RandHyp: Preventing Attacks via Xen Hypercall Interface

Feifei Wang, Ping Chen, Bing Mao, and Li Xie

State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University

Abstract. Virtualization plays a key role in constructing cloud environments and providing services. Although the main jobs of the hypervisors are to guarantee proper isolation between domains and provide them services, the hypercall interface provided by the hypervisor for cross-layer interactions with domains gives attackers the possibility to breach the isolation or cause denial of service from inside the domains. In this paper, we propose a transparent approach that uses randomization technique to protect the hypercall interface. In our approach, even facing a total compromise of a domain, the security of the virtualization platforms can be guaranteed. We have built a prototype called RandHyp based on Xen. Our experimental results show that RandHyp can effectively prevent attacks via Xen hypercall interface with a small overhead.

Keywords: Xen, Hypercall, Protection, Randomization.

1 Introduction

Nowadays, virtualization has gained an increasingly concern in both industry and academic world. Cloud hosting providers have exploited the abstraction and isolation provided by the hypervisor to allow individual paying customers to share large-scale datacenter facilities, such as Amazon EC2[2], and Linode[3]. A lot novel projects also take advantage of hypervisor's supervision property in intrusion detection systems[19], workload isolation[20], attack investigation and debugging[21], and system monitoring[4].

However, all these applications are based on the belief that the hypervisor is sufficiently trustworthy to provide services and maintain isolation. But is it really like that? While the hypervisor itself can be regarded as secure due to its small size and a well-defined narrow interface, a number of different commodity operating systems are coresident on the same host[14], numerous vulnerable applications communicate with the uncertain outside world[13]. Under these circumstances, it is very likely that by exploiting the vulnerabilities of applications and operating systems, attackers can compromise a domain in the same way as they do in conventional operating systems. After that, attackers can escalate priviledges by applying to the hypervisor for illegitimate resources, which may lead to information leakage or denial of service. Specifically, if dom0 is compromised, attackers are able to access other domains' memory and I/O informations, and even create or shutdown other domains at will. In situations where

D. Gritzalis, S. Furnell, and M. Theoharidou (Eds.): SEC 2012, IFIP AICT 376, pp. 138–149, 2012.
© IFIP International Federation for Information Processing 2012

dom0 is simplified to reduce the vulnerabilities exposed[6], attackers can turn to compromise domUs. And according to CVE bug reports (e.g. CVE-2011-1898, CVE-2010-4238)[1], domU users can gain dom0 privileges and can either cause a denial of service or read arbitrary files in dom0. Further, any domain can be dominated to keep requesting critical resources, such as cpu and memory, which may lead to denial of services to other domains.

Since the hypercall interface is used to access hardware resources and execute sensitive instructions, the malicious goals listed above have to be achieved by invoking hypercalls. Therefore, preventing the hypercall interface from being used is very essential.

In this paper, we present solutions to protect domains by using Xen hypercall interface randomization technique. In our approach, all hypercall invocations are classified into trusted ones or untrusted ones. For the trusted ones, we randomize each hypercall's arguments, including its hypercall number. We also add paddings to the arguments and permute both the paddings and the arguments to further make our approach less breakable. The untrusted ones do not get randomized and are regarded as invalid by the hypervisor. In such a case, even attackers can get into a domain, he can not further elevate his priviledges by exploiting the hypercall interface, and thus avoid more serious damage to the whole system.

We present RandHyp, a modified version of Xen with para-virtualized Linux platforms as dom0 and domUs. Our experimental results show that RandHyp can successfully prevent untrusted hypercalls from executing while incurring low performance penalty.

The rest of this paper is organized as follows. The next section presents a short related work section about existing security machenisms in virtualization platforms. Section 3 identifies the threat model, explicits our design goals, and gives the design overview. After that, the implementation of RandHyp is described, following by a brief security analysis. The evaluation results are given in section 6, and section 7 concludes this paper.

2 Related Work

Existing security measures, like mandatory access control (MAC)[15] and trusted platform module (TPM)[16], can not be directly applied to protecting the Xen hypercall interface. Most previous efforts concentrate on reducing the probability of domains being compromised by monitoring domains from the hypervisor level[11,12]. However, designers may not know all threats, and new exploitation techniques can appear, and attack vectors cannot be totally eradicated, thus domains are still untrustable.

Other solutions are based on the assumption that dom0 is malicious (dom0 is a high-valued attack target, resulting from its control over other domains), and aim at protecting domUs against a malicious dom0. For example, CloudVisor[8] introduces a tiny security monitor to protect resources, Xoar[9] breaks the control domain into single-purpose components to reduce the damanage attackers

can introduce, Chunxiao Li *et al.*[10] removed the control domain from TCB, and so on.

However, all these solutions are difficult to implement in the commercial products for the reason that they need to modify the existing virtualization architectures, which requires professionals and may lead to problems in updating systems.

Hoang[13] has proposed two innovative approaches, authenticated hypercalls (MAC) and hypercall access table (HAT), that aim at protecting the hypercall interface. The MAC approach is not practical given the limitation of the number of arguments that can be passed to the hypervisor. The HAT approach records the addresses of trusted hypercall invocations and stores them in an HAT table in the hypervisor. However, in kernels of the same version, the addresses are fixed. Hence, getting the addresses of the trusted hypercall invocatons are easy. Moreover, the size of the HAT table stored in the hypervisor will increase as the number of domains increases, which contradicts the design concept of Xen[5].

Unlike existing solutions, RandHyp maintains the design concept of Xen with a small modification of the Xen hypervisor and guest operating systems and is transparent to guest applications. Besides, since the changes of the operating systems are mainly in some head files, they would not hinder updating systems or inserting of modules.

3 Xen Architecture, Threat Model, and Design Goals

Since RandHyp is based on Xen, this section first describes the architecture of Xen, especially the hypercall interface. Then the threat model is identified, and our design goals are articulated.

3.1 Architecture Overview

In Xen architecture, the hypervisor is the most priviledged software layer that virtualizes the hardware resources and partitions them dynamically to the overlying domains. A single administrative domain (dom0) has the ability to manage all other domains (domU).

Since the hypervisor is responsible for monitoring all privileged state, domains have to transfer control into the hypervisor when executing sensitive instructions, which is realized by using hypercalls[7]. Hypercalls are made to execute high-priviledged operations, such as exception handling, scheduling, physical memory management, access-control of disks and network devices, virtual CPU operations, and inter-domain communications.

Hypercalls are similar to system calls in conventional operating systems. A software interrupt `INT $0x82` transfers the control from the domain into the hypervisor, where operations are validated and applied, and when completed, the control is returned to the calling domain[5]. The particular hypercall to be invoked is contained in `rax` with all the arguments contained in `rbx, rcx, rdx, rsi, rdi` (x86_64 Linux) or `eax, ebx, ecx, edx, esi, edi` (x86_32 Linux).

An example use of hypercall is to update an individual segment descriptor in the GDT or LDT by using hypercall `HYPERVISOR_update_descriptor(unsigned long ma, unsigned long word)`.

3.2 Threat Model

We assume the hardware and hypervisor are trusted, which means that we are not concerned with the violation of security by the service providers. We focus on the threat when a well-behaved domain is compromised and used as an entry to breach isolations among domains.

In a virtualization environment, since domains are usually commodity operating systems with vulnerabilites, it is prudent to assume that they may be compromised to behave with evil intentions. Thus, the attacker in our model is a domain aims at violating the security of other domains, including violating the data integrity or confidentiality of the target domains or causing denial of service to other domains with whom it is sharing the same underlying resources.

In order to achieve the malicious goals, the attacker inevitably has to go through the hypercall interface to achieve hardware resources. For example, domains cannot directly apply for extra memory space or modify critical data structures such as page tables, but these operations can be done through hypercall requests. In such a case, the attacker needs to invoke a series of dedicately arranged hypercalls to make his malicious goals realized.

Since hypercalls are much similar to system calls, hypercall attacks could be in any form known for system call attacks such as argument hijacking or mimicry[13]. Faking a series of hypercalls to sniff other domains' information or cause denial of service is applicable.

Information Leakage Attack. Information leakage attacks are mainly caused by a malicious dom0, for the reason that dom0 can access memory and I/O informations belonging to domUs. Given the evidence that information leakage may bring more profits to attackers, dom0 turns out to be a high-valued attack target.

Most projects assume a small-sized dom0 and deduce that the vulnerabilities exposed to be rare. However, this is undesirable as demonstrated by Colp[9] that dom0 in a mature virtualization platform is actually larger than a conventional server operating system for it is often relied on to provide additional shared services, such as drivers for physical devices, device emulation, and administrative tools.

The potentially malicious dom0 is often ignored in discussing the security of virtualization systems. Once dom0 is compromised, personal informations of customers can be stealed, which would be a total disaster.

Denial of Service Attack. Both dom0 and domUs can launch denial of service attacks through keeping occupying resources. Further, dom0 can cause a denial of service to certain domains by changing the scheduling flows.

3.3 Design Overview

Rather than exploring techniques to construct a bug-free system, a more pragmatic goal is to provide an architecture that can prevent the hypercall interface from being used, so that attackers in subverted domains cannot further mount successful attacks against other domains. With this in scheme, RandHyp is designed to guarantee that a malicious domain cannot issue arbitrary control transfers into the hypervisor and the execution context cannot be faked.

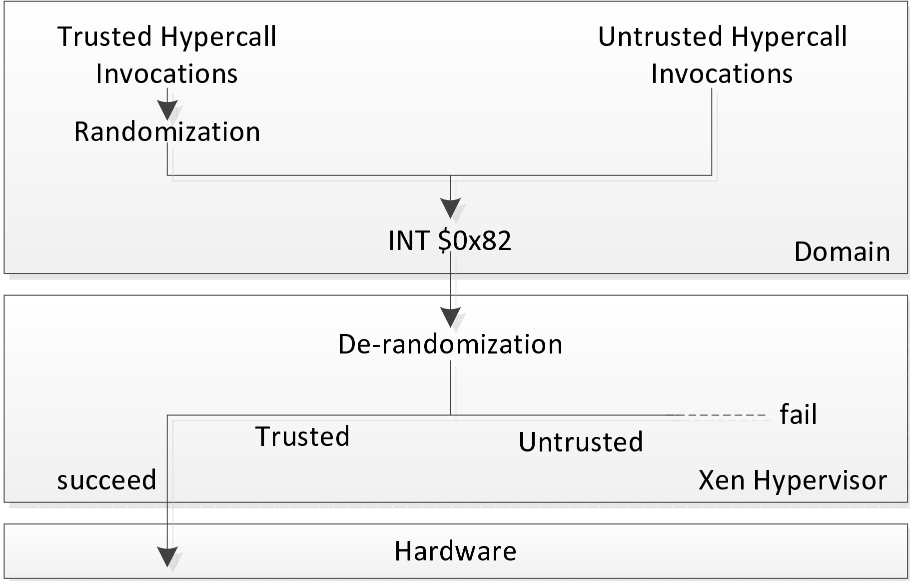

Fig. 1. RandHyp architecture

Fig. 1 shows the architecture of RandHyp. The general idea is to make the untrusted hypercall invocations unrecognized by the hypervisor, thus avoid attackers to execute priviledged operations. The method we choose is to randomize trusted hypercall invocations without the attackers' consciousness, and de-randomize them in the hypervisor where attackers cannot access.

4 Implementation

In this section, we present our design of randomization and de-randomization schemes in RandHyp. RandHyp protects domains against all other domains, including dom0. Hypercall `HYPERVISOR_update_descriptor` is used as an example to typify all the rest hypercalls.

4.1 Randomization

In x86_64 XenLinux, the hypercall number of `HYPERVISOR_update_descriptor` is `10`, the arguments are `ma` and `word`. What we need to do is re-assign the numbr and the arguments new values generated by some algorithm to make them not only unaccessable by attackers, but also difficult for them to guess. The randomization process is described as below.

Kernel-Level Randomization. Kernels, including loadable kernel modules and drivers, invoke hypercalls directly by calling the function `HYPERVISOR_update_descriptor (unsigned long ma, unsigned long word)`, so that is the place we do randomization operations.

Firstly, we add paddings to maximum the number of arguments. After that, the function looks like `HYPERVISOR_update_descriptor(ma, word, pad1, pad2, pad3)`.

Secondly, every argument is encrypted to make a new, randomized argument using the following equation:

$$\mathrm{arg}'_i = R_K(arg_i, key)\ . \tag{1}$$

R_K is our randomization algorithm using a key key. In RandHyp, we choose RC4[17] encryption algorithm which is one of the most secure symmetric encryption algorithms in the world. Other algorithms are also applicable. After this step, the hypercall invocation would be `HYPERVISOR_update_descriptor(arg1', arg2', arg3', arg4', arg5')`.

Thirdly, to make our scheme more aggressive, RandHyp permutes the roles among `rbx, rcx, rdx, rsi, rdi` (`ebx, ecx, edx, esi, edi` in x86_32 Linux) registers.

Lastly, due to the design of Xen[5], hypercall numbers are limited between 0 and 128, so it is impossible for RandHyp to randomize hypercall numbers in the way it does to hypercall arguments. In this way, RandHyp adjusts by choosing a number between 0 and 128 randomly for each hypercall.

User-Level Randomization. Hypercalls may be indirectly used in user-level. The randomization operations in user-level are the same as in kernel-level, while the difference lies in where the operations are done. To invoke hypercalls in user-level, a structure `privcmd_hypercall_t`, which contains the hypercall arguments and number, are passed to the kernel by a kernel driver `privcmd`. In function `privcmd_ioctl` the arguments are written into registers and the software trap `INT $0x82` is executed. Consequently, the hypercall arguments and number should be randomized when `privcmd_hypercall_t` is constructured.

4.2 De-randomization

No matter where a hypercall comes from, the execution of the hypercall instruction `INT $0x82` generates a software trap into the Xen hypervisor, where the

hypercalls are validated and executed by the same handler. Note that we have randomized trusted kernel-level hypercalls and trusted user-level hypercalls in the same method, so we don't need to differentiate the de-randomization scheme.

RandHyp has constructed a correlation between the hypercall numbers in domains and the hypercall numbers in the Xen hypervisor. Thus, when the hypercall interrupt handler catches a hypercall, the hypervisor can pass it to the right handle function according to its number, in our cases, it is the `do_update_descriptor` function.

In order to avoid the randomized hypercall arguments from being used by the hypervisor, RandHyp de-randomizes the hypercall arguments immediately when entering the `do_update_descriptor` function.

Firstly, real arguments are selected from the paddings.

Secondly, given the encryption algorithm we use is symmetrical, the de-randomization algorithm is the same as the randomization algorithm. RandHyp recovers the original hypercall arguments $arg_i = R_K^{-1}(arg_i', key)$ using the same key used during the randomization process. After that, Xen can do what the hypercall requires to do.

5 Security Discussion

Attacks Using Direct Hypercall Invocation. An attacker may directly invoke hypercalls in kernel-level or user-level. RandHyp can defeats such straignt-forward attacks.

Since hypercalls created by attackers are untrusted, they are not randomized before trapping into the hypervisor, so that they will be de-randomized into meaningless informations and fail to execute.

Attackers may attempt to acquire the randomization key directly, which is also defeated by RandHyp. The reason is that the randomization key is stored in the memory space of Xen, where attackers in domains can't access. Attackers are forced to scan the kernel code memory and collect the semantics of the instructions to get the key, which is hard to perform.

Even if the attacker can successfully get the key, the probability of him to create a right argument would be very small, for the reason that there are 38 hypercalls in a system, and each hypercall contains 5 arguments, the possibility for an attacker to get the right arguments of `HYPERVISOR_update_descriptor` is $\frac{1}{C(5,2)\times 38}$, which is about 0.26%.

Attackers may try to construct plaintext-ciphertext pairs to brute-force the key. RandHyp makes this very difficult. Firstly, a strong encryption algorithm and a long key makes it almost impossible to crack the key. Secondly, because we have added paddings to the arguments, and permuted the roles among the registers, and the hypercall number is re-assigned, the attackers face difficulties in constructing plaintext-ciphertext pairs.

Attacks Using Indirect Hypercall Invocation. Instead of invoking hypercalls directly, an attacker may try to reuse existing hypercalls. In this situation,

an attacker has to achieve the memory address of the desired hypercall instructions accurately, and then jump there to eventually invoke the intended hypercalls. Although RandHyp cannot directly prevent this kind of attacks currently, existing techniques such as address space layout randomizatoin (ASLR)[18], which makes the memory location of pre-existing code hard to predict, can be utilized to enhance security. Moreover, even if attackers can successfully hijack the control flow of a process, the hypercalls used by kernel only can accomplish basic and simple functionalities. Combining them to achieve malicious goals is almost impossible.

6 Evaluation

In this section, we first present RandHyp latency measurement results, and then present a number of attack experiments that cover all hypercalls.

6.1 Experimental Setup

Our experiments were run on a Lenovo desktop PC with a 2.53GHz Intel(R) Core(TM)2 Duo CPU processor and 2GB of RAM. All Xen modifications necessary for RandHyp were implemented on Xen 4.0.1 and Linux 2.6.32.13 kernel with Xen patches applied. The changes required to Xen were minimal, spanning only a handful of source files including the files containing the hypercall service routines and several head files. On the guest domain kernel, we modified the files containing the routines which invoke hypercalls and the file that defines hypercall number. For convience, we give the modified system an alias, Rand-Domain, to distinguish the unmodified system, Orig-Domain.

6.2 Performance Evaluation

The performance of RandHyp should be evaluated in the following aspects: 1) Disk I/O throughput; 2) Network I/O throughput; 3)Overall system performance.

Disk I/O Performance. Disk performance is tested by using *dd*. Fig. 2 shows the results of these tests with different configuration parameters. Overall, disk throughtput is down by 1-4.5%. The reading and writing latency incurred by small files is larger than that by big files, which may be caused by more frequent transversions into the hypervisor.

Network I/O Performance. *Netperf* is used to test network performance. Fig. 3 shows the results. RandHyp only causes 1.5-3% drop in throughput when using TCP, and less than 0.2% drop while using UDP. Because TCP is a connection-oriented protocol, more hypercalls have to be used to keep the connection.

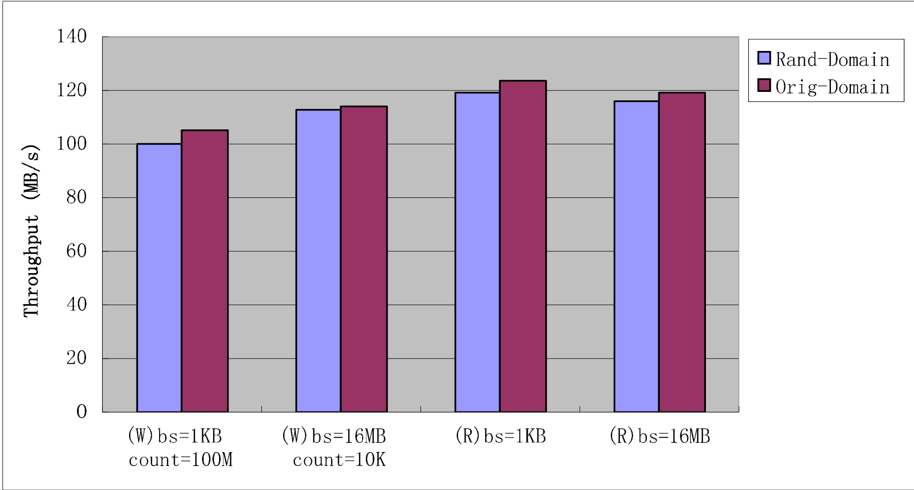

Fig. 2. Disk I/O performance using dd (higher is better)

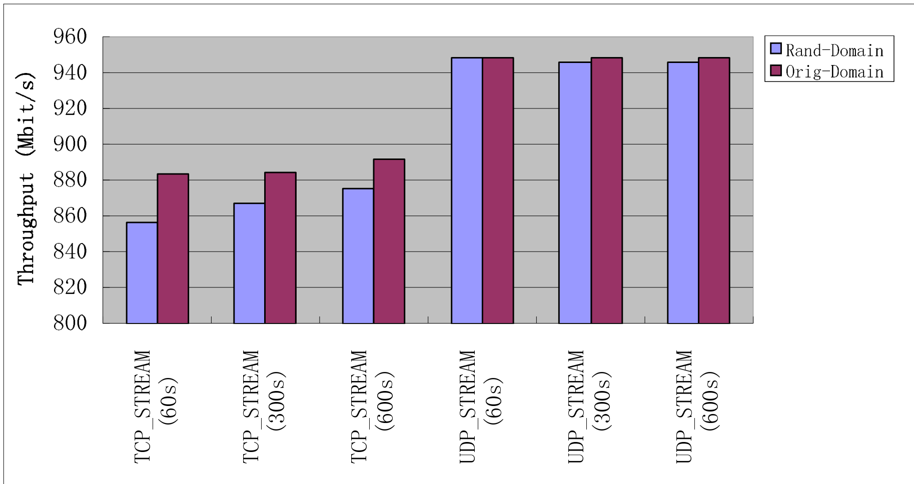

Fig. 3. Network I/O performance using Netperf (higher is better)

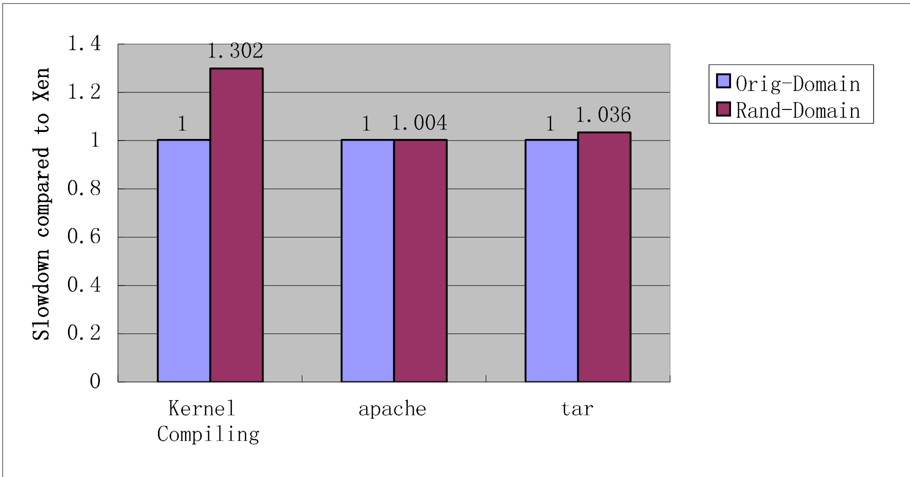

Fig. 4. Real-word benchmarks (lower is better)

Real-world Benchmarks. We use several popular applications for RandHyp latency measurement, including Linux kernel compilation, Apache *ab* benchmarking tool and *tar*. As Fig. 3 shows, the kernel compiling overhead added by RandHyp is about 30%, the Apache web server serving a 4KB static webpage 10000 times for 500 simultaneous clients adds only 0.4% overhead, and the *tar* operation adds about 0.36% overhead.

6.3 Effectiveness Evaluation

The commonly received approach to evaluate a security measure is to subject it to existing attacks. We have discussed the possible attack scenario in Section 2. However, attacks target at crushing virtualization platforms from the overlying domains have not gained widely applied in reality, which makes evaluating security a challenging job. Moreover, constructing the attacks is out of our boundaries. Given this situation, we make an attempt to demonstrate the improvement to the state of security for hypervisors by exposing them to loadable kernel modules (LKM) that are used to imitate the intrusion attacks. Each LKM tries to execute one or a set of non-randomized hypercalls as a real attacker may do. We assume that such LKMs could be successfully injected into the domain through conventional operating system security breaches in real world.

We have tested all hypercalls, and the experiment results show that these hypercalls cannot execute correctly, which verifies that the hypercalls which are not in our protection coverage can be caught by RandHyp.

These simple experiments are, though very specific and even quite contrived, they serve the purpose of verifying the effectiveness of our prototype nevertheless.

7 Conclusion

While virtualization is becoming widely accepted in both industry and academic world, its security is put on the agenda. Enhancing the security of virtualization platforms would bring more values to their popularity and usability. Although the sizes of hypervisors are a lot smaller than that of conventional operating systems, the domains running on is untrusted, and can perform illegal operations through the hypercall interface. This paper focuses on protecting the hypercall interface so as to maintain normal services to domains and guarantee the isolation between them. A transparent approach, namely RandHyp, was proposed on Xen. In RandHyp, any hypercall out of our protect range will be detected and refused to execute. RandHyp achieved this by using randomization techniques. Both hypercall's number and arguments are randomized in order to mess the attackers. Experiments show that RandHyp can effectively counter illegal hypercalls while incurring only a small overhead.

Acknowledgments. This work was supported in part by grants from the Chinese National Natural Science Foundation (61073027, 60773171, 90818022, and 61021062), and the Chinese 973 Major State Basic Program(2009CB320705).

References

1. CVE, `www.cve.org`
2. Amazon EC2, `www.amazon.com`
3. Linode, `www.linode.com`
4. Xenaccess library, `http://code.google.com/p/xenaccess/`
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: 19th ACM Symposium on Operating Systems Principles, pp. 164–177. ACM Press, New York (2003)
6. Steinberg, U., Kauer, B.: NOVA: a Microhypervisor-Based Secure Virtualization Architecture. In: 5th EuroSys, pp. 209–222
7. Xen Interface Manual, `www.xen.org`
8. Zhang, F., Chen, J., Chen, H., Zang, B.: CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In: SOSP 2011 Proceedings of the 23th ACM Symposium on Operating Systems Principles, pp. 203–216 (2011)
9. Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., Warfield, A.: Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In: SOSP 2011 Proceedings of the 23th ACM Symposium on Operating Systems Principles, pp. 189–202 (2011)
10. Li, C., Raghunathan, A., Jha, N.K.: A Trusted Virtual Machine in an Untrusted Management Environment. In: 3rd IEEE International Conference on Cloud Computing, pp. 172–179 (2010)
11. Chen, X., Garfinkel, T., Christopher Lewis, E., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating System. In: 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 2–13. ACM Press, New York (2008)

12. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering Kernel Rootkits with Lightweight Hook Protection. In: 16th ACM Conference on Computer and Communications Security, pp. 545–554. ACM Press, New York (2009)
13. Hoang, C.: Protecting Xen Hypercalls. Intrusion Detection/Prevention in a Virtualized Environment. MS Thesis, Univeristy of British Columbia (July 2009)
14. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS 2009 Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 199–212 (2009)
15. McCune, J.M., Jaeger, T., Berger, S., Caceres, R., Sailer, R.: Shamon: A System for Distributed Mandatory Access Control. In: ACSAC 2006: Proceedings of the 22nd Annual Computer Security Applications Conference, pp. 23–32 (2006)
16. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: 19th Symposium on Operating System Principles
17. Rivest, R.L.: The RC4 Encryption Algorithm. RSA Data Security, Inc. (March 1992)
18. Shacham, H., Page, M., Pfaff, B., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: 11th ACM Conference on Computer and Commuications Security, pp. 298–307 (2004)
19. Wang, Z., Jiang, X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, CA (May 2010)
20. Gupta, D., Cherkasova, L., Gardner, R., Vahdat, A.: Enforcing Performance Isolation Across Virtual Machines in Xen. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 342–362. Springer, Heidelberg (2006)
21. Kamble, N.A., Nakajima, J., Mallick, A.K.: Evolution in kernel debugging using hardware virtualization with xen. In: Proceedings of the 2006 Ottawa Linux Symposium, Ottawa, Canada (July 2006)