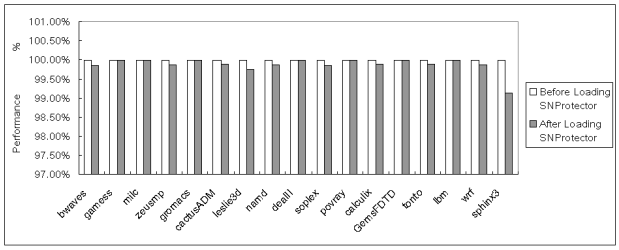

Fig. 8. **SPEC CFP 2006 Benchmarks.**

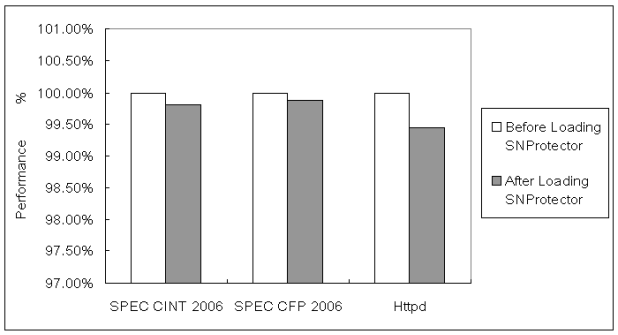

Fig. 9. **Application Benchmark Summary.**

VI. RELATED WORK

Hardware Enabled Virtualization Technology enables transparent intercepting guest OS exceptions and interrupts. This has been leveraged in the New Blue Pill project to intercept external timer source like HPET and Real Time Clock [8]. Also, it enables another step of translating guest physical address into real machine physical address. Systems which employ this ability can be used in virtualizing physical address [9] and transparent page sharing [10] , as well as transparent VM migration among physical machines [11]. Our approach directly modifies the guest OS's page table to provide different views of memory to guest machine and hypervisor.

Another example is Overshadow [6], 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 [12] runs its protected applications in trusted VMs, with the ability of using the resource in the untrusted VM. Similarly, iKernel [13] 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 [14] is utilized to enable secure level code running in different VMs. Hypervisor based monitoring on malicious behaviors [15, 17] 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 due to the optimized design. Proxos [12] requires code modification to both application and the kernel. While Igor Burdonov's work [14], Ether [15] and Lares [17] pose a high performance penalty and iKernel [13] hasn't range its experiment statistics on temporal overhead test.

Intel's Trusted Execution Technology [16] 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. Especially, the hypervisor constructed by HBSP Framework can be install/uninstalled on

the fly and hides its space from the OS view with the help of Memory-Hiding Strategy.

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 a hypervisor layer application or software protector to store sensitive data and codes.

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 brings in little overhead to the existed environment. Still, a series of interesting research opportunities and topics remain for the future study, such as pre-OS Hypervisor Loading and enabling Self-Verification based on Intel TxT technology. We believe it is a practical approach to protect software in commodity operating systems.

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] Devanbu, P.T.; Stubblebine, S.G. *Stack and queue integrity on hostile platforms.* Software Engineering, IEEE Transactions on Volume 28, Issue 1, Jan. 2002 Page(s):100 - 108.

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

[6] 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.

[7] 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.

[8] Invisible Things Lab. *Is GameOver() Anyone?* In Black Hat USA 2007, May 2007.

[9] E. Bugnion; S. Devine; M. Rosenblum. *Disco: Running Commodity Operating Systems on Scalable Multiprocessors.* In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, pages 143-156, October 1997.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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

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