Preventing Secret Data Leakage from Foreign Mappings in Virtual Machines

Hanjun Gao[1], Lina Wang[1,2], Wei Liu[1], Yang Peng[1], and Hao Zhang[1]

[1] School of Computer Science, Wuhan University, Wuhan 430072, China
ghjwhu@sina.com
[2] The Key Laboratory of Aerospace Information and Trusted Computing, Ministry of Education, Wuhan, 430072, China

Abstract. The foreign mapping mechanism of Xen is used in privileged virtual machines (VM) for platform management. With help of it, a privileged VM can map arbitrary machine frames of memory from a specific VM into its page tables. This leaves a vulnerability that malware may compromise the secrecy of normal VMs by exploiting the foreign mapping mechanism. To address this privacy exposure, we present a novel application's memory privacy protection (AMP^2) scheme by exploiting hypervisor. In AMP^2, an application can protect its memory privacy by registering its address space into hypervisor; before the application exists or cancels its protection, any foreign mapping to protected pages will be disabled. With these measures, AMP^2 prevents sensitive data leakage when malware attempts to eavesdrop them by exploiting foreign mapping. Finally, extensive experiments are performed to validate AMP^2. The experimental results show that AMP^2 achieves strong privacy resiliency while incurs only 2% extra overhead for CPU workloads.

Keywords: Direct Foreign mappings, Virtual machine, Hyprevisor, Privacy, Secrecy, Data leakage.

1 Introduction

In recent years, virtual machine monitors (VMMs, or hypervisor) have been widely adopted in modern computing systems, such as Xen[1], VMware[2] and KVM[3] etc. The distinguishing security features of hypervisor, especially in VM introspection (VMI), have aroused many researchers' attentions. For example, Livewire[4] proposes the concept of VM introspection and applies it in the field of intrusion detection. AntFarm[5], Xenprobes[6], XenAccess[7] and VMwall[8] incorporates VM introspection to monitor real-time memory status and disk activity of Guest OS, and consequently infer guest-internal events, such as running processes, file-system operation and network connections etc. VMwatcher[9] is implemented for detecting malwares and kernel rootkits, which are difficult to be done in conventional methods. SBCFI[10] is used to protect the control flow integrity of guest OS and improve its reliability and security. With the help of hypervisor, Lycosid[11], Patagonix[12] and Manitou[13] can effectively detect and

M. Rajarajan et al. (Eds.): SecureComm 2011, LNICST 96, pp. 436–445, 2012.

identify hidden processes.These efforts effectively exploit the fact that hypervisor can easily fetch memory pages from the target guest OS.

Related Work. In spite of various managemental gains as illustrated above due to the privileged ability in Xen hypervisor, we observe that it is also desirable to enforce some restrictions to this privilege to avoid misuse and/or abuse. In Xen hypervisor, any software running in the Dom0 use-space can obtain arbitrary memory pages by making direct foreign mappings. This non-restricted memory sharing mechanism may potentially undermine the privacy of guest OS. For instance, when a user logs in his bank account, his account's password will be temporarily stored somewhere in the memory, and malware residing in Dom0 may eavesdrop the password by performing direct foreign mappings. Murray*et al.* [14] suggested to remove all uses of the direct foreign mapping operation from Dom0 user-space to protect the privacy of virtual machine. Unfortunately, Dom0 is designed to serve as a managing domain, and a simple removal of all uses may undermine its availability and corrupt other security measures such as VM introspection which has been widely used to solve system security problems (e.g.,[4,5,6,7,8,9,10,11,12,13,15]).

Several efforts have been devoted to privacy protection of virtual machines without significantly undermining their availability. Yang and Shin proposed SP^3[16] which exploits hypervisor to prevent application information from unauthorized exposure and does not require the operating system to be trustable. And Chen et al[17] also proposed their scheme to protect the privacy and integiety of application data based on the same assumptions. However, if malware resides in Xen's privileged domain, it can still eavesdrops the application data by foreign mapping. Borders *etal.* proposed Storage Capsules [18] which allow users to view and edit sensitive files in a compromised machine without leaking confidential data. The key technique is to take a checkpoint of current system state and disable device output. When editing files and re-encrypting are done, the system is restored to original state and device output is resumed. However, this methodology leaves the gap that if storage capsules are equipped in Xen, malware residing in Dom0 can steal confidential data by foreign mapping.

Our Contributions. In this paper, we propose a novel scheme to protect application's memory privacy in DomU even when there are malwares attempting to eavesdrop them by direct foreign mappings. The scheme is called application's memory privacy protection (AMP^2) which is designed to mainly protect data resided in memory, such as decrypted secrets, password entered to login bank account etc. Whereas files stored on disk are out of our concern, because they can be properly protected via encryption. Compared to the SP^3 [16], the Overshadow[17] and the Storage Capsules [18] proposals, our methodology makes special efforts to protect secret data in the case that malware resides in privileged domain(Dom0) in Xen, which enables our scheme to be complementary to these above three proposals and to provide stronger privacy protection.To keep availability, instead of removing the foreign mappings as in the Murray *et al.* solution [14], our scheme restricted them in a way such that a memory

page allocated to protected application is unable to be mapped by Dom0 or any other privileged domains. To this end, we present a kernel module to accept the request from the user-space and created a hypercall to send the protection request to the hypervisor. Also, we carefully strengthen the page table updating handler to intercept any mapping operation so that it can dynamically protect application's memory pages.

2 AMP^2 Scheme

In AMP^2, when an application needs to be protected, it issues the request to hypervisor. Hypervisor maintains a protected applications memory page counter table (AMPC table) which is used to keep the page counters registered by the application.[1] When foreign mapping to DomU's pages occur, hypervisor will look up AMPC table to get the counters and decide whether the foreign mapping can be done. At the same time, AMP^2 also maintains a foreign mapping tracking table (FMT table) to record all foreign mapping operations. If a memory page which has been mapped by foreign mapping is dynamically allocated to a protected application, the previous foreign mapping will be redirected to some other public page, such as shared info page etc, and the relevant entries in FMT table will be cleared too. Finally, AMP^2 must be aware of the events of application exiting, memory protection canceling and DomU destroying, and consequently update AMPC table lest legitimate foreign mapping cannot be performed.

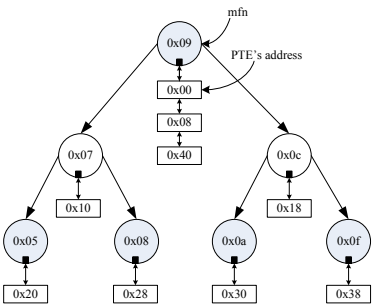

Fig. 1. Foreign mapping tracking table, FMT table

In the following, we illustrate with an example how AMP^2 works. It is assumed that Dom0 has first 4 page out of a total 16 ones and the rest belongs to DomU. AMP^2 intercepts all foreign mapping operations and maintains a FMT

[1] In our scheme, the AMPC table's size is proportional to that of machine memory, and one memory page correspond one entry in AMPC table. When a memory page is registered, the corresponding entry in AMPC table is increased by 1. It is possible that multiple processes sharing the same memory pages register its memory space for protection. In this case, the values of some entries are larger than one.

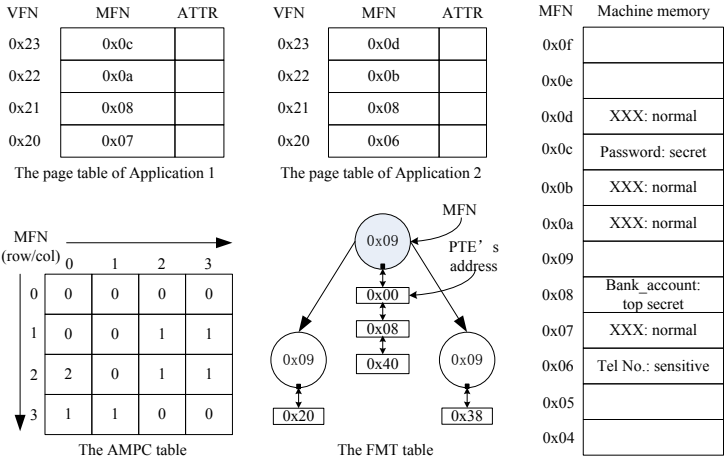

Fig. 2. AMP^2 example

table to record these operations, as shown in Figure 1. This FMT table shows that seven machine page frames in DomU have been mapped by Dom0 and these records are stored in a red-black tree, where the key of node represents the foreign mapped machine frame number (mfn). Taking the root as an example, the page frame 0x09 has been foreign mapped 3 times by Dom0, and the corresponding PTEs' machine addresses are 0x00, 0x08 and 0x40 respectively, all at page frame 0x0.Figure 2 shows the process that AMP^2 protects applications' memory pages when they apply protection to hypervisor. In the figure, there are two applications applying protection to the hypervisor. They occupy seven pages in total, the mfn of which are 0x6, 0x7, 0x8, 0xa, 0xb, 0xc and 0xd respectively. The corresponding entries in AMPC table are increased by 1, except 0x8, which is increased by 2 because it is occupied by two applications simultaneously. Then, AMP^2 look up FMT table to check whether these pages have been recorded. In our example, there are four pages having been foreign mapped, the mfn of which are 0x7, 0x8, 0xa, and 0xc, respectively. Base on the mfns, AMP^2 can quickly locate the target nodes and remove them from FMT table. Meantime, AMP^2 can easily get corresponding PTE's address and modify PTE to redirect to public page, such as shared info page etc.

3 AMP^2 Design

3.1 Restricted Foreign Mapping

When a foreign mapping opeartion occurs, AMP^2 captures it and parses the mapped machine page frame number (mfn) and the corresponding PTE's address. Then it checks whether the mapped page's counter in AMPC table is

above zero or not, which shows that whether some applications have applied protection. If the counter is more than zero, AMP^2 fails this mapping request. Otherwise the mapping can be performed and the operation is recorded in FMT table. The reason to maintain FMT table is that, with FMT's help, AMP^2 can effectively redirect previous established foreign mappings when an application, which is not protected before, requests for protection.

3.2 Application Applying for Protection

In the hypervisor-based implementation, we define a hypercall for applications to issue protection requests.When AMP^2 is aware of the protection request, it firstly obtains the head of the list virtual memory area (that is mmap) and the page global directory (pgd) base on the PID of the application, and then parses the mfn of the occupied pages,including page directory, page table, and currently occupied machine frames. Secondly, AMP^2 updates AMPC table according to these mfns (The index of AMPC table is mfn, and the value of the table entry represents the counters). Because the request is for protection, the value of corresponding enntries is increased by 1.

In morden OS, the memory page is allocated to a process until it is actually needed. Therefore, AMP^2 will capture all the events of normal pages mapping in DomU, retrieve the page allocated to the protected application, and eventually register it for protection in the application's runtime. The detail is illustrated in section 4. At last, AMP^2 looks up FMT table to check whether there exists any recorded mapping. If a mapping is found in FMT table, AMP^2 will modify the mapping to redirect to the public page, such as shared info page which is designed for share infomation between Dom0 and DomU. Furthermore, any child process created by the protected application will also be automatically protected.

3.3 AMP^2 Page Table Updating

AMP^2 page table updating extends the interface of Xen's. We implement our checking logic by intercepting all Xen's page table updating routines. In these routines, the eventual control structure to be handled is a simple pair:$\langle ptr, val \rangle$,the ptr is machine address of PTE, and the val is new contents (the key is mfn) of PTE. Figure 3 illustrates the AMP^2 page table updating framework. It first checks the P (present) bit of val to determine that the updating is mapping or unmapping. If it is a mapping operation, and even is a foreign mapping operation, AMP^2 will ensure that the counter of entry whose index is val.mfn in AMPC table is equal 0. Only in this case, the foreign mapping can be performed and meantime the operation will be recorded in FMT table.(In the opposite case, the foreign mapping failed.) Otherwise, if it is a normal guest domain page mapping, AMP^2 will check whether the ptr (that is machine address of PTE) locates in a protected process's address space or not. If it does, AMP^2 will increase the page's counter by 1 in AMPC table based on the val.mfn. In the meantime, AMP^2 checks against FMT table to redirect previous foreign mapping to a public page if this memory page had been foreign mapped before.

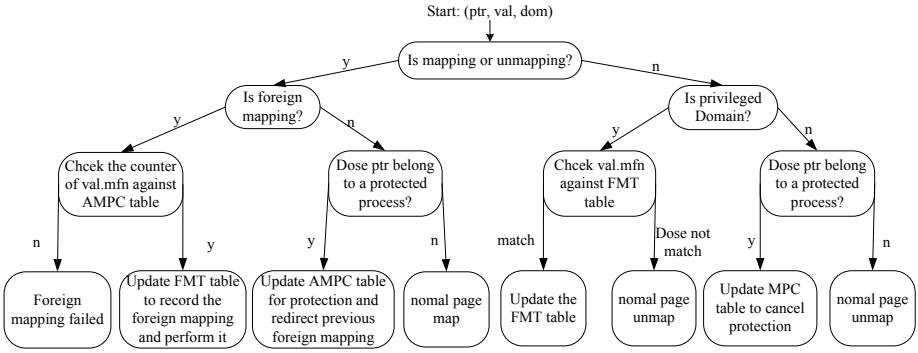

Fig. 3. AMP^2 page table updating

On the other hand, if the updating is an umapping operation, and it is from a privileged domain, AMP^2 will check the val.mfn against FMT table. If a record is found, it is shown that this is a foreign unmapping operation. And AMP^2 will clear the relevant entry in FMT table based on the mfn. Otherwise, if the unmapping is from the guest domain and the address of PTE belongs to a protected application, AMP^2 will update AMPC table to cancel the memory page's protection.

4 AMP^2 Implementation

In order to accept the request for protection from the application, we provide a hypercall and a kernel module. User explicitly issues a register request, which triggers the kernel module. Handler in the module parses corresponding page tables based on the pid, wraps up all mfns as a request, and invokes the hypercall to pass the request to AMP^2. It increases the corresponding entries in AMPC table and check whether the pages for protection have been mapped by Dom0 in the past. If it is, AMP^2 will redirect the foreign mapping to other public page such as shared info page in read-only mode for security.

Due to on-demand paging, it is insufficient to only protect the pages which the application actively registers. We add codes into the Xen's handler responsible for PTE updates to protect the memory page which is dynamically allocated to the application. In the para-virtualization, OS can update a PTE either by using hypercall, or with the help of writeable page table. Either way, the hypervisor can intercept PTE updates. It is no doubt that hypercall always trap into hypervisor by definition. Meantime, a modification to a PTE incurs a page fault which always traps into hypervior too. Therefore, we modify the Xen's handler for PTE updates to achieve our goal. The relevant modified handlers include do_mmu_update, do_update_va_mapping, and ptwr_emulated_update.

In AMP^2, besides explicitly canceling its protection by issuing a hypercall, the exit of a protected application also results in canceling protection. Therefore,

AMP^2 needs to intercept page unmaping events for lifting the page's protection. Unfortunately, normal page unmapping goes through a fast path for the sake of optimization and never traps into hypervisor. The only exception is that the page unmapping caused by foreign mappings. The reason is that Xen modifies the mm_struct.context of an application to add a has_foreign_mappings field in it. When the page unmapping occurs, the system call will check whether the field is set. If it not, hypervisor will unpin the page table. It means that modifying the page table will not trigger any page fault. If it is set, clearing the PTE will arouse the page fault and the hypervisor will emulate this direct page table write. Therefore, we also add an is_protected filed in that structure (mm_struct.context) and modify the do_exit handler to implement our check logic.

Finally, when a domain exits, the relevant resources allocated to it will be recycled, and the protection about an application in the very domain will also consequently be lifted. Therefore, we modifiy the resources recycling routine, especially the memory pages recycling handler: relinquish_memory, to clear corresponding entries in AMPC table to lift protection when a domain exit.

5 Evaluation

In this section, we first analytically examine the security guarantees provided by AMP^2. Then we measure the performance overhead. The machine used in our evaluation has a 3.0 GHZ Core 2 processor with 1GB of RAM. The version of hypervisor is Xen 3.3.0, and the kernel's version is XenoLinux 2.6.18. There are two virtual machine instances(one is Dom0, the other is DomU). Xen allocates 512 MB of RAM to Dom0, and the rest is allocated to DomU.

5.1 Security Analysis

As memtioned above, FMT table and AMPC table are key data structures to achieve our goal. Therefore, the integrity of them(including codes of AMP^2) should be guaranteed. According to our design, all of them are kept in hypervisor space, which runs in the highest privileged level. And there is no supported method to modify the Xen code in runtime even taking control over Dom0. In other words, it is difficult to bypass AMP^2 by patching out its check codes or tampering data structures without recompiling the Xen. Although there was a backdoor to subvert hyperviosr by overwriting Xen code and data structures by conducting DMA to Xen's memory[19], and it is indeed a real threat to AMP^2. Fortunately, however, Wang[20] proposed HyperSafe that endows Xen hypervisors with a unique self-protection capability to provide lifetime controlflow integrity. With the help of HyperSafe, the integrity of AMP^2 can be effectively protected.

In real usage, whenever an application needs to make sensitive operations, it just applies a protection request to AMP^2. And before the application exists or cancels its protection, any foreign mapping to protected pages will be disabled. And the pages which are foreign mapped before will be redirected. Therefore, AMP^2 don't detect whether malware is running in Dom0 or hides its presence.

5.2 Performance Evaluation

To evaluate the performance overhead introduced by AMP^2, we measured the runtime overhead with some CPU and memory intensive workloads, including two programs from the SPEC CPU 2000 integer benchmarks, and two other real world applications.

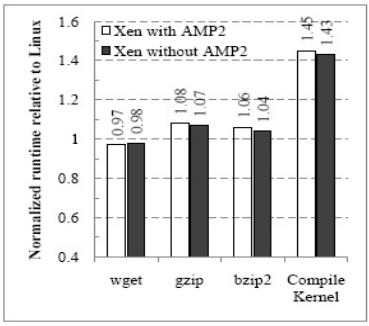

Fig. 4. Applications performance normalized to native Linux. (The numbers on top of bars represent runtime of applications normalized to native Linux without Xen).

We tested the application's performance in the guest OS in three scenarios: native Linux, Xen with and without AMP^2. First, we executed these applications in native Linux and measured the runtime. Then, these applications were executed in Xen without AMP^2. Last, we measured the runtime overhead in the Xen with AMP^2. The final performance result is shown in Figure 4. The performance overhead is presented as a relative runtime normalized to native Linux. Since these applications run in the guest OS, the mainly performance penalty comes from PTE updating, maintaining AMPC table and checking against FMT table. Therefore, the performance of an application with a frequent page table updating will be influenced dramatically. Overall, AMP^2 increases applications execution time by only 2% CPU workloads.

Another possible performance penalty may exist in the foreign mapping in Dom0. When the foreign mapping request is sent to hypervisor, AMP^2 will search FMT table, and decide whether the mapping can be performed. And if the mapping is valid, AMP^2 will record the pair of pte's address and target mfn. Figure 5 shows the times consumed to execute foreign mapping in Dom0. We tested 16 sets of data in total, ranging from two pages to thirty-two pages, and compared the consumed time. As the mapped pages increase, the size of FMT table and the time consumed to manipulate it increase too. However, using foreign mapping to map large amount pages is not always needed except for security reasons, so we can tolerate the performance penalty in most circumstances.

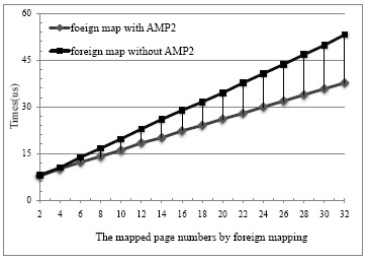

Fig. 5. Normalized performance of foreign mapping (The x-axis shows the page numbers mapped, and the y-axis shows the times consumed to complete foreign mappings)

6 Conclusion

This paper proposed AMP^2 to protect the application's memory data privacy from malware's evil eavesdropping via foreign mapping. When foreign mappings to DomU pages occur, hypervisor will decide whether the mappings can be done based on security requirements. We detailed the modifications and extensions made to hypervisor. To protect the target application, we presented a kernel module to accept the request from the user-space and created a hypercall to send the protection request to hypervisor. Finally, we strengthened the page table updating handler to intercept any mapping operation so that it can dynamically protect application's memory pages. Extensive practical experiments were carried out and the results shows that AMP^2 can successfully protect the memory data privacy without significant performance penalties.

Acknowledgement. The authors would like to thank my colleagues and the anonymous reviewers for their insightful feedback. This work is supported by National Natural Science Foundation of China under Grant No. 60970114.

References

1. Barham, P., Dragovic, B., Fraser, K., et al.: Xen and the Art of Virtualization. In: 19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, pp. 164–177 (2003)
2. Waldspurger, C.A.: Memory resource management in VMware ESX Server. In: 5th Symposium on Operating Systems Design and Implementation (OSDI), New York, pp. 181–194 (2002)
3. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux virtual machine monitor. In: The 2007 Ottawa Linux Symposium, Ottawa, pp. 225–230 (2007)
4. Garfinkel, T., Rosenblum, M.: A Virtual machine Introspection-Based Architecture for Intrusion Detection. In: 10th Network and Distributed System Security Symposium (NDSS), San Diego, pp. 191–206 (2003)

5. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: Tracking processes in a virtual machine environment. In: Proceedings of the 2006 Annual USENIX Technical Conference, Boston, pp. 1–14 (2006)
6. Quynh, N.A., Suzaki, K.: Xenprobe: A lightweight user-space probing framework for xen virtual machine. In: USENIX Annual Technical Conference, San Diego (2007)
7. Payne, B.D., Carbone, M., Lee, W.: Secure and Flexible Monitoring of Virtual machines. In: The Annual Computer Security Applications Conference (ACSAC), Miami Beach, pp. 385–397 (2007)
8. Srivastava, A., Giffin, J.: Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 39–58. Springer, Heidelberg (2008)
9. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-based "out-of-the-box" Semantic View Reconstruction. In: 14th ACM Conference on Computer and Communications Security (CCS), Alexandria (2007)
10. Petroni, N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: 14th ACM Conference on Computer and Communications Security, CCS, Alexandria (2007)
11. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: International Conference on Virtual Execution Environments (VEE), New York, pp. 91–100 (2008)
12. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: 17th Conference on Security Symposium (USENIX SECURITY), San Jose, pp. 243–258 (2008)
13. Litty, L., Lie, D.: Manitou: A layer-below approach to fighting malware. In: The Workshop on Architectural and System Support for Improving Software Dependability (ASID), pp. 6–11, San Jose (2006)
14. Murray, D.G., Milos, G., Hand, S.: Improving Xen Security through Disaggregation. In: 4th International Conference on Virtual Execution Environments (VEE), New York, pp. 151–160 (2008)
15. Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)
16. Yang, J., Shin, K.: Using hypervisor to provide Data Secrey for User Applications on a Per-Page Basis. In: Proc. of the 4th International Conference on Virtual Execution Environments (VEE), New York, pp. 71–80 (2008)
17. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, et al.: Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In: Proc. of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Seattle (2008)
18. Borders, K., Weele, E.V., Lau, B., Prakash, A.: Protecting Confidential Data on Personal Computers with Storage Capsules. In: 18th USENIX Security Symposium (USENIX SECURITY), Montreal (2009)
19. Wojtczuk, R.: Subverting the Xen Hypervisor. In: Black Hat, USA (2008)
20. Wang, Z., Jiang, X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: Proc. of the 31st IEEE Symposium on Security & Privacy (SSP), Oakland (2010)