Electronic Communications of the EASST
Volume 17 (2009)

Workshops der Wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2009 (WowKiVS 2009)

ProMoX: A protocol stack monitoring framework

Elias Weingärtner, Christoph Terwelp and Klaus Wehrle

10 pages

Guest Editors: M. Wagner, D. Hogrefe, K. Geihs, K. David
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

ProMoX: A protocol stack monitoring framework

Elias Weingärtner, Christoph Terwelp and Klaus Wehrle

Distributed Systems Group
RWTH Aachen University
Aachen, Germany
Email: {weingaertner,terwelp,wehrle}@cs.rwth-aachen.de

Abstract: In this paper, we present a preliminary glance on our framework for protocol stack monitoring using Xen (ProMoX). ProMoX uses the Xen hypervisor to virtualize entire instances of operating systems which may execute any arbitrary protocol implementation. By utilizing system virtualization for external monitoring, ProMoX can transparently inspect any protocol state and performance metrics of protocol implementations carried by a guest operating system. This way, ProMoX supports both the indentification of faults within early prototypes as well as the evaluation of new protocol designs.

Keywords: system virtualization, protocol stack monitoring, development support

1 Introduction

Over the last decade, overlay networks and peer-to-peer technologies have drawn much attention in the network research community. Moreover, file sharing services like BitTorrent [4] as well as overlay-based VoIP services such as Skype [3] have recently proved the potential of this paradigm and are nowadays used by millions of users. Due to the popularity of Skype and BitTorrent and standardization efforts like P2PSIP [1], it can be expected that more overlay-based services will be commonly used in a little while and hence will diversify the Internet landscape.

Directly linked to the proliferation of overlay services is the development and the implementation of respective network protocols. Network protocol stacks are tightly coupled to the underlying operating systems, and core services are often implemented in the system kernel. Adequate tools are required for debugging and monitoring the implementations' execution, as well as for evaluation purposes. In this regard, debugging tools such as GDB [13] and KGDB [8] as well as full-system simulators like Simics [10] are commonly used to study the behavior of a protocol implementation.

In the following, we focus on a different approach for the analysis of protocol implementations. We utilize system virtualization to facilitate transparent monitoring of a protocol implementation at run-time. A guest operating system which carries a network protocol implementation of choice is virtualized using a virtual machine monitor (VMM). The VMM is controlled from a privileged control context. The privileged control context is able to inspect any resource used by the virtualized guests, in particular network and memory usage. This way, the virtualization environment can be used to explore the run-time behavior and the performance of a protocol implementation. Moreover, the exploration of the implementation can be entirely carried out in

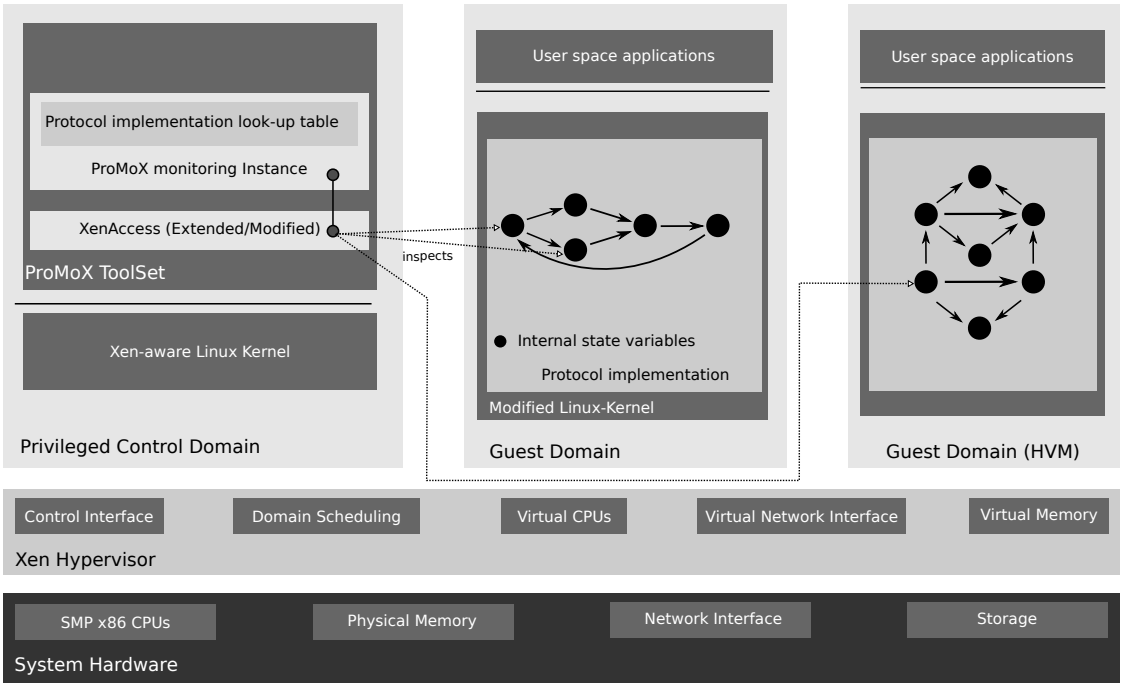

Figure 1: ProMoX architecture

the control context, and hence, it is completely transparent for the system under investigation. From this point, we will refer to this virtualization-assisted external monitoring as *introspection*.

The idea of applying introspection to the analysis of protocol implementations is the driving motivation behind our framework for Protocol stack Monitoring using the Xen hypervisor (ProMoX). ProMoX allows one to transparently monitor any protocol state using a privileged control context. A detailed description of the ProMoX introspection architecture is given in Section 2, before we discuss first insights about the system performance in Section 3. In fact, this is not the first time that system virtualization has been used to ease such investigations of protocol stacks. In Section 4, we compare ProMoX to related approaches. As ProMoX is in early stages, there is a lot of room for improvement and enhancements. We outline this future work in Section 5, before we conclude this paper in Section 6 with the essential findings of this paper.

2 ProMoX Architecture

In this section, we describe the architecture of ProMoX as depicted in Figure 1. ProMoX is based on the Xen hypervisor that virtualizes instances of multiple operating systems. After a brief description of Xen and its virtualization approach, we describe how the ProMoX framework can be applied to monitor the internals of a protocol implementation using our modified version of the XenAccess [11] introspection library.

2.1 Xen-based virtualization

Since its introduction in 2003, Xen [2] has become one of the most used virtualization platforms for x86 systems today. Xen is a virtual machine monitor that allows to execute multiple operating systems concurrently on one physical machine. Such an operating system instance is called a *domain* in the context of Xen. The Xen virtual machine monitor runs directly on top of the hardware and coordinates the domains' concurrent accesses to the shared resources of the host system. Xen is based on the concept of para-virtualization, where the virtualized operating system is aware of the virtualization environment and cooperates with the hypervisor.

In Xen one mandatory privileged control domain, the so-called *domain 0*, is in charge of controlling the execution of host systems on the hypervisor using Xen's control interface. For example, the privileged control domain instantiates new domains and is capable of suspending or terminating their execution at any time. In addition, it is noteworthy that all hosted operating systems make use of the device drivers of domain 0 in order to access the system hardware, for example a network interface.

In the x86 architecture, four different protection rings exist: ring 0, 1, 2 and ring 3. While common operating systems only use ring 0 for the execution of the kernel and ring 3 for applications, this is different for a system where a Xen hypervisor is in place. In Xen's implementation of para-virtualization, the domains are executed in ring 1 and cooperate with the hypervisor which resides in ring 0. In order to execute an operating system in ring 1, its kernel needs to be modified. Operating system commands, which need to access system resources that are managed by Xen are replaced with so-called *hypercalls*. These hypercalls are handled by the Xen hypervisor. Data between Xen and the hosted domains is exchanged using shared memory pages. The major advantage of para-virtualization is the increased virtualization performance, as no system calls have to be trapped, and no hardware has to be emulated in order to virtualize an operating system.

A major drawback of earlier Xen versions was that they only supported para-virtualized guest domains. The major disadvantage with para-virtualized guest systems is that they require modified operating systems kernels. Modified kernels are available for a couple of operating systems, most notably Linux and OpenSolaris. However, if one wants to enable the use of para-virtualization for a particular operating system, he needs access to the kernel source code, which essentially constrains the choice to open-source operating systems. To overcome this problem, Xen in version 3.0 introduced hardware assisted virtualization (HVM) as an alternative to para-virtualization. Intel and AMD have equipped their processors with architectural extensions (AMD-V, Intel VT) to support virtualization on CPU level. Starting with Xen 3.0, Xen uses these extensions as a basis to allow the execution of arbitrary x86 operating systems. This is especially important, as closed source systems like Windows can now be used on top of Xen. Our ProMoX implementation operates with both types of domains and hence, conceptually facilitates external monitoring of any protocol implementation based on a x86 operating system.

Of direct interest for the later discussed introspection is the memory management of Xen based systems, where three different memory address types can be distinguished:

- *Machine memory addresses* are physical addresses which directly address the hardware memory. These addresses are only directly used by the Xen hypervisor and XenAccess.

- Xen allocates a fraction of the physical memory to every domain, which is seen by the domain as its physical memory. The domains physical addresses are Xen's *pseudo-physical memory addresses* which are mapped to a *machine memory addresses*.
- Almost all guest operating systems rely on *virtual memory* and hence nearly all memory addresses used in this context are in fact *virtual memory addresses*. As described in the standard literature [12], this core concept is required for a couple of reasons, for example paging.

Most remarkably, Xen provides every domain with an own pseudo-physical memory range. Every domain operates only within this pseudo-physical address range, which yields to a strong isolation of the domains against each other. Solely the privileged domain is able to access the memory resources of other domains via Xen using specific hypercalls. Therefore, the ProMoX monitoring instance as described in the following always needs to be placed in domain 0.

2.2 ProMoX Introspection

After the introduction to the Xen virtual machine monitor, we now describe how its virtualization approach is used in ProMoX to facilitate the monitoring of arbitrary protocol stack implementations on x86 systems. ProMoX is based on two distinct building blocks, the ProMoX monitoring instance and the XenAccess introspection library [11].

ProMoX monitoring instance

The central element of our monitoring framework is the ProMoX monitoring instance, which is a user-level application being executed within the privileged domain. The monitoring instance maintains a look-up table which contains virtual memory addresses of variables that contain the system states one is interested to observe. These virtual addresses are obtained by using virtual addresses from the system map as starting points and then traversing the data-structures of the kernel. All accesses to the kernel memory of the guest system are established by using the XenAccess library, which will be discussed in more detail later.

For example, if one is interested in exploring the connections of a TCP/IP stack, one required entry in the look-up table would be the address of the struct which contains the information about open connections, their port numbers and their destination addresses. However, the XenAccess library only provides bare access to logical memory addresses and is not able to interpret the raw memory stored at these addresses. Hence, a core functionality of the ProMoX monitoring instance is to provide a view which abstracts from raw memory content and instead uses more meaningful state descriptors. As a first step towards a comprehensive protocol-stack inspection framework, we have implemented a set of methods to inspect a range of properties of the Linux 2.6 TCP/IP stack for guests which operate this kernel. So far, the ProMoX implementation allows the inspection of open TCP or UDP connections and facilitates the access to important protocol information, such as the current TCP window size. Moreover, it is noteworthy that supplying an abstract view on a protocol state requires transparent memory lookups with a partial re-implementation of the protocol logic inside ProMoX. For example, in order to output a list of all open TCP connections of a hosted Linux system, ProMoX first obtains the global protocol

state and iterates over all allocated socket buffers, for which each an individual XenAccess look-up is required. As such introspections may take a significant amount of time, it is beneficial that our ProMoX implementation and our modified XenAccess implementation in fact also facilitates the analysis of suspended domains. Moreover, Xen allows to take snapshots of a running domain at any point in time which ProMoX is capable to inspect as well.

The maximum achievable monitoring rate of ProMoX is directly dependent on the domain scheduling provided by the Xen hypervisor. Xen schedules the execution of domains in a similar fashion like a OS scheduler coordinates execution of concurrent processes. Hence, the achievable monitoring rate is directly dependent on the used scheduling scheme. The Xen distribution currently provides two schedulers in this regard, a credit-based scheduler and a scheduler that implements a EDF scheme. For a different purpose, we have extensively modified the EDF scheduler of Xen to allow domains to be executed for an exact amount of time [14]. By utilizing this modified scheduler, it is possible to achieve a monitoring rate of up to 10μs.

XenAccess Introspection Library

The core task of the XenAccess library is to enable the convenient access to virtual addresses within a executed guest domain, as this is required to access a protocol stack's internals from the control context. The actual look-up process of a protocol state descriptor involves several steps:

1. The ProMoX monitoring instance determines the virtual address of the state descriptor variable using the look-up table.

2. The XenAccess library identifies the address of the page table within the corresponding domain.

3. XenAccess traverses the guest domain's page table in order to determine the machine address (in case of HVM the pseudo-physical address) of the protocol state descriptor.

4. Once the corresponding machine address (in case of HVM the pseudo-physical address is automatically translated to the corresponding machine address) has been identified, the corresponding memory region is mapped into the address space of the privileged domain, thus enabling the introspection using a set of shared memory pages.

As a part of our work on ProMoX, we have extensively modified and extended the original implementation of XenAccess, first by completing its implementation for the inspection of HVM domains. Moreover, we added 64bit support to the introspection library. XenAccess now executes on any 64bit system and is even capable to inspect 32bit guest domains if the hypervisor and domain 0 operate in 64bit mode. Another issue is given by the size of state descriptors: The original XenAccess only allowed to inspect symbols which are smaller than one memory page (4kb). However, if one is interested in the introspection of more complicated data structures, a problem may arise if such a data structure spans across multiple memory pages. In order to handle this problem, we extended XenAccess in a way that it supports data structures which exceed the page size limit. In addition, we rewrote the entire XenAccess page table look-up procedure to support large page tables and Xen versions greater than 3.0.

Domain type	Caching	No Caching
Para-virtualized	13μs	59μs
HVM	13μs	26μs

Table 1: ProMoX introspection times

3 ProMoX Performance

As ProMoX is slated to monitor a protocol stack in a concurrent fashion, it is important that protocol state descriptors can be inspected efficiently. In order to keep track of sudden state changes, it is usually required to sample a protocol state descriptor at a high rate. Thus, it is important to carry out the look-ups efficiently, with the goal of keeping the impact of the overall system performance in reasonable bounds. In the following, we provide first results regarding the monitoring performance of ProMoX and a discussion of the achievable monitoring resolution. All performance measurements were conducted on an AMD Athlon 64 3800+ workstation with 2GB of RAM, running ProMoX on top of Xen 3.2. For both the control domain and the inspected guest domain, we used Linux with the kernel version 2.6.18.8-xen.

3.1 Introspection performance

In order to investigate the look-up performance, we measured the duration of one introspection of a kernel-level address inside the Linux guest domain. The results were obtained by averaging over 5000 measurements. Table 1 shows these introspection times for both domain types. As a matter of fact, XenAccess internally maintains a cache for the mapping of virtual addresses to pseudo-physical addresses. In order to measure the influence of this cache, we repeated the experiment, performing the introspection on 5000 different virtual addresses instead, which leads to a cache miss for every introspection. As our results show, caching the pseudo-physical addresses improves the performance significantly. There is a remarkable performance difference between a non-cached introspection in a HVM domain and the para-virtualized case. We explain this result with the hardware support of the used processor. The evaluation machine was a AMD Athlon 64 processor with AMD-V extensions. AMD-V supports the virtualization of page tables in hardware. XenAccess is able to directly access the guest domain's page table in the HVM case. In contrast to this, XenAccess first needs to obtain a guest domain's page table by mapping it as a shared memory page if para-virtualization is used.

3.2 Monitoring resolution

Of direct interest for monitoring protocol stack implementations is the ability to observe protocol state changes accurately. As our approach of monitoring a protocol stack is entirely passive, the state descriptors must be inspected regularly at an adequate rate in order to detect such state changes. While the required monitoring rate is highly dependent on the respective protocol state, the achievable monitoring rate and hence the maximum monitoring resolution is limited by the virtualization environment, as the scheduler of the virtualization environment essentially assigns discrete run-time slices to the domains.

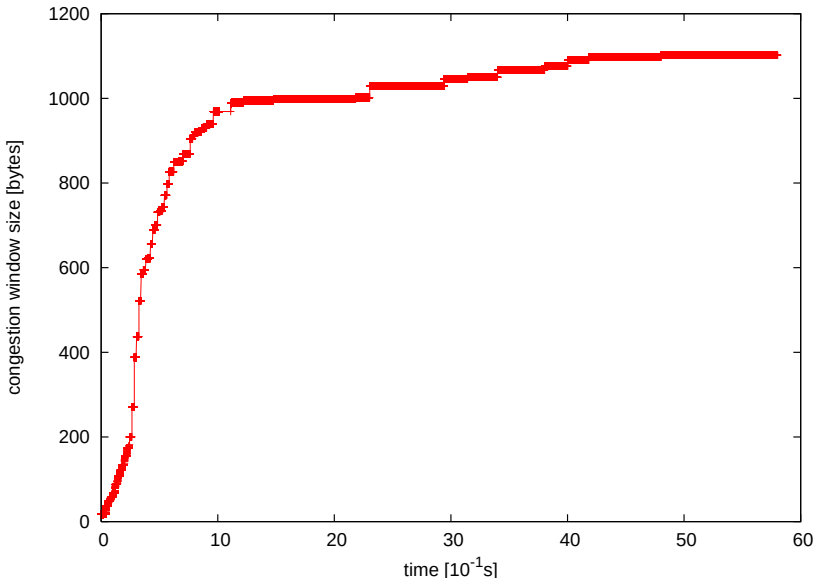

Figure 2: Partial trace of TCP Congestion window size (obtained using ProMoX prototype)

In order to determine the performance of ProMoX in this regard, we traced the congestion window size of the Linux TCP/IP stack for one connection from the guest domain to a different host. A partial trace of the congestion window size is depicted in Figure 2. The results obtained with ProMoX reflect the well-known slow start behavior of TCP. This preliminary result shows that ProMoX is able to capture the behavior of rapidly changing protocol state descriptors, given the fact that the congestion window size changes with every received acknowledgement packet.

4 Related Work

The investigation of network protocol stacks and related debugging tasks are often carried out using kernel-level debuggers or full-system simulators. Kernel-level debuggers such as kgdb [8] or the Rasta Ring 0 Debugger [5] allow the inspection of kernel-level implementations. In comparison to a virtualization-based approach for the investigation of network protocol implementations, debugging tools debuggers are executed in the same context as the investigated subject. While this eases the access to data structures and protocol state descriptors, the monitoring process is less isolated from the studied protocol implementation.

Full system simulators such as Simics [10] model the entire machine hardware and its peripherals in software, and unmodified operating systems can be executed on the modeled system. The key strength of full system simulators is their flexibility, as any property of the modeled machine can be changed in an arbitrary fashion. Many full system simulators provide a plug-in interface to the system simulation, which one could use for protocol monitoring. However, the simulation of the entire machine hardware naturally bears a lot of overhead, and in spite of many optimizations, full system simulations execute at least 5-10 times slower than a real system. For

this reason, it is hard to employ full system simulators for protocol monitoring, as one is typically interested in observing real-time interactions with multiple hosts.

A popular application area for virtual machine introspection is intrusion detection as first proposed in [6]. In their paper, Garfinkel and Rosenblum use a modified version of VMWare Workstation for Linux as virtual machine monitor to provide a sandbox for an operating system, which can be introspected from a control context. The actual intrusion detection is then carried out by analyzing the information retrieved from the introspection, for example by hashing memory regions of the running guest operating system. In this case, the hash values are used to detect a possible tampering of the running operating system kernel. While the architecture and the notion of introspection is very similar to our work, the authors are merely interested in identifying persistent changes in a virtual machine, for example the installation of a root-kit. However, if introspection is applied to the monitoring of protocol stack implementations, one is typically interested in observing short-term state changes.

In general, the need for development tools of massively distributed systems has been addressed lately by others as well. A recent and very exciting example is D3S [9], which allows a distributed system to be checked against predicates while it is executed. In their paper, the authors demonstrate its capabilities by applying it to a Chord and a BitTorrent implementation. D3S uses binary code instrumentation in order to export internal system states to the observing instances. While binary code instrumentation works well for application-layer protocol implementations, it is difficult to employ for protocol stacks that are tightly integrated in the operating systems kernel.

5 Future Work

As mentioned earlier, ProMoX is in very early stages, and hence our future work mainly focuses on the extension of the framework. A major challenge of ProMoX is given by the fact that it currently requires a lot of knowledge about the implementation, for example, the exact structure of a protocol state descriptor. Currently, we are working on a more convenient solution which automatically provides ProMoX with this required information, for example by marking the protocol state descriptor in the implementation's source code.

Up to now, ProMoX merely samples a protocol state descriptor in a periodic manner. However, very high sampling rates are required to observe communication links of high bandwidth. Therefore we currently investigate an alternative event-based scheme. More specifically, we explore the possibility of monitoring stack internals at the occurrence of particular system events. We are currently working on the invocation of ProMoX at the time network packets are exchanged in this regard, by using a modified driver back-end in the privileged control domain. This way, we hope to facilitate the intercorrelation of network traffic with internal state changes of a protocol implementation.

Another challenge to be addressed by future work is the application of ProMoX to closed-source operating systems. So far, ProMoX is able inspect the TCP/IP stack of Linux 2.6, besides custom protocol implementations. In the case of closed-source operating systems, the look-up process has to be modified in order to match the guest operating system virtual memory management.

6 Conclusion

In this paper, we provided a first outlook on our ProMoX architecture for the virtualization-based external monitoring of protocol stack implementations. ProMoX is based on the Xen hypervisor and utilizes the XenAccess introspection library for its monitoring purposes. ProMoX is able to access the protocol stack internals of both running and suspended virtualized systems. Our preliminary evaluation results have shown that ProMoX is able to monitor a fast-changing protocol state variable accordingly.

In comparison to established monitoring approaches such as kernel level debuggers, frameworks such as ProMoX that are based on system virtualization deliver a higher degree of transparency. The monitoring process is executed in a privileged context and not within the context of the system under investigation. However, this comes at increased costs in terms of implementation complexity and run-time efforts, due to the need of externally traversing through the system's memory management and due to the necessity of parsing internal data structures.

All in all, we regard our work as a proof of concept for virtualization-based external monitoring of protocol implementations. We anticipate systems such as ProMoX not only for research and development purposes. It is imaginable that future systems will always be executed within a virtualization environment. In this case, monitoring frameworks like ProMoX could be used to monitor a protocol stack continuously, both for the detection of faults and for general run-time analyses of the deployed system.

Bibliography

[1] P2PSIP working group website.
http://tools.ietf.org/wg/p2psip/.

[2] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, Oct. 2003. ACM.

[3] S. A. Baset and H. G. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, pages 1–11, April 2006.

[4] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[5] Droids Corporation. Rasta ring 0 debugger (RR0D).
http://rr0d.droids-corp.org/ (accessed 10/2008).

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2003)*, San Diego, CA, Feb. 2003. Internet Society.

[7] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked*

System Design and Implementation (NSDI'08), San Francisco, CA, USA, 2008. USENIX Association.

[8] J. Lamphere. Remote debugging of loadable kernel modules with kgdb: a knowledge-based article for getting started. *Embedded Linux Journal*, 2:49–54, Mar./Apr. 2001.

[9] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: debugging deployed distributed systems. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[11] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. *Proceedings of the 32rd annual Computer Security Applications Conference (ACSAC)*, pages 385–397, Dec. 2007.

[12] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[13] R. M. Stallman, R. Pesch, S. Shebs, et al. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, pub-GNU-PRESS:adr, 2002.

[14] E. Weingärtner, F. Schmidt, T. Heer, and K. Wehrle. Synchronized network emulation: Matching prototypes with complex simulations. In *Proceedings of the First Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics'08)*, Annapolis, MD, 2008. ACM, SIGMETRICS PER.