Recursive Virtual Machines for Advanced Security Mechanisms

Bernhard Kauer
Universidade de Lisboa
Faculdade de Ciências
LaSIGE
kauer@fc.ul.pt

Paulo Verissimo, *Fellow, IEEE*
Universidade de Lisboa
Faculdade de Ciências
LaSIGE
pjv@di.fc.ul.pt

Alysson Bessani
Universidade de Lisboa
Faculdade de Ciências
LaSIGE
bessani@di.fc.ul.pt

***Abstract*—Virtualization contributes to the optimization and modularization of resource usage in a machine. Furthermore, many systems have relied on a virtualization layer to provide extra security functionality. Both features rank amongst the most important of the technological capabilities enabling cloud computing, improving performance and security.**

The availability of hardware support for x86 virtualization allows to run virtual machines (VMs) with very low overhead. However, using hardware virtualization inside the OS makes it unavailable for any additional security code as the hardware supports only a single layer of VMs. Stacking virtual machines recursively is one solution to this problem. Unfortunately, current implementations induce an overhead that grows exponentially with the stacking depth.

In the paper we address this conflict by describing a novel design that mitigates the performance issues of recursive virtual machines. Once this solved, the doors are open for the design of advanced security mechanisms that are implemented in the intermediate layers and provide additional security features to the system. We suggest concrete ways to further explore this avenue.

I. Motivation

Virtualization has contributed to a better resource utilization of a machine. Its generalized use is bound to be amplified by the emerging cloud computing paradigm. In fact, cloud computing is essentially a business model, made possible by some technological advances that enable modularization of resources. Virtualization is probably the most important of these technological capabilities and as such has been deserving considerable attention in cloud-computing related research, since it is not exempt from drawbacks. One issue we address in this paper is the conflict between performance and security.

Virtualization was used to improve the security of a machine's OS or applications: hypervisors spawn virtual machines to protect the kernel code [13] or application data [5]; they provide a trusted execution environment [10] and implement fault tolerance [4]; they detect intrusions [8] and help in analyzing them [6]. There are different reasons why virtual machines are an appealing environment to improve an operating system. First, they allow to execute the target OSes unmodified. This minimizes the development effort and allows to reuse closed source software. Secondly, a virtual machine provides an already established interface that is independent of OS APIs. Porting a solution to another OS is therefore relatively easy. Finally, security-critical code can be protected against attacks by relying on the virtualization layer for additional security defense lines.

A drawback of VMs is the performance overhead of a fully virtualized hardware architecture. Fortunately, the introduction of hardware support for CPU virtualization [16] and nested paging [3] have significantly reduced these overheads [15]. Furthermore, they have simplified hypervisor implementations, because they made older techniques such as binary translation [1] obsolete. However, the success of hardware support has led to another problem: operating systems itself like to run hardware-accelerated virtual machines. Windows 7, for example, relies on VMs to start legacy XP applications. The security layers now compete with the OS to be the hypervisor, because the current hardware only supports one of them in the system.

Such a restriction can be avoided by nesting two hypervisors [9], [18]. One becomes the outer hypervisor (HV) and runs directly on the hardware, whereas the other one will run inside the virtual machine the first HV provides. However, nested virtualization is not sufficient, if more than two layers need to be available. Generalizing nested virtualization leads to *recursive virtual machines*, where the nesting depth is only limited by the platform resources [12]. Figure 1 shows an example where two OSes are running inside recursive virtual machines. The first OS runs directly on the root hypervisor. The second OS is wrapped by three hypervisors. The intermediate hypervisors (HV_1, HV_2) can be used to provide additional security features to the system.

Nevertheless, there is a major problem to be solved before the benefits of recursive virtualization can be fully enjoyed. Previous work has argued that hypervisors based on hardware architectures can not be efficiently nested, as the performance *worsens exponentially with* the *stacking depth* [7]. One challenge of this paper is to show that virtual machines can indeed be stacked with a performance overhead that is (largely) independent of the nesting level. Once this solved, the doors are open for the design of advanced security mechanisms, and this avenue will be further explored in the paper.

This work was partially supported by the EC through FP7-ICT project TCLOUDS, and by the FCT through the Multiannual and the CMU-Portugal Programmes.

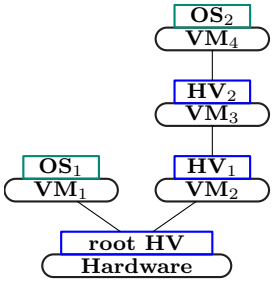

Fig. 1. Recursive virtualization with three hypervisors. The root HV multiplexes the hardware, whereas the other hypervisors wrap the OS to implement security functions.

Structure: The rest of this paper is organized as follows: We first review related work (§ II). Then we analyze the overheads of existing nested VM approaches (§ III). We introduce a new design for recursive virtualization (§ IV) that avoids these performance penalties. Furthermore, we suggest several ways in which to take advantage of high-performance recursive virtualization to improve OS and application security, in § V. Finally, we draw conclusions (§ VI).

II. Related Work

The theoretical requirements of recursive virtualization were first described in the 70s by Popek and Goldberg [12]. Multiple hardware architectures were proposed to build recursive virtual machines for example by Belpaire and Hsu [2].

More recent work has focused on nesting two VMs on top of hardware support for virtualization. Graf and Roedel [9] show for the first time that nested VMs can be efficiently implemented with AMD processors. Later, Yehuda at al. [18] extended nested virtualization to Intel's virtualization extension (VMX). They also described the folding of two layers of nested page-tables and the use of direct device assignment to optimize the I/O path. For more than two layers of virtual machines they assumed that the *emulation of VMX can work recursively*. However, a recursive emulation leads to a cascade of traps, which is known for overhead that scales exponential with the nesting level [7]. We show in this paper how this overhead can be avoided.

Poon and Moon [11] have proposed a hardware extension that reduces the overhead of event delivery in recursive virtual machines from exponential to linear runtime. Our solution does not rely on any hardware extension for synchronous events but still requires only linear or even constant overhead. Furthermore, we will explain why their extension is insufficient for asynchronous events such as interrupts.

In the meantime nested virtualization has become an established feature in x86 virtualization environments such as KVM [18] or VMWare ESX [1].

From an operating system point of view, Fluke by Ford et al. [7], is most similar to our work. Fluke is a software-based virtualizable architecture that provides *recursive* OS containers or *virtual machines*. They aimed not at security but modularity and extensibility of the operating system. Within the VMs, Fluke provides a virtualized system-call interface instead of an existing hardware architecture. Interestingly, they excluded VMs and hypervisors based on hardware interfaces for two reasons. First, they cause exponential overhead. Second, they can not provide shortcuts in the chain of virtual machines. We will show in this paper how these two issues can be solved.

AMD SVM	Intel VT
`clgi`	`vmread(exit-reason)`
`vmload(child-state)`	`vmread(exit-qualification)`
`vmrun(child)`	`vmread(instruction-pointer)`
`vmsave(child-state)`	`vmread(instruction-len)`
`vmload(parent-state)`	`vmwrite(instruction-pointer)`
`stgi`	`vmresume(child)`

Fig. 2. Virtualization instructions usually executed by a hypervisor to handle a single trap on AMD and Intel CPUs. A nested hypervisor will need six virtualization instructions to emulate one virtualization instruction from its child.

III. Nested Virtualization

Nested virtualization is a feature of a hypervisor that allows its VMs to act as hypervisors and start virtual machines themselves. Even though a hypervisor might support only a single layer of nesting, recursive virtualization can be implemented with this technique, if all intermediate hypervisors support nested virtualization.

In previous sections we have claimed that nested virtualization should not be used as the building block for recursive virtualization due to its low performance in deeply nested scenarios. This mainly stems from the use of trap-and-emulate style of virtualization: The parent hypervisor multiplexes the CPU between its child VM and the nested grandchild VM, by trapping the virtualization instructions executed by the child and emulating their behavior.

In the following we will analyze the overheads of nested virtualization. This should give us a better understanding how the performance scales with the nesting depth and what problems our design should avoid.

Classifying the Overhead: We can distinguish three classes of overhead when using nested virtualization multiple times: constant, linear, and exponential.

Constant $O(1)$ overhead occurs when an event can be completely handled within a single layer (hardware or hypervisor) without causing further events. One example are untrapped instructions, if we do not consider cache and TLB effects. Another example are trapped instructions such as `cpuid` that can be completely handled by the parent hypervisor.

The overhead scales *linearly* with the nesting depth n, if a single event is propagated through the hierarchy: $O(n)$. Such an event can flow down the hierarchy. One example is the delivery of interrupts, where the parent hypervisors will propagate the IRQs to one of their children. The event can also flow up the hierarchy. If a child programs one of its virtual devices, for instance a timer, and the parent needs to program its timer as well, to emulate the side effects of the original operation.

Branching Factor	Interrupts per Second			
	1	10	100	1000
2	22	19	15	12
4	11	10	8	6
6	9	8	6	5
8	8	7	5	4
10	7	6	5	4

Fig. 3. Maximum number of nested VMs for different branching factors that can sustain a given interrupt frequency. Calculated for a 3 GHz machine where a single trap can be handled within 1000 cycles.

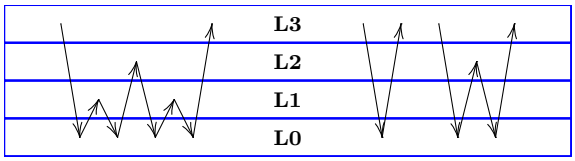

Fig. 4. Event-flow within three levels of VMs. With nested virtualization (left) an event causes an exponential number of traps to be forwarded through the hierarchy. In our design either the root hypervisor can handle the event itself (middle) or it forwards it directly to the upper layer (right).

The worst case are cascading effects which will lead to *exponential* overhead: $O(b^n)$ with a branching factor $b >= 2$. This can happen if a single event of a child causes the parent to generate multiple events of the very same type. One example are the emulation of virtualization instructions. Figure 2 shows that emulating a single virtualization instruction requires the parent to execute six virtualization instructions itself for the AMD and Intel versions of x86 CPU-virtualization extension. A grandparent that emulates these six instructions would already need 36 of them. Another example is the page-table lookup in multiple layers: a single memory operation can result in five memory accesses[1] with a four-level page-table and in 25 when two layers of page-tables are involved [3].

Nesting Limit of VMs: Based on these observations we can now estimate the maximum number of virtual machines that can be nested until the system will only handle interrupts and will not make other progress anymore. Since the exponential part has the most influence, we will ignore the constant and linear overhead in this estimation.

We assume a simplified scenario of nested virtual machines: At the beginning the deepest VM runs. When an external interrupt, such as a scheduling timer tick for the deepest VM, arrives at the root hypervisor all nested VMs are stopped. The nested hypervisors then recursively forward the timer interrupt and resume its virtual machines. This generates an exponential number of events depending the number of virtualization instructions used in a single layer to resume a child VM. This number is equivalent to the branching factor of the event tree.

We now calculate for this scenario the maximum number of virtual machines that can be nested and still sustain a given interrupt rate. We vary the IRQ frequency between 1 Hz and 1000 Hz and consider a branching factor between 2 and 10. We assume a 3 GHz machine that can handle a single trap of a virtualization instruction within 1000 cycles. This is slightly faster than what was previously reported [15]. Furthermore, we do not take into account that handling certain traps will be more costly than others.

Figure 3 shows the results of our estimation. If we are interrupting the virtual machine once every second and assume a branching factor of six, to emulate the six instructions from Figure 2 to resume a virtual machine, we are restricted to nine nested VMs. If we assume a more realistic frequency of 1000 Hz for a scheduling timer, we cannot nest more than five VMs!

[1]Excluding access- and dirty-bit updates.

IV. A Novel Design

The previous section has shown that nested virtualization cannot be used as building block for recursive virtualization, because exponential overheads limit it to a handful of nesting levels. Any design for recursive virtualization should therefore avoid any exponential overheads. In this section we present a novel design that achieves this.

The core idea of our design is the following: instead of repeating the support for nested virtual machines in every layer, we just implement recursive VMs in the root hypervisor. This approach has several advantages. Foremost, all virtualization instructions can be already emulated at the root and need not to be propagated trough the hierarchy. Other virtualization events can be directly forwarded to the right upper layer, since the root hypervisor knows all VMs in the system (See Figure 4). Furthermore, the root hypervisor can aggressively cache intermediate values to reduce the overhead. Finally, upper layers can be simpler as they do not need to be involved or in most cases even aware of recursive VMs. In the following we will describe how we implement this idea.

A. CPU Virtualization

If an upper layer parent wants to start a child VM, it uses an architecture specific instruction, such as `vmrun`. This instruction traps to the root hypervisor which can not directly execute it on behalf of the parent, since the child VM state is specified relative to the parent VM. For example, the memory for the new VM is a subset of the parent address space, but to execute a VM on a physical CPU, its memory has to be taken from the host memory.

Thus, the root hypervisor synthesizes for the child VM a corresponding *shadow VM* that can be directly executed on the hardware. This approach is similar to a shadow page-table algorithm, where the hardware is using shadow page-tables synthesized by the hypervisor, instead of guest specified page-tables.

Creating Shadow VMs: Shadow VMs are created by *merging* the parent and child VM state in the following way:

- All guest state is directly taken from the child. For example the CPU registers, the event injection, and the exit information are reused unmodified.
- The access rights to the platform resources are calculated by taking the intersection of both VMs' access rights. The two nested page-tables for example are folded and the two I/O permission bit maps are AND-ed together.

We thereby guarantee that the parent can isolate its child VMs and that a child has no more rights than its parents.
- The intercept bitmaps of instructions, exceptions and asynchronous events for the shadow VM are the union of what both VMs would use. This makes sure we do not lose any event the parent or the grandparents are interested in.

Please note that the hypervisors can be used unmodifed, as no support is needed from them for creating shadow VMs. Furthermore, merging is independent of the VM nesting depth, since only the parent and child state are involved. A hierarchy of virtual machines is flattened into a single layer by the root hypervisor by applying this process recursivly.

Lazy and Eager Merging: Merging can be an expensive operation, as it might touch several megabytes of memory. We therefore optimize this process through *lazy merging*. The nested page-tables are a good example for this. In the beginning the shadow VM starts with an empty nested page-table. Whenever the VM accesses memory that is not present, the root hypervisor will observe a nested page-fault and merges the corresponding page-table entries. In fact all resources can be lazily merged, if their absence generates a trap to the root hypervisor and if the operation that caused the trap can be resumed. This includes I/O permission and MSR bitmaps but excludes the I/O MMU page-tables, because DMA operations are usually not restartable.

Recursive virtualization with an Intel VT interface can also benefit from *eager merging*: Because the root hypervisor observes all updates (`vmwrites`) to the child VM state, it can directly merge the data into the shadow VM. When the child should run, the shadow VM can be activated without the need to merge anything anymore. This drastically reduces the runtime overhead since only the subset of the state that was modified has to be merged.

Forwarding Virtualization Events: The root hypervisor receives all virtualization events and has to forward them to the correct upper-layer hypervisor. If a virtualization event occurs, the root hypervisor inspects the original intercept bitmaps of all parents of the currently running virtual machine. Inspection is done in either top-down or bottom up direction, depending on the type of event. Asynchronous events such as interrupts, and machine check exceptions are delivered bottom-up and synchronous events, such as exceptions and instruction intercepts, are delivered in a top-down manner. The first intermediate hypervisor that has the corresponding bit in the intercept bitmap of its child set will receive the event. The overhead for event delivery scales at most linearly with the nesting depth, because only the state of the current VM and its parents has to be inspected. Please note that we do not require an interface extension for this, in contrast to previous solutions [11].

Event delivery can also be performed in constant time, because the destination for an event can be cached in a per shadow VM lookup table, if the root hypervisor can observe all state updates that invalidate this cache. This holds true for example with Intel VT based VMs, where only `vmwrite` instructions, which are emulated by the root hypervisor, can be used to modify VM state.

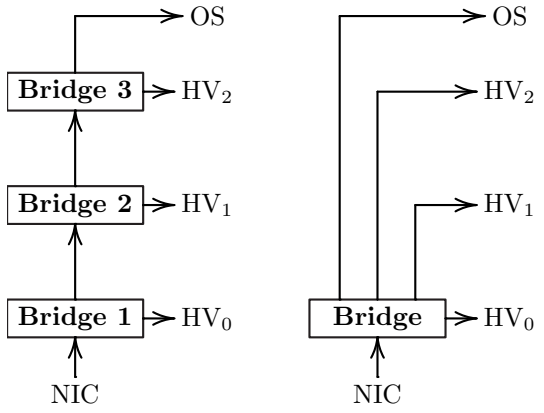

Fig. 5. Bridging cascade (left) vs. direct assignment of virtual NICs (right). Traditionally every virtualization layer implements a bridge to multiplex its network card with his child VMs. This forces packets through a deep bridging cascade. In our design the root layer can emulate multiple virtual NICs and recursively assign them to upper layer VMs. Thus only a single bridge is involved in network packet handling.

B. I/O Virtualization

The I/O path can have a huge influence on the overall performance of a virtualized system. [18] has shown that directly assigning a hardware device to a nested VM improves the performance, because hypervisors are no longer involved in most device operations. A register access for example will not cause a trap anymore, but will go directly to the hardware. Furthermore, DMA requests issued by a device are validated with an I/O MMU. This restricts DMA to the memory of the nested VM which drives the device. This allows to put drivers in untrusted VMs while still protecting the hypervisor from DMA attacks. Thus, we also use this technique and directly assign physical devices to recursive VMs.

However, not all VMs can benefit from direct assignment because there are usually more VMs than hardware devices in a system. This holds especially true where saving physical resources is one of the main incentives such as in server consolidation scenarios. Therefore, we also *assign virtual devices* through the VM hierarchy. The root HV, for example, can create multiple virtual NICs, which are recursively given to upper level VMs. In this way a received packet needs to be bridged only once, instead of going to a bridging cascade (See Figure 5). Since every bridging usually involves a copy of the packet, the performance gain of this approach will be significant.

Deep Intercepts: Direct assignment of virtual devices allows to implement shortcuts between different layers. Say for example a layer two (L2) hypervisor implements a virtual network card for a L3 VM, that is directly assigned by L3 to L4 and by L4 to L5. If L5 accesses a register on this virtual NIC, it will trap to the root hypervisor which forwards it to L2, thereby removing L3 and L4 from the chain.

This example leads us to the *deep intercept problem* of recursive VMs: In some point in time L2 needs to handle a trap from L5. However, L2 does not known L5 directly, but for handling the trap it needs to access the state of L5. One reason might be to figure out which of the memory it has given to L3 is available to L5. Therefore, L2 has to be able to recover the state of L5 by just looking at its direct child L3.

To handle deep intercepts a HV needs to know whether its child runs a nested VM and where the state of this VM can be accessed. Unfortunately, neither Intel's VT nor AMD's SVM currently provide these two pieces of information. Because we want to support deep intercepts for performance reasons, we extend the virtualization interface in the following way: We introduce a new *VM running* exit reason to decide whether the VM has exited or if it is still running and a nested VM has caused this exit. Furthermore, we publish the VMCS pointer of a VM in its shadow VMCS for Intel VT based VMs. This makes it accessible to its parents via `vmread`. On AMD SVM, we use the RAX register which already points to a child VMCB if the last operation of the VM was to run a child. These two simple extensions are sufficient for a hypervisor to search for the nested VM that caused the trap and handle it.

IRQ Forwarding: We rely on direct assignment and I/O MMUs to speed up the control and data path to a device. This leaves interrupts as the *the biggest cause for the remaining overhead* [18]. In fact, interrupt (IRQ) delivery is the only operation that still goes through the VM hierarchy. It is therefore desirable to minimize the number of IRQ forwarding steps. Another approach would be to reduce the interrupt frequency through IRQ coalescing. However, this causes a larger latency which might be undesirable for certain applications.

An interrupt can be directly delivered to the leaf VM, if the root HV would know to which child VM the intermediate hypervisors will forward the IRQ and how they will translate the IRQ vector. Vector translation is important with Message Signaled Interrupts, where a VM can freely choose its destination vector for a device interrupt. Unfortunately, neither AMD nor Intel currently allow to specify this information in their virtualization extensions. Related work [11] has proposed a simple redirection bitmap that decides whether an interrupt is forwarded or not. However, this is insufficient as it does not allow vector translation. Thus, to optimize the interrupt path for recursive VMs we developed our own extension.

We extend the VM state with a pointer to an IRQ redirection table. This table is indexed by the interrupt vector and reveals the destination vector and a pointer to a child VM. If the root hypervisor should deliver an IRQ, it traverses the hierarchy of redirection tables until a leaf is reached or a maximum number of forwarding steps is done. Limiting the number of lookups avoids a DoS attack against the root HV, where an untrusted VM creates a loop inside the table or a large number of VMs. At the destination VM the root HV sets the corresponding bit in the virtual APIC IRR, as it cannot directly inject the IRQ, because the VM might be in an IRQ shadow or even not running at all. If the virtualization interface does not provide a virtual IRR register, it can be shadowed by the root HV. This hardware extension can drastically reduce the number of events even in single HV scenarios. Please note that we do not optimize the handling of end-of-interrupts. In contrast to previous work [18], the overhead for them is independent of the nesting depth in our design, because we can handle them locally in the parent hypervisor.

C. Discussion

We solved the two issues mentioned by Ford et al. [7] that made recursive virtual machines based on a hardware interface infeasible. Our design avoids the exponential overhead caused by recursive trap-and-emulates, because the root hypervisor flattens the hierarchy and handles all virtualization instructions itself. Furthermore, shortcuts in a chain of virtual machines can be implemented by directly assigning virtual devices through the hierarchy.

Our design shows that a hypervisor does not need to provide the very same interface to its virtual machines which it gets from the underlying hardware. This is in-line with the observation that current hypervisors provide only a subset of the hardware functionality to their virtual machines. Especially seldom needed and optional features, for instance CPU power management, or FPU extensions such as 3DNow!, are usually ignored or only partially implemented. A virtualized OS will still work by detecting missing features and adopting accordingly.

However, we can go a step further and extend or partially replace the interface by the one that leads to the best performance for a given scenario. Announcing and supporting, for example, nested paging for child VMs is possible, even if the underlying hardware does not support it. We can also virtualize Intel VT extensions on an AMD CPU and vice versa. This has the nice consequence that all hypervisors running on top of the root hypervisor can be simpler than state-of-the-art implementations. If the root hypervisor provides a richer interface and already supports for example recursive virtualization, nested paging and direct execution in real-mode, there is no need to have support for nested virtualization, a virtual TLB, or a real-mode emulator in the higher layers anymore.

Finally, the root hypervisor can easily benefit from hardware acceleration. This includes our virtualization extensions, merging nested page-tables in hardware and providing a selective `vmwrite` intercept. Our design allows a clear upgrade path, were parts of the root HV functionality can move step-by-step from software into the CPU.

V. Advanced Security Mechanisms

By solving the performance problem of hypervisor-based recursive virtualization we opened the doors for the design of advanced security mechanisms. In the following we will shortly discuss the different ideas:

TCB reduction: Recursive virtualization allows the fine-grained decomposition of the virtualization environment into multiple hypervisors. This could even further reduce the TCB compared to [15].

Minimal Hypervisors: A minimal hypervisor has to reserve some memory for itself and can give all other resources to a child VM. It can periodically wakeup or trigger on a register access to perform its (security) task. Except for the initial configuration, it need not be involved in virtualizing the child VM. Furthermore, it does not need to virtualize devices, if it just assigns them to its child. This should make a single upper-level virtualization layer very small.

Thin Security Layers: Thin virtualization layers will improve the security of legacy operating systems, without degrading performance. These layers can be independently developed from different parties and later freely combined into a single system, because they implement the very same machine interface.

Defense in Depth: Multiple security layers below the OS can provide in-depth barriers of several kinds, such as firewall-like filters, wrappers, failure and intrusion detectors, etc.

Intrusion and Fault Tolerance: Sophisticated micro-middleware structures could achieve fault and intrusion tolerance of an OS and upper layer hypervisors. Diverse replicas can be used to transparently mask intrusions, and to provide fault-free support [17]. VMs may also be rejuvenated periodically for proactive recovery [14] to heal potential intrusions.

Cloud computing: Fast recursive virtualization is especially interesting for a cloud-computing environment as it provides security and flexibility with very little overhead. Supporting it in a cloud would allow a user to freely select its hypervisor. It would improve diversity as multiple management layers can coexist in a single system. Finally, it makes hypervisor updates easier, because it allows to move a hypervisor one layer up and start the new version in parallel. Virtual machines can then migrate locally from the old to the new version.

Further developing these ideas is part of ongoing work.

VI. Conclusions

In this paper, we showed that current approaches for recursive virtualization on x86 are plagued by exponential overheads, and we proposed a new design that can avoid these overheads, scale beyond a handful of layers and minimizes the burden for higher layer hypervisors.

Furthermore, we discussed that fast recursive virtual machines will open the field for new security-related mechanisms. We advanced several ideas that concurred to this objective. We are currently working to finish our implementation and evaluate the performance characteristics of recursive virtual machines on x86.

References

[1] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The Evolution of an x86 Virtual Machine Monitor. *SIGOPS Oper. Syst. Rev.*, 44:3–18, December 2010.

[2] G. Belpaire and N.-T. Hsu. Hardware Architecture for Recursive Virtual Machines. In *Proceedings of the 1975 annual conference*, ACM '75, pages 14–18, New York, NY, USA, 1975. ACM.

[3] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two–Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–35. ACM, 2008.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[5] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, March 2008.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.

[7] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernel meet recursive virtual machines. In *USENIX 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[8] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[9] A. Graf and J. Roedel. Nesting the Virtualized World. Linux Plumbers Conference, September 2009.

[10] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland 2010)*, May 2010.

[11] W.-C. Poon and A. K. Moon. Bounding the Running Time of Interrupt and Exception Forwarding in Recursive Virtualization for the x86 Architecture. Technical Report TR-2010-003, Vmware Inc., Palo Alto, CA, USA, October 2010.

[12] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17:412–421, July 1974.

[13] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *SIGOPS Oper. Syst. Rev.*, 41:335–350, October 2007.

[14] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[15] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2010.

[16] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[17] P. Verissimo, M. Correia, N. F. Neves, and P. Sousa. *Intrusion-Resilient Middleware Design and Validation*, pages 615–678. Emerald, 2009.

[18] M. B. Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.