Hypervisor-based Protection of Sensitive Files in a Compromised System

Junqing Wang, Miao Yu, Bingyu Li, Zhengwei Qi, Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiaotong University, Shanghai 200240, China
{ junqingwang, superymk, justasmallfish, qizhwei, hbguan }@sjtu.edu.cn

ABSTRACT

One of the most fundamental issues in computer security is protecting sensitive files from unauthorized access. Traditional file protection tools run inside the target operating system, which hosts sensitive files. This makes previous approaches vulnerable in face of a compromised OS. To address this limitation, recent approaches seek for a good isolation by putting file system into a dedicated virtual machine or by using a network file system. However, they suffer a sharp increase in trusted computing base size which degrades their reliability.

In this paper, we present Filesafe, a special purpose hypervisor aimed at protecting sensitive files in a compromised operating system. It bridges the semantic gap between guest OS and hypervisor by reconstructing file hierarchy from raw data, which incurs no runtime overhead. By enforcing security policies created in hypervisor, Filesafe could prevent sensitive files from unauthorized access even if they have kernel privileges in guest OS. We have implemented a proof-of-concept prototype on Windows XP with FAT32 file system. Furthermore, we evaluate Filesafe's performance and code size to demonstrate it is practical in real world scenarios.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

security

Keywords

file protection, semantic gap, hypervisor, hardware assisted virtualization

1. INTRODUCTION

The widespread of commercial operating systems (OS) makes it easier and cheaper to host large amount of digital content. How to protect sensitive content from unauthorized access has become a major concern for governments, corporations and individuals. Unfortunately, commercial OSes like Windows and Linux are large and complex, making them hard to be protected from malicious software exploitation. Once compromised, all the files including sensitive files are exposed to intruders. This raises the problem of how to protect sensitive files in a compromised system.

Protecting sensitive files in a compromised system is difficult. Many existing file protection mechanisms [10, 12, 14] reside in the same space as the victim OS. They have an excellent view of the file system and could be implemented with a small code size by utilizing existing OS interfaces and drivers. Nevertheless, if the OS is compromised, these approaches become vulnerable to intruders. In the untrusted environment, isolation is an indispensable ingredient of feasible solutions.

Therefore, recent works focus on introducing isolation by putting file system into a dedicated virtual machine (VM) [11, 20] or by using a network file system (NFS) [15, 18]. These approaches provide a strong isolation while still maintaining a regular view of the file system. However, including another VM or the NFS server into systems, will greatly increase the trusted computing base (TCB) size of these systems and therefore degrade their reliability. Besides, the configuration cost in these approaches is noticeable because a general-purpose virtual machine monitor (VMM) or a NFS server has to be setup first.

In this paper, we present Filesafe, a lightweight hypervisor leveraging both BitVisor [16] architecture and hardware assisted virtualization (HAV) to protect sensitive files in a compromised OS. It offers strong isolation to the guest OS, and it is more resistant to attack due to small TCB size. Hypervisor or VMM, offers isolated execution environment from the untrusted guest OS. If the guest OS is compromised, it will not affect the security of the hypervisor. Hypervisor also provides the ability to interpose on hardware interface, which means we could monitor the interactions between the hardware and the guest OS. Therefore this enables intrusion detection and hardware access control.

Using hypervisor to protect sensitive files poses a problem well-known as semantic gap [4]. It is a gap between the view of the guest OS from the outside and the view from the inside. For example, we only see disk blocks from the outside of the guest OS while we could see files and directories, which are semantic objects of the OS, from the inside. To address this problem, we need to reconstruct internal semantic objects, such as directories and files, of the guest OS

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

from the outside. This approach has already been proved to be feasible to solve this problem [9].

Using hypervisor to protect sensitive files faces another problem: the security of hypervisor. It is not acceptable to trust in a hypervisor if this hypervisor is hardly more secure than guest OS. Reducing the TCB size of hypervisor is an effective approach to improve reliability. Unfortunately, traditional VMMs including hypervisors are large because they are designed for general purpose. They may consist of device drivers, virtual devices and resource managers. On the contrary, our hypervisor is a special purpose hypervisor, so it only needs some special device drivers, which makes it small.

In this paper, we make the following contributions.

- We implement a special purpose hypervisor named Filesafe, which successfully protects sensitive files in a compromised system. It bridges the semantic gap between guest OS and hypervisor by reconstructing file hierarchy from raw data instead of relying on guest OS interface.

- We utilize both BitVisor architecture and HAV to minmize TCB size of Filesafe, which is at least an order of magnitude smaller than that of contemporary virtualization environments. (e.g The TCB size of Filesafe is only 6% of that of Xen [2].)

Besides, we deploy Filesafe on off-shelf hardware platform with legacy OS. That means Filesafe requires no specific hardware support and no modification to the guest OS.

The rest of the paper is organized as follows. Section 2 describes the threat model and assumptions underlying our hypervisor. Section 3 and Section 4 present the design and the implementation of our hypervisor. Section 5 illustrates the experimental results obtained by measuring the code size and performance of our hypervisor. Section 6 discusses the related work in this research field. Finally, we conclude the paper in Section 7.

2. THREAT MODEL AND ASSUMPTIONS

We assume the following adversary model:

Our adversary or the intruder aims at stealthily reading the content of sensitive files, distorting sensitive files by modifying them, destroying sensitive files by deleting them. By exploiting vulnerabilities of the victim guest OS, the intruder may have the highest privilege level of it (e.g., the Administrator privileges in Windows). That means the intruder could easily bypass the file access control system in the guest OS and make any operations to sensitive files. We assume that intruder does not have physical access to the machine on which the system deploys and the hardware and firmware of that machine are considered trusted.

We assume the load-time integrity of our hypervisor. This could be achieved by leveraging the recent Intel TXT technology [1] which provides a reliable way called *measured late launch* to load a clean hypervisor. This approach is already listed in our future work. We also assume a trustworthy hypervisor with no vulnerabilities. We have spent great efforts on minimizing TCB size of our hypervisor, which will be detailed in the following section. Code size of our hypervisor is relatively small compared with that of a legacy OS. Besides, our hypervisor provides no interface to the guest OS.

We protect sensitive files that are most important to users, such as users' private data, and do not protect system-critical files and other files. Restricting access to system-critical files may lead to instability of guest OS, especially when the guest OS is patching or updating. Our current implementation cannot protect system critical files. However, OSes have many methods to protect themself such as Kernel Patch Protection [7] in Windows Vista, which is out scope of this paper.

3. DESIGN OF FILESAFE

3.1 BitVisor Architecture

Filesafe is based on BitVisor project. BitVisor is a tiny hypervisor designed for enforcing I/O device security [16]. It introduces a hypervisor architecture called *para-passthrough* for I/O access. The hypervisor only intercepts the I/O access that is configured to be watched, and others are passed through to the device. It is designed to minimize the code size of the hypervisor at the cost of running only one guest OS at the same time. Since BitVisor is a special purpose hypervisor for security, it is worthwhile to trade off the capability of running multiple guest OSes for the following benefits:

- It does not need any virtual devices which are shared among guest OSs. Including these virtual devices, such as QEMU [3], will greatly increase the code size of the hypervisor.

- The performance of the I/O devices in such architecture is almost equivalent to the native performance, because the guest OS operates physical devices directly.

In pursuit of simplicity and high performance, we leverage the cutting-edge Intel Virtualization Technology (Intel-VT) to make BitVisor core even smaller and faster. We replace SPT implemented by software with EPT feature supported by hardware. We eliminate real-mode instruction simulator by utilizing Unrestricted Guest feature supported by hardware. These will be detailed in Section 4.

3.2 Bridging The Gap

As mentioned in Section 1, we need to bridge the semantic gap between the view of the guest OS from the outside and the view from the inside. Therefore, File System Parser is designed to address this problem.

File System Parser is the bridge mapping files to blocks. It parses file system information from disk following specific file system standard. It cooperates with user when user is setting policy in hypervisor before guest OS starts up, so it does not incur any overhead to guest OS.

As shown in Figure 1, the flow of how to set policies is represented by dotted lines. After the machine is booted, user could log in our hypervisor to set his/her access policy of sensitive files before starting the guest OS. A console is offered to let user interact with our File System Parser. User could use command line to browse the file system, just the same as what we do in the OS. Then user could set read/write restriction to sensitive files, and after that both the block range in which these sensitive files are stored and access permission bits will be recorded in the block-level Policy List.

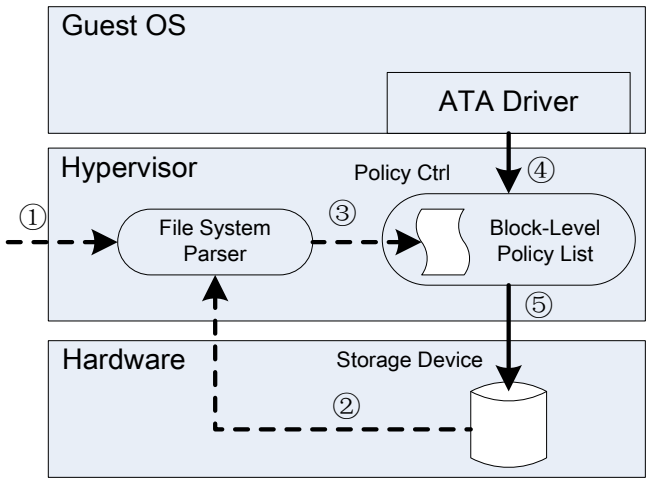

Figure 1: Design of Filesafe. Dotted lines represent the flow of how to set policies. Solid lines depict the flow of how policy control works.

Solid lines in Figure 1 depict the flow of how policy control works. When the guest OS is running, every read or write request to disk blocks will be intercepted to check whether it violates the policies in the Policy List, and if so, it will be silenced and the guest OS will receive a notification indicating that the operation is failed.

Currently, the Policy List is stored in the memory and will be lost after machine is powered off. It should be stored somewhere in the hard disk before machine is powered off and be read from disk after the hypervisor starts up. We will make this improvement in the future.

4. IMPLEMENTATION

4.1 EPT Implementation

Extended Page Table(EPT) is a feature supported in Intel-VT technology. The AMD equivalent is called Rapid Virtualization Indexing (RVI), formerly known as Nested Page Tables (NPT).

Due to the lack of hardware support for memory management unit (MMU) virtualization in first CPUs that support virtualization technology, virtual environments which support full virtualization introduce Shadow Page Table(SPT) to manage memory virtualization. SPT is a table maintained by the hypervisor. It maps virtual address to machine address while the guest OS still maintains its own page table which maps virtual address to physical address. The crucial defect of SPT is that each MMU address translation needs to be trapped into the hypervisor, which is termed as VM exit, and travels another execution path to fetch the real address. This introduces more than 10 times CPU cycles compared with the native MMU address translation.

To reduce this overhead, both AMD and Intel provide hardware assists to facilitate address translation. The hypervisor can now rely on hardware to eliminate the need for shadow page tables. EPT translates guest physical address to machine address, and any updates to guest page table will no longer incur a VM exit, which avoids much of the overhead.

SPT is implemented in BitVisor which has about 2,000 lines of code (including other related code), and high overhead is noticed in memory-bound benchmarks. In our work, we replace the SPT with EPT, which has only about 200 lines of code. To implement EPT, we first allocate memory space for this 4-level table and fill every page entry at each level with appropriate values. After that, the machine memory address of this table should be provided to VM. VM will do address translation according to this table. Our EPT implementation supports up to 4GB memory, the hypervisor lies at the top of 4GB memory, other memory belongs to the only one guest OS. Though we greatly reduce the code size, the overhead is not reduced as much. We observe more than 20% overhead in some benchmarks. This will be discussed in Section 5.

4.2 Unrestricted Guest Implementation

Unrestricted Guest is a feature supported in Intel-VT technology.

The first Intel processors to support Intel-VT technology require CR0.PE and CR0.PG to be enabled when starting a VM. This restriction implies that guest OS cannot run in unpaged protected mode or in real-address mode. In order to address this problem, BitVisor implements an instruction emulator to emulate instruction execution in real-address mode and the switch between real-address mode and protected mode. The instruction emulator has about 2,500 lines of code and incurs overhead at runtime.

Later processors which support Unrestricted Guest feature allow guest OS to run in unpaged protected mode or in real-address mode. The processor operates instructions of VM in these modes natively. Since these two modes do not use paging, each linear address is passed directly to the EPT for physical address translation, so EPT feature mentioned in the section above should be enabled as well.

Implementation of Unrestricted Guest is simple and straightforward. After EPT is setup, we enable Unrestricted Guest feature by setting corresponding bit in the secondary processor-based VM-execution control and set the right value to some registers. With the help of Unrestricted Guest feature, our implementation does not require an instruction emulator.

4.3 File System Parser Implementation

File System Parser helps the user map file to blocks the file occupies when he is setting file security policy. Usually, the most convenient way of doing that is utilizing the file system interfaces in OS, such as [5] does. But in that case, system vulnerabilities may be introduced in due to these untrusted interfaces. As to how to implement such a tool is file system specific, we only implement the support for FAT32 file system to prove our design is feasible.

File allocation table 32 (FAT32) [6] is a simple and prevailing file system. It consists of three important data structures: boot sector, file allocation table (FAT) and directory table. Boot sector is the first sector in a FAT32 partition. It contains BIOS parameter block (BPB), which portrays the key information of this partition. FAT is a table with 32-bit entry in FAT32 file system. Each entry represents the status of a cluster, which is the minimum unit of file allocation. All the files and directories are stored in one or more clusters, which are not necessarily continuous in the disk. Linked list data structure is used and FAT is the table that describes this structure. The value in each entry may indicate a free

cluster, the next cluster number of a file, this cluster being the last cluster of a file, a bad or reserved cluster. Directory table is a special type of file that represents a directory. Each file or directory stored within it is represented by a 32-byte entry in the table. Each entry records information of each file, such as name, attributes, size and especially the cluster number of the first cluster of file data.

To get the blocks that the file named A occupies, File System Parser first locates the root directory entry of the FAT32 partition. If A is in root directory, by matching the file name of each entry in this directory, the parser could get the file entry of A as well as the cluster number of first cluster of A's data. Next, it should query FAT to get the next cluster in linked list with the results of Formula 1 and Formula 2. When all the cluster numbers of A's data have been collected, cluster number should be converted to sector number with Formula 3 and Formula 4. At last, Logic Sector Number (LSN) is converted to Logical Block Address (LBA) by adding an offset because LSN is the sector index in this partition while LBA is the sector index in the whole disk. If A is in the subdirectory, as directories are treated as files that contain the directory information, the subdirectory should be accessed recursively the same as the procedure described above. The abbreviations used in formulas are list in Table 1.

Once we have the mapping from file to blocks, we could set the security policy, such as read only or not accessible, for all the blocks of sensitive files. When the guest OS is running, Policy Control will enforce these polices to protect sensitive files. Guest OS usually does disk access operations by specifying three parameters: LBA, number of sectors, and sector data. Operations will be intercepted and these parameters will be sent to Policy Control. In the view of guest OS, any attempt to read a read protect file will get an empty file and to write a write protect file will result in a failure. But deleting a write protect file will always be succeed, because in FAT32 file system, deleting a file will only set the first byte in the directory entry to 0xE5 rather than erase all the data of that file. It could be recovered with ease. Formatting a partition will also work well since formatting will only erase the data in FAT. As all the sector numbers of sensitive files are listed in Policy List, FAT could be reconstructed for these sensitive files. Filesafe does these recover work after the guest OS is shutdown.

$$FATEntrySN = BPB.RSC + (CN * 32/8)/BPB.BPS \quad (1)$$

$$FATEntrySO = (CN * 32/8)\%BPB.BPS \quad (2)$$

$$FirstDataSN = BPB.RSC + BPB.NF * BPB.SPF \quad (3)$$

$$LSN = FirstDataSN + (CN - 2) * BPB.SPC \quad (4)$$

5. EVALUATION

In this section, we first show the comparison of the TCB size of various contemporary virtual environments. Then we evaluate the performance of our implementation on an Intel machine with Core i5-650 3.20 GHz, 3GB DDR3 RAM and 250 GB Seagate SATA2 disk. A Windows XP with sp3 runs on this machine, and it will become a guest OS after our hypervisor is installed in.

5.1 TCB Size Comparison

Abbr.	
BPB	BIOS Parameter Block
RSC	Reserved Sector Count
NF	Number of FATs
SPF	Sectors Per FAT
SPC	Sectors Per Cluster
BPS	Bytes Per Sector
SN	Sector Number
SO	Sector Offset
CN	Cluster Number
LSN	Logic Sector Number

Table 1: Abbreviation

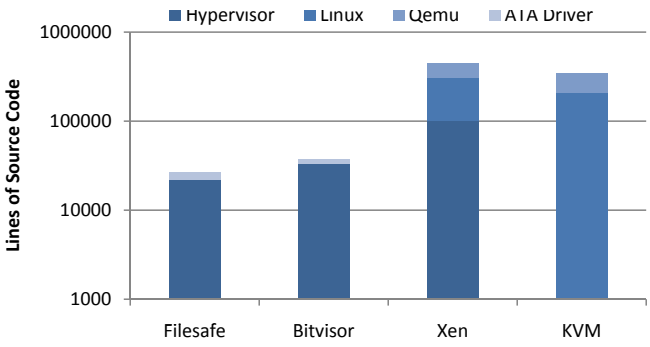

Figure 2: Comparison of the TCB size of virtual environments.

In Figure 2, we compare the size of the TCB for contemporary virtual environments. The total height of each bar indicates the TCB size of an operating system when it runs inside that VMM or hypervisor as a guest OS.

Filesafe consists of a modified BitVisor core that has approximately 20KLOC with a FAT32 file system parser that has 2.3KLOC, and the ATA driver which has 4.4KLOC. For Xen and KVM, we assume that all unnecessary functionality, such as unused device drivers and file systems, has been removed from the Linux kernel. The code size of that kernel is about 200 KLOC. By removing support for non-x86 architectures, code size of QEMU can be reduced to 140 KLOC [17].

5.2 Impact on Performance

We execute both compute-bound and I/O-bound applications with Filesafe. For compute-bound applications, we use the SPECint 2006 suite. The SPECint suite consists of a series of long-running computationally intensive applications intended to measure the CPU and memory management performance. For I/O-bound applications, we use Iometer to evaluate disk I/O performance.

Results of SPECint benchmarks are presented in Figure 3. On average, the overhead of our hypervisor is less than 8%, but the gcc and mcf benchmarks show more than 20% overhead. We attribute this to EPT performance overhead and hope the hardware will be improved in the near future. Besides, We also observe high overhead (up to 80%) measured by SPECint in other works [13, 19, 21] implemented with EPT support.

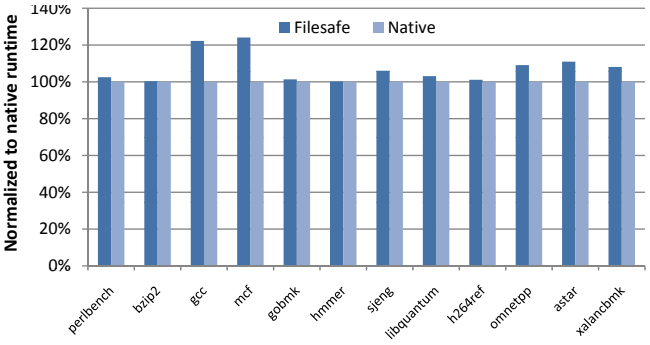

Figure 3: SPEC CINT 2006 Benchmarks. It shows the percentage of runtime overhead relative to the native Windows. Lower is better.

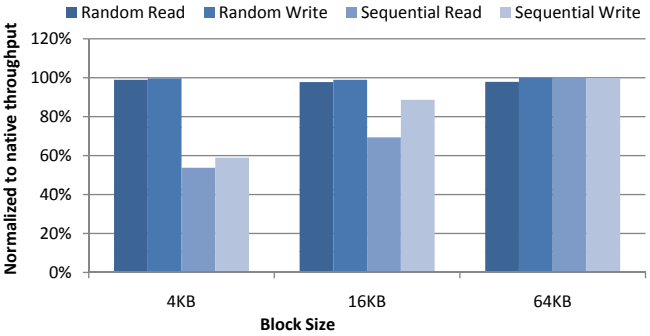

Figure 4: Iometer Benchmarks. It shows the percentage of I/O throughput overhead relative to the native Windows. Higher is better.

Results of Iometer benchmark are presented in Figure 4. We evaluate disk I/O performance of Filsafe with various block size and various operation types. When the block size is increased to 64KB or higher, the performance overhead is negligible. For the sake of brevity, results of block size larger than 64KB are not shown in the figure.

This figure indicates that the I/O operation overhead of Filesafe is a constant value, rather than in proportion to block size. Firstly, we focus on the bars whose block size is 4KB. We know that random operation is much slower than sequential operation. Sequential operation overhead of Filesafe is notable because sequential operation takes short time, while for random operation, overhead is not notable because random operation itself costs much more time. Secondly, we compare the bars whose block size is 4kb with others, as the block size increases and the I/O operations cost more time, and the overhead incurred by Filesafe becomes negligible.

6. RELATED WORK

File security issues such as confinement and protection have been studied extensively, and many approaches have been proposed to address these issues. In this section, we first introduce the out-of-the-box approaches that leverage virtualization technology, and then we describe classical in-the-box approaches. Finally, some other approaches are summarized.

6.1 Out of the Box Approaches

HyperSpector [11] is a virtual distributed monitoring environment that employs multiple Intrusion Detection Systems (IDS) to protect distributed system. A specific IDS VM mounts the file system of the target OS as shadow file system and enforces the access control on that file system. SVFS [20] takes a similar approach of using a Data Virtual Machine store and protect sensitive files for other application VMs. Though this approach may be a good solution for distributed systems, it is not suitable for our problem because introducing another VM inevitably increases the TCB size and performance overhead. Our approach does not need any additional VMs but only a thin hypervisor.

Livewire [8] is a VMM-based IDS, first proposing the methodology of inspecting a VM from the outside, which is termed as virtual machine introspection (VMI). Livewire does the introspection by primarily engaging in passive checks on the guest OS, which is similar to what Tripwire does, while our prototype intercepts malicious file accesses actively. The successor VMWatcher [9] is a malware detector rather than an IDS. It moves anti-malware out of the VM while still maintaining native view of that VM by using the technique named guest view casting, reconstructing internal semantic views of a VM from the outside [9]. Both Livewire and VMWatcher encounter the semantic gap problem and their solutions give us lots of inspiration.

BitVisor [16] is a thin hypervisor for enforcing I/O device security. It proposes the *parapass-through* architecture, which makes most of the I/O requests from the guest OS go directly to devices instead of handling by the hypervisor. The later work [5] utilizes BitVisor to prevent persistent rootkits by protecting system-critical files from writing operations. Besides, the later work introduces a mode named *Administrator Mode* in which they make security policy by using user-level tool in guest OS, while in this mode guest OS is assumed to be secure. However, our approach makes no assumption about the guest OS. We leverage the methodology of VMI instead of depending on OS interfaces.

6.2 In the Box Approaches

Classical approach for protecting sensitive files is to use a mandatory access control system [12], which commonly belongs to the operation system. The access control is enforced at the OS kernel level, which makes it is hard to be bypassed by user-level intruders.

Tripwire [10] and I^{3}FS [14] are both file integrity checkers, which validate specified files by comparing cryptographic hash values of them with trusted values. The essential difference between them is Tripwire runs in user mode and I^{3}FS runs in kernel mode. They may detect any file modifications but they do not protect sensitive files from malicious access.

These in-the-box approaches all depend on one assumption that the guest OS kernel is trustworthy. However, under our threat model, if the whole guest OS is compromised, these approaches may hardly take effect.

6.3 Other Approaches

NFS [15] could be configured to protect sensitive files by enforcing access control on NFS server. This approach is more convenient and it also provides strong isolation. However, including external NFS server and network connection

makes it vulnerable to network related attacks.

Another idea proposed by Strunk et al. [18] is that using disk-level versioning protects data in a compromised system. Old versions of data are stored in a NFS-like server named Self-securing storage server (S4) for a window of time, within this window, administrators have a wealth of information for data recovery. The longer window of time is, the more disk space is consumed. It may cancel malicious writing operations of sensitive data by recovering from old versions, but it offers limited confinement to keep intruder from reading the sensitive data. Besides, it does not prevent malicious file accesses in real time.

Finally, file encryption could be regarded as an auxiliary approach where access control could not be enforced (e.g. Intruder has physical access to hard disk).

7. CONCLUSION

We presented a special purpose hypervisor called Filesafe, which could protect sensitive files from unauthorized access even if the OS is compromised. We described two primary challenges to design this kind of hypervisor. One is the semantic gap between hypervisor and guest OS. The other is the security of hypervisor itself. We implemented a prototype of our design, which has a File System Parser to bridge the semantic gap. Moreover, our implementation has a very small code base that makes verification feasible. Our work demonstrates that such a hypervisor could be built with negligible performance overhead and such a small TCB size, thanks to recent hardware virtualization features and BitVisor architecture.

In order to guarantee the load-time integrity of hypervisor, we plan to apply Intel TXT technology to Filesafe in the future. We will also make Filesafe support various prevailing OSes and file systems. And some technical details will be improved to make Filesafe more practical.

8. ACKNOWLEDGEMENT

This work is supported by Key Lab of Information Network Security, Ministry of Public Security, National Natural Science Foundation of China (Grant No. 60873209, 60970107, 60970108,61073151), the Key Program for Basic Research of Shanghai (Grant No.10511500100, 10DZ1500200), IBM SUR Funding and IBM Research-China JP Funding.

9. REFERENCES

[1] *Intel Trusted Execution Technology Architecture Overview*. Intel Corporation, Denver, CO, 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Technical Conference*, pages 41–46, 2005.

[4] P. M. Chen and B. D. Noble. When virtual is better than real. In *Workshop on Workstation Operating Systems (now HotOS)/Workshop on Hot Topics in Operating Systems*, pages 133–138, 2001.

[5] Y. Chubachi, T. Shinagawa, and K. Kato. Hypervisor-based prevention of persistent rootkits. In *SAC*, pages 214–220, 2010.

[6] M. Corporation. Microsoft extensible firmware initiative fat32 file system specification. (12), 2000.

[7] M. Corporation. Kernel patch protection: Frequently asked questions, Jan. 2006.

[8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium*, 2003.

[9] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Computer and Communications Security*, pages 128–138, 2007.

[10] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, CCS '94, pages 18–29, New York, NY, USA, 1994. ACM.

[11] K. Kourai and S. Chiba. Hyperspector: virtual distributed monitoring environments for secure intrusion detection. In *International Conference on Virtual Execution Environments*, pages 197–207, 2005.

[12] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Technical Conference*, pages 29–42, 2001.

[13] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[14] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok. I3fs: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the 18th USENIX conference on System administration*, pages 67–78, Berkeley, CA, USA, 2004. USENIX Association.

[15] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. 2000.

[16] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *VEE*, pages 121–130, 2009.

[17] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys*, pages 209–222, 2010.

[18] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. *Foundations of Intrusion Tolerant Systems*, 0:195, 2003.

[19] M. Xia, M. Yu, Q. Lin, Z. Qi, and H. Guan. Enhanced privilege separation for commodity software on virtualized platform. In *ICPADS*, pages 275–282, 2010.

[20] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *IEEE Security in Storage Workshop*, pages 21–28, 2005.

[21] M. Zhu, M. Yu, M. Xia, B. Li, P. Yu, S. Gao, Z. Qi, L. Liu, Y. Chen, and H. Guan. Vasp: virtualization assisted security monitor for cross-platform protection. In *ACM Symposium on Applied Computing*, pages 554–559, 2011.