Improving Flask Implementation Using Hardware Assisted In-VM Isolation

Baozeng Ding, Fufeng Yao, Yanjun Wu, Yeping He

Improving Flask Implementation Using Hardware Assisted In-VM Isolation

Baozeng Ding[1,2] Fufeng Yao[1,2] Yanjun Wu[1] Yeping He[1]

1(Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
2(Graduate University, Chinese Academy of Sciences, Beijing 100049, China)
{sploving1, yffbrave, yanjun.wu}@gmail.com yeping@iscas.ac.cn

Abstract. The Flask architecture, which mainly contains object manager (OM) and security server (SS), is widely used to support flexible security policies in operating system. In nature, OM and SS should be isolated from each other to separate decision from enforcement. However, current implementation of Flask, such as SELinux and SEBSD, puts both OM and SS in the same address space. If one component is subverted, the whole system will be exposed to the attacker. In this paper, we present hardware assisted in-VM isolation to improve the security of the Flask implementation. The key of our approach is the separation of SS from other parts of guest OS by constructing hardware assisted page tables at the hypervisor level. In this way SS can execute in a strongly isolated address space with respect to its associated guest OS, and therefore can provide a trustworthy and centralized repository for policy and decision-making. Our experiment shows that our method introduces moderate performance overhead.

Keywords: Flask Architecture, Virtualization, Security, In-VM Isolation, Extended Page Tables

1 Introduction

Since operating system kernels occupy a privileged position in the software stack of computer systems, exploiting kernel-level vulnerability gives attackers completely control of systems. So protecting the integrity of operating system is very important. The Flask architecture [1], which mainly contains object manager (OM) and security sever (SS), is widely used to support flexible security policies in operating system. Various security modules such as SELinux [2], SEBSD [3], have been implemented using the Flask architecture. Since such modules play an important role in the system security, it is important to protect them from being attacked. In nature, OM and SS should be isolated from each other to separate decision from enforcement. However, current implementation of Flask, places both OM and SS in the same address space. If one component is subverted, the security module will be broken and the system will be exposed to the attacker.

As far as we know, there is no effective ways to protect the Flask implementation. Kernel-level integrity protection of Flask is not enough, as these protection mechanisms reside in the same privilege level as the operating system kernel. They could be potentially bypassed or subverted. Virtualization gives us a chance to improve the security of Flask implementation. As the virtual machine monitor (VMM), also known as the hypervisor, exists on the lower level than the kernel, it could be used to detect and prevent the attacks against the Flask. Livewire [4], VMwatcher [5], Lares [6], Patagonix [7] raised to protect the guest kernel integrity utilizing virtualization. These approaches leverage a separate trusted VM to monitor the untrusted VM. Although these out-of-VM approaches have good security effect, they are suitable only for monitoring the events that occur infrequently during system running because switching between VMs causes large overhead. So it's not appropriate for protecting Flask architecture which is used to actively monitor those events occurring frequently.

SIM [8] is an In-VM monitoring framework that puts security tools in the same VM. It utilizes shadow page tables (SPT) mechanism [23] to create a separate virtual address space called SIM address space in the guest VM. It's a one-side view address space and exists in parallel to the system address spaces used by guest kernel. That means the code in the kernel address space cannot visit SIM address space directly but the code in the SIM address space can visit the kernel code. Security monitor is placed in the SIM address space so that it can be isolated from the untrusted kernel, and therefore is protected. It also leverages a feature of Intel VT to reduce the overhead when switching between the system address space and the SIM address space. This way gives SIM the performance benefits and the same security level as out-of-VM method. However, the SPT mechanism that SIM uses will result in expensive VM exits and context switches when synchronizing the shadow page tables with guest page tables [19]. Moreover, the source code structure of the shadow page tables is quite complex and not easy to maintain [19].

In this paper, we utilize Intel Extended Page Table (EPT) technology [21] to design a hardware assisted in-VM framework to isolate SS from the kernel. Isolated SS provides a trustworthy and centralized repository for policy and decision-making. EPT is an Intel hardware virtualization technology that is used to support the virtualization of physical memory. It controls the translation from guest physical addresses to the host physical address. The extended page table is shared by all the process in the guest VM and we call it system EPT which constructs system address space. In order to construct an isolated address space for SS, we create an independent EPT called SS EPT. This EPT maps all the physical addresses that belong to the guest VM, which constructs security server address space. Then we modify the origin system EPT to eliminate the mapping for SS code and data. In this way, the code in the kernel cannot access SS code and data directly. It has to enter security server address space by switching from system EPT to SS EPT to call SS function. We use UnixBench [11] to evaluate our prototype. The result shows the overhead is moderate. As AMD Nested Page Tables (NPT) [24] technology is similar to EPT, our work can be extended to the system with an AMD CPU. To the best of our knowledge, our work is a first step towards constructing in-VM isolation using hardware assisted page tables to improve Flask implementation.

2 Background

2.1 The Flask Architecture Overview

Flask [1] is an operating system security architecture that aims at supporting diverse security policies, as shown in Figure 1. OM is used to enforce security policy decisions returned by SS. Access Vector Cache module (AVC), is used to store the access decisions provided by SS for future use so that it can reduce the performance overhead.

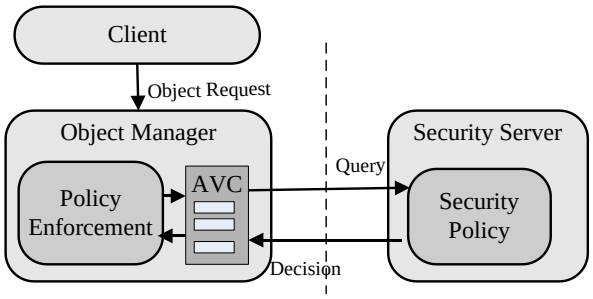

Fig. 1. The Flask architecture

When the system executes an operation, opening a file for example, OM will enforce security policy decision for this operation. OM queries AVC for the access decision. If AVC has stored such rules, it passes the result directly to OM. If no valid entries exist, AVC will send the query to SS. SS will look up the policy database, find the access decision and return it to AVC. AVC saves this decision, and then returns the result to OM.

2.2 EPT Overview

Without virtualization, memory management unit (MMU) is used to translate virtual addresses (VA) to physical addresses (PA) using page tables. But in a virtualized environment, the hypervisor takes full control of the host memory which is shared among virtual machines. So the guest VM can't see PA and guest page tables can't be used directly. In order to abstract the physical memory, the hypervisor presents guest physical addresses (GPA) to VM and maintains the mapping from GPA to host physical addresses (HPA). This gives the VM an illusion that it owns the actual physical memory.

EPT mechanism [21] is an Intel hardware virtualization technology that can be used to support the virtualization of physical memory. It is used to map GPA to HPA. Figure 2 shows the address translating process using EPT technology. First, the guest page tables pointed by CR3 register are used to translate VA to GPA. Then the EPT

pointed by EPT base pointer is used to translate GPA to HPA. EPT has a structure of four level page tables. The top level is page map level 4 table, the second level is page directory pointer table, the third is page directory table and the last is page table.

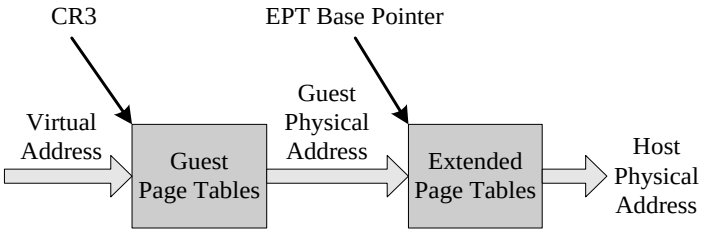

Fig. 2. Address translation using EPT

3 Assumption and Threat Model

Our framework is based on the following security assumptions. First, we assume that the operating system runs on a virtualization environment. Second, we assume that the underlying hypervisor is trusted. Recent hypervisor rootkits [13-14] make the hypervisor insecure. There are already some research such as HyperGuard [15], HyperCheck [16], HyperSentry [17] and HyperSafe [18] which aim at protecting the integrity of the hypervisor. In our prototype, we assume that the attacker cannot control the hypervisor to do malicious things. Especially the attacker cannot manipulate the EPT maintained by the hypervisor. Third, we assume OM hooks can be protected so that they cannot be removed or changed maliciously to bypass the security checks enforced by SS. Protecting these hooks are out of the scope for our work. It can be done using other research work like HookSafe [12]. In this way, we can guarantee the security operation cannot be bypassed. Finally, the boot of guest VM is trusted and the guest is clean before we enable our setting.

Our threat model allows an attacker to gain access to the kernel address space and to carry out the following type of attack:

1. **SS code attack:**
   The attacker can modify SS code or inject malicious code to make SS not functional.
2. **SS data attack:**
   The attacker may carry out control flow attack targeting SS stack or heap data. For example, stack buffer overflow or heap overflow attack. Also, non-control data attack is possible. It is pointers out that non-control data, like global variable, can be exploited to launch attack [22]. For example, a Boolean variable in SS that used to determine whether authentication is passed or not can be flipped to circumvent the

authentication. Moreover, the attacker may take control of AVC entries and security policy data which may cause the Flask architecture not work as expected.

4 Overview of Our Approach

In order to make SS trustworthy to provide a centralized repository for policy and decision making, we isolate SS from kernel, including OM and other parts of the kernel, using hardware assisted in-VM isolation technique, as shown in Figure 3.

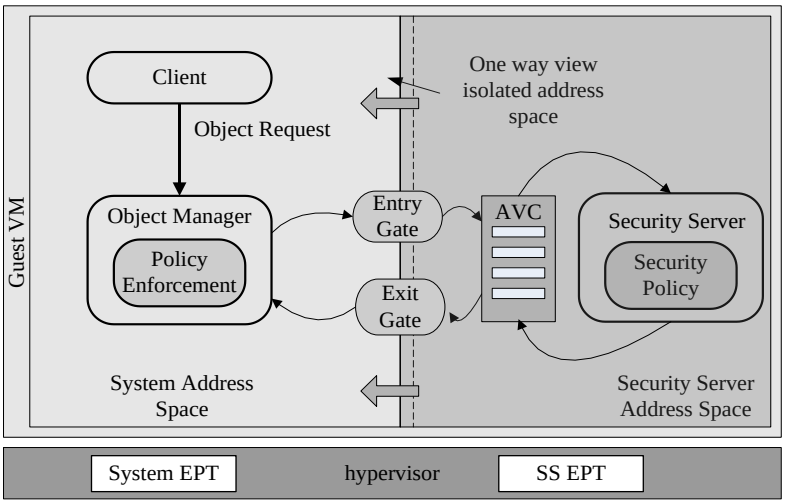

Fig. 3. Hardware assisted in-VM isolated Flask framework

We introduce a security server address space that is independent from and invisible to the system address space used by the operating system. We place SS together with AVC in it, as AVC is subverted, it makes no sense to isolate and protect SS. For clarity, when we refer to SS in the following section, it includes SS and AVC together. We create security server address space by constructing an independent EPT, called SS EPT, at the hypervisor level. This EPT is a synchronized copy of system EPT and is only used by SS for address translation. At the same time we remove the mapping to SS in the original system EPT. In this way, the code in the system address space cannot view and access SS code and data directly.

The entry and exit gates are used to switch address spaces by loading different EPTs. When the kernel calls SS function, it uses the entry gates to enter security server address space by switching from system EPT to SS EPT. When the call is finished, it uses the exit gates to return to system address space by switching from SS EPT to system EPT. The entry and exit gates also switch the stacks as SS need have its own to prevent attacks against stack data. The entry gates perform the transfer out of system stack into security server stack. The exit gates perform the switch from security

server stack to system stack. The entry and exit gates are the only regions mapped into both address spaces so that a transfer between system address space and security server address space can only happen through code contained in these regions. They are write-protected so that their contents cannot be modified by any code in the kernel.

5 Implementation

In our prototype, we use Xen as the hypervisor and SELinux in the HVM DomU with 2.6.32 Linux kernel as the implementation of the flask architecture.

First, we construct an independent address space for SS by creating a separate EPT. Then we create entry and exit gates to achieve the switching between two EPTs. At last, we load a kernel module to take all the settings into effect.

5.1 Isolated Address Space Construction

Generally, SELinux which includes OM and SS is compiled into the kernel. SS code and kernel code may be sharing one physical page. SS data is similar. In order to isolate SS from other parts of the kernel, we need modify the Linux kernel linker script file to guarantee SS code and data are page aligned. So they are not mixed with code and data from the kernel. Take x86 architecture for example, when the kernel is compiled, a link script file called "vmlinux.lds" is generated according to the source file "vmlinux.lds.S". It's used to control the kernel linking. Here we modify its source file so that the changes will take effect every time vmlinux.lds is generated. In vmlinux.lds.S, we make SS code and data page align and add four indicators to record the start and end address of SS code and data. After the kernel building is completed, we can lookup System.map table to find the values of indicators to see whether the change is applied successfully.

After that, in order to construct an EPT that is only used by SS to execute address translation, we add a hypercall named "ept_create". It is invoked only once. After guest VM is booted up, we load a kernel module to notify the hypervisor to invoke this function. It completes the following two tasks.

First, this hypercall creates a new EPT. For clarity, we would refer the EPT used for guest VM as system EPT, and refer the EPT we create for SS as SS EPT. This hypercall uses the system EPT as a template. Since EPT has a structure of four level page tables, we need allocate the physical pages for them and set their entries. First of all, the hypercall allocates a physical page as the new top level page table for SS EPT. Then it checks the top level page table entries of system EPT. For each valid entry that points to a second level page table of system EPT, the top level page table of SS EPT will be set the corresponding entry to point to a new page allocated as a second level page table of SS EPT. As for the access permission of the entry, the hypercall will copy it from the top level page table entry of system EPT. We do the similar operations for the second and the third page table of SS EPT. The last level page table is dealt with differently. As for the page table entries, we assign them exactly the

same value as those in system EPT. When all is done, we get an independent EPT which has its own top level, second level, third level and last level page tables. But it maps the same physical memory as system EPT.

Second, the hypercall changes access permissions in system EPT. In order to isolate SS from the guest OS, SS code and data can only be seen and used in the security server address space. So the hypercall removes the mapping for them from system EPT. The hypercall "ept_create" takes four parameters when it's called. Two of them represent start and end of guest frame number of SS code section. The other two are for SS data section. For each guest frame number of code section, we find the entry of page table in system EPT and make it invalid. For SS data section, we do the same setting.

5.2 Entry Gate and Exit Gate Implementation

We use entry and exit gates to switch the address spaces. First, in order to implement the switching between system EPT and SS EPT, we need to define another hypercall named "ept_switch". The main task of entry and exit gates is issuing this hypercall to switch EPTs. This hypercall is designed to take an integer as parameter to indicate switching direction. If we want to enter SS to execute code, the physical address of the top level page table of SS EPT will be written to a control field in virtual machine control structure. If we want to exit SS and return to the kernel, the physical address of the top level page table of system EPT will be written to that field. Both the above physical addresses are stored in a structure of the hypervisor, so they cannot be obtained and operated by attackers.

In order to prevent the execution from diverting to somewhere else due to interrupt, we disable interrupts at the beginning of entry gates and enable interrupts at the end of exit gates. Moreover, we also need switch the stacks in the entry and exit gates. In other words, the security server address space has its own stack. The entry gates perform the transfer out of the stack in the system address space into the stack in the security server address space. The exit gates do the opposite thing. In this way, SS stack is not viewed and accessed by the kernel code, and therefore our framework can defend attacks against stack data.

There is still one detail we need to deal with. After SELinux completes the initialization, the entry and exit gates should begin to work. But we don't have the SS EPT created at the moment, so the entry and exit gates actually do nothing before the "ept_create" hypercall is issued. We define a global integer variable H in the kernel to illustrate this condition. Before SS EPT is constructed, we assign zero to H. The entry and exit gates will read this value and do nothing. After creating SS EPT, we assign one to H. The code in both gates will perceive the change and begin to work.

5.3 Loading a Module to Enforce the Settings

The last step is to load a kernel module to take the entire configuration into effect. The module does the following two things.

First it will issue the "ept_create" hypercall to notify the hypervisor to create SS EPT. The four parameters of this function can be obtained through simple calculation. In section 5.1, we define four indicators, which are saved in the System.map file. They record the start and end address of SS code and data. The addresses are all virtual addresses. But the kernel lies in the direct mapping area, so we can get physical address by subtracting a fixed offset from virtual address. Then we can get guest frame numbers that the hypercall needs by shifting right the physical address a page size. When this hypercall is called successfully, SS EPT is created. Second, the module will set the global variable H, which is described in 5.2, to be one. The entry and exit gates check this variable and start to work.

Since the global variable H is in the system address space, attackers may change its value to be zero at run time. However, if it is set to be zero, the entry gates and exit gates will do nothing, and the guest VM can only use system EPT. In this way OM cannot find SS when it calls SS functions, as the system EPT has no mapping to them. So it returns a segmentation fault and terminates current process.

6 Evaluation

We have implemented our prototype on a DELL desktop with Intel Core i5-650 processor, which supports Intel VT technology. We used Xen 4.0.1 as the hypervisor. The host OS is 32-bit Ubuntu 10.04. So is the guest VM. It's booted up as HVM DomU with one VCPU and 512M RAM. We evaluate our work from the aspects of performance and security.

6.1 Performance Overhead

As many security hooks exist in the kernel and they are frequently called, we should pay attention to the performance overhead. In our experiments, we used UnixBench [11] to evaluate the system performance. UnixBench provides a basic reference of the performance of a Unix-like system. We tested the performance of the guest VM in two scenarios: one was with SELinux enabled natively and the other one was our prototype that used hardware assisted in-VM isolation.

First we ran UnixBench in DomU with SELinux enabled natively. It means SS and OM are not isolated from each other and both of them are in the kernel address space. We used the results as the base of our test. Then we ran UnixBench again using our prototype. SS is isolated from OM and it is in its own address space. As is shown in Figure 4, we divided the latter results by the formal ones and expressed the final results as percentages. Each result is the average of 5 trials. Most of them demonstrate that the overhead of our prototype is small except for the shell script test. This test made a process start eight concurrent copies of a shell script which did operations to a data file. SELinux would check process creation, shell script running and file access for each copy, so the performance went down due to frequent switching of EPTs.

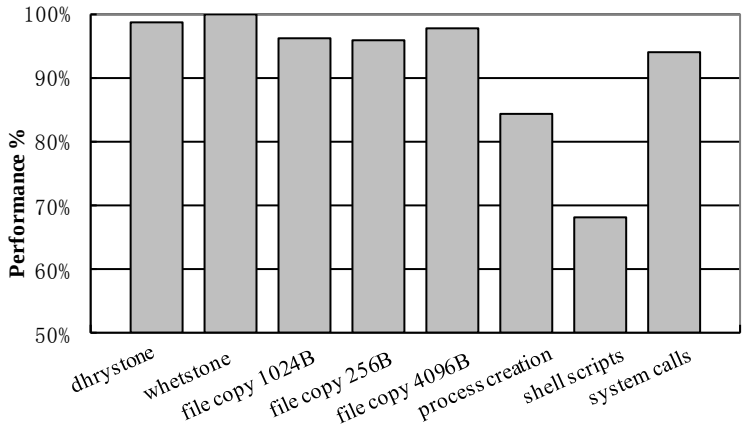

Fig. 4. Performance Overhead

Besides, as we put AVC in the security server address space to improve security, every time OM queries AVC for the access rules, it would switch to the security server space. This is a large part of performance overhead. We can improve the performance by moving AVC out of the security server address space but we need extra effort to protect it from being attacked.

6.2 Security Analysis

One important design consideration is to stop attacks against SS code and data. It is easy to protect SS code. For example, we can use write-protected technology or attestation technology to protect SS code. However, it is nontrivial to protect SS data. It includes dynamic data that may cause control flow attack. For instance, the heap overflow can be used to modify function pointers and stack buffer overflow can be used to change return addresses. Our approach can prevent such attack, as SS has its own address space, including its own heap and stack. Attackers may also launch non-control-data attack [22] to bypass the authentication. Since we isolate SS strongly from other parts of the kernel, the attacker who even obtains the kernel privilege cannot see SS non-control data. So our approach can prevent such attack too. As we put AVC and security policy data in the security server address space, it is also very difficult for the attacker to change them maliciously.

Since SS can load policy files into memory at runtime, attackers may breach the security of SS by loading an invalid policy. To prevent this we can maintain a white list which stores valid hash values of policy files. When SS loads a policy file, we can check whether the hash value of this file is in the list. If it is, the file is allowed to be loaded into the memory. Otherwise, SS refuses to load the policy.

7 Related Work

Our work is related to projects that use secure monitors to protect the integrity of OS kernels. Livewire [4] is a VMM-based prototype that aims at protecting the guest VM kernel code and critical data structures from being modified. VMwatcher [5] is a VMM-based malware detection approach, which moves the anti-malware facilities out of the monitored VM in order to achieve stronger tamper-resistance. Lares [6] is an architecture that places hooks inside the guest VM that can invoke a security application residing in a separated VM. Patagonix [7] is a hypervisor-based prototype that detects and identifies covertly executing binaries without making assumptions about the OS kernel. All these approaches are out-of-VM monitors which are inappropriate for Flask as it is a fine-grained secure architecture that the hooks are called frequently during system execution. They would incur heavy overhead due to frequent switching between the guest VMs and the hypervisor.

SIM [8] is an in-VM monitor which places the security tools in the same VM and uses the hardware virtualization technology to avoid the hypervisor involvement when switching address spaces. It is more efficient than the out- of-VM approaches, and also guarantees the security. However, the shadow paging mechanism it uses is quite complex and not easy to maintain. So we design a hardware assisted in-VM isolated framework that takes advantage of Intel EPT technology to improve the Flask implementation.

8 Conclusions and Future Work

In this work, we improve the Flask implementation using hardware assisted in-VM isolation. We introduce a separated EPT in the hypervisor to construct an address space for SS so that we can isolate SS from other parts of the kernel. Since all the improving is done at the hypervisor level, the attacker cannot subvert them at the guest VM level. Our approach only works with a Type I VMM, which is implemented directly on the physical hardware, since we need the lower level VMM to protect guest VMs running on it. Our work is a first step towards creating in-VM isolation using hardware assisted page tables to improve Flask implementation. It can be applied to other security tools, such as integrity measurement agent, instruction detection tools and so on.

Our future work is to protect OM, especially the hooks used by OM to call AVC and SS functions. If they are bypassed, SS will have no effect at all. Only using write-protect method to protect them is not enough and we will utilize the research results of HookSafe [12] to protect the hooks from being hijacked.

Acknowledgments. Our work is supported by the National Natural Science Foundation of China under Grant No. 90818012, National Science and Technology Major Project No.2010ZX01036-001-002, 2010ZX01037-001-002, and the Knowledge Innovation Key Directional Program of Chinese Academy of Sciences under Grant No.KGCX2-YW-174, No.KGCX2-YW-125.

References

1. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: system support for diverse security policies. In Proceedings of the 8th USENIX Security Symposium, pages 123-139, Washington, DC, (1999)
2. Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, (2001)
3. C. Vance, R. Watson. Security Enhanced BSD. Network Associates Laboratories, (2003).
4. T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Proceedings of the 10th Annual Network and Distributed Systems Security Symposium, (2003)
5. X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In Proceedings of the 14th ACM Conference on Computer and Communications Security, pages 128-138, (2007)
6. B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In Proceedings of the 29th IEEE Symposiumon Security and Privacy, pages 233–247, (2008)
7. L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In Proceedings of the 17th USENIX Security Symposium, pages 243–258, (2008)
8. M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In Proceedings of the 16th ACM Conference on Computer and Communications Security, pages 477–487, (2009)
9. Intel Virtualization Technology. http: //www.intel.com/technology/virtualization
10. Inel Inc. Intel 64 and IA-32 Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 1, (2006)
11. Tux.Org (1996). http://www.tux.org/pub/tux/benchmarks/System/unixbench
12. Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In Proceedings of the 16th ACM Conference on Computer and Communications Security, pages 545-554, (2009)
13. The Blue Pill Project. http://bluepillproject.org/
14. S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: implementing malware with virtual machines. In Proceedings of the 2006 IEEE Symposium on Security and Privacy, (2006)
15. R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy. In Black Hat conference, (2008)
16. J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A hardware-assisted integrity monitor. In Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection, (2010)
17. A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In Proceedings of the 17th ACM Conference on Computer and Communications Security, pages 38-49, (2010)
18. Z. Wang, and X. Jiang, HyperSafe, a lightweight approach to provide lifetime hypervisor control-flow integrity. In Proceedings of the IEEE Symposium on Security and Privacy, (2010)
19. X.Wang, J.Zang,Z. Wang,Y.Luo, and X.Li. Selective hardware/software memory virtualization. In proceedings of the 7th ACM Conference on Virtual Execution Environments, pages 217–226, (2011)

20. S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent, 6397242, (1998)
21. Intel Inc. Intel 64 and IA-32 Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 2, (2007)
22. C. Shuo, J. Xu, E. C. Sezer, P. Gauriar, and R. K. lyer. Non-control-data attacks are realistic threats. In Proceedings of the 14th conference on USENIX Security Symposium, (2005)
23. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In Proceedings of the Symposium on Operating System Principles, (2003)
24. Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.12 edition, (2006)