External Monitoring of Protocol Stacks

Rheinisch-Westfälische Technische Hochschule Aachen
LuFG Informatik 4 Verteilte Systeme

Diploma Thesis
Marko Ritter

Advisors:
Dipl. Inf. Elias Weingärtner
Prof. Dr.-Ing. Klaus Wehrle

Registration date: 16. February 2009
Submission Date: 14. September 2009
I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 09. September 2009
Abstract

This work targets distributed external monitoring with the help of virtual machines. Monitoring and debugging applications is a troublesome task, especially in the case of distributed protocol implementations. To generate a consistent global view of an erroneous situation a single debugger attached to a local process is not sufficient. This gets even more complicated when the protocol is implemented inside an operating system kernel. Kernel debugging itself is elaborate as it mostly requires an external debugging access from a second machine.

With virtualization, the privileged position of the hypervisor enables external access to the virtualized hardware of the virtual machine. Our work utilizes this access to monitor and debug protocols in the virtual machine. A consistent global state of the participating systems is achieved by synchronizing wallclock- and runtime of the virtual machines.
Acknowledgments

I would like to thank all people who made it possible to write this thesis:

My advisor, Elias Weingärtner for always giving me helpful suggestions and advice.

Prof. Dr. Wehrle, for giving me the opportunity to write this thesis at the distributed systems group, and for being my supervisor.

Prof. Dr. Rumpe, for taking his time to be my second supervisor for this thesis.

Florian Schmidt and Martin Lindner for implementing and revising the time synchronization solution used in the evaluation.

All other members of the distributed systems group, for contributing ideas, helping me solve prickly problems, and for contributing to the excellent work atmosphere.

Stefan Hahn, Stephan Peltzer, Axel Langhoff and Maik Glatki who helped me to improve the quality of my written english.

My girlfriend Fatima Godusevic for giving me the endurance and motivation to complete this work.

(My headache tablets for helping me keep a clear mind even in hard phases.)

Finally, my parents, for introducing me to computers when I was six years old and for making my studies possible.
Contents

1 Introduction 1

2 Background 5

2.1 Memory Handling on i386 5
2.2 Virtual Machines 9
2.3 Xen Hypervisor 14
2.4 Kernel Programming and Debugging 21

3 System Design 27

3.1 Why to use virtual machines 28
3.2 Usability vs. Versatility 28
3.3 Architecture Concept 30
3.4 Distributed Setting with Time Synchronization 36

4 Implementation 39

4.1 Implementation of Front- and Backend 39
4.2 GDBserver-Xen 42
4.3 LibMiGDB - GDB/Machine Interface Library 44
4.4 Apache Thrift 44
4.5 Python Monitoring Script-Interface 45

5 Evaluation 47

5.1 Goal and Test-Setup 47
5.2 Basic Tests and Microbenchmarks 48
5.3 Evaluating Protocol-Monitoring 54
5.4 Distributed Protocol-Monitoring and Virtual Time 56
Introduction

Protocols are conventions that enable the connection, communication and data transfer between multiple endpoints. They are implemented in user space applications and system kernels. Utilizations of network protocols range from file sharing, P2P based telephony to multiplayer online games. Over the last decade, the number of distributed applications has grown very fast and so has the relevance to debug them.

For a protocol implementation to work correctly, itself and the design of the protocol must be flawless. The reason why it is particularly difficult to find all bugs, is that the global state of the test-environment is distributed, too. Each participating node is part of the global state. Furthermore, the bug could exhibit itself only after a long sequence of distributed events, involving special conditions of the network or remote machines.

A cyclic debugging approach, using diagnostic statements (e.g. `printf`) at particular locations in the code and interpreting them during test runs, is not working with non-deterministic bugs. The non-determinism is a consequence of the variance in network delays and is already known from race conditions due to the scheduling of multiple threads.

A common approach to cope with this problem is to collect timestamped debugging information from all nodes in a central location. There they are ordered by their timestamp and analyzed by the developer. This approach is very laborious and error-prone, as the amount of collected information grows very fast.

There also exist two advanced debugging approaches for non-deterministic bugs. The first, “online monitoring”, allows the developer to write distributed predicates, which usually define distributed invariants that are continuously checked during the execution. Violations are announced to developer, along with the sequence of events that lead to this violation.

The second is “deterministic execution replay”. It is a two phase approach, in which the information on the program execution and all non-deterministic events are gath-
ered during the record phase. Using these information the developer can debug the application by replaying its execution during the replay phase.

With virtual hardware, we can simulate such a distributed test-environment on one physical machine. Furthermore, the hypervisor has direct control on all running virtual machines. It can decide when to run or when to pause them and is able to change their virtual clock. Thereby a synchronization of the virtual machines can be achieved.

The concept of virtualization can also be used to analyze kernel applications. The challenge with kernel debugging is that a normal debugger on the test-system cannot attach to the system-kernel as itself depends on it. The kernel has to be accessed from the outside. The common approach for this is to use a serial console to connect to a debugging stub build in the kernel. This requires a second development machine and preparations of the kernel itself.

Using virtualization it is possible to externally monitor the kernel or user space application of a virtual machine (VM). The hypervisor has the privileges to access all virtual hardware of all virtual machines. It can thereby access the complete memory structure of the VM and read or manipulate the desired values. Event capturing is enabled by installing hooks in form of breakpoints inside the VM kernel.

One main challenge with this is to bridge the semantic gap from the view of the hypervisor, seeing a flat memory block without structure, to the real structured contents of kernel and user space applications. The operating system (OS) of the virtual machine is in charge to decide what content goes in which place of the memory block. The static kernel memory is mostly a zoned part of this block. Information on the memory handling procedures of the used OS are required to find and interpret a target variable or function in the memory of the VM.

The second challenge is that the physical memory is handled by the memory management unit of the processor together with virtualization algorithms of the hypervisor. This represents another barrier which has to be bridged.

For a couple of years, processors have been steadily improved to provide special virtualization abilities in hardware. Hardware assisted virtualization is available on many current computer systems and makes it possible to virtualize unmodified operating systems like Windows without unbearable performance losses.

The goal of this work is to provide a scriptable toolset to debug distributed protocols using external monitoring of virtual machines and time synchronization. Our proof of concept implementation uses tools like GDB and GDBserver-Xen to access the VM. The required kernel information are taken from the debugging symbols provided with the kernel. To avoid protocol timeouts a time synchronization server is used to keep all VMs in sync.

We chose a client-server model for the tool’s back- and frontend, with a well defined interface. That makes it possible to use different kinds of those together. The implemented toolset has a scriptable Python frontend which makes it possible to automate the complete monitoring session and provides the whole power of external virtual machine monitoring at the fingertips of the developer.
To our knowledge there is no other tool available that uses external virtual machine monitoring as an approach for distributed debugging, with a flexible scriptable frontend and the ability to use multiple physical machines.

Our proof of concept implementation can be used to monitor any given operating system virtualizable with Xen that provides the debugging symbols in a GDB compatible format.
2

Background

Before delving into the depths of monitoring virtual machines, one needs to understand several procedures inside current computers: The memory handling on physical and virtual i386-based computers, how virtual machines are realized and what benefits come from them, the Xen Hypervisor[4] implementation specialties and some basics of kernel programming and debugging.

2.1 Memory Handling on i386

Most computers run a modern operating system which provides multi-tasking. It is the process of running multiple programs on a computer at the same time [19]. The programs or tasks have to share and compete for the computers resources, which they need to run.

Scheduling and managing the resources and processes is the job of the operating system kernel.

Two main resources - which programs have to compete for - are CPU runtime and memory. CPU runtime can be divided in time slices and scheduled to all the running processes more or less fairly, which makes it a rather endless resource. System memory on the other hand is always limited. This makes it often difficult to load and keep all running programs in the memory at the same time. There exist several techniques to manage and allocate the system memory to the running processes. Frequently used - and hence very important - are Virtual Memory, Paging and Segmentation.

In all Unix operating systems the system memory or random access memory (RAM) is split into two portions. Some few megabytes are statically used by the kernel, storing its image and data structures. The remaining RAM is handled by the memory management system of the kernel. It is used for dynamic kernel structures, process memory and caches.
From the view of a User Mode process, each has its own private address space containing private stack, data and code areas [18]. Parts of the process address space could be shared among several processes. Mostly because it is static and used by more than one process (libraries etc.). Sometimes the shared memory is used for inter-process communication (IPC) purposes as propagated by System V [51].

Virtual memory is the abstraction that gives the process the impression that its address space is one linear continuous area of system memory, isolated from all other processes, which may even exceed the maximum size of the real memory. This has several advantages as it allows[18]:

- to run processes concurrently
- processes to use more memory than physically available
- for partial loading of programs
- static portions of memory to shared (e.g. system libraries)
- machine independence for the programs (in respect of memory organization)
- relocatability of processes

A 32-bit system is able to address 4 gigabyte (GB) of memory using a single 32-bit unsigned integer. We have to distinguish three kinds of addresses as seen in Figure 2.1 [18]:

Logical addresses are used in the machine language to address an operand or instruction. They consist of two parts: The segment identifier and an offset which is the distance from the start of the segment to the actual address. The CPU transforms a logical address into a linear address with the segmentation unit.

Linear addresses are 32-bit unsigned integers, mostly represented in a hexadecimal notation ranging from 0x0000000 to 0xffffffff. A second hardware circuit, called the paging unit, is used to transform a linear address into a physical address.

Physical addresses point to the real memory cells and correspond to the electrical signals send over the memory bus. They are represented by an unsigned 32-bit integer, too.
2.1. Memory Handling on i386

2.1.1 Segmentation

There exist two address translation modes: real and protected mode. The older one is called real mode. The linear addresses in real mode are 20 bits long and allow only one megabyte (MB) to be used directly. It still exists for compatibility reasons and operating systems have to bootstrap in real mode and can then switch to protected mode. The protected mode was introduced by Intel with the 80386 model.

In protected mode, the segment identifier or segment selector - as part of the logical address - is a 16-bit field, while the offset is a 32-bit long integer. The processor provides special segmentation registers to hold these segment selectors. There exist descriptor tables called Global Descriptor Table (GDT) and Local Descriptor Table (LDT) in the main memory that contain information on type and content of the segments. Whenever a segment selector changes in a segmentation register, the processor loads the corresponding segment descriptor in a non-programmable register for easy and fast access.

Segmentation is used as a virtual memory mechanism under MS DOS and MS Windows. However, since one design goal of Linux is portability and segmentation is supported by several RISC processors only in a very limited way, Linux prefers paging as virtual memory mechanism.

As a short comparison one can say, “segmentation can assign a different linear address to each process, while paging can map the same linear address space into different physical address spaces” [18]. When segmentation should be used, the program has to be split into logical parts during development. Paging on the other hand is used completely transparent to the developer.

2.1.2 Paging

As already said, the hardware paging unit translates linear addresses into physical ones. Basic access right validation is also done. The paging unit raises a page fault exception if the memory access is invalid, for example when a page is not present in memory.

Starting with the i80386, the available memory is partitioned into fixed-length page frames in size of 4 kilobyte (KB) or 8 KB. Since the Pentium i80x86, even 4 megabyte (MB) page frames were possible in extended paging mode [18]. Pages of data have the same length as a page frame and may be stored in any page frame of the memory. As not all pages of data need to be present in memory at all times, they may be swapped out of memory onto a disk.

The paging unit uses page tables, located in the main memory, to map the linear to physical addresses as seen in Figure 2.2. The tables are managed by the system kernel. The CPU register cr3 contains the base address of the global page directory.

32-bit Intel processors implement a so called “regular two level paging” with pages of 4KB size. The linear address consists of three parts (Figure 2.2) [18]: Directory entry address, Table entry address and Page offset.

Each of the two translation steps is based on a translation table, to map the address. In the first step, the page directory is used to look up the address of the page table.
Figure 2.2 Paging by Intel 80x86 processors

for step two. Consequently, the page table is used in the second step to look up the page address. At last, the offset has to be added to the page frame base address. The resulting memory address points to the physical position of the paged linear address.

Besides the addresses, these translation tables contain flags describing the status of the page frame and access rights. The flags indicate whether the page is present in current system memory, if it was accessed, if it was written and now is so called dirty, the page size (4 KB or 4 MB), the access rights (read/write or read and user or supervisor) and some other information.

When the page size in a page directory indicates a 4 MB page, the extended paging mechanism is used as second step. In extended paging, the whole remaining 22 bits (32 bit minus the 10-bit directory address) of the linear address are used to represent the offset inside the 4 MB page.

As in recent years the new microprocessors adopted a 64-bit architecture and so allowed more than 4 GB system memory, the need for a new memory addressing scheme arose. Using two level paging with the extended system memory would lead to huge page tables. As Linux should be compatible with the 64-bit architecture, it adopted a three-level paging model. The 64-bit linear address used in the alpha architecture, which is used by Linux as example model, is split as follows:

- 21 bits are set to zero
- 13 bit are the offset for the 8 KB page
- 3 x 10 bit are used as table entry addresses in the 3 translation steps

Linux is using a comparable division of the 64-bit linear address, but the bit lengths vary depending on the computer architecture used.

When used on a 32-bit microprocessor, the middle translation step is essentially eliminated by setting the address bits to zero, setting the number of entries for the
2.2 Virtual Machines

A virtual machine (VM) is the software representation of a physical machine. Through virtualization, it is possible to run a guest machine within another host machine, with little or no modification to the guest operating system (OS). There are two basic types of VMs. System virtual machines provide a complete system platform and allow complete operating systems to be run on them. Process virtual machines, on the other hand, just allow one process to be run in a virtualized environment. A famous process virtual machine is the Java Virtual Machine (JVM) contained in the Java Runtime Environment (JRE) [34].

The piece of software providing the virtualization layer for the virtual machines is called virtual machine monitor (VMM) or hypervisor. The Xen hypervisor for example, runs directly on the hardware.

To understand the differences of the virtualization techniques, some background on the privilege rings concept for CPU instructions is given next. As seen in Figure 2.3 [21] the concept provides four protection rings, ordered from highly trusted (ring 0) to least trusted (ring 3). The kernel runs in ring 0 and has direct hardware access in it. In contrast, user programs - such as web browsers - run in ring 3. They do not have any direct hardware access. All hardware access requests from outside ring 0 have to be done with system calls, provided by the operating system. These system calls trigger the execution of trusted code inside ring 0. Although the concept intended ring 1 and 2 for a more granulated access level distinction, few operating systems make any use of them. This is again because of portability reasons in the operating system design, as not every hardware has support for the four privilege rings. The new processors from AMD and Intel support a virtual “ring -1”, called

Figure 2.3 x86 privilege rings

Page Middle Directory to one and then mapping this single entry directly into the Global Page Directory. This makes it possible to run the same code on 32-bit and 64-bit architectures.
Table 2.1 Different Hypervisors

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Ring</th>
<th>Supported Guest Architecture</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual PC (Microsoft) [6]</td>
<td>0</td>
<td>Emulates an x86 system in software</td>
<td></td>
</tr>
<tr>
<td>Hyper-V (Microsoft) [57]</td>
<td>-1</td>
<td>x86 systems</td>
<td>Very similar to Xen</td>
</tr>
<tr>
<td>VMware (VMware Inc.) [8]</td>
<td>-1 / 0</td>
<td>Emulates x86 and x86-64 systems</td>
<td></td>
</tr>
<tr>
<td>Xen (Citrix) [4]</td>
<td>-1 / 0</td>
<td>x86 and x86-64</td>
<td>Full HVM, Paravirtualization</td>
</tr>
<tr>
<td>Cooperative Linux (GPL) [16]</td>
<td>0</td>
<td>x86 Linux</td>
<td>Virtual Hardware, Emulated Devices</td>
</tr>
<tr>
<td>Simics (Virtutech) [30]</td>
<td>3</td>
<td>Arbitrary</td>
<td>Simulates the complete hardware, heterogeneous target architectures</td>
</tr>
<tr>
<td>Virtual Box (Sun) [15]</td>
<td>0 / -1</td>
<td>Emulates x86 and x86-64 systems</td>
<td></td>
</tr>
<tr>
<td>QEMU (GPL) [28]</td>
<td>3</td>
<td>Emulates all x86 and x86-64 systems</td>
<td></td>
</tr>
</tbody>
</table>

VMX, specifically designed for virtualization. It allows the hypervisor to run in a privilege ring below the normal operating system and the OS to stay in ring 0. More on the benefits of this later. [29]

An important characteristic of any virtual machine is its limitation to the resources and abstractions provided to it by the hypervisor. It is the reason why it cannot break out of its virtual world and is very important for the security of the hypervisor and other VMs running on this machine.

There exist too many hypervisors/VMMs of different developers to list them all here. Table 2.1 gives brief information on a few important.

The implementations differ in the way they virtualize a guest system. Some hypervisors run in ring -1, leave the OS running in ring 0, where it was supposed to run, and the user programs remain in ring 3 (Hyper-V, Xen). Other hypervisors run in ring 0 and degrade the OS to run in ring 1 or 3 (Xen). In both cases, system calls from ring 3 will be caught by the hypervisor, potentially altered and then rerouted to the OS. Again, other hypervisors may run itself in ring 3 and simulate the whole computer for the virtual OS (Simics).

2.2.1 Advantages

Virtual machines (VMs) have many advantages compared to physical computers: Using VMs makes the system installations platform independent. One can host several different operating systems in parallel, on a single physical machine. This is useful for debugging purposes, or to overcome operating system incompatibilities.
Besides this, offline system migration between different physical machines is straightforward and consists more or less of suspending the VM, transferring the system image and VM configuration to the new computer and starting it there.

Backing up the whole VM is just as easy. Analogous to offline migration, one has to stop the VM and copy the system image to the backup location. Current software allows snapshots of running systems. These snapshots can be used as restore points, which are very much like classic backups [9].

Virtual machines can be relocated without being stopped. Though the process being rather complex and the relocation taking some time to complete, the resulting system downtime can be as low as 60ms [24]. Which wouldn’t even be noticed on a game server for example.

Like with multiprocessing, virtual machines allow the user to exploit more system resources, which would otherwise stay unused, using only one application or (virtual) system at a time.

One most significant advantage of virtual machines is the separability of different services. System security is vastly improved when the provided services run each in their own virtual machine [20]. So, when a certain service is compromised, allowing the attacker possible control over the operating system, everything outside that virtual machine stays untouched.

Because of the hardware for the virtual machine being a configurable software layer, the system is much more flexible and adaptable to the changes in the needs of the VMs. Several tasks that are hard or even unable to accomplish on real hardware, are easy to do with a software simulated one. An example for this is the state manipulation of a virtual machine. It can for example be saved, cloned or encrypted.

### 2.2.2 Challenges

There are two major challenges in providing services with virtual machines. The first is performance. The virtualization overhead always impacts overall performance of applications. Even when the user code in ring 3 can run natively on the CPU, all system calls have to be trapped, processed and forwarded by the hypervisor. This overhead has to be outweighed by the benefit of running the service inside the VM.

Although being very popular, virtualizing x86 systems gives us additional problems. For a system to be virtualizable, all instructions that change the state of the system in a way that impacts other processes (sensitive instructions), have to be privileged instructions. A privileged instruction has to be executed in ring 0 or traps otherwise. In x86 systems, not all sensitive instructions are privileged instructions. This is a violation of the formal virtualization requirements for a system, stated first by Popek and Goldberg, and has to be dealt with [53].

One way, in which for example VMware deals with this problem, is called binary code rewriting [41]. The idea is to rewrite the running binaries to force these special silently failing instructions to trap anyway. This overhead has additional impact on the performance.

The other challenge in virtualizing services is that they now operate on top of the abstractions provided for the guest operating system. A service, which for example
checks the file system integrity for the physical computer, now has to use the virtualized hardware and must be provided the knowledge of the real on-disk structures [20].

2.2.3 Implementation Techniques

CPU Virtualization

In theory, virtualizing a CPU is straightforward. One process runs on it for a while, using it exclusively. After that, the current CPU state is saved and exchanged and the next process runs for its time slice. This is already done in every multi-tasking OS and typically occurs every 10ms. During the phase when the operating system is swapping the CPU state data, the CPU runs in privileged mode, allowing access to all memory addresses.

This process has to be extended from physical CPUs to virtual ones. The hypervisor has to do some additional work, translating from virtual to physical addresses and back, before the OS is allowed to swap CPU state data of different processes.

As mentioned, Popek and Goldberg stated a set of requirements in their paper of 1974 “Formal Requirements for Virtualizable Third Generation Architectures” [53], which have to hold for a CPU to be completely virtualizable. They required that all instructions that attempt to change the configuration of resources in the system, such as updating memory mappings and device communication, and those that act differently depending on the configuration of resources, such as all load and store operations on virtual memory, have to be privileged instructions. I.e. an instruction that either executes in privileged mode or traps otherwise.

As this is not the case with the x86 architecture, which has 17 instructions which do not trap - but fail silently - when executed outside privileged mode, the virtualization solution has to cope with this nuisance.

I/O Virtualization

The CPU is not the only thing one needs to virtualize for a virtual operating system to run. The physical memory is one of the I/O devices which is fairly easy to be virtualized. As written in the reference for this subsection [22], it can be partitioned, with one designated partition for every VM, and the memory access instructions trapped and translated to the corresponding partition range. Modern CPUs contain a Memory Management Unit (MMU), which does these translations in hardware, when provided with the required information by the hypervisor.

This mechanism can be transferred to other block devices such as harddisks. But there are devices clearly not designed with virtualization in mind, which need specialized approaches. Actual graphic cards for example, cannot be simply partitioned into virtual areas. Because of 2D and 3D acceleration, they do have lots of internal states, to which mostly no save-restore mechanism exists. So even switching the screen between several running VMs is difficult.
Many I/O devices interact via Direct Memory Access (DMA) with the system. The devices write directly into the system memory to an address designated by the device driver. This raises another problem as the devices, existing outside of the framework of the operating system, are limited to the use of physical memory address spaces. This will not work together with a virtualized operating system and a virtualized address space. One could tackle this problem by trying to trap and translate these DMA accesses. The reasons why this is not possible with DMA are the following. First, the performance overhead for this translation would be too big. Second, even if it wasn’t, each device driver has its own protocol for the DMA access and so the hypervisor would have to know the protocol, parse it, translate the addresses and change the device communication. It would be less work and more efficient to write special virtualizable drivers for the device in the first place.

Some hardware platforms provide an Input/Output Memory Management Unit (IOMMU), similar to the MMU for the memory. Where provided, it can be used to make the physical-virtual address translation for the devices.

**Binary Code Rewriting**

One approach to solve the x86 virtualization problem is binary code rewriting as propagated by VMware. It allows most parts of the virtual machine to run in user space, at the cost of performance. The hypervisor has to scan the stream of instructions for the privileged ones. They are then translated to their emulated counterparts and reinjected into the stream.

The VM performance is typically between 80-97% that of the host machine. It is worse in code segments containing many privileged instructions, so I/O intensive work is especially slow [22].

The implementation uses breakpoints, which interrupt the running process and allow the inspection and emulation to take place. They are inserted on any jump and on any unsafe instruction. So before a jump occurs, the instruction reader has time to scan the target area for unsafe instruction.

The rewriting process strongly benefits from caching replacements, as the same instructions often occur repeatedly in the stream, and the replacement calculations can be omitted the second time. The caching process works at the expense of system memory of course.

**Paravirtualization**

Yet another approach, this time popularized by Xen, is paravirtualization. The idea is to not simulate an x86 system, but the closest system to x86, to which the virtualizing requirements hold. In other words, the hypervisor silently ignores the problematic instructions and the operating systems itself has to deal with them. To make this work, the operating system code has to be rewritten, but because this happens at compile time it causes no performance penalty. The main problem is that this limits the available guest systems to especially patched versions, which are mostly only provided for some open source operating systems like Linux, NetBSD, OpenSolaris or FreeBSD [63].
An operating system being paravirtualized with a 32-bit Xen runs in ring 1 instead of ring 0. So it cannot directly execute any privileged instructions. Like user processes make system calls to communicate with ring 0, the guest OS - aware of being virtualized - uses hypercalls to communicate directly with the hypervisor. Hypercalls are conceptually similar to system calls and there exists one for every privileged instruction.

The fact that the guest operating system can be optimized for virtualization also allows for better I/O performance during runtime. The guest OS uses lightweight interface drivers to communicate with the real driver layer inside a privileged domain. This real driver talks to the physical device. This eliminates the overhead to emulate a real device for the guest, which could then use its default driver. The lightweight interface driver only needs to forward request and answer - forth to and back from the hypervisor.

**Hardware-Assisted Virtualization**

The new processors from Intel and AMD have a special set of instructions specifically designed for virtualization. AMD’s feature is called AMD-V, formerly Pacifica, and Intel’s is called (Intel) Virtualization Technology (VT). The concept behind both is the same. A new privilege ring (ring -1 or VMX) is added below the operating system in ring 0. Allowing the OS to stay in ring 0 removes the need for hypercalls, so that unmodified operating systems can be run as guest. The new privilege mode allows the hypervisor to trap and emulate even those instructions, which previously silently failed. Additionally, when the CPU is in VMX mode, a new set of instructions is available to the hypervisor. They are used to store CPU states in the memory, start and stop VMs and for other VM management tasks.

A virtual machine using hardware assisted virtualization is called Hardware Virtual Machine (HVM). Several procedures in a HVM are faster compared to a paravirtualized guest. The hardware accelerates system calls and AMD-V provides hardware support for nested page tables, which are used for the memory address translations by the hypervisor. On the other hand, an unmodified OS produces overhead, as it needs emulated virtual devices matching the OS’s driver layer. This particularly slows down I/O operations.

A hybrid virtualization solution, using lightweight interfaces and hardware acceleration, offers the best performance [63, p. 340-341].

### 2.3 Xen Hypervisor

There are two main reasons why Xen is a good platform for monitoring a virtual machine’s kernel. First, Xen is open source, which allows deep insight and tampering of the hypervisor itself. Second, Xen’s main guest system, especially for paravirtualization, is Linux. Linux is open source, too. A prerequisite for effective kernel debugging is the kernel source or at least the needed debugging symbols.

To get a detailed overview on Xen “The Definitive Guide to the Xen Hypervisor” [22] is recommended, which serves as a main source for this chapter along with [17].
2.3.1 System Design

Xen has been developed with one key idea of every good system design: the separation of policy and mechanism. The Xen hypervisor is the mechanism by which the policies, set in Domain 0 (Dom0), are implemented. Dom0 is an especially privileged guest VM (or domain) on the system. The other unprivileged domains are called DomU. Xen provides mechanisms to give a guest system direct access to hardware. The guest OS has to implement the device driver. When a device should be shared between more than one guest, the default device driver has to be extended. Xen only provides a mechanism to share memory pages among guest systems (the grant table), which is used by the special device driver to provide device access to other guests (split driver model). The presence of such a device needs to be propagated. This is done in the XenStore, which implements a file-system-like hierarchy and is used as a discovery service for the shared pages.

As can be seen, the Xen hypervisor only implements these simple mechanisms. This is because of another design idea: “Less is More”. The hypervisor runs at the system’s highest privilege, which makes it very security relevant. A compromise of the hypervisor not only affects one system, but all running virtual machines. So, to make the Xen source code as secure and bug free as possible, it is to be as small and manageable as possible, too.

As the policy and the drivers are implemented by the OS in Dom0 and not by Xen, which only provides the available mechanisms, Xen is very flexible with respect to the supported hardware and devices.

2.3.2 Domain 0

Though being just one of maybe many running VMs, Domain 0 (Dom0) has a special role. As said, most system drivers are implemented here and the hypervisor’s policies are configured. When the hypervisor is started by the boot loader, one of the first things it does is to boot Dom0. All the Xen user space tools - needed to start, stop and configure Xen - are also located here. It is because of this elevated position that the security of Dom0 is equally important as the security of the hypervisor itself.

Dom0 provides the tools to install a new guest VM and configured it. Depending on the type of disk used by the guest VM, the harddisk image or dedicated partition is also managed in Dom0. Most of the user space tools (like `xm`) are Python scripts that control the hypervisor with hypercalls.

The main operating system used for Dom0 is Linux. Even though NetBSD and Solaris can also be used and other operating systems are likely to add support for it in the future.

Sometimes the line between Dom0 and DomU is not so clear. A DomU could be given direct access to a hardware device to test an untrusted device driver, separated from the others in Dom0. Still, the other guests are vulnerable to erroneous DMA requests and possible I/O outages on systems without an IOMMU, but are shielded against the driver possibly running haywire. A system with an IOMMU can prevent the buggy driver completely from interfering with other drivers.
2. Background

Figure 2.4 x86 Paravirtualization

Figure 2.5 x86-64 Native and Paravirtualization

2.3.3 Xen Implementation

The next subsection describes how Xen works with the privilege rings.

How a paravirtualized VM, running on an Intel IA32 processor, would work is shown in Figure 2.4. The hypervisor takes control of ring 0 and pushes the OS up to ring 1. The user programs remain in ring 3 as normal. AMD tidied up the IA32 architecture in the process of creating the x86-64. As a result (non-Pacifica) AMD processors support only two privilege rings. Xen deals with this by moving the OS up to ring 3 instead of ring 1, shown in Figure 2.5. There it runs together with its user programs. The same solution is used for every other processor architecture supporting only two privilege rings.

When run on a HVM supporting architecture, Xen moves itself to the new ring -1 and leaves anything else in place: OS in ring 0, applications in ring 3.
Hypercalls

In the case of paravirtualization, when the OS runs in ring 1 or 3, it now has the same problem as user space applications, when interacting with hardware. It is not privileged to do so. The former introduced hypercalls solve this problem. They provide a controlled way to bypass the privilege restriction. Like system calls, they are provoked by writing the arguments in CPU registers or on the stack and then calling a well known interrupt (82h). Newer CPUs have a special instruction to make system calls. Since Xen Version 3, calling an interrupt to make hypercalls is deprecated. Instead, a hypercall is performed by calling an address of a special hypercall memory page, mapped into the guest’s address space.

Xen Event Model

There are other equivalents, like system calls and hypercalls. The Xen Event Model very much corresponds to Unix signals. Signals are an asynchronous mechanism, which allows a single bit of data (event data) to be delivered to a program from the outside. The source could be any external entity like the kernel, another program or a user. To receive and process signals, programs have to set up special handlers. Analogous to that, guest kernels should register a callback for the event delivery mechanism of Xen. Events can be used by Xen itself or any other guest. They may represent hardware or virtual interrupts, or can be used to build more complex asynchronous communication paths, for example to indicate incoming data in a ring buffer - located in a shared memory page.

In contrast to Unix signals, events cannot be send without an active preparation on the receiver’s side. Xen events are sent via channels with a receiving respective sending port on each end. The receiving guest first has to create a new port and advertise its existence via the XenStore. The sender also creates a new port and binds it to the receiver’s to form a channel, before it can send an event.

Xen Shared Memory - Grant Tables

The Xen Shared Memory was also mentioned before: an analogous mechanism to POSIX shared memory. Each domain has an own grant table, through which the domain can tell Xen what kind of permissions other unprivileged domains have on its allocated memory pages. Of course every privileged domain can access any memory page of the system. This option is used to analyze, debug and tamper with DomUs from Dom0.

The shared memory pages are needed because Xen Events are not sufficient to implement enhanced IPC mechanisms like pipes, message queues and shared memory itself. As Xen is a minimalistic hypervisor and pipes and message queues can be implemented solely by using shared memory, it is the only mechanism provided.

The two possible interdomain operations on memory pages are sharing and transferring. Sharing simply gives both domains access to the contents of the page. It maps the memory page into the address space of both guests. Transferring can be thought
of as a message passing mechanism with the granularity of a page size. This is realized by mapping the memory page into the target’s address space and removing the same from the source’s.

Shared memory pages can be used to implement shared ring buffers. A standard data structure, used for a producer-consumer scenario. This scenario often applies to device I/O communications and so shared ring buffers are frequently used by the split drivers. Figure 2.6 gives an indication how producer and consumer are using the buffer. Xen allows responses only to be written over requests.

Simple shared memory ring buffer algorithms need a lot of polling for incoming data, which in particular is not very efficient. Hence, most ring buffer algorithms in Xen are merged together with the Xen Event mechanism, to indicate incoming data. This eliminates the need for polling. For a VM guest to know, where to look for available split driver ring buffers etc., it has to search in the XenStore.

**XenStore**

The XenStore is used to communicate the existence of shared drivers, Xen Event Channels and other information between running VMs. It is a virtual storage system shared between the Xen guests and has a hierarchical file system composed of directories and subdirectories. A daemon process (xenstored) running in Dom0 maintains this service and access to it is provided through shared memory pages and an event channel for the guests.

The XenStore is not intended to collect large amounts of data, but to provide an extensible method of transmitting small amounts of information between domains. It is also used to store information on the running VMs, to be used by the administrative tools in Dom0.
2.3. Xen Hypervisor

QEMU Drivers and the Split Device Driver

There are two types of drivers that provide guest VMs with virtualized devices. First, emulated devices based on the QEMU project which are mainly used for HVMs. Xen uses the QEMU-Daemon (qemu-dm) running in Dom0 as a backend for the device multiplexing [63, p. 38]. I.e., the VM sees some sort of default hardware like a RTL8139 network card and uses its build in drivers to communicate. The result is additional overhead compared to the split drivers, which are mainly used in paravirtualized guests, as the HVM first has to encode the I/O request with its driver layer and the QEMU driver has to decode the same before it can be forwarded and recoded to the real device driver. Only the last step is shared with the split driver of a PVM. The first two are replaced by a simple forwarding mechanism using memory buffers, which are shared using grant tables and advertised with the XenStore. While special split device drivers have to be supported for the guest VM, implementing them is relatively straightforward and the performance gain outweighs this effort easily.

To get an idea which steps for example a network packet has to run through coming from a paravirtualized guest system, Figure 2.7 shows an overview. An application sends the packet using the local TCP/IP Stack. It is then send to the frontend of the Split Device Driver and put into the Shared Memory Segment. The split device driver uses a Xen Event to signal its backend of incoming data. The backend then reads the data and puts it in its local TCP/IP Stack. Here, the routing and multiplexing takes place. The new data packet is then treated like every other packet and send to the real network card using the real network driver in Dom0.
Modern operating systems use the concept of virtual or protected memory to provide every running application with a flat and contiguous amount of memory. The operating system itself, normally working on physical memory, is also used to work on its own address space. While running on top of a hypervisor this layer is called pseudo-physical memory. Figure 2.8 shows how the additional layer of indirection is added to the default paging model by the hypervisor. The new indirection layer is needed because the hypervisor has to ensure the principle of isolation. Xen uses a technique to catch and process all page table operations, which is called shadow page tables. It allows each guest to have a copy of the page table mapped in a set of write-protected pages. When the OS tries to update the page table, it causes a fault caught by the hypervisor. The update is then validated and translated from pseudo-physical to machine addresses by the hypervisor, which updates the real page table and continues.

For a HVM with an unmodified operating system, this has to be done invisible for the OS. Maintaining shadow page tables - for the guest - in software is actually quite expensive. Fortunately the VT-x and AMD-V CPU extensions both include hardware support for that.

Paravirtualized modified guest operating systems have another possibility to solve this. There exist hypercalls corresponding to MMU updates, which the real MMU interactions have to be replaced with. Although being a very fast solution in the end, it means a lot of work to translate all existing page table updates in the OS source code.

**Time Keeping in Xen Virtual Machines**

A physical system has access to a hardware clock (also called BIOS or CMOS clock), which is powered by a battery and is running, even when the PC is turned off. It can issue interrupts at a maximum frequency of 8192Hz, resulting in a resolution of approximately 122 milliseconds [18]. As access to it involves hardware I/O, it is relatively slow to read, compared to software clocks. Most systems only read the hardware clock once, at boot time, copy the time value to a software clock and use this from that point. The resolution of the software clock is given by the operating system.
A computer system distinguishes two types of times. First, the wall clock or real 
time. It is used mainly for user space applications to show the current time, run 
scheduled tasks, timestamp logfiles etc. The granularity of the ticks is not too 
important here.

The other is virtual time - the aggregated time the system is running. It is needed 
to schedule running tasks of the computer. The same applies to guest domains. Its 
virtual time only increases when the guest is in running state. This is implemented 
in Xen by sending a periodic tick every 10ms to a scheduled guest [22].

Three time values are used to calculate the current real time by the OS.

**Initial system time** is the wallclock time of the point when the system started or 
resumed and the software clock began to run. It is exported by Xen in the 
shared info pages.

**Current system time** is the real time duration from that same point. It is up-
dated whenever the system or guest is scheduled.

**TSC time** called after the Time-Stamp Counter (TSC), it contains the number of 
CPU cycles that have elapsed since a point in the past. The difference of such 
two TSC timestamps is used for the real time calculations.

An example implementation of the gettimeofday() function is given in [22, page 56].

Although the TSC is very fine grained, its accuracy depends on the timing circuitry 
in the system, which is mostly disposed to a variable skew. To combat the resulting 
drift of the system clock, the Dom0 guest is expected to synchronize its clock and 
system time with a high-resolution time source, for example using an NTP client.

For analysis or debugging purposes, it is sometimes useful not to synchronize the 
running guest VMs with real time but with a simulated time. This can be stopped 
and resumed by the analysis tool so that the target guests do not notice the missing 
time period, when they were stopped for investigation. This is particularly useful for 
the analysis of network protocol, which would encounter protocol timeouts otherwise.

VM time synchronization is discussed in the thesis of Florian Schmidt in great detail 
[60].

### 2.4 Kernel Programming and Debugging

Since this approach concentrates on monitoring kernel protocols and kernel code 
with other tasks, some background on the specialties of programming and debugging 
kernels is needed. Other basic mechanisms like symbols or breakpoints are relevant 
to know, too.

#### 2.4.1 Symbols

Symbols or symbol tables are not only used inside kernels. A symbol table maps 
instructions in a compiled program (binary) to their corresponding variable or func-
tion in the source code. Symbol tables may be stored in separate files or embedded
into the program file. But they are not always created by default. The compiler must be told to create a debug version. For example, the GCC compiler supports the inclusion of the debugging symbols with the \texttt{-g} flag. A normal or retail build is hence smaller, and it is more difficult to reverse-engineer it.

Symbols can also be explicitly exported by a program, the kernel or kernel modules to provide their internal functionality to external users. For example, a kernel part could export a symbol to one of its functions, which is then used by a kernel module to call the same [58].

The symbol information are also required by debuggers to map variable- and function-names to memory addresses. They can be thought of as a map of the memory area.

### 2.4.2 Available Functions

When writing kernel code, functions provided by standard libraries are not available. The reason for this is: Many functions provided by libraries make systems calls to do the real work. This would bring the CPU in kernel mode, which kernel code is already in. Hence this would not work. So what is available? The functionality (symbols) provided by the kernel and of all loaded modules is gathered in \texttt{/proc/kallsyms}. A famous kernel function is \texttt{printk()}, which is alike \texttt{printf()} provided by the standard C library, libc [59].

### 2.4.3 Interrupts

One main task of the operating system kernel is the communication with the hardware. It consists of the CPU giving orders to the hardware and the hardware talking back to the CPU. The second is realized with “hardware interrupts”. They are handled by special functions in the OS called “interrupt handlers”. It is difficult to implement these interrupt handlers. The reason for this is the CPU has to deal with the hardware requests anytime they arrive and whatever it may be doing at the moment that happens. Typical hardware devices have only a very small amount of RAM to buffer data. So when the CPU takes too long to read it out of the buffer, it may be lost.

There are two types of hardware interrupts (IRQs - InterruptReRequests). The first are short uninterruptable interrupts, which are expected to run only for a very short time. The second are long interrupts. They are allowed to process longer, and other interrupts with higher priority (from other devices) can preempt them.

When the CPU is interrupted, it stops the current process (unless is is already processing a more important interrupt), saves some parameters on the stack and calls the interrupt handler. As the interrupt handlers are supposed to run only for a short time, they mostly only save the buffer data and schedule the postprocessing to their so called bottom half. A Linux kernel uses \texttt{request_irq()} to register an interrupt handler for a certain interrupt number.
2.4.4 Signal Handling

Unix Signals were already mentioned in comparison to the Xen Event Model. They are like a notification service for processes and can be compared to interrupts, only for interrupting processes. The signal handlers are functions contained with the rest of the program and have to be registered during runtime. So when a signal is received, the operating system stops the execution and calls the corresponding signal handler in the program. Execution is continued when the signal handler returns.

Signals can be send to a process with the \texttt{kill} command, by the operating system or any other process with enough permissions [42].

2.4.5 System Calls

The preceding chapters showed that system calls are used by processes running in user mode to contact the operating system. Common requests are file I/O or printing to the screen.

The kernel maintains a table of system calls (\texttt{sys\_call\_table}) which maps system call numbers to function addresses. These are run when the system call is processed. Tampering with this table may render the system unuseable and should be avoided on productive systems.

However, to register a new system call or replace an existing one, the kernel programmer has to make the reference in the \texttt{sys\_call\_table} point to his function. For kernel modules, this is the job of the \texttt{init\_module} function. Equally important is to remove this reference or make it point back to its old location before the module is unloaded, mostly done in the \texttt{cleanup\_module} function of the same kernel module [18].

2.4.6 Kernel Timers

Kernel Timers are part of the kernels internal API and are used to let specific events happen at some point in the future. One fundamental limitation is that kernel timers depend on the kernel clock interrupt. On a x86 architecture running a 2.6.13 kernel with the default clock interval, this limits the timers to a maximum resolution of 4ms [25]. This may be precise enough for many applications, but is not for some real-time and desktop multimedia applications.

Improvements to this resolution can be achieved by using especially patched kernels or other specialized real time operating systems. Current computer hardware brings support for higher timer resolution, so called High Precision Event Timers (HPET). The Linux kernel supports these in current versions. The HPETs have a considerably higher frequency of 10 MHz leading to a timer resolution of 100nano seconds.

If one needs even more real time abilities for the Linux system, there exist frameworks like RTAI [7], which uses patched kernels and special services to provide a real time application interface.
2.4.7 Debugging Programs

Debugging is used to remove bugs. It does not directly remove them, it only helps the developer to find them. Many code errors are found at precompile- or compile-time. Runtime errors are more difficult to find. The erroneous part of the program has to be found and inspected.

As mentioned, to map binary parts of the program to real names from the source code, the debugger needs the symbol information created by the compiler.

To find bugs, the content of selected variables has to be watched, the program has to be stopped on certain conditions (watchpoints) or at specific points in the code (breakpoints). Even though this could be done by implementing the functionality inside the code, it is easier and much more efficient to use a debugger for it. A debugger provides several additional features [52]:

- Showing the current stack frame of the program
- Backtraces (a kind of description of how the program got to that point)
- Changing variable contents on the fly at runtime
- Examine arbitrary memory contents
- Examine the contents of CPU registers

Even single stepping through every code-line of the program or used libraries is possible, when the debugging information for these are available to the debugger, too.

To begin analyzing a program, the debugger could start the program itself or attach to an already running process. It is then under direct control until the debugger unattaches itself.

One famous open source debugger, and the one used in this thesis, is GDB.

2.4.8 GDB

GDB can be used to debug programs written in C or C++. Partial support exists for other languages like Pascal, Fortran and Modula-2 [64].

It supports all features mentioned in the last section and a manual can be found at [64].

As seen in Figure 2.9, GDB can attach to local processes directly or connect to the GDBserver, running in another location, which itself locally attaches to the target process and is remotely controlled by GDB. The connection between GDB and the GDBserver may be established over a serial line or a TCP connection.

Before GDB can attach to a process, it has to be configured for the target. GDB has to be set to the right target architecture specifications like 32-bit/64-bit, little/big endian or processor type. Furthermore, GDB needs local access to the debugging information to load them.
2.4.9 Scripting GDB and GDB/MI

GDB cannot only remote control a GDBserver, it is often integrated in other software and is therefore remote controlled itself. Many integrated development environments (IDEs) integrate GDB and provide a graphical user interface (GUI) to the debugging procedure (for example Eclipse [13] or DDD [14]).

For this reason, GDB provides multiple command interpreters as shown in Figure 2.10. A simple console command-line interpreter (CLI), which is used by default and is intended for console use by humans, and a special machine interface (MI). There exists an old (mi1) and a new (mi2) version of the MI protocol, but the old version only still exists for backward compatibility.

The task of the MI is to provide GDB as a backend for debugger GUIs or IDEs. It is done by providing a machine readable (easy to parse) input and output syntax for received commands and produced responses [12].

If one wants to integrate GDB into its project, only a GDB/MI interface connection has to be implemented.
2.4.10 Kernel Debugging

As in normal application debugging, it is possible to implement own debugging messages (via printk etc.) directly into the kernel. But as before, a debugger offers more features.

The main problem with kernel debugging is that a debugger running in user mode on the same machine cannot just attach to the kernel and debug it, as itself depends on it. Hence, other approaches are taken to debug kernels. Nearly always the kernel has to be patched for this procedure.

The Linux kernel for example already has several debugging outputs built in, which only need to be enabled in the kernel build configuration. The configuration option CONFIG_LL_DEBUG enables low level access to the serial port, which then can be used as output console and read from another system connected to it. This is especially useful when porting the kernel to a new architecture and no other way of reading the debug output exists [59].

There exist various debugging tools for the kernel, of which some are explained in detail in the related work Chapter 6. Many kernel debuggers need to patch the kernel before they work and use the serial port for communication. This implies that a second computer is needed, where the debugger frontend runs on and the serial port is connected to.

To debug the kernel of a windows system the same approach is taken. As the source code for Windows is not openly available, Microsoft provides the Windows debugging tools and the debugging symbols for their kernels online. A nice introduction into Windows kernel level debugging is given in [67]. It shows, how Windows XP has to be started with special parameters, to open up a serial debugging connection to the second machine, which runs the debugger. The debugger loads the needed kernel symbols from the Microsoft Symbol Server, which provides the symbol information for other Microsoft products, too. The debugging session is controlled by the user of the debugger. One tool that can be used for Windows Kernel debugging is WinDbg.

With virtualization, the two debugging computers could be replaced by one, running two virtual machines. The serial connection is simulated by the hypervisor and connects the two VMs. But virtualization enables another approach, too, as introduced in the next section.

Interim Résumé

Provided with all necessary background knowledge it is now possible to understand how the concept of virtualization can be used to debug an operating system kernel. The target kernel is put into a virtual machine and the hypervisor is used to access it. Monitoring and tampering from a privileged domain is then used to debug the target kernel, running in the test VM. The hypervisor exploits the ability to read, modify, start and stop all simulated hardware, including system memory and CPU registers [40]. This is the approach chosen during the design and implementation of this tool.
System Design

The goal of this work is to provide the developer with a tool that allows easy and flexible distributed monitoring of kernel protocol stacks. It is designed to work for three classes of use cases.

**Class one** are unconditional monitoring situations for a single kernel application or protocol. It consists of a developer machine, running a virtual machine with the kernel under development. The monitoring tool will be run on the same machine, in the privileged domain. The task for the tool is to monitor selected variables at scheduled times, for example to read the task list of the kernel.

The use case class number one can be extended to conditional monitoring, a class where more variables should be monitored under certain circumstances during the process. The virtual machine could be paused or snapshotted when variables break out of predefined limits. Still only one target VM is allowed here.

**Class two - reactive monitoring** consists of situations where virtual machines should be analyzed when certain events occur, for example predefined network packets arrive. Appearing events will be scanned and the VM will be shortly paused for deeper analysis, on selected incidents. To scan for example the network traffic, a hook must be installed at some point in the system. It could be in the driver layers of the system or VM, or in the kernel protocol stack of the VM.

**Class three - distributed monitoring** The third use case is a distributed setting with at least two physical machines. The object of analysis is for example a network protocol with two or more participants, of which all are examined at the same time. It is important to synchronize the time of all virtual machines that are communicating over the protocol, because occurring protocol timeouts would falsify the result. Hence, when one virtual network node is paused, all others must stop, too. This use case requires a tool with network functionality
3. System Design

as the distributed VMs have to be controlled from a central point for the analysis.

3.1 Why to use virtual machines

The approach of taking hypervisors to monitor systems has been previously addressed. The elevated position for an inspection of the VM is very practical and can not be easily reconstructed with real hardware.

A former work by the Distributed Systems Group of the RWTH Aachen University is using XenAccess [49] to monitor selected kernel variables - exported as a symbol - with a tool running in Dom0. It has several drawbacks regarding flexibility and will be described in Chapter 6.1.1 in more detail.

Virtualization software like VMware and User Mode Linux (UML) is already being used to help debug kernels. While UML starts the kernel as a user process and is then able to attach a local debugger, VMware simulates the two machines needed for kernel debugging over a serial console with the help of GDB [31].

Computer security systems take the approach of monitoring virtual machines to verify their integrity. They are able to detect and prevent incoming attacks on a whole different level than for example firewalls can do. The approach of Garfinkel and Rosenbaum uses checksums of memory parts to verify the integrity of the running virtual machine [33]. Furthermore, they use hooks inside the hypervisor (VMM) to validate requests from the VM before they are executed. Garfinkel and Rosenblum explain: “...the VMM provides the ability to interpose at the architecture interface of the monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events.”

This is extended by Nance et al. [46]: “This makes it possible for security tools — such as virus scanners and intrusion detection systems — to observe and respond to VM events from a ‘safe’ location outside the monitored machine.”

The ability to react to certain incoming packets - like a security system or firewall - is one aspect needed in our use case class number two.

We decided to extend the ideas of these approaches and implement the tool using a hypervisor to monitor the kernel protocols.

3.2 Usability vs. Versatility

The planned tool should be as powerful and easy to use as possible. Many powerful tools like for example a ring 0 debugger [11] are hard to use. One way to make a tool easy to use is to provide an intuitive graphical user interface (GUI). The drawbacks resulting from a GUI are mostly a reduced flexibility and increased implementation effort.

Our first approach was to provide a static command line tool with a flexible configuration file. It is no problem to cover use case class number one with such a
solution. The target variable, kernel and virtual machine only have to be entered in the configuration and the program can be used. The more complex use case classes two and three require more flexibility. One way is to extend tool and configuration file with options for these use cases, but it would mean further work on the tool for additional situations. The extended use case class one requires a way to specify the conditions under which a deeper analysis should happen. Use case class two requires a way to specify the packets which trigger the inspection. No configuration file could be flexible enough to cover all possible use cases the tool could be used for.

With or without a GUI, a static tool always limits the user to the monitoring/debugging procedures thought of during development. So, the design decision is to provide a scriptable frontend to the monitoring tool as backend. This provides the pure functionality without boundaries, resulting from a limited amount of work-modes.

Another aspect of flexibility concerns the target. If possible the exact target variables or functions should be changeable without patches to the target kernel or parts of the tool itself. So that the debugging focus can change, even at runtime. With XenAccess, the kernel has to export symbols for the variables explicitly and changes require a recompilation of the kernel.

The architecture of the tool could limit the application possibilities, too. The use cases, in which the tool could be used, should be as many as possible.

As before, a simple stand-alone tool - running directly on the real machine of the target virtual machine - would be sufficient for use case class one. Even use case class two could be realized with a stand-alone solution. The distributed settings in use case class number three can not. The possibility to synchronize and control several instances of the monitoring tool - running in parallel - requires a network remote control mechanism. An external entity that does control the instances is also needed.

The proposed architecture of the tool - to separate it in front- and backend - should make it adaptable to various test infrastructures consisting of arbitrary physical and virtual machines. When split into front- and backend the always desirable isolation of mechanism and policy of the tool can also be achieved.

The monitoring backend has to be developed for - and run on - the architecture used as hypervisor platform. The frontend however does not necessarily need to be build for the same platform. The network remote control allows it to be run on any developer machine in the network. When the frontend is implemented in a scripting language the need to compile it for an architecture is eliminated, too. Of course different versions can be implemented. For example as command line program or one with a Windows GUI.

The separation enables the developer to use different kinds of front- and backends together in one monitoring setting. He could for example use a GUI based frontend to control a Linux backend and a Windows backend.

The architecture could also be extended to monitor user space applications. One way would be to implement a monitor backend running inside the virtual machine, which directly attaches to the target process. Using the same network command interface, it would easily integrate into a distributed kernel monitoring only setting.
Another way would be to use knowledge of the kernel structure to read the process table information and find the process in the virtual machine’s memory. This is of course more complicated and less comfortable than a directly attached debugger.

3.3 Architecture Concept

Figure 3.1 shows the architecture concept for the tool. It consists of a frontend, running on the computer of the developer, and a backend, running on the same machine the target virtual machine does. The tool is controlled by the developer using the frontend, which controls the policy. The backend does all the work, providing the mechanism and is merely remote controlled. It must run in a privileged domain on the target machine so that it can control the target VM and can read/write to the VM’s memory.

The following two figures demonstrate the versatility of the chosen design. Figure 3.2 shows how multiple types of front- and backends can be mixed together in debugging sessions. Even different architectures can be interconnected. Potential approaches to debug user space applications with the toolset are pointed out in Figure 3.3.

3.3.1 Monitor Frontend

The monitor frontend has to provide access to the functions of the tool. It should be scriptable, so that monitoring and debugging can be automated or complex monitoring settings can be adequately configured. To provide a distributed environment the frontend has to communicate with the backend over default network sockets. This makes it possible to use the frontend on the test-machine itself or on a remote machine. The network requests and replies have to be marshalled and unmarshalled so that they can be send over the communication channel.
Figure 3.2 Toolset adapts to different infrastructures

Figure 3.3 Access to user space applications with the toolset
The network feature could be replaced by a local communication method when front-
and backend run on the same machine, but is definitely required for distributed
monitoring settings like in use case class three.

The first step planned is to build a frontend configurable in any script language
to program the planned session. Later steps could include the development of an
enhanced graphical user interface for the frontend.

The advantage of using a script language is a combination of more flexibility and
straightforward usage, compared to a static complex configuration file.

One drawback on the other hand is, the user must learn the script language of the
tool to work with it. Another is the overhead resulting from the use of a network
channel to communicate with the tool’s backend, which could be avoided in non-
distributed monitoring cases by merging front- and backend.

Use case class one and two are such issues, where a stand alone tool would be
faster and sufficient, as the network feature is not needed. The scripting interface is
however required for all use case classes but the first non-extended one. The extended
use case class one already needs a way to define the conditions under which a deeper
analysis should happen. As does use case class two, to script the events and define
the network packets that should trigger them.

The configuration of the third distributed use case class is obviously too complex to
be mapped in any configuration file. It also requires the separate frontend to control
the various existing distributed backends in this setting. These could be running on
several physical machines with different architectures. The backend only has to be
implemented for these architectures and must use the same communication interface
to the frontend.

Requirements to the scriptability of the frontend are straightforward syntax and
functional completeness, so that every thinkable case can be covered. One suggested
way to do this is to provide an object for an already existing scripting language to
which these requirements hold.

3.3.2 Monitor Backend

The monitor backend represents the mechanism of the separation scheme. Its task
is to provide the functionality of the tool to the frontend. The backend contains
modules, implementing this functionality. As back- and frontend use a well defined
communication interface, different versions or implementations can be used together.

The backend needs to be run in a privileged domain on the same host as the target
virtual machine. It needs high level access to the hypervisor. Hence, it has to be
build for the same computer architecture. One backend can attach to only one
virtual machine at a time, so there has to be one backend running per target VM in
the setting.

The configuration of the backend can be done in a plain configuration file, contain-
ing the target VM’s name, target architecture and the path to the kernel symbol
information.
As the backend only answers to requests, the application logic is straightforward. First things to do are: to open a port for the network communication, read the symbol information and attach to the virtual machine. After that, it waits until a request arrives. The request is processed by the according modules of the backend and the answer is sent back.

The benefits gained by this design are a pure functionality without boundaries of a static controller and the possibility to use the backend with any compatible frontend. An advantage for the development-process is that the backends can be implemented separated from the frontend. A backend version can be build for every architecture appropriate.

The drawback is the same as with the frontend: the network communication has a small overhead and time dilation which could be eliminated for very simple monitoring situations with a stand-alone solution.

The backend itself can be segmented into three logical parts, as seen in Figure 3.4: network module, virtual machine control and memory handling. In an object oriented programming language, these will probably be represented by separate classes. The following subsections give detailed information on each of the modules.

**Network Module**

The network module has the task to handle all network communication for the backend. First, it has to open the communication port to the outside - over which the frontend talks to the backend. The incoming requests must then be unmarshalled and dispatched to the other backend modules for processing. The resulting answer has to be marshalled and send back as reply to the frontend.

The communication model is that of a default client-server situation. There already exist many frameworks and libraries to handle such tasks. Therefore it can be chosen to use an appropriate library or to write own network code. Using an existing approved working library has the major advantage of building the monitoring server on a firm foundation. In contrast, writing own network code is like inventing the
wheel just another time, as no new challenges exist in this part of the toolset’s source code.

There exist several libraries for remote procedure calls (RPC) which is exactly what is appropriate for the toolset. The libraries can be categorized, as they are working on different levels in respect to transport layer usage and the utilization of other standards like XML. SOAP, formerly used as acronym for Simple Object Access Protocol, is a request-response mechanism based on the hypertext transfer protocol (HTTP) [32]. It uses XML as data format to envelop the messages. SOAP is the successor of the XML-RPC protocol which also uses XML and HTTP but is more lightweight and easier to use. There exist libraries for both RPC protocols. Of course, other libraries exist that do not base on XML-RPC or SOAP, but use own protocols. As these do not have to use XML as message format, they can be more optimized and faster like for example Google’s Protocol Buffers [65] or Apache Thrift [62]. Which library exactly is chosen remains an implementation decision.

One drawback of choosing the RPC model for the client-server communication is that the RPC server can only react to requests and is not able to initiate a connection. In other words, the RPC server cannot tell the RPC client anything, it can only answer. When an initial communication from the monitoring server to the frontend is needed, it has to implement an RPC server as well.

Module Controlling the VM

This module is controlling the VM’s state. It is responsible for pausing and resuming. Saving and restoring VM states could also be provided by the module. These tasks require direct communication with the hypervisor as only he has the ability to process them. If possible, the work can be dispatched to the existing VM control tools already on the system of the privileged domain.

The ability to pause and resume the VM is needed for debugging, when it comes to breakpoints. When analyzing a state of the VM, reading multiple values successively, the feature is also needed, as the values could constantly change when the VM is still running and the read results would not fit together. With this module, the system can be stopped and the memory read in paused state. Distributed monitoring and debugging needs synchronization of the participating VMs, which also depends on this ability.

Saving and restoring a state are advanced features that could be implemented additionally. They can be used to save specific states, for example containing rarely occurring buggy situations. Later, these could be restored and analyzed. In general, the current memory, disk and register contents of the virtual machine are stored on disk to save the state. The advantage of using saved states is that the monitoring script can be altered and rerun on the exact same state of the virtual machine. This way the developer could analyze entirely different variables, he maybe never thought of before, with the contents of the same old situation.

One drawback with this kind of external control of the VM is that this module can only handle it as a black box. It has no indicator in which state the VM is when it triggers the pause command. Neither can it control the VM to run only for a
3.3. Architecture Concept

certain amount of CPU steps, which makes “single stepping” impossible. This kind of internal control falls into the responsibility of the memory handling module.

Memory Handling Module

The third module handling all the memory accesses to the virtual machine is very important. It is needed to do the actual reading and writing of variables in the monitoring process. The following section will give details on its complex tasks.

Reading the memory contents of a virtual machine may sound easier than it is. First of all, the tool needs the system permission to access the memory of another virtual machine running on our physical machine. This is one of the reasons why the backend has to run in a privileged domain.

Second, the other virtual machine has a whole independent memory address range, which cannot be accessed with the default addressing methods. Both domains are using pseudo-physical addresses, only the hypervisor is handling the real physical memory addresses. A mechanism to read/write these pseudo-physical memory addresses of the target VM from the privileged domain needs to be found. The memory parts of interest have to be mapped into the address range of the privileged domain. After that, they can be accessed like a normal memory page. The mapping process can only be done by the hypervisor, which has to provide a control interface for this mechanism.

The third and most complicated challenge is to find the correct position in the target virtual machine’s memory. As explained in the background chapter, without additional information, the target VM’s memory is only a block of data without structure. The variables one wants to read/write could be anywhere within. Hence, the module needs the symbol information of the target VM’s kernel. They can be read at startup and are used from that point to map static kernel variable- or function-names to memory addresses. As this work concentrates on kernel monitoring, this mapping information is indispensable. The memory addresses of the symbol file are of course pseudo-physical as they are valid inside the virtual machine’s kernel.

Static kernel variables are the simplest case, as they always have the same position. Dynamic memory is more complicated. The paging mechanism handling the dynamic memory for the kernel and user space applications is using the page tables to save the mapping from linear application addresses to pseudo-physical memory addresses. So, to find a variable in the dynamic memory, the module has to find the page tables and traverse them.

Another problem with dynamic memory is the fact that it is not only dynamically distributed in the pseudo-physical memory, but also may be dynamically swapped in and out, as decided by the paging unit. Hence, the desired page could just be not present in the memory at the time of the access.

One advantage of accessing the memory from a layer below the kernel is the unlimited access to all memory regions. Even that of interrupt handlers or other otherwise critical areas inside the kernel.

As seen, challenges are swapped out pages of dynamic memory, the page table traversal, the required symbol information for the kernel and the access to user space applications.
3.4 Distributed Setting with Time Synchronization

Depending on the requirements for use case class three, it is possible that narrow time synchronization is needed. The monitoring result of a distributed setting could be falsified by divergences in the system timers. Events may appear out of order, when ordered by their system-timestamp. Timeouts would occur, if one machine has to wait for a packet from another VM currently paused or under analysis.

The former work of Florian Schmidt [60] approaches this problem and serves as the base reference for this section. It introduces a synchronization server, which communicates with all participating VMs and allocates runtime slices to them. Only when all VMs acknowledged the finalization of their former time slice to the server, the next round of slices is allocated to the VMs. So all participants wait for their slowest member and synchronization is warranted. The need to always stop all VMs at the same time is also eliminated, as stopping one VM is sufficient with synchronization and results in almost the same scenario.

For the timeouts to not occur, it is also needed to simulate a fake wallclock time for all VMs. The participants must not notice that any virtual machine was paused during the process. The fake wallclock time is of course ideally the same on all VMs.

Figure 3.5 shows the buildup used for the synchronization. The synchronization server can be run on any computer accessible in the local network. The synchronization clients are kernel modules - run in the privileged domain on any participating physical machine. The clients could also be implemented and used as user space applications for the privileged domains, but that would involve considerably more context switches and hence more overhead for the synchronization procedure.

To achieve time synchronization is not an easy task. Besides the requirement of an additional synchronization server and client, the hypervisor itself and partly even the kernel of the target VMs have to be patched for this purpose. We are not going to simplify these complex tasks, but as this exceeds the scope of this work and is merely a requirement for a use case setting, the efficiency is assumed preexisting from here on.

One arising drawback - in comparison to the non-distributed use cases - is the additional setup work, as the synchronization facilities have to be installed and managed. Furthermore, the time slice synchronization leads to more performance and traffic overhead. The size of the time slice affects the overhead negatively proportional. Smaller time slices involve bigger overhead.
Figure 3.5 Time Synchronization Buildup
4

Implementation

The design for the toolset allows different combinations of front- and backends intended for different architectures - in terms of operating system of the host, hypervisor used, virtual machine type and the operating system of the target VM. We had to decide on one combination for the proof of concept implementation. Further versions of front- and backend can be implemented as future work and would broaden the deployment possibilities of the designed tool.

4.1 Implementation of Front- and Backend

The monitoring server (backend) is implemented in GNU C++ (Version 4.3.2) on a Ubuntu Linux 8.04 LTS system in combination with Xen. C++ was chosen as the programming language because it is free, powerful, object oriented, close to hardware and much related software like the Linux Kernel, Xen and its libraries are written in C or C++, which mostly enables direct library access from C++. Used libraries are for example: log4cxx [10] for logging purposes, libconfig [43] to read the configuration files and Apache Thrift [27] for network remote procedure calls (RPC). Apache Thrift is introduced in more detail in Section 4.4.

Xen was chosen as the virtualization software mainly because of being open source and hence modifiable and extensible.

The monitoring frontend is implemented as a Python script object. Python was chosen, because it is a straightforward and clear scripting language already known to a broad range of developers. All monitoring examples are hence Python scripts which include and use the monitoring object. The developer using the tool has full control of the monitoring procedure and targets as he is free to script any possible sequence in Python.

An implementation overview is shown in Figure 4.1.
Figure 4.1 Implementation Overview

The Python object is an RPC client, generated by the Apache Thrift framework. Thrift uses a generic interface definition of the RPC server as input to generate server and client stubs in different languages. The stubs encapsulate all needed network code. All relevant functions of the monitoring server are provided to the Python frontend via this RPC mechanism.

On the other side, the generated RPC server stub (also denoted “Apache Thrift Object” in the Figure) is included in the C++ monitoring server. The server must run in a privileged domain on the same host as the target virtual machine. It is configured using a configuration file written with common libconfig syntax [43]. An example configuration file can be found in the Appendix A.4.

The unstripped kernel image of the target virtual machine has to be accessible to the server. The functional modules of the server are organized as classes as can be seen in Figure 4.2. The server stub is being initialized at the beginning of the program - together with all other backend modules - after reading the configuration file. The class representing the server stub is called `ServerDispatcher`. All incoming requests are dispatched by it to the respective classes and the result is send back as answer.

The virtual machine control functionality is implemented by the `VirtualMachineControl` class. It reads the VM name from the configuration file and uses the `xm` console tool to find out its ID and running state. The same tool is also used to forward the pause, unpause and save-state commands.

The memory handling module denoted “GDB” is using three sub-elements (GDBserver-Xen, GDB and LibMiGDB) to read and write to the target virtual machine’s memory. Each of them is described in detail in the following sections. The functionality is split in two classes:
Figure 4.2 Implementation Overview - GDB + GDBserver-Xen

GdbserverControl is used to spawn a GDBserver-Xen process and reroute its input and output. GDBserver-Xen is a modified version of the GDBserver. It is part of the Xen project and instead of attaching to a running process, it can attach to any running domain on the host. For more details on GDBserver-Xen, goto Section 4.2.

GdbControl is using the LibMiGDB library to create and control a GDB process. GDB is commanded to load the kernel symbol information and to connect to the running GDBserver-Xen. Section 4.3 presents more details on LibMiGDB.

4.1.1 Monitoring Server Interface and Linking Scheme

The interface definition for communication between front- and backend is presented in Listing 4.1.

The calls getVariable and setVariable are used to read and set variables contained in the virtual machine’s memory. The breakpoints for specific functions or code-lines can be managed with setBreakpoint, delBreakpoint and removeBreakpoints. To continue the VM and/or wait for a breakpoint to occur, the call wait is used. The remaining four are management calls to pause or unpause the VM, save the current state into a file (saveState) or terminate the monitoring server at the end of a session (stopServer).

All calls are dispatched in the backend’s network module. Table 4.1 shows the linking scheme for the ServerDispatcher processing these calls.
string getVariable (1:string varName),
bool setVariable (1:string varName, 2:string varValue),

i64 setBreakpoint (1:string bp) //return breakpointID,
bool delBreakpoint (1:i64 bpID),
bool removeBreakpoints (),

i64 wait(), //return breakpoint ID

bool pause (),
bool unpause(),

bool saveState (1:string filepath),

oneway void stopServer()

Listing 4.1 Monitoring Server Interface Definition

<table>
<thead>
<tr>
<th>Method</th>
<th>GdbControl -&gt; LibMiGDB -&gt; GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>getVariable</td>
<td></td>
</tr>
<tr>
<td>setVariable</td>
<td></td>
</tr>
<tr>
<td>setBreakpoint</td>
<td></td>
</tr>
<tr>
<td>delBreakpoint</td>
<td></td>
</tr>
<tr>
<td>removeBreakpoints</td>
<td></td>
</tr>
<tr>
<td>wait</td>
<td></td>
</tr>
<tr>
<td>pause</td>
<td></td>
</tr>
<tr>
<td>unpause</td>
<td></td>
</tr>
<tr>
<td>saveState</td>
<td></td>
</tr>
<tr>
<td>stopServer</td>
<td>GdbControl, VMControl, GdbserverControl, ServerDispatcher</td>
</tr>
</tbody>
</table>

Table 4.1 ServerDispatcher Linking Scheme

4.2 GDBserver-Xen

The modified GDBserver is provided by the Xen project and can be build with the Xen sources. It must run in a privileged domain and is utilizing Xen hypercalls to access the target virtual machines memory and state. To control the GDBserver-Xen, an unmodified GDB can attach to it via the remote debugging feature.

It uses the xc_prtrace interface from the libxenctrl library - also part of the Xen project - to access the VM’s processor states and memory contents [40].

The internal details of this process can be seen in the following example - in which processor register are read [40].

Figure 4.3 shows how GDB’s request to get and set processor registers is implemented in the libxenctrl library by xc_prtrace(). Listed are multiple call options for the function. The xc_prtrace() function itself uses fetch_regs() to fetch virtual processor registers and calls the xc_vcpu_setcontext() function to set them. The dom0_op(DOM0_GETVCPUCONTEXT) hypercall to get the context of the guest
4.2. GDBserver-Xen

VCPU is performed by `fetch_regs()`. The hypercall is accomplished by making an `IOCTL_PRIVCMD_HYPERCALL` I/O-control on the `/proc/xen/privcmd` Dom0 xen device.

These implementation details are all encapsulated in GDBserver-Xen. The library `libxenctrl` also detects the current paging mode of the VM: real mode or protected mode, paging enabled or disabled, physical address extension (PAE) enabled or disabled and 32- or 64-bit.

It further does the required page table traversal when paging is enabled and dynamic memory is accessed.

The memory access to the target VM is realized by mapping the VM's memory into the address space of the GDBserver-Xen process.

4.2.1 Breakpoints

When a breakpoint is set by GDB, GDBserver-Xen places an `int3` instruction at the desired location in the code area. When the CPU executes the `int3` instruction, it raises an exception which results in a VMexit into the hypervisor. The hypervisor reacts on this exception depending on the exit reason. A breakpoint exception causes the hypervisor to `pause` the domain, which is noticed by GDBserver-Xen. It reads the virtual processor register states and passes them on to GDB. GDB can then determine where the guest was paused by looking at the `eip/rip` registers. A following `continue` from the GDB user causes GDBserver-Xen to `unpause` the domain and wait till it gets `paused` again.
### 4. Implementation

#### 4.2 Limitations of GDBserver-Xen

As basic element of our implementation, limitations of GDBserver-Xen are also
limitations of the current monitoring toolset prototype.

Only software breakpoints with the help of the `int3` instruction are available. Hardware supported breakpoints and watchpoints are not supported, as the debugged processor is only virtual. Single stepping of processes is not available, because the use of the virtual interrupt `int1` is not yet implemented. It could be emulated by GDBserver-Xen by inserting an `int3` instruction in every code line, but that is also not implemented in the current version.

A challenge with using GDBserver-Xen is that it is fairly undocumented and still in development stage. During the evaluation multiple hypervisor panics were encountered, when the GDBserver-Xen was shutdown uncleanly without deleting all existing breakpoints first.

#### 4.3 LibMiGDB - GDB/Machine Interface Library

LibMiGDB is a GNU DeBugger/Machine Interface (GDB/MI) library for C and C++. It implements the GDB/MI protocol as a library, so that a GDB frontend can be developed without writing the dialog with GDB.

The Machine Interface (MI) is used instead of the Command Line Interface (CLI) because it is explicitly intended for programs. An extract of an example communication dialog in the appendix A.2 shows the differences of MI and CLI and why MI is better parseable by programs.

The LibMiGDB library is heavily used in the `GdbControl` class to control the GDB process. Table 4.2 shows some used methods with their corresponding class method and GDB/MI command as example.

Further documentation to GDB/MI and LibMiGDB can found in the manual [12] and the documentation of the library itself [1].

#### 4.4 Apache Thrift

Thrift is a software framework for the development of scalable cross-language services. It allows the developer to define data types and service interfaces in a defi-
4.5 Python Monitoring Script-Interface

The definition file, which is compiled and used to generate code. The generated code is to be used to easily build RPC clients and servers that communicate smoothly across programming languages.

The main part of the interface definition language (IDL) file for our implementation was already shown in Section 4.1.1.

Apache Thrift can currently generate RPC client stubs for the following languages: C++, C#, Cocoa, Erlang, Haskell, Java, OCaml, Perl, PHP, Python, Ruby, Smalltalk. It supports basic data-types like for example booleans, strings and integers of different bit length. Supported are also special types like a sequence of unencoded bytes (called binary), structs - containing uniquely named fields - and three thrift containers types, which can be mapped to the list, set and map types in other programming languages.

The calls to the server can be asynchronous if they have a void return type. They must be synchronous otherwise. The toolset uses only synchronous RPC. All functions defined for the service can throw standard or user-defined exceptions on errors.

The transport layer implemented by Apache Thrift is totally hidden from the developer using it. Thrift generates a common interface for bidirectional raw data transport in every supported language. It leaves the developer with the choice of using, for example: TCP stream sockets, raw memory data, or files on disk as communication medium. The same applies to the protocol used to encode the data for the transport. It is immaterial to the developer writing the application code and can be chosen during initialization of the network stubs. The toolset uses the TBinaryProtocol in Thrift for data encapsulation as it is suggested in the Thrift tutorials and supposed to be space-efficient [62].

4.5 Python Monitoring Script-Interface

The monitoring frontend we implemented for the monitoring server backend is a Python script using the RPC object. The RPC client stub generated by Apache Thrift with the help of the IDL file needs to be included and initialized before usage (Listing 4.2).

After that, all provided methods can be called through the client object and replies from the server are saved as return value. At the end of the session, the transport just needs to be closed by transport.close(). Example monitoring scripts can be found in the evaluation Chapter 5.

It is worth mentioning that local variables of functions - called during the execution - are only accessible straightforward as long as the function is being executed and hence present in the memory. Therefore it is generally a good idea to set a breakpoint somewhere in the function and to read the local variable when the breakpoint is hit.

Additionally, when the monitoring server is in waiting state for a breakpoint that does not occur, it is possible to trick it into a break by pausing the virtual machine. Though the return value of the wait command will be invalid, the execution of the script can at least continue.
MI example:

```python
#Include
from thrift.transport import TTransport
from thrift.transport import TSocket
from thrift.transport import THttpClient
from thrift.protocol import TBinaryProtocol
import Introspection

#Config
host = 'localhost'
port = 10099

#Network Init
socket = TSocket.TSocket(host, port)
transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Introspection.Client(protocol)
transport.open()
```

**Listing 4.2** Thrift Python Init

**Interim Résumé**

Up to this point, we developed the design for a flexible monitoring toolset for distributed protocol implementations. It consists of a scriptable frontend, which is used to define what to monitor, and a remotely accessible monitoring backend server, which provides the means to realize that.

The design does not depend on specific architectures. Thus, we had to decide on one for the implementation of the proof of concept prototype. The frontend is implemented in Python and the monitoring server is build for a 64-bit Xen in combination with Linux and targets kernel debugging as main object.

It is evaluated on different test cases for protocol monitoring and a distributed buildup. The settings and results are presented in the next chapter.
5

Evaluation

5.1 Goal and Test-Setup

The goal of this evaluation is to verify the functionality of the implemented proof of concept prototype. This chapter describes the test cases run and their results. Test cases from every use case class described in the design chapter are chosen.

As our implementation has the task to monitor virtual machine kernels, this is also the main focus for the evaluation.

In the first part, every basic feature will be validated individually. We constructed straightforward test cases for this purpose. After that, a micro-benchmark will evaluate the timings for the features (Subsection 5.2.4).

The second part evaluates more complex cases of protocol monitoring. Test Case 5 and 6 hook into the kernel TCP network stack and analyze incoming packets.

The last part has the goal to test the prototype in a distributed setting. Hence, Test Cases 7 and 8 need more than one physical machine, and are processed in a different buildup (Test-Environment B). It consists of two equal computers, connected over a Gigabit LAN. The configuration of them is shown in Table 5.2.

All non-distributed cases are tested in Test-Environment A. It consists of only one computer. The configuration is shown in Table 5.1.

Software Buildup of the Test Machines

The virtual machines are using image files - hosted in Dom0 - for their virtual harddisks. 512MB of memory are allocated to each VM. The monitoring server is always running in Dom0 with superuser rights. GDB and GDBserver-Xen are spawned by the server with the same rights. They communicated over an arbitrary non-reserved local TCP port. We used one network port for the single VM tests,
5. Evaluation

<table>
<thead>
<tr>
<th>AMD Athlon(tm) 64 X2 Dual Core Processor 4800+</th>
</tr>
</thead>
<tbody>
<tr>
<td>3528704 kB of main memory</td>
</tr>
<tr>
<td>160 GB harddisk with one Linux partition</td>
</tr>
<tr>
<td>virtual machines use harddisk-image-files</td>
</tr>
<tr>
<td>Ubuntu Linux 8.04 LTS (Dom0 + DomU)</td>
</tr>
<tr>
<td>Xen Version 3.3.1</td>
</tr>
<tr>
<td>Dom0 kernel: 2.6.18.8-xen #1 SMP</td>
</tr>
<tr>
<td>PVM-DomU kernel: 2.6.18.8-xen #1 SMP</td>
</tr>
</tbody>
</table>

**Table 5.1** Test-Environment A

<table>
<thead>
<tr>
<th>Intel(R) Core(TM) 2 Quad CPU Q9650 @ 3.00 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>6977536 kB of main memory</td>
</tr>
<tr>
<td>ext3 Linux partitions</td>
</tr>
<tr>
<td>virtual machines use harddisk-image-files</td>
</tr>
<tr>
<td>Debian 5.0.2 (Dom0 + DomU)</td>
</tr>
<tr>
<td>Xen Version 3.3.1 (patched for time synchronization)</td>
</tr>
<tr>
<td>Xen Scheduler used: <em>sync SEDF</em> (<em>credit</em> is default)</td>
</tr>
<tr>
<td>Sync SEDF is a modified version of the SEDF scheduler and was originally developed by Florian Schmidt [60].</td>
</tr>
<tr>
<td>Dom0 kernel: 2.6.18.8-xen #1 SMP</td>
</tr>
<tr>
<td>PVM-DomU kernel: 2.6.18.8-xen #1 SMP</td>
</tr>
</tbody>
</table>

**Table 5.2** Test-Environment B

and two ports for the tests with two servers running on the same machine (Test Case 7 and 8).

The monitoring server itself listened on one TCP port in the single VM tests, and two different ports were used for the two servers in the other cases.

### 5.2 Basic Tests and Microbenchmarks

As shown in the interface definition in Listing 4.1, the basic functionality of the monitoring toolset is to: read a variable, write a variable, set and remove breakpoints, wait for a breakpoint, and pause or unpause the virtual machine. The `saveState` call is only implemented for future use and not part of this evaluation. `stopServer` is a management function to shutdown the server, hence there is no need to benchmark it.

#### 5.2.1 Test Case 1 - Basic Read Access and Control

**Counting Kernel Module**

To evaluate the basic functions and benchmark them, a dummy kernel module was developed and compiled into the kernel.

Its only task is to define a global integer variable, increment it every five seconds by calling the `test_func` function and echo out the current value on the kernel console.
The function uses two local variables. One is used to increment the global counter variable and the other is simply echoed out. The local variables are only integrated to test read and write accesses to them with the toolset.

Of course, we could have tested the basics with other target variables in the kernel, but the contents of none of them are so easily verified like that of a dummy variable, which is echoed out on the kernel console every five seconds.

**Results of the Test Case**

The goal of the first test case is to monitor a single variable, inside the kernel of the running virtual machine. We chose a variable inside our test module to easily compare the read contents with the output on the kernel console. At the end of Test Case 1, the pause/unpause functionality is validated.

The buildup for the first test case can be seen in Figure 5.1. It is also used for the other basic Test Cases 2, 3 and 4. We are using Test-Environment A with one paravirtualized virtual machine. The VM is up and running and the monitoring server started and attached to the VM without errors.

The Python test script is also started in Dom0 and the main part of it can be seen as cutout in Listing 5.1. The complete script can be found in appendix A.1.

The script loops 9 times through the `getVariable` routine, printing out the result, with a 2.5 seconds pause between each lookup. At the end, the virtual machine is paused for 5 seconds.

Test Case 1 already covers all possible examples from the unconditional use case class one. The conditional part is covered in Test Case 2.
... # Introspection Part
for i in range(1, 10):
    pp.pprint("getVariable:") + client.getVariable("acid_test_c",));
    time.sleep(2.5);

print ("pause now");
client.pause();
time.sleep(5);

print ("unpause now");
client.unpause();
...

Listing 5.1 Test Case 1 - Python script (cutout)

Python output:                      Console output:
result:                          ACID Counter 67924, 666
'getVariable:67924'             ACID Counter 67925, 666
'getVariable:67924'             ACID Counter 67926, 666
'getVariable:67924'             ACID Counter 67927, 666
'getVariable:67925'             ACID Counter 67928, 666
'getVariable:67926'             ACID Counter 67929, 666
'getVariable:67926'             ACID Counter 67930, 666
'getVariable:67927'             ACID Counter 67931, 666
'getVariable:67927'             ACID Counter 67932, 666
'getVariable:67928'             ACID Counter 67933, 666

Figure 5.2 Test Case 1 - Output showing corresponding variable contents

The results in Figure 5.2 show the corresponding output from the Python script and the kernel console, which verifies that the tool is working correctly. The variable content matches the output on the kernel console. The status output of `xm list` displayed in Figure 5.3 shows the status change of the VM when it gets paused. The state of `vm01` goes from “b” to “p” and back again.

The direct access to the variable by its name indicates that the symbols are loaded and interpreted correctly by GDB and that GDBserver-Xen is able to access the memory and read its contents.

The timings for functions - used in this test case - can be seen in Subsection 5.2.4, where the results of the microbenchmarking are presented.

To test the functionality of the tool for hardware virtualized machines (HVM), we changed the first build-up a little by replacing the PVM with a HVM. As the HVM is not required to use a Xen-patched kernel, the current unmodified version 2.6.26.6 is used. The installed operating system in the HVM is Ubuntu 8.04.3 LTS and 512 MB of main memory are allocated to it.

As the kernel is an unpatched version, the dummy kernel module cannot be chosen as target. The global variable used for the test is `idle_timer` and no problems were encountered reading it.
5.2. Basic Tests and Microbenchmarks

5.2.2 Test Case 2 - Conditional writing

The goal of Test Case 2 is to cover the conditional cases from use case class one, which were not yet tested in Test Case 1. The target is the same variable in the dummy kernel module, which this time is going to be changed on certain conditions.

The buildup is exactly the same as in Test Case 1 (Figure 5.1) and the according Python script cutout can be found in Listing 5.2. It basically tests the target variable every 0.5 seconds and resets it to 50000 when it raises above 50010.

The results in Figure 5.4 show that it is working as expected and that the tool is able to react on certain conditions inside the VM. When the first value above 50010 is read, the variable is reset. As the variable changes only every 5 seconds, but is probed every 0.5 seconds, the results show repeating values until the change occurs.

```python
for i in range(1, 100):
    testvar = client.getVariable("acid_test_c");
    pp.pprint("getVariable:" + testvar);
    testvar = int (testvar); #make integer
    if testvar > 50010 :
        testvar = 50000;
        result = client.setVariable("acid_test_c", str(testvar));
    time.sleep(0.5);
```

Listing 5.2 Test Case 2 - Python script (cutout)

5.2.3 Test Case 3 - Reactive Monitoring and Local Variables

To achieve reactive monitoring, needed to cover use case class two, Test Case 3 uses breakpoints to install hooks in the kernel. Breakpoints can be set anywhere in the
kernel code and so every event - represented by its according event handler code line - can be listened to.

To use breakpoints inside a function, enables us to read and write to local variables. The test target is again the dummy module. This time, a code line in the \textit{test\_func} function is selected as breakpoint and local variables are read and written.

The Python script is presented in Listing 5.3 and shows that the implementation of the procedure in Python is straightforward. It sets a breakpoint and goes in a loop which waits for the breakpoint. When it is hit, the script continues and reads/writes the local variables. At the end the breakpoint is removed and the execution continued.

The buildup configuration for the test is still the same as in Test Case 1 and 2 shown in Figure 5.1.

The result (Figure 5.5) can be verified by analyzing the kernel console log, as the second variable (\textit{acid\_test\_b}) is part of the output to the console and the first is used to temporary hold the global variable, which is also printed on the console. So the counter IDs, which were monitored and modified, have to show a changed second variable in the console output.

It shows that the tool is able to hook into certain events in the VM and modify their behavior during runtime.

```
breakpoint = client.setBreakpoint("acid_test.c:76",) # in counter module
pp.pprint("breakpoint ID:" + str(breakpoint))
for i in range(1, 5):
    pp.pprint("wait:" + str(client.wait()))
    pp.pprint("getVariable:" + client.getVariable("acid_test_a"))
    pp.pprint("setVariable:" + str(client.setVariable("acid_test_b","777")))

pp.pprint("delbreakpoint:" + str(client.delBreakpoint(breakpoint,)))
pp.pprint("wait:" + str(client.wait()))
```

\textbf{Listing 5.3} Test Case 3 - Python script (cutout)

During the evaluation of the test case, the problem of missing local variables was encountered. Meant are variables that are removed by the compiler during optimization steps. One solution to cope with this problem is to disable the compiler optimization for the debugged module.


5.2. Basic Tests and Microbenchmarks

Python output:

'breakpoint ID:1'
'wait:1'
'getVariable:68046'
'setVariable:True'
'wait:1'
'getVariable:68047'
'setVariable:True'
'wait:1'
'getVariable:68048'
'setVariable:True'
'wait:1'
'getVariable:68049'
'setVariable:True'
'delbreakpoint:True'
'wait:0'

Console output:

ACID Counter 68044, 666
ACID Counter 68045, 666
ACID Counter 68046, 777
ACID Counter 68047, 777
ACID Counter 68048, 777
ACID Counter 68049, 777
ACID Counter 68050, 666
ACID Counter 68051, 666

Figure 5.5 Test Case 3 - Output showing local variables being read and written

<table>
<thead>
<tr>
<th>Function</th>
<th>average Timing (in micros.)</th>
<th>standard deviation (in micros.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getVariable</td>
<td>2553.67</td>
<td>159.43</td>
</tr>
<tr>
<td>setVariable</td>
<td>6310.32</td>
<td>4753.92</td>
</tr>
<tr>
<td>getLocalVariable</td>
<td>1487.64</td>
<td>409.92</td>
</tr>
<tr>
<td>setLocalVariable</td>
<td>2762.33</td>
<td>1659.03</td>
</tr>
<tr>
<td>pause</td>
<td>183290.71</td>
<td>2729.13</td>
</tr>
<tr>
<td>unpause</td>
<td>187581.47</td>
<td>76406.57</td>
</tr>
<tr>
<td>setBreakpoint</td>
<td>3349.99</td>
<td>529.33</td>
</tr>
<tr>
<td>removeBreakpoints</td>
<td>609.30</td>
<td>85.20</td>
</tr>
</tbody>
</table>

Table 5.3 Microbenchmarks for every basic function

5.2.4 Test Case 4 - Microbenchmarks

Test Case 4 is a microbenchmarking script to determine the timing for each function in the script. The basic buildup is that of Test-Environment A.

To measure the execution time of each of the functions, the Python module timeit was used.

The benchmark-script first tests getVariable and setVariable for global variables, followed by the other available functions. Each is tested a 1000 times, to calculate an average time.

The benchmarking results show that the average timings in Figure 5.3 should be taken more as an estimate, as the measured values vary from run to run.

We attribute these fluctuations to CPU usage impacts from other programs running in Dom0 or DomU on the host machine or other non-deterministic influences, e.g. of the hypervisor or operating system scheduler. Even so the surrounding was not changed between multiple test runs, the values still varied strongly.
5.3 Evaluating Protocol-Monitoring

After verifying the functionality on the dummy kernel module, the next step in the evaluation is to test the toolset on a real-live protocol. The target of choice is the TCP protocol, as it is widely used nowadays.

5.3.1 Test Case 5 - Protocol Monitoring

The goal of the fifth test case is to validate the correctness of read values from the protocol stack with the parallel operation of Wireshark. The network analysis tool Wireshark [36] can be used to capture and disassemble packets on local network devices.

The Python script reads the congestion window size - used at the TCP socket inside the kernel - when packets arrive. Wireshark captures the packets at the same time. After the test run, the results of the frontend are verified with the values read out of the header of the packets captured by Wireshark. In a working TCP network stack, these values should perfectly match.

The buildup is slightly more complex, compared to the former test cases as shown in Figure 5.6. Still the basis of the setting is Test-Environment A. The virtual machine and the monitoring server are up and running. In this test case traffic has to be generated on the network. So, additionally to the user running the Python script, we have to log in a user into the VM over the Xen console. This user has to generate traffic during the test-session by using, for example the lynx [2] browser or the network pipe tool netcat [3]. In parallel to monitoring the packets with the
5.3. Evaluating Protocol-Monitoring

![WindowSize per Time](image)

**Figure 5.7** Test Case 5 - Comparing the window size readings

Python test script, *Wireshark* has to be set up for recording the same packets, by capturing the traffic on the local network devices.

The test-hook (breakpoint) is set in the `tcp_ack_update_window` kernel-function that is used to acknowledge the congestion window size updates for incoming TCP sessions (TCP-ACK-Packets). *Wireshark* is set to filter out only these packets from our Test-VM, as any other noise on the line can be ignored for this test.

When the breakpoint is hit, multiple selected values are captured: the destination IP of the socket, the source and destination port, the congestion window size, window scale and at least the `syn` status of the current packet. These are used to identify the packets and calculate the effective congestion window size for the current packet.

One minor issue in interpreting the data, read by the Python script, is that it first has to be converted into a human readable format. The IP address in the TCP-socket structure for example is encoded as 32-bit long in network byte order and has to be converted into the “dotted 4-tuple” format.

The values from the two captures are shown in the plot in Figure 5.7. It displays both results, mapping the congestion window size captured to the timestamp of the packet. The two plots are identical as expected.

The example exhibits how the toolset can be used to monitor protocol implementations inside the kernel and read real live values from the protocol stack. It uses a hook inside the network stack to react on the event of incoming messages.
5. Evaluation

packet: 0
'wait: 8'
'getVariable: saddr: 137.226.59.194'
'getVariable: daddr: 82.98.82.50'
'dport: 80'
'sport: 45920'
'getVariable:*iocb:{ki_run_list = {next = 0xffff88001eb977c8, prev = 0xffff88001eb977c8},
    ki_flags = 0, ki_users = 1, ki_key = 4294967295, ki_filp = 0xffff88001dc19380,
    ki_ctx = 0x0, ki_cancel = 0, ki_retry = 0, ki_dtor = 0, ...}
'getVariable:*sk:{__sk_common = {skc_family = 2, skc_state = 8 '\b', skc_reuse = 0 '\0',
    skc_bound_dev_if = 0, skc_node = {next = 0x0, pprev = 0xffff8800074fa418}, ...
'getVariable:*msg:{msg_name = 0x0, msg_namelen = 0, msg_iov = 0xffff88001a001da8, msg_iovlen = 1,
    msg_control = 0x0, msg_controllen = 0, msg_flags = 0}'}

Figure 5.8 Test Case 6 - Output showing detail-information on selected packets

5.3.2 Test Case 6 - Capturing Network Packets

The second protocol monitoring test case has the goal to capture information on selected incoming packets. The test script is configured to monitor HTTP traffic (Port 80 is filtered) with a selected host (IP).

Test-Environment A is used again. Virtual machine and monitoring server are running and a user in the VM is generating HTTP traffic with lynx.

The breakpoint is set at the beginning of the tcp_recvmsg kernel-function, which is called on receiving packets. The script then reads the packet information needed to filter it. In this example: destination IP address and destination port. As before, the information first have to be converted into the common human readable format for the comparison.

When the filter criteria match, further information on the packet are read by the Python script. The output in Figure 5.8 shows the dumped values for a filtered packet. The selected three structures - used in the kernel-function - contain deep information on the packet. The example output is of course cut off.

The test case shows how the toolset can be scripted to monitor network traffic of the virtual machine. It can further be used to scan for malicious content and instantly freeze the VM on arrival, before any harm can be caused. Of course other countermeasures can also be undertaken.

5.4 Distributed Protocol-Monitoring and Virtual Time

The last two test cases target the evaluation of distributed monitoring and should thereby cover the use case class three. As explained in the design Chapter 3.4, this requires advanced preparations, to synchronize the time and the execution of the test run, between all participating virtual machines.

Both test cases are realized in Test-Environment B, consisting of two physical hosts. Furthermore, each machine hosts two virtual machines - in addition to Dom0 - which are all part of the distributed setting. An overview can be seen in Figure 5.9.

The synchronization server, as well as the according client kernel module, from the work of Florian Schmidt [60] have undergone some updates and adaptations by Martin Lindner, to work properly on the current version of the Xen Hypervisor.
5.4. Distributed Protocol-Monitoring and Virtual Time

Test-Environment B
Distributed Monitoring Setting

Host Machine A
Privileged Domain
Python Frontend
Monitoring Server
Xen Hypervisor

Host Machine B
Privileged Domain
Unprivileged Domain
Kernel
Monitoring Server
Xen Hypervisor

Unprivileged Domain
Kernel
Sync. Client
Gigabit LAN

Figure 5.9 Monitoring buildup for the Distributed Setting

Test-Environment B
Distributed Monitoring Setting

Host Machine A
Privileged Domain
Synchronization Server
Kernel
Sync. Client
Xen Hypervisor

Host Machine B
Privileged Domain
Unprivileged Domain
Kernel
Sync. Client
Xen Hypervisor

Unprivileged Domain
Kernel
Gigabit LAN

Figure 5.10 Synchronization buildup for the Distributed Setting
The buildup needed for the synchronization is shown in Figure 5.10. The kernel module was used in the distributed setup on each of the two host machines. It runs in Dom0 and is synchronizing both DomUs on the host. The server, to which both synchronization clients connect, runs only on host A.

To initiate the synchronization, the VMs have to run, the synchronization server started and the client kernel module loaded, with the appropriate parameters (server address, virtual machine ID, etc.), on both hosts. The exact parameters used and available are shown in the Appendix A.3.

### 5.4.1 Test Case 7 - Global Snapshot

Test Case 7 is realized in the environment described above. The test script initializes the network connection to all four monitoring servers, instead of only one till now. In the main part, it is pausing the first virtual machine, which leads to a paused state in all synchronized VMs, and reads a randomly selected global variable (the `pidmap_array` - Process ID Map) from all VMs. It can thereby create a global snapshot for the distribution of this variable in the entire scenario.

To see why all other virtual machines have to wait - when one gets paused - see the time slice allocation scheme of the synchronization server in Figure 5.11.

![Figure 5.11 Allocating time slices to the virtual machines](image)

The resulting output of the script can be seen in Figure 5.12. In each round shown, a global snapshot is generated. We chose only one variable for the example snapshot, but any number of values - the developer is interested in - can be read in this state for every virtual machine. The content of the `pidmap_array` variable for every VM in shown in the `getVariable` lines in a shortened form.

This first distributed test case shows that the monitoring toolset works in combination with time synchronized virtual machines. The synchronization mechanism itself works too, as expected. Also, the network feature of the tool, used by the frontend to communicate with multiple backends over the local area network, is also tested fully functional with this test case.

### 5.4.2 Test Case 8 - Distributed Breakpoints

The last test case tries to overcome the challenge of waiting for a breakpoint on all virtual machines simultaneously. As the `wait` function is synchronous and blocking,
the main program thread can only wait for one virtual machine at a time. The chosen approach to cope with this is multithreading. The buildup is the same as in the last test case.

The test script’s main thread spawns four new worker threads, right after the network initialization. With these it is possible to set breakpoints in and wait for all four participating VMs. When the breakpoint occurs, the normal state-analysis - by reading and writing to the variables of the VM - can follow. The output of the different threads is marked by the thread id in brackets at the beginning of the line.

As an additional possibility it is tested to “cross-read” variables of the other three VMs, when one hits a breakpoint.

Unfortunately, not all elements of this test case could be evaluated successfully with the first prototype. Although the main challenge of waiting simultaneously is overcome by using multiple threads, the analysis after a breakpoint is limited to the virtual machine, which hit the breakpoint. The first implementation of the monitoring server containing one Apache Thrift RPC server does not allow additional requests to be handled when it is working on a synchronous call like wait. With it, there is no possibility to interrupt the other three waiting VMs, to make intermediate accesses with the Python client (“cross-reading”). The problem is demonstrated in a sequence diagram of two example virtual machines in Figure 5.14.

To cope with this problem, the prototype was extended with a second identical RPC server in a parallel thread. So even when the first RPC server is blocked waiting, the spare RPC server can process a pause command to simulate a breakpoint, which brings back the main RPC server. The solution is also demonstrated in a sequence diagram (Figure 5.15). When the VM gets paused, the GDBserver-Xen assumes it hit a breakpoint and returns control back to GDB, respective the Python client. After that the frontend can “cross-read” the virtual machine to generate the global state for the target variables and the VM can be resumed.

As future work, the RPC library could be changed to allow multiple synchronous calls natively without a second RPC server running.
The output shown in Figure 5.13 is the shortened result of a test run using the extended prototype implementation of the monitoring backend. It can be seen how the four threads start, insert a breakpoint and wait. When thread number one hits a breakpoint by receiving a network packet with the given details, it simulates a breakpoint on the other virtual machines and generates a global view on the example variable `pidmap_array`. After that all VMs are continued and wait for a breakpoint again.

The last test case evaluated the functionality of the toolset even in a very complicated distributed solution with simultaneous waiting for breakpoints. Many other test cases containing even more complicated buildups and tasks for toolset are thinkable, but they would be more or less only a composite of the presented cases in this chapter.
Figure 5.14 Failing cross-reading attempt

Figure 5.15 Successful cross-reading attempt
5. Evaluation
Related Work

This chapter is presenting multiple related works, reaching from pieces we used in our toolset or approaches similar to ours, to the use of hypervisors for computer security and forensics. At the end, details on multiple kernel debuggers are given.

6.1 Hypervisor Monitoring

As mentioned in the design chapter, there exist other related works that use hypervisors like Xen to debug or monitor kernel implementations.

6.1.1 ProMoX and XenAccess Introspection

One of those works is that by the Distributed Systems Group of the RWTH Aachen University [66]. It uses XenAccess [49] to monitor selected kernel variables - exported as a symbol - with a tool running in Dom0.

The work resulted in a protocol monitoring framework called ProMoX. The goal of the framework is to assist the development process of complex, distributed kernel protocols. One type of network protocols that fall into this category are peer to peer protocols.

The examination of a protocol stack often requires the analysis of an implementation running on an actual system, not in a classic debugging environment. That is why ProMoX uses virtual machines to test the protocols in. They simulate a real live system and can be stopped and analyzed at any time.

The used XenAccess [50] is a library that allows the state inspection of running guest domains with the Xen hypervisor. It helps the developer of a monitoring tool by handling the address translations and page table traversals done inside the hypervisor that are needed to access the memory of the VM. These abstractions help a lot when developing monitoring tools.
The ProMoX framework is one of those. In comparison to our toolset, ProMoX has the drawback of less flexibility, as the target variables have to be exported as symbols. Such a change in the kernel source code requires a new kernel build run. Furthermore, a special kernel spying module has to be run inside the target VM that helps looking up memory addresses and offsets.

The need to patch the kernel makes it impossible to be used on closed source operating systems that only provide debugging symbols.

### 6.1.2 Evolution in Kernel Debugging

The work [40] from the Open Source Technology Center of the Intel Corporation is taking the same approach of debugging a virtual machine with the Xen hypervisor. Like the memory handling module of the monitoring server in this work, they are using the GDBserver-Xen to attach to a running guest domain and use GDB to debug its kernel at source level.

By using the same technologies at this point, the work is exposed to the same limitations and possibilities in respect to target compatibility and possible debug actions. Only that our toolset extends these, by using additional software modules like for example the VM handling module or the scriptable frontend.

The main limitations of their work are: No single stepping mode for the virtual CPU (VCPU), no scriptable frontend, hence no possibility to automate the debugging or monitoring task, and no distributed debugging.

A compatible target is every operating system virtualizable with Xen, which provides the required debugging symbol information.

Using only GDB as frontend without additional user interface layers has advantages, too. First, the overhead of using such layers is removed. Second, the user has a more interactive access to GDB and can react to certain situations more directly.

### 6.1.3 GDBSX

GDBSX is a kernel debugging tool presented in [55] by the Oracle Corporation. Like XenAccess and the GDBserver-Xen approaches, it uses the Xen hypervisor as platform to access a target VM’s memory for debugging purposes. Unlike the other approaches, GDBSX makes the hypercalls to the hypervisor directly, not using the libxc library. Furthermore, GDBSX is using a patched hypervisor extended with four new hypercalls and is supporting single stepping for VCPUs.

GDBSX provides a GDBserver stub, comparable to GDBserver-Xen, to which any debugger - using the GDB serial debugging protocol - can connect. It is limited to a 32-bit binary running on a 64-bit hypervisor only, but is able to attach to 64-bit and 32-bit target kernels.

Like GDBserver-Xen, GDBSX recognizes when the VM is paused and breaks into GDB [56].
The target compatibility is the intersection of that of GDB and Xen. Every system virtualizable with a 64-bit Xen hypervisor that GDB can attach to using the provided debugging symbols, is a valid target.

As with the former work in Subsection 6.1.2, GDB is the frontend of the toolset. Every additional interface layer provided by our tool is missing, and so are the benefits from them.

Because of being nearly a replacement for GDBserver-Xen, it is a good candidate for inclusion into the memory handling module of our toolset, too.

### 6.1.4 Xenprobes

The Xenprobes framework [54] transferred the Kprobes [45] approach of kernel debugging from a classic machine to virtualization. Kprobes provide a lightweight interface to automated dynamic kernel debugging. They allow the injection of breakpoints and simulated faults into the running kernel, without any recompilation of it.

Xenprobes realize the same idea using virtual machine introspection on a Xen hypervisor.

The advantages compared to Kprobes are the following: Xenprobes handle the complete work in user space (Kprobes run in kernel mode), they are operating system independent, can be put into any space inside the kernel and cannot be accidentally chained to loops.

Xenprobes uses probing samples that are written in C and have to be compiled. These probes are run in Dom0 in user mode, which makes them considerably easier to write than the Kprobes that have to run in kernel mode.

The implementation of Xenprobes uses `libxc` as library for memory accesses as well as the interrupt `int3` for breakpoints. The kernel symbol information are needed for the probes to run, just like for the other debuggers before.

One advantage compared to our tool is that the Xenprobes can handle a shutdown of the target virtual machine. They register a shutdown notice from the hypervisor and reinstall all previously installed hooks (breakpoints), when the system is back online.

Another one is the usage of the virtual `VIRQ DEBUGGER` interrupt, which is not implemented by GDBserver-Xen for example. It makes the polling for the `paused` state of the VM obsolete, as they get notified by the hypervisor. Therefore they have to register the interrupt in Dom0. When the hypervisor pauses the VM, because of a hit breakpoint, it sends the virtual interrupt and the probe can react immediately.

One minor drawback, in comparison to our toolset, is that the probes have to be statically compiled and written in C, while our monitoring frontend is a flexible Python script.

The lack of support for real distributed debugging is a major drawback, as it limits the developer to one physical machine and hence only a virtually distributed setting.

So are the missing means for a time-synchronization of the virtual machines, which is a required feature for protocol debugging to avoid the timeouts.
6.1.5 PDB - Pervasive Debugging with Xen

Another work [38], which shares the goal - to debug distributed applications or grids - with our toolset, is “PDB: Pervasive Debugging With Xen”. It uses a hypervisor to run multiple virtual machines on one physical and uses the elevated position of it to debug applications in all of them together.

The classic peer debugging approach, described in the next chapter, requires network communication to control and synchronize multiple local debuggers. As the network may delay or even reorder packets, this may falsify the timing of events in the analysis. As the PDB’s targets run all on one host, these network problems are removed.

A major difference to our toolset is that the pervasive debugger is located inside the hypervisor. It thereby has direct control over the virtual machine and its environment. All control and debug actions in our toolset have to use hypercalls from user space.

One substantial drawback is the lack of a network feature to connect multiple PDBs for one debugging session, which limits the PDB debugger to only one physical machine and their resources. This feature is planned for a future release.

The implementation includes an enhanced GDB, called “PDB Client” that communicates with the debugger backend in the hypervisor, using an extended GDB serial communication protocol. The protocol had to be extended with an addressing scheme for the different naming and security contexts that is needed to target the right virtual machine when sending GDB commands. Breakpoints are realized using the virtual \texttt{int3} interrupt, too.

One advantage compared to our tool is the support for watchpoints in the debugging session. They are realized using either hardware support or software mechanisms.

As in other related works using a GDB like frontend, it allows interactivity at the cost of the possibility to script and automate the session.

A planned feature for PDB was the utilization of virtual time and time roll-back techniques to synchronize all test VMs and use roll-back for these that are out of sync. This is at least comparable to the time-synchronization solution already used in our work. But as last publications [26] on this work are from 2005 it seems to be no longer under development.

6.1.6 Classic Distributed Debugging Approaches

To compare our work with the classic approach on debugging distributed applications, we will describe the common architectures and challenges.

The architecture of many current online debuggers for grid systems consists of a central console that coordinates several independent debuggers. They attach locally to their target applications.

One of the two main challenges is the reliance on the underlying network and the difficulty to synchronize the activities of coordinator and debuggers. That makes
it difficult to generate a consistent global view. Unfortunately the network reliance is one element shared with the distributed buildup for our debugging toolset. The synchronization difficulties on the other hand are resolved by the separate client-server-synchronizer solution we are using for all virtual machines. It ensures that all virtual machines, and hence all processes inside, are always in the same time slice. A global view can be generated straightforward by pausing one VM and reading the required information from all systems.

Their second main challenge is that the local debuggers operate in user space and are thereby limited to the debugging constraints of the local operating system. The visibility is generally restricted to the target process’s virtual address space and CPU state [38]. The ability to alter and analyze the environment of the process is very restricted. With full control and access of the operating system, our toolset has way more possibilities and thereby another significant advantage.

**Advanced Distributed Debugging Approaches**

There are also advanced approaches to debug distributed applications. They try to cope with the inability of cyclic debugging approaches to detect non-deterministic bugs reliably. The non-determinism in the distributed debugging setting is introduced by variances in network communication delays and scheduling differences.

The approaches can be classified into three basic types: Online monitoring, deterministic replay and declarative debugging. All types are shortly introduced together with an example implementation.

**Online Debugging with Modist and D3S**

The online debugging approaches mainly have the function to notify the developer of arising problems in the current state.

Erroneous situations are specified by distributed predicates written by the developer and continuously checked during the execution. Common errors like segmentation faults, deadlocks or infinite loops are also detected.

When an error occurs, the system reproduces the sequence of events that led to it. The error can thereby be manually reproduced and analyzed by the developer. The events are given timely relations using vector timestamps.

Implementations for this approach are D3S [44] and its successor MoDist [68]. Both are only able to debug userspace processes. This is because of the used technique: Interposition between operating system and application. No modification of the hardware, operating system or application are required.

In MoDist, all WinAPI functions are intercepted by the frontend. The event is then send to the backend via RPC and processed there. The backend collects all information and does the model checking and tests for global assertions. For comparison, our toolset interposes at the architecture level so that the operating system in the VM can be targeted.
Although MoDist cannot directly replay the external events to drive an example application, it can inject errors into the system and drive it directly into erroneous situations.

MoDist also provides a virtual clock for all hosts, which is controlled by the backend. The virtual clocks are used for MoDist’s internal tasks, for example to time when to inject bugs, and to provide a consistent view of the clock to all target systems. This is comparable to our time-synchronization solution.

The virtual clock is used by MoDist to speedup the test session where possible. It uses it for example to skip sleep periods by forwarding the clock.

Even though being a good approach to find bugs in distributed systems, it lacks several important features. It is not possible to “zoom” in on a bug, or to travel back in time to replay the same situation directly. Furthermore, it cannot be used to set breakpoints in the applications.

**Deterministic Execution Replay**

The approach called “deterministic execution replay” tries to fight the problematic non-determinism directly. It consists of two main phases.

First, the *record phase*. All information on the execution of the application, including all non-deterministic events are recorded and stored in a log file. The events are timestamped using Lamport clocks. A non-deterministic event is for example an incoming network message.

The second phase is called “replay phase” and consists of executing the application locally using the captured events from the log file.

Major drawbacks of this solution are the space requirements for the large event logs.

“Liblog” [35] is an implementation using this debugging approach. It is a user-space library intended to debug distributed C/C++ applications.

The technique used to capture the events is library interposition. Liblog is interjected between *libc* and the application. It depends on the main assumption that all action are processed through the target library. Furthermore, it assumes that the application execution itself is deterministic. This is especially problematic with applications using threads. Liblog copes with this by using a special scheduler in the operating system and controls it from user-space.

Liblog works without modifications in the application or operating system and no special hardware is necessary.

As frontend, liblog uses an extended version of GDB to debug the applications at source-level.

The approach also allows to debug mixed environments, in which only some applications are using liblog.
Declarative Debugging

The key idea of declarative debugging is to not move the captured logs from the nodes to a central location, but to process them in place. The debugging queries are sent directly to the nodes and processed there.

An implementation for the declarative debugging approach is “D3” [23].

The used queries have to be specified in declarative way using a formal specification language, for example NDLog.

The concept is to define “what to query” in the high-level queries, which are sent to the nodes. These are then converted into low-level commands, which tell the distributed query engine exactly “how to query” the logs.

D3 can use static and dynamic logs for computation. Static log information are for example the system configuration or offline data of the application. Dynamic logs contain online data from execution traces or resource utilization monitoring.

What to log on the nodes has to be taken care of by the developer. He also has to provide a log-adapter which describes how D3 has to read the logs.

Using D3, the developer is able to check invariants of distributed systems, but he is strongly limited to the supported query types of the system.

6.2 Computer Security and Intrusion Detection

The use of the elevated position of the hypervisor is not only beneficial for debugging purposes. Intrusion detection systems (IDS) are no longer directly vulnerable, when located in the safe context of another virtual machine, compared to the common position as a system service inside the target machine.

As the IDS has to have superuser rights inside the target machine, being a system service or even kernel application are the only ways on simple physical machines. The virtualization of the machine and the possibility to access the memory contents from outside, make the context change into a safer location workable.

Computer security systems take the approach of monitoring virtual machines to verify their integrity. They are able to detect and prevent incoming attacks on a whole different level than for example firewalls can do.

“This makes it possible for security tools — such as virus scanners and intrusion detection systems — to observe and respond to VM events from a ’safe’ location outside the monitored machine [46].”

6.2.1 A Virtual Machine Introspection Based Architecture for Intrusion Detection

The approach of Garfinkel and Rosenbaum [33] uses checksums of memory pages to verify the integrity of the running virtual machine. Furthermore, they use hooks
inside the hypervisor to validate requests from the VM before they are executed. The hooks are used to notify the event driven policy engine of the IDS when state changes occur.

The required knowledge of the operating system is provided by an OS interface library and the debugging information inside the kernel binary. The implementation prototype called “Livewire” is using a modified version of the Linux “crash dump” tool that is normally used to interpret the /dev/kmem device, to access the VM’s kernel memory. The hypervisor exports a /dev/kmem like device, containing the target’s memory in form of a flat file, to which Livewire can attach from the outside.

The hypervisor used in their implementation is a modified VMware workstation for Linux. It provides “the ability to interpose at the architecture interface of the monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events”. The ability to interpose at the hardware interface is also used to control the hardware access of the VM.

In addition to the abilities of a normal host based intrusion detection systems (HIDS), it is possible to detect compromises of the operating system itself. A HIDS is sometimes unable to do that, because the detection methods rely on the OS itself. Furthermore, when an OS is compromised, the malware could easily turn off all installed HIDS to prevent detection.

The VM based IDS has another benefit. Even when the OS in the target VM is compromised, the IDS still can mediate hardware accesses to force compliance to a given policy.

The detection methods provided are, e.g.: a lie detector (recognizing inconsistencies in responses from default system libraries), per memory page integrity checks (ala. tripwire), malware signature detection, a raw socket detector (often used by stealthed malware) and a memory access enforcer, which marks sensible memory parts as read only, leading to error notifications when someone tries to overwrite them.

### 6.2.2 Automated Virtual Machine Introspection for Host-Based Intrusion Detection

The work from Captain Brett A. Pagel [48] of the U.S. Air Force implements an intrusion detection system for Windows.

It is using hardware virtualization with Xen and Fedora Linux as Dom0. The goal of his work is to detect rootkits by scanning for the known mechanism, they use to install themselves into the operating system.

For example, many rootkits hook into the system service dispatch table (SSDT), which is going to be watched for unwanted entries.

One major challenge in his work is the detection of the correct operating system. As every build, service pack or even hotfix for an OS may have different offsets for the relevant variables, the exact version has to be detected. The technique used to do this for Windows OSes, is to scan the memory block for a special signature, located in the kernel VS VERSION INFO structure. When the structure is found,
the version information string is saved and the offset is used to set the kernel base address.

To interpret the memory block of the VM correctly, the exported symbol information for the kernel are needed. These are retrieved with the help of the `dumpbin.exe` utility, part of the Microsoft Visual Studio, and the guest’s kernel binary `ntoskrnl.exe`.

The gathered information are forged into a XenAccess configuration file, for use with a modified XenAccess 0.4 library. Provided with the kernel symbols, XenAccess is used to analyze the SSDT and detect or remove unwanted entries.

## 6.3 Comparison to classic Kernel Debuggers

This section gives a short description on Ring0 debuggers, a more detailed one on two actual kernel debuggers for Linux and compares them to our approach, using virtual machine inspection.

One major disadvantage of all classic debuggers, compared to our toolset is the lacking possibility to script and automate a debugging session, while on the other side more interactivity is given to the developer. Another drawback shared by all classic kernel debuggers is the limitation on one machine, which makes a distributed debugging session for multiple machines impossible.

The required time-synchronization or virtual time for the target machines - to avoid timeouts during protocol debugging sessions - is one more major drawback of the classic kernel debuggers.

### 6.3.1 Ring0 Debuggers

Debuggers that are working directly on the CPU, and even below every operating system installed, are called Ring0 debuggers, as that is the place they operate in. One example for such a debugger is the “Rr0d Rasta Ring0 debugger”. It does not use any source code or symbolic information of the running software, but disassembles the machine code instructions. The CPU is controlled directly via interrupts.

Ring0 debuggers are used for very low level debugging and reverse engineering of applications. The debugging results only consist of basic CPU instructions, registers contents and memory addresses.

The next abstraction level of debugging is done with kernel debuggers. Now at the core of an operating system, and mostly still located in Ring0, they connect the source code, or at least the symbolic information provided, with the debugging results. That makes it possible to map for example variable names to memory contents etc.

Two kernel level debuggers for Linux are KDB and KGDB.
6. Related Work

6.3.2 KDB

The Linux Kernel Debugger (KDB) [47] is a patch to the kernel which extends it with several debugging features. The kernel can be debugged from the same system by jumping into the debug mode on every kernel panic or dedicated key press (PAUSE key). Once set up, KDB can also enable the serial console and the kernel can be remotely debugged from another system.

When the debugger is actually running, it uses only one core of the CPU. Mostly the one on which the unusual activity occurred, which leads to the break into the debugger.

KDB is self-contained, meaning it does not rely on any other kernel code. Even though this implies some duplicate code, it enables the possibility to debug the whole rest of the OS’s kernel.

The installation procedure consists of patching the kernel and compiling it with multiple debug options enabled for better usability. For example: `CONFIG_KDB` and `CONFIG_FRAME_POINTER`.

The symbol information created by the build process are later used by the debugger. KDB does not support source level debugging.

During runtime the debugger is enabled by echoing a “1” into the `/proc/sys/kernel/kdb` file. It provides the developer with common debugger features like for example: reading and writing memory contents, displaying and modifying CPU registers, setting breakpoints, stack tracing and disassembling.

It is notable that the KDB kernel debugger is installed in memory regions where the real memory addresses equal the effective addresses. This makes it possible to run it at stages where the CPU has the memory translation still turned off.

One major drawback in comparison to our toolset is that the kernel itself has to be patched. This is not possible for any closed source operating system. Leading to the next disadvantage: the target operating systems are limited to Linux versions.

Furthermore, the debugging mode of KDB, using the serial console, requires two development computers.

Another drawback is that KDB cannot debug the boot loader or BIOS code of the guest kernel.

Advantages compared to our solution are: a VT-capable processor is not required to debug common kernels, it can debug arbitrary device drivers while our implementation is limited to these supported by Xen, single stepping is possible, and it supports special kernel-aware commands like `ps`, `btp` and `bta` to show the running processes and their backtraces.

6.3.3 KGDB

The Linux Kernel Source Level Debugger (KGDB) [5] is a kernel patch too, but it patches a GDB stub into the kernel. It is comparable to a GDB server, only running inside the kernel, which can be remotely controlled with a GDB over a serial console.
The fault handlers are patched too, to allow the developer to analyze unexpected faults, too.

The patched kernel is given the port number and connection speed, for the serial line, as bootup parameter. New versions of KGDB provide the debugging console over an ethernet connection as alternative option.

Like KDB before, it provides the developer with common debugging options. For example: setting breakpoints, accessing the CPU registers, switching between kernel threads, etc.

When the remotely attached GDB is provided the source code, source level debugging is possible.

A major drawback is that a KGDB buildup always requires two systems, one for the kernel and one for the debugger.

Another is the limitation to Linux as the target OS, as well as the need to patch and recompile the kernel.

The advantages compared to our tool are mainly the same as with KDB: no VT-capable CPU is required, every arbitrary device driver can be debugged and single stepping of the CPU is possible.

## 6.4 Kernel Debuggers coupled with Virtual Machines

The next section shows how these classic debugging approaches can be aided by virtualization techniques without using virtual machine introspection.

### 6.4.1 UML - User Mode Linux for Kernel Debugging

In User Mode Linux [31], a Linux kernel can be loaded as normal user space process on top of the the running operating system. Besides the advantages of not risking that a kernel panic freezes the whole system, and preventing the kernel under development from ruining the file system of the host, it enables the developer to attach a normal user space debugger like GDB to the running target kernel.

This removes the need for a second development machine for source level debugging. Another advantage is that multiple test machines can be run on one physical machine.

A major drawback is the limitation to Linux as target system. It is also limited to only one physical machine, and a GDB debugging session can only attach to one virtual machine kernel at a time.

UML does not allow the developer to debug actual kernel device drivers, since it does not provide physical hardware access to the virtual machines.
6.4.2 Kernel debugging with VMware or Virtual Box

A very straightforward approach [31] to using virtualization for kernel debugging with classic kernel debuggers is to emulate both development machines that are needed for debugging over a serial console (COM port). The serial cable is also emulated and connects both VMs.

The VMware and Virtual Box hypervisors can both be used for this task.

The target operating system is only limited by the compatibility of the used kernel debugger and that of the hypervisor. As most operating system have a compatible debugger available, nearly all of these are supported.

The serial connection is a weak point, regarding performance, in this debugging scenario. The connection speed is very limited (around 10KB/s) and the VMs have to permanently poll for incoming traffic, which produces a lot of CPU usage.

There exist solutions for VMware and Virtual Box that replace the serial console with a technique using local named pipes for the communication. They are called “Virtual Machine KD Extensions (VMKD)” [39] and “Windows Kernel Debugger booster for Virtual Machines” [61]. Using named pipes removes both limitations, the maximum connection speed and the need for polling.

The drawbacks of this whole solution in comparison to our tool, are the same as with the classic kernel debuggers. There is no mechanism for distributed debugging provided, the Linux kernel debuggers have to be patched into the kernel, and BIOS code and boot loaders cannot be debugged.

6.5 Other Related Work

6.5.1 Forensics examination of volatile system data using virtual introspection

The work [37] presents early concepts of how virtual machine introspection can be used by investigators during forensic analyses.

A classic forensic analysis uses only gathered static data, like for example raw hard-disk contents. While a wealth of useful data can be retrieved by this static analysis, many interesting information stay hidden. There are for example: the process list, open network ports, loaded kernel modules or volatile memory contents. These information can be retrieved with a live analysis of the machine.

The problem with a live analysis on a physical machine is its obtrusive behavior. Every action of the investigator on the running system can influence the data on the disk and may destroy valuable forensic evidence.

When the target system is a virtual machine, there is another approach to consider. Virtual machine monitoring or introspection can be used to investigate the live information of the VM without the risk of contamination.
The introspection instrument used in the work is the VIX Tool Suite described in [46]. It is used, for example, to read a genuine process list of the virtual machine. Even malicious rootkits cannot hide from this external access.

A detection of the introspection procedure by internal services of the VM cannot be excluded, but is very unlikely.

**Interim Résumé**

The related work chapter showed that there exist very different approaches to distributed debugging and kernel debugging.

Many of them are very specialized and therefore provide only features for kernel or distributed debugging. For example the distributed debugging approaches: Online debugging, deterministic execution replay or declarative debugging cannot be used to debug kernel applications. On the other hand, classic kernel debuggers like KDB or KGDB have no means for virtual time or distributed settings.

Recent publications sharing goals with our work are for example PDB, Xenprobes or ProMoX. They all lack one or more features in comparison to our work, e.g.: A flexible scriptable frontend for the debugging procedure, the possibility to use multiple physical machines for the distributed buildup and a time-synchronization solution for a virtual time.

Also shown are works that use virtual machine monitoring for completely different purposes like security checks or forensic investigations.
Conclusion

The goal of this work is to externally monitor virtual machines to debug internal protocol implementations. The designed toolset has a flexible scriptable frontend which is able to control different monitoring backends via network in distributed settings. Furthermore a separate client-server solution for time-synchronization is used to make all virtual machines run in corresponding time-slices.

The implementational result of this work is a functional proof of concept prototype consisting of a frontend - scriptable in Python - and a monitoring server backend utilizing a remote procedure call (RPC) communication mechanism.

During the evaluation of distributed debugging settings with breakpoints, the back-end had to be extended with a second RPC server to make multiple parallel synchronous RPC calls possible.

All that is required to debug a virtualized kernel with the toolset are the compatible debugging symbol information. With these provided, the monitoring tool can react on special conditions inside the kernel and arbitrary extended action can be undertaken. It can also listen to kernel events, like for example incoming packets in the network stack, and react before even the operating system kernel processed it.

This powerful functionality is provided to the developer in the form of simple Python script commands. He is able to debug even a distributed setting using a single Python script.

For the developer using the monitoring tool there is no big difference whether he is targeting multiple machines or only a single one. Except, of course, for the additional work to setup the distributed environment itself.

The modular design of the toolset allows a broad range of versatile appliances and a mix of different architectures. To our knowledge this is not possible with any other tool yet.

A main challenge during the implementation process was to find an appropriate way to bridge the semantic gap between the external view on the virtual machine's
memory as block and the contained contents. After a profound background research, we decided to use a debugger (GDB) for this task, as it is comparable to what it does with a user space application given the debugging symbols. So all that is needed to interpret the memory contents are the kernel debugging symbols in a GDB compatible format.

The next challenge was to realize the memory access into the VM using hypervisor technologies while coping with the issues of memory address translations and page table traversals. Our first approach was to use a modified version of the XenAccess library, but was later replaced with GDBserver-Xen, which fitted way better to our requirements. It encapsulates the memory handling issues and provides a GDB stub. GDBserver-Xen is part of the open source Xen project, hence it is no fixed black box and could be modified to our needs in the future.

The tool can be applied in many different situations. It mainly depends on the development effort put into the script for the monitoring session. However, the toolset does not change the fact that debugging an application is still a troublesome and complex task.

For example, I had the problem of variables not existing in the binary code during evaluation. This was due to compiler optimizations replacing them with advanced expressions during the compilation process. It should be considered to turn them off for the procedure.

The next challenging matter was to make this functionality accessible via a central frontend for distributed settings and to make this frontend as flexible as possible. This lead to the design decision of splitting the tool into a monitoring server and a separate frontend, implemented as Python script object.

It was noticed during the evaluation that the control of the test VMs, the monitoring servers and the synchronization clients is a rather elaborate and repetitive work. It is therefore a grateful target for automation and the integration into the server should be planned as future work.

One remark on the synchronization solution used. It contains a Xen patch, implemented by the Distributed Systems Group of the RWTH Aachen University, for an older version. It had to be revised and patched into the used 3.3.1 version of Xen. This indicates that the synchronization solution is either limited to this version or has to be patched into future versions of Xen, too.

As the evaluation shows, the extended proof of concept implementation is already usable for complex distributed kernel debugging settings, and all test cases could be evaluated properly, but there is much room for further improvements and the integration of more helpful features.

7.1 Future Work

A number of ideas for future improvements and extensions are presented in the last section.

The created toolset is already not only useful for protocol developers, it can be used by deployment managers to surveil all running virtual machines in a server farm by
polling changes in status variables in the VMs. He can thereby verify that all system
are up and running normally.

Another scenario for the tool would be to use it as a security monitor for VMs. One
can inspect vital services encapsulated in the virtual machine or run integrity checks
on the active operating system inside.

Further advanced features for the tool would be to integrate more VM management
functionalities that make it possible to dynamically create VMs on the fly, start further
instances of monitoring servers on the machine etc. This would help the developer to
ease the distributed deployment procedure and would even make dynamic changes
of the whole setting possible.

With the help of the used time synchronization, the developer can debug constel-
lations that wouldn’t even run in real-time on his machine or would require an
expensive setup of multiple physical machines otherwise. Additionally, the virtual
machines can be coordinated to run together in groups intermittently while the
others have to wait, or other thinkable scenarios.

The design of the toolset, utilizing an open and well defined interface, makes it
possible to mix front- and backends of different types and architectures. A soliciting
extension would therefore be to port and implement different versions of the backend
for other architectures.

One example could be Windows as target operating system. The fact that Microsoft
does provide the debugging symbols for Windows only in \textit{pdb} format, which is not
compatible with GDB, makes it difficult to debug Windows with the current im-
plementation. There do exist other debuggers, e.g. included in VMware, that can
be used to debug a virtualized Windows with the help of these debugging symbols.
One future work would be to integrate one of those debuggers, which provide a GDB
stub as frontend, into a compatible monitoring server for our toolset.

As we focussed on kernel debugging during implementation and evaluation, it would
be interesting to develop example scripts for the frontend to monitor user space
applications in a selected operating system. In a second approach, a compatible
monitoring server could be developed straightforward that runs inside the target
virtual machine and attaches to a user space application directly.

Possible alternative approaches for the frontend also leave much room for future
work. One could develop a graphical user interface that directly attaches to the
monitoring servers or helps to build a script for a debugging session. Thinkable is
also an interactive frontend that provides direct control of the debugging process
(aliike GDB), but is able to communicate with and address multiple backends at the
same time.

The problems during the evaluation showed that it would be better to use an alter-
native RPC library that supports multiple synchronous calls natively. It could be
used to completely replace “Apache Thrift” in front- and backend.

One motivation for the development of our virtual machine monitoring toolset was
the possibility to extract “real live” protocol stack information for use in network
simulations. Of course, it would be a future work to develop a test scenario with VMs
connected to a network simulator, in which the simulator is fed with live information
from the VM, and evaluate the toolset on it.
As already mentioned in the conclusion, the work to control and set up a test environment is very expensive. It would be a very handy feature if the monitoring server could start up the target virtual machine or load the client for the synchronization server by itself.

Given these automations, it would be less troublesome to evaluate the toolset on larger numbers of target virtual machines. This would also make it easier to answer the still open question how the overall performance of the tool scales with the number of monitored virtual machines.

The current toolset is fully functional even for complex distributed situations and kernel developers all over the world can now begin to use it. During the writing of this work it was already used in the Distributed Systems Group of the RWTH Aachen University to debug timing problems inside a patched Xen kernel.

Until more developers start to use the tool, it may require more usability improvements and optimizations. Furthermore the developers first have to learn that such a tool exists.
Bibliography


List of Figures

2.1 Logical address translation ................................................. 6
2.2 Paging by Intel 80x86 processors ........................................ 8
2.3 x86 privilege rings ........................................................... 9
2.4 x86 Paravirtualization ....................................................... 16
2.5 x86-64 Native and Paravirtualization .................................... 16
2.6 Ring-Buffer with Consumer and Producer ............................. 18
2.7 Split Device Driver Model .................................................. 19
2.8 Xen Memory Layers .......................................................... 20
2.9 GDB and GDBserver ....................................................... 25
2.10 Using the GDB .............................................................. 25

3.1 Concept Design .............................................................. 30
3.2 Toolset adapts to different infrastructures ............................. 31
3.3 Access to user space applications with the toolset .................. 31
3.4 Backend Modules .......................................................... 33
3.5 Time Synchronization Buildup ............................................. 37

4.1 Implementation Overview .................................................. 40
4.2 Implementation Overview - GDB + GDBserver-Xen .................. 41
4.3 GDBserver-Xen reading VCPU registers using Libxenctrl .......... 43

5.1 First buildup in Test-Environment A .................................... 49
5.2 Test Case 1 - Output showing corresponding variable contents .... 50
5.3 Test Case 1 - Virtual machine state-change .......................... 51
5.4 Test Case 2 - Output showing the variable being reset ............ 52
5.5 Test Case 3 - Output showing local variables being read and written 53
5.6 Protocol debugging buildup in Test-Environment A ................ 54
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Test Case 5 - Comparing the window size readings</td>
<td>55</td>
</tr>
<tr>
<td>5.8</td>
<td>Test Case 6 - Output showing detail-information on selected packets</td>
<td>56</td>
</tr>
<tr>
<td>5.9</td>
<td>Monitoring buildup for the Distributed Setting</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>Synchronization buildup for the Distributed Setting</td>
<td>57</td>
</tr>
<tr>
<td>5.11</td>
<td>Allocating time slices to the virtual machines</td>
<td>58</td>
</tr>
<tr>
<td>5.12</td>
<td>Test Case 7 - Output - Two global states of one distributed variable</td>
<td>59</td>
</tr>
<tr>
<td>5.13</td>
<td>Test Case 8 - Output - Distributed breakpoints with multiple threads</td>
<td>60</td>
</tr>
<tr>
<td>5.14</td>
<td>Failing cross-reading attempt</td>
<td>61</td>
</tr>
<tr>
<td>5.15</td>
<td>Successful cross-reading attempt</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Different Hypervisors ........................................ 10
4.1 ServerDispatcher Linking Scheme .......................... 42
4.2 Example LibMiGDB methods and GDB/MI representation 44
5.1 Test-Environment A ........................................ 48
5.2 Test-Environment B ........................................ 48
5.3 Microbenchmarks for every basic function ............... 53
Appendix

A.1 Python Script used in Test Case 1

#!/usr/bin/env python
#
# Introspection Test Case 1
#

#Imports
import sys
import time
import pprint
from urlparse import urlparse
from thrift.transport import TTransport
from thrift.transport import TSocket
from thrift.transport import THttpClient
from thrift.protocol import TBinaryProtocol
import Introspection
from ttypes import *

import Introspection
from ttypes import *

#Config
host = 'localhost'
port = 10099

#Network Init
socket = TSocket.TSocket(host, port)
transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Introspection.Client(protocol)
transport.open()
pp = pprint.PrettyPrinter(indent = 2)

#Introspection Part
for i in range(1, 10):
    pp.pprint("getVariable:" + client.getVariable("acid_test_c",));
    time.sleep(2.5);

print ("pause now");
client.pause();
time.sleep(5);

print ("unpause now");
client.unpause();

#Close Network Connection
transport.close()
A.2 GDM/MI Dialog

In the MI example, ‘-’ means that the following line is passed to GDB as input, while ‘-’ means the output is received from GDB.

MI example:

-> -exec-run
<- `running
<- (gdb)
<- *stopped,reason="breakpoint-hit",disp="keep",bkptno="1",thread-id="0",
   frame={addr="0x08048564",func="main",
   args=[{name="argc",value="1"},{name="argv",value="0xbfc4d4d4"}],
   file="myprog.c",fullname="/home/nickrob/myprog.c",line="68"}
<- (gdb)
-> -exec-continue
<- `running
<- (gdb)
<- *stopped,reason="exited-normally"
<- (gdb)

CLI example:

(gdb) break 15
Breakpoint 1 at 0x8048848: file STL_vector_int.cpp, line 15.
(gdb) r
Starting program: /home/userx/a.out

Breakpoint 1, main () at STL_vector_int.cpp:15
15       cout << II.size() << endl;
A.3 Parameters of the Synchronizer Client Kernel Module

All available options are listed in this table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_as_server</td>
<td>integer</td>
<td>Run module in server or client mode?</td>
</tr>
<tr>
<td>server_address</td>
<td>string</td>
<td>Address to send (un)register/completion packets to (if in client mode).</td>
</tr>
<tr>
<td>server_port</td>
<td>integer</td>
<td>Port Number</td>
</tr>
<tr>
<td>broadcast_address</td>
<td>string</td>
<td>Address to send run-permission packets to (if in server mode).</td>
</tr>
<tr>
<td>client_port</td>
<td>integer</td>
<td>Port Number</td>
</tr>
<tr>
<td>cli_id</td>
<td>integer</td>
<td>Numeric ID used when registering at the sync-server.</td>
</tr>
<tr>
<td>cli_descr</td>
<td>string</td>
<td>Description</td>
</tr>
<tr>
<td>sync_domain</td>
<td>integer (array)</td>
<td>Xen-ID or comma-separated list of Xen-IDs of synced domain(s).</td>
</tr>
<tr>
<td>netback_limit</td>
<td>integer</td>
<td>Bandwidth limit for PVMs (in MBit).</td>
</tr>
</tbody>
</table>

The options used in some of our test cases are the following:

```plaintext
modprobe kernelsync
run_as_server=0
server_address=10.0.0.1
server_port=17543
client_port=17544
broadcast_address=127.0.0.1
sync_domain=1,2
```
A.4 Libconfig Configuration File Example

The configuration file shown is that of a monitoring server used in many test cases.

gdbserver:
{
    host = "localhost";
    port = 9999;
    executable = "/usr/local/bin/gdbserver-xen";
    vm_name = "vm01";
    target_architecture = "i386:x86-64";
    symbol_file = "vmlinux";
};

introspection_server:
{
    host = "localhost";
    port = 10099;
    port_spare = 10100;
};

A.5 Grub Settings used for the Xen Setup

title    Xen 3.3.1 / Ubuntu 8.04.2, kernel 2.6.18.8-xen
root     (hd0,0)
kernel   /boot/xen-3.3.1.gz sched=sedf
module   /boot/vmlinuz-2.6.18.8-xen root=UUID=b0ea2d8f-63aa-444c-9d90-31
module   /boot/initrd.img-2.6.18.8-xen
quiet

Important is the sched=sedf option, which activates the proper scheduler.