# Virtual Machines Jailed

## Virtualization in Systems with Small Trusted Computing Bases

Michael Peter, Henning Schild, Adam Lackorzynski, Alexander Warg
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
{peter, hschild, adam, warg}@os.inf.tu-dresden.de

## ABSTRACT

The trusted computing base of legacy applications can be reduced significantly by separating their security–critical parts into dedicated protection domains. As yet, paravirtualization has been used to host the non–secure portion. The applicability of this approach is limited by the need of source code access. We show how to implement efficient virtual machines in a microkernel–based system enabling the reuse of arbitrary operating systems. We found that the performance is on par with other virtual machine implementations, while security–sensitive applications retain their small trusted computing base. In fact, the kernel growth is marginal (500 SLOC), other security–critical components are not affected.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security Kernels; D.4.8 [**Operating Systems**]: Performance

## General Terms

Design, Security, Performance

## Keywords

Virtualization, Small Trusted Computing Base, Secure Systems, Microkernel

## 1. INTRODUCTION

Visiting the web page of his bank, a user must trust software on his machine that can easily comprise millions of source lines of code (SLOC). Contemporary web browsers, such as Firefox, which alone accounts for 2.5 million SLOC[1] are targeted recurrently[2]. Private data may be exposed possibly resulting in personal distress or financial losses.

Considering the size of trusted components, the situation is similar at the system level. A minimal configuration of the Linux kernel contains about 200,000 SLOC and the majority of production configurations are significantly larger[2]. With such a size, vulnerabilities are unavoidable[3]. Even worse, the attacker does not need to compromise the kernel, as it is sufficient to gain control over one of several processes running with superuser privileges. For example, current X11 implementations comprise 1.25 million lines of code. Buffer overflows, as observed in the past [4], have the potential to seize complete control over the machine.

With about two defects per 1,000 SLOC [17] produced even by leading software-development organizations, error-free software is elusive in the foreseeable future and other mitigations should be considered. Proposed for more than three decades [22], subdividing software into trusted and untrusted components has eventually been recognized as the best practice [24, 12, 21, 20, 11].

Microkernels have a record for being a suitable foundation upon which highly decomposed systems can be built. As for any novel system, lack of applications is an issue. From the technical point of view, building a completely new operating system stack with desirable properties followed by porting applications is the most appealing solution. But it also comes with the highest costs. A more pragmatic approach is to port whole legacy systems which obviates changes to applications. Nonetheless the kernel has to be adapted to run on top of the microkernel instead of bare hardware. Although possible, porting is labor–intensive or even impossible without source code access.

In this paper, we show how a microkernel can be extended to support CPU and memory virtualization both of which are crucial for the implementation of virtual machines (VMs). We start with a small system where the security critical core system, i.e. the kernel and base infrastructure, contains less than 150,000 SLOC. Adding support for virtualization resulted in a modest growth of 500 SLOC in kernel size, which was the only increase of the trusted computing base (TCB) for security–concerned applications.

We will proceed as follows: Section 2 will revisit the fundamentals, before the design is described in Section 3. Our evaluation in Section 4 shows that the performance is on par with established solutions, which is noteworthy because previous secure systems often had to pay a security tax. Related work will be discussed in Section 5 and will be followed by concluding remarks.

[1] Firefox 3.0 measured with David A. Wheelers `SLOCCount`

[2] Linux 2.6.29 comprises about 7 million SLOC

## 2. BACKGROUND

### 2.1 Microkernel-based Systems

Microkernels grew out of the insight that error–prone device drivers should not reside in a location where isolation cannot be enforced; and neither should resource allocation policies, which have the potential of yielding much better performance, if user–level knowledge is incorporated.

Following the L4 [16] philosophy, L4/Fiasco aims at minimality in the kernel. Functionality is only admitted into the kernel if either it cannot be implemented at user–level without compromising on security, or an user–level implementation had a severely adverse impact on performance. Page table related operations (address space construction) are an example for the former, scheduling is one for the latter.

As communication overhead had proven to be crucial for the viability of a microkernel, early work on L4 was primarily focused on inter-process communication (IPC) performance, which was a prerequisite for fine-grained isolation. When this problem was solved the attention turned towards security, where the global name space was swiftly identified as problematic. Various development lines arose all with the goal to adopt a capability–based access control mechanism, which had been advocated as the method of choice [19]. Our efforts have resulted in L4/Fiasco, a microkernel which employs capability mediation for all kernel objects.

In the remainder of this section, we will briefly introduce L4 mechanisms as far as they are needed for the comprehension. An L4 task embodies a protection domain and serves as a container for resources such as memory. L4 threads, the unit of scheduling, execute in tasks and communicate via synchronous IPC. Apart from simple message exchange, IPC is also used for capability transfer and fault reflection. In L4–speak, granting capabilities via IPC is called *mapping*, the revocation is called *unmapping*. A *pager* is a program that fulfils the role of handling faults by mapping capabilities.

The size of an appropriately configured development version of L4/Fiasco is approximately 26,000 SLOC, which is two orders of magnitude smaller than current commodity monolithic kernels.

A microkernel is of little use on its own. It is rather the microkernel–based system that provides utility to the end user. The execution environment needs to supply basic functionality, such as memory management, CPU and device resources, infrastructure and service discovery, or common application libraries.

Based on the kernel mechanisms we developed a runtime environment that provides this set of abstractions. It's overall structure is hierarchical and allows the construction of isolated compartments with differing characteristics.

Some flavor of virtualization is the most convenient way to provide access to legacy applications. Compared to full virtualization, paravirtualization has lower demands on the kernel but requires source code modification to the OS. Our software stack offers a paravirtualized version of Linux, L$^4$Linux [13], which is binary compatible with its original version. This allows to run unmodified Linux programs on L4/Fiasco.

Despite its potential to remove the source code dependency, the integration of virtual machines into microkernels was hold back by virtualization deficiencies of the IA32 ISA.

### 2.2 Splitting Applications

The conceptionally most appealing approach — rebuilding an application with a small TCB — is impractical in most cases. This can be illustrated with the example of a contemporary web browser: Given that a browser runs on a secure operating system, it still has a TCB of hundreds of thousands, if not millions of lines of code. Modern web pages apply sophisticated transformations to adapt their presentation. They often use scripting, which is indispensable for rich web applications, and employ plugins for more elaborate dynamic content.

If an application has to be available then there is no other choice than to make sure that *all* contributing components are also available. Fortunately, security [3] has less stringent requirements. For many use cases it is acceptable that an application fails as long as there are no harmful effects.

An example is an online banking transaction as described in [23], where a full featured web browser is usually involved. The user authorizes a bank transfer with a *Transaction Authentication Number* (TAN) where it needs to be made sure that the TAN is not stolen or data of the transaction is modified by intruders. In a split application scenario the user uses his regular web browser but the final transaction is displayed by a secure component and the TAN is entered only there. The secure side also handles the encryption of the transaction, which is finally sent to the bank via the legacy operating system. This scenario handles threats like keyloggers or viruses that exploit vulnerabilities somewhere in the sizable software stack.

## 3. DESIGN AND IMPLEMENTATION

The objective of our design is the support of virtual machines in an environment that at the same time allows for secure applications with a small TCB. Secure applications may use VM–hosted functionality through the *split application* design pattern (see Section 2.2).

### 3.1 Virtualization Aspects

The implementation of a virtual machine can be broken down into virtualization of the CPU and memory and device virtualization.

CPU and memory virtualization are tightly coupled and critical to the performance of a VM. It is desirable to execute as many instruction as possible natively on the processor. Recent virtualization extensions in commodity processors allow for efficient CPU virtualization with low software complexity. Switching CPU execution modes is a highly intrusive operation that involves page tables defining the execution environment. Since only the kernel can guarantee the validity of page tables, this operation has to be integrated into the kernel's protection domain management. The guest in a VM employs it's own virtual memory defining it through page tables in (guest) physical memory which translates guest virtual addresses[4] to guest physical ones. On an actual machine, these physical addresses would be used to refer to a memory location. Virtualization adds an additional translation step wherein a guest physical address is further translated into a host physical one.

[3] In this paper, we restrict ourselves to confidentiality and integrity when referring to security. Availability cannot be assured by splitting applications.

[4] IA32 terminology speaks also of linear addresses.

In the general case, the number of VMs exceeds the number of actual devices which necessitates device sharing. Since commodity hardware often does not provide virtualization support, the arbitration has to be done in software. Typical strategies are sharing (e.g. timers), partitioning (e.g. mass storage devices) or time multiplexing (e.g. network devices). Best practice is to expose a device model to the the guest, intercept interactions with it, analyze those and translate them into requests for the actual host devices.

## 3.2 Kernel Extension

Representing protection domains, the microkernel's tasks were a natural choice to represent the execution environment of VMs as well. In our design each VM has a task associated. Memory, mapped into this task's address space, will appear as physical memory in the virtual machine. The secure memory delegation mechanism of L4 ensures that VMs can only access memory that has been granted to them.

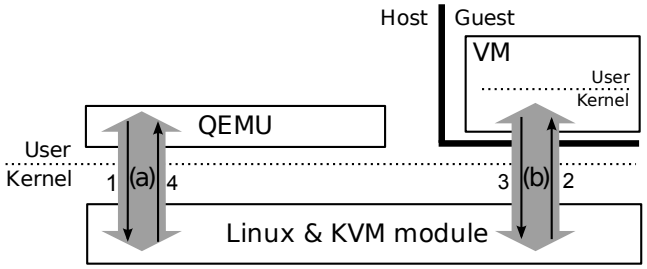


**Figure 1: KVM control flow**

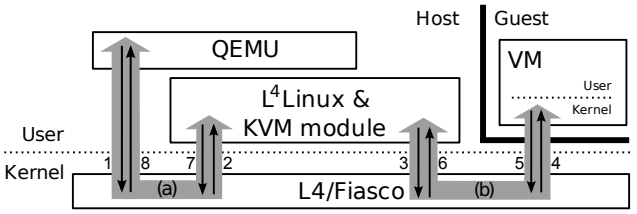


**Figure 2: KVM-L4 control flow**

The host has two choices how to implement the translation from guest virtual to host physical addresses[5]. It can run the guest with page tables that translate directly from guest virtual to host physical addresses. For this to work, the host needs to intercept all guest page table manipulations and translate each guest page table into a corresponding shadow page table. Unfortunately, the overhead for shadow page tables is significant. The frequent and expensive page table intercepts can be avoided if both guest and host translation are implemented in hardware. A processor therefor uses two distinct page tables. It is noteworthy that only the page table that translates to host physical addresses is security–critical which obviates the interception of guest page table modifications.

Figure 1 shows a world switch on a monolithic kernel. KVM is a typical representative of a Linux-based virtual machine monitor (VMM), any other VMM would also serve well as an example. Running a virtual machine starts with the switch (a) from QEMU to the Linux kernel (1). KVM will then execute a world switch (b) to the VM (2). Switching back occurs when events need to be handled (3). After events that do not require a return to QEMU, such as expiring timers driving the Linux scheduling, KVM resumes execution (2). For more complex assistance such as device emulation control is transferred to QEMU (4).

As L$^4$Linux runs in user–mode KVM lacks the CPU privileges to execute a VM switch directly. To compensate we added a system call to L4/Fiasco which takes a CPU state and a task capability as arguments. The task defines the physical memory of the VM and the execution resumes with the given CPU state. We denote the version of KVM adapted to L4/Fiasco KVM-L4. Figure 2 illustrates how a world switch looks like in our design. Compared to Figure 1, the number of privilege transitions has doubled. Transferring control from QEMU to L$^4$Linux requires an L4 IPC (1,2) instead of a single syscall. The same applies to the step where the KVM-L4 code within L$^4$Linux switches to QEMU (7, 8). Our new syscall enters L4/Fiasco (3) and switches to the VM (4). Control transfer from the VM back to KVM-L4 again involves two privilege transitions (5,6).

The format of the page tables used to translate guest physical to host physical addresses — often referred to as nested page tables — is specific for a virtualization extension. In the general case, the kernel has to derive a nested page table from the VMs task page table. Such a derivation is straightforward and not performance–critical but increases the kernel memory footprint. Our implementation capitalizes on the fact that AMD's SVM uses the same format for both nested and regular page tables. Since the kernel already maintains page tables for the VM task's address space, we could reuse them with little modifications. The only difference is that an address space used for a virtual machine does not contain the microkernel[6].

The code we had to add to the microkernel to support all the described functionality comprises about 500 SLOC.

## 3.3 Virtual Machine Monitor

We use the term *virtual machine monitor* for two different meanings. In the broader sense, the virtual machine monitor is the sum of all components that are needed to create the illusion of a machine. In that case the VMM is not a single component but rather comprises multiple parts as depicted in Figure 3.

In a stricter sense, VMM refers to the component that coordinates the execution of a VM. While doing so it may make use of kernel services, e.g. for CPU and memory virtualization, or secure device managers for device virtualization. In our architecture (Figure 3), QEMU acts as VMM. Under this meaning, the VMM is a regular user activity and not particularly interesting from a security point of view. In most cases, the intended meaning should be clear from the context so that we do not explicitly denote the subcase.

The device model can be rather complex. Since it is implemented in a protection domain, it does not pose a security risk so that an untrusted existing VMM can be reused.

Regarding device drivers, the VMM has to be treated

[5]Processor execution modes without address translation enabled are of no practical interest and can be handled by software emulators.

[6]The L4/Fiasco kernel normally resides in the upper portion of each task which accelerates kernel–user transitions.

as any other activity accessing devices. Each component with access to DMA–capable devices may circumvent page table based isolation as DMA operations are not subject to this mechanism. If the VMM hosts device drivers it has to be trusted. Pragmatically, we complement our VMM with L$^4$Linux and its rich assortment of device drivers. The question of secure device drivers is beyond the scope of this paper.

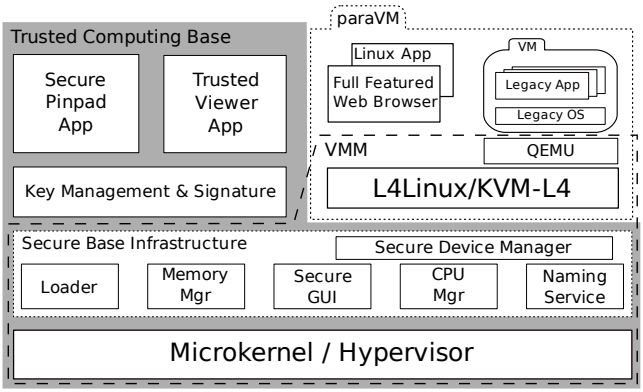


**Figure 3: System Architecture.**

*Kernel–based Virtual Machine* (KVM) [1] is an open source virtualization implementation for the Linux kernel. It comprises a kernel module, which is responsible for CPU and memory virtualization making use of the virtualization extensions available with recent processors [6, 14]. Guest device emulation is handled by a slightly modified version of the QEMU [8] machine emulator.

All our adaptations are limited to the part of KVM running inside the Linux kernel, KVM-L4 uses the KVM–QEMU as it is. The changes to the KVM code comprise about 100 SLOC. Although this number is not relevant for the TCB of secure applications running next to virtual machines , it is still noteworthy as we can easily benefit from the ongoing KVM development with modest merge overhead.

The slight modifications needed for KVM suggest that the adaption of other VMMs, e.g. VirtualBox, should be straightforward. Running a VMM directly on L4/Fiasco is also conceivable with the possible use case of a rudimentary VMM that only virtualizes core devices, such as timers and interrupt controllers, which are owned by the microkernel. All other devices would be exposed directly to the guest possibly with their activities restricted by an IO–MMU. Such an arrangement would allow to run a Linux much less modified than L$^4$Linux.

## 3.4 Limitations

Using L$^4$Linux's device drivers without further provisions makes L$^4$Linux part of the TCB. It has been shown that device drivers can be contained by an additional validation layer albeit at a noticeable performance cost. The impending availability of IO–MMUs [5, 15] gives reasonable hope for a solution that integrates well into a microkernel system and is both general and well performing.

Our implementation does not yet support direct communication of applications inside virtual machines with processes running outside a VM or in another VM. This feature is required for the split application scenarios described in Section 2.2. We expect the implementation of a communication service through guest device drivers to be straightforward and applicable to a wide range of guest operating systems, including closed source products.

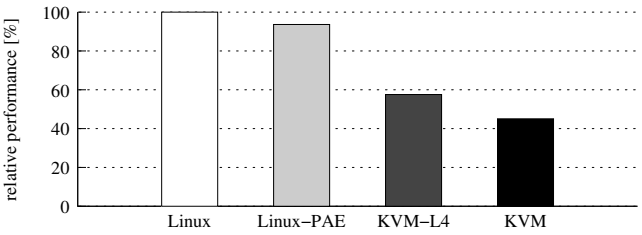


**Figure 4: Virtual Memory Benchmark**

# 4. EVALUATION

## 4.1 Performance

In this section we present the results of our experiments. We used an AMD Phenom$^{TM}$ 9550 Quad–Core CPU on an ASUS M3A78–EM motherboard equipped with 2GB DDR2–800 RAM and a Samsung HD080HJ hard disk. On the host we ran version 2.6.27 of both, L$^4$Linux and Linux. KVM-L4 is based on KVM version 79. Hardware virtualization ran always with nested paging enabled and used 32–bit mode for both, the host and the guest. In all our experiments only one of the four available cores was used. The respective VMs were always assigned one virtual CPU and 512MB RAM. For the guest's hard disk we used *virtio*, which relies on paravirtualized drivers.

The first benchmark we ran was a microbenchmark measuring the duration of context switches from the host domain to the guest domain and back. This benchmark helped us to quantify the overhead introduced by our design modifications. Subsequent application benchmarks revealed that the increased context switch duration of KVM-L4 only has a slight impact on the performance. To our surprise, KVM-L4 even performed slightly better than KVM in some cases. After some investigations we concluded that the most likely explanation can be found in KVM's PAE overhead.

### 4.1.1 Microbenchmarks

Slipping a microkernel/hypervisor underneath invariably increases the number of context switches. Our first microbenchmark quantifies the induced overhead. KVM ships with a benchmark for measuring the average duration of a world switch (Figures 1 and 2 scheme (b)). We implemented another benchmark that measures the duration of complete round–trips including the switch to QEMU and back (scheme (a) and (b)). Concerning the world switch, we measured an average of 2742 cycles for KVM and 3684 cycles for KVM-L4. The switch, which also includes switching to QEMU and back (scheme (a) and (b)), takes 4361 cycles for KVM and 14132 cycles for KVM-L4. Hence, the detour through the microkernel increases the duration of the world switch by 34%, for the full round–trip the increase is 224%. The high increase for the round–trip is because address space switches and cache flushes are included in our implementation.

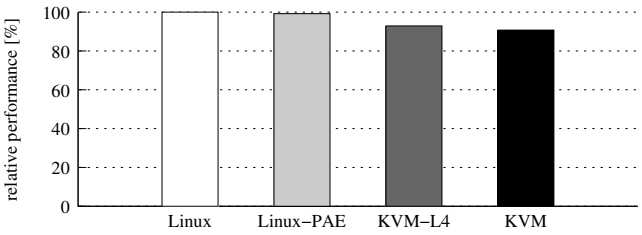


**Figure 5: Linux–kernel compilation**

To our surprise, KVM-L4 showed better performance than KVM in some of our application benchmarks. At first, we did not have an explanation because all of our design changes should degrade performance due to additional context switches and larger TLB and cache footprints.

We also ran a benchmark addressing virtual memory management. The benchmark reads 256 MB virtual memory in a long running loop. The area is constructed out of 2 MB of system memory using alias mappings. The results of that benchmark are illustrated in Figure 4. Notice that in this experiment KVM-L4 actually performs better than KVM. We suppose that this is because of different TLB fill costs that depend on the number of levels of the page table. Under KVM nested paging can only be used with three levels (PAE enabled on the host). The depth of the page tables affects the page walk complexity and the TLB pressure. Enabling the third level in native Linux (`Linux-PAE`) decreased performance by 6% without virtualization being involved. The depth of the nested page table goes in as a multiple of the guest page table depth when calculating the page walk complexity. It therefore also affects TLB pressure. Bhargave et al. [9] go into details on nested paging and suggest keeping the number of nested levels low in order to reduce the memory management overhead.

### 4.1.2 Application Benchmarks

To evaluate the application performance of KVM-L4, microbenchmarks are not sufficient because they stress specific functionality only, whereas application performance depends on a larger set of features. A widely used benchmark, compiling the Linux kernel (version 2.6.27.9) was our choice for a complex workload. The results of this evaluation are illustrated in Figure 5. Although the control flow switches are slower on our adapted version of KVM, the overall performance for kernel compilation turned out to be slightly (about 2%) better. Using a two level nested page table compensates for the slower switching in this benchmark.

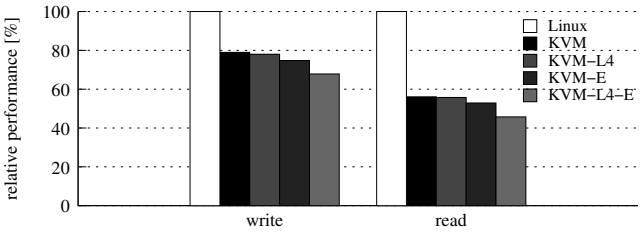


**Figure 6: `dd` input–output Benchmark**

Compilation primarily stresses virtual memory management and the CPU. Another limiting factor for virtual machine performance is input–output (I/O). Workloads issuing I/O operations involve QEMU interaction, as already shown these switches are much more expensive on KVM-L4 than they are on KVM. In order to analyse the impact of slower switching on I/O performance, we used the Linux tool `dd` to read/write 512 MB from/to hard disk in blocks of 1 kB. Before each run we cleared the Linux buffer caches to really cause I/O operations. When measuring a VM, we also flushed the host's caches. Paravirtual devices reduce the number of context switches induced by I/O and therefore improve performance in contrast to full device emulation. To show this improvement, we also ran this benchmark without the paravirtualized hard disk provided by *virtio*. The results of that benchmark are given in Figure 6 . It illustrates that the I/O performance of KVM-L4 is slightly worse in comparison to KVM. The big loss against native Linux is implied by using a virtual machine and also applies for KVM. Comparing the scenarios using *virtio* to the ones using full device emulation (`KVM-[L4]-E`) shows that KVM-L4 benefits more from that technique than KVM. This is also what one would expect keeping in mind that KVM-L4 has slower round–trip switching and paravirtualized devices reduce the number of these switches.

## 4.2 Guest Support

Besides Linux, which we used for the performance evaluation, we also ran several other guest operating systems successfully on our VM. Among them, L4/Fiasco, Microsoft Windows XP and Windows 7 Beta 1, OpenSolaris, and FreeBSD. Except for the current limitation to 32–bit guests, we expect KVM-L4 to be feature compatible to KVM.

# 5. RELATED WORK

The reduction of the TCB size has been investigated by various researchers for some time now. Singaravelu et al. [23] proposed a very similar architecture. Limited by the absence of hardware virtualization support, they had only paravirtualization at their disposal, which excludes closed source operating systems and applications tied to them.

Biemüller and Dannowski [10] proposed an extension to the L4 API to enable virtual machines on L4–based systems. The kernel version used as starting point did not support capabilities, which is a drawback, if dynamic in the secure part shall be supported. With the proposal, neither architectural aspects above the kernel are covered, nor is there an evaluation of a reference implementation available.

Murray et al. [21] showed how the TCB of a *Xen*–based [7] system can be minimized. The lack of light–weight protection domains and a capability–based access control mechanisms is likely to hamper the evolvement of secure applications that act more dynamically.

Flicker[18] employs a different architecture and forgoes a kernel. Instead it relies on hardware support to ensure the integrity of its execution environment. This approach is limited to a single secure application at any given instant in time which may become problematic for secure environments that consist of multiple components.

# 6. CONCLUSION AND OUTLOOK

In this paper we presented a design and implementation of a VM in a secure environment. The size of the TCB has only

marginally increased and remains with 150,000 SLOC about two order of magnitudes smaller than those of contemporary monolithic kernels. Experiments have shown that hosting a virtual machine on a system with a small trusted computing base is feasible and does not incur significant performance losses.

The upcoming challenges of microkernel systems, such as multiprocessor support and kernel resource management, will necessitate compromises between simplicity and performance. If performance–critical activities can be contained in virtual machines, chosing the simpler version might be acceptable. Future versions of KVM-L4 may add support for Intel's virtualization technology, including it's version of nested paging, as well as support for systems without nested paging. We are currently working on the implementation of a multi-core version of L4/Fiasco, which will enable guests to benefit from multi-core machines. Besides all the discussed points, the foremost problem with virtualization in secure environments is secure device virtualization. forthcoming IO–MMUs should facilitate the removal of device drivers from the TCB.

## 7. ACKNOWLEGDEMENT

We would like to thank Björn Döbel and the reviewers for their constructive criticism on the way to this version of the paper. Authors of this paper have been partially supported by the European Research Programme FP7.

## 8. REFERENCES

[1] KVM - Kernel-based Virtualiztion Machine. White paper, Qumranet Inc., 2006.

[2] Mozilla foundation security advisories, 2009.

[3] Vulnerability Report: Linux Kernel 2.6.x, 2009.

[4] Vulnerability Report: X Window System 11 (X11) 7.x, 2009.

[5] Advanced Micro Devices. *AMD I/O Virtualization Technology (IOMMU) Specification*, rev 1.20 edition, 2007.

[6] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, rev 3.14 edition, 2007.

[7] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, New York, NY, USA, 2008. ACM.

[10] S. Biemueller and U. Dannowski. L4-Based Real Virtual Machines - An API Proposal. In *Proceedings of the MIKES 2007: First International Workshop on MicroKernels for Embedded Systems*, pages 36–42, Sydney, Australia, Jan. 16 2007.

[11] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[12] M. Hohmuth, M. Peter, H. Härtig, and J. Shapiro. Reducing TCB size by using untrusted components - small kernels versus virtual machine monitors. In *in Proc. of the 11th ACM SIGOPS European Workshop*, page 22. ACM Press, 2004.

[13] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of $\mu$-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, 253669-028us edition, 2008.

[15] Intel Corporation. *Intel Virtualization Technology for Directed I/O*, rev 1.2 edition, September 2008.

[16] J. Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, Dec. 1995.

[17] Y. K. Malaiya, Y. K. Malaiya, J. Denton, and J. Denton. Estimating Defect Density Using Test Coverage. Technical report, 1998.

[18] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, New York, NY, USA, 2008. ACM.

[19] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability Myths Demolished, 2003.

[20] D. G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *EUROSEC '08: Proceedings of the 1st European workshop on system security*, pages 40–46, New York, NY, USA, 2008. ACM.

[21] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, New York, NY, USA, 2008. ACM.

[22] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[23] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.

[24] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.