

# Distributed System Monitoring and Failure Diagnosis using Cooperative Virtual Backdoors

Benoit Boissinot

`Benoit.Boissinot@ens-lyon.fr`

E.N.S Lyon

sous la direction de

Christine Morin

`Christine.Morin@irisa.fr`

IRISA/INRIA Rennes

Liviu Iftode

`iftode@cs.rutgers.edu`

Rutgers University

June 16, 2006

## 1 Introduction

In computing environment, failures are not the exception. Computers are subject to operating system bugs, hardware failures or malware attacks. In a clustering or grid environment, management tools are needed to improve the reliability of the system (detecting failures and recovering from them), load-balancing and intrusion detection.

The backdoor architecture is a mechanism for external monitoring and intervention in an operating system. Since it is separated from the running operating system, the backdoor will survive a system crash or a rootkit attack.

In a cluster environment, the backdoors running on each node can interact and exchange local data to build a global system state.

In this report, we present a backdoor architecture based on virtual machines, which can be used to build a cooperative monitoring system.

In the first part, we present what is a virtual machine and virtual machines classification. Then, the related work on virtual machine based monitoring and the backdoor architecture is presented. Later on, we describe the functionalities that our backdoor system should meet. We follow by a description of the implementation and the limitations of our system. Finally, we describe the experiments we used to test the backdoor system.

## 2 Virtual Machines

The virtual machine concept, developed by IBM in the late 60's [4], is a way to run multiples instances of an operating system on the same hardware in an isolated way. The idea is that each operating system running on a virtual machine will behave exactly the same as if they were running on the bare hardware. It was first used as a way to securely partition a mainframe, so that production and development code could run at the same time without sacrificing reliability.

A virtual machine monitor (VMM) is a program that provides an abstraction of the system (processor, devices, ...) and isolation for guests operating systems. One of the first VMM is the VM/370 used on IBM System/370 hardware. It provided the illusion for the guests operating systems to run on their own mainframe.

Popek and Goldberg [11] defined in 1974 the requirements for an architecture to be fully virtualizable. But as demonstrated in [12], the x86 does not fit at least one requirement: there are 17 sensitive instructions that can be used in non-privileged mode and does not generate an exception. This means those instructions cannot be virtualized by the VMM without doing an emulation (reading all the instructions and rewriting them if needed) instead of a virtualization.

To workaroud the limitation of the x86 architecure, different methods have been used [13]:

- Emulation of sensitive instructions. For each branch, the VMM analyzes the instructions and sets a hardware breakpoint if it detects a sensitive instruction or a jump. When it encounters a sensitive instruction, the VMM will emulate it. If it encounters a jump, the VMM will analyze the code block starting at the target address. A cache of already analyzed code is kept in memory. Since some instruction could rewrite the already analyzed code, the pages are write-protected after they have been scanned for jump and sensitive instructions.
- The operating system is modified to use a software interface to communicate with the VMM. This approach is more simple, but requires porting the operating system to this interface before being able to run on it. This is called paravirtualization.
- Latest processors from Intel (Intel-VT) and AMD (Pacifica) incorporate a new mode and new instructions so that all the sensitive instructions will trap. Using this extension, the x86 instruction set fits the requirements from Popek and Goldberg.

Popek and Goldberg distinguish two type of VMMs [8]:

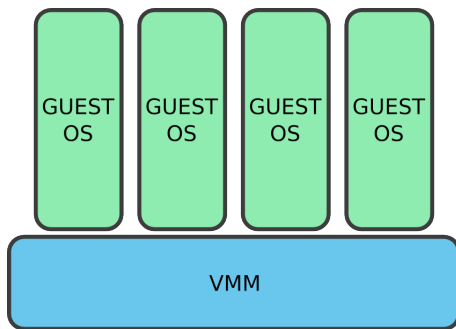


Figure 1: Architecture of a type I VMM. The VMM runs directly on the bare hardware.

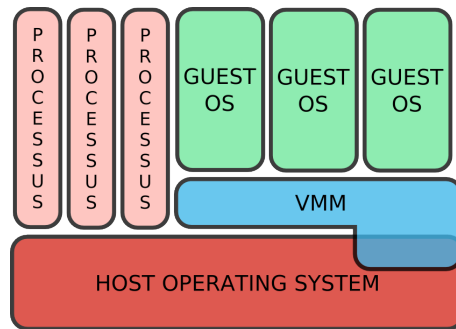


Figure 2: Architecture of a type II VMM. The VMM is hosted by a host OS.

- Type I VMM: the VMM runs directly on the hardware. It must provide scheduling and virtualization of the resources to the virtual machines. A type I VMM is sometimes called hypervisor.
- Type II VMM: the VMM runs in an host OS. It is also called hosted VMM. The VMM relies on the host OS for the drivers, scheduling and resources allocation.

Actually, a virtual machine technology rarely fits in one or the other category. For example, User Mode Linux and Xen virtual machines are not precisely in one category.

User Mode Linux is a special architecture for the Linux kernel that runs a kernel as a userspace process. Since the guest kernel needs some modifications, it is a hybrid type II VMM. Both the VMM and the guest kernel runs in user mode, the VMM uses ptrace and signals to catch the systems calls and the exceptions.

The Xen virtual machine monitor is a hybrid type I VMM [5]. For simplicity and performance reasons, the guest kernels need to be adapted to the Xen interface to run.

Moreover, there is a third type of virtualization technology: Vserver and BSD jails partition a system in such way that every partition will have only a portion of the resources. This too can be seen as a very lightweight VMM. In fact, there is still only one operating system running, but the kernel virtualize the resources (filesystem namespaces, process ID namespaces, scheduling) for each group of processes.

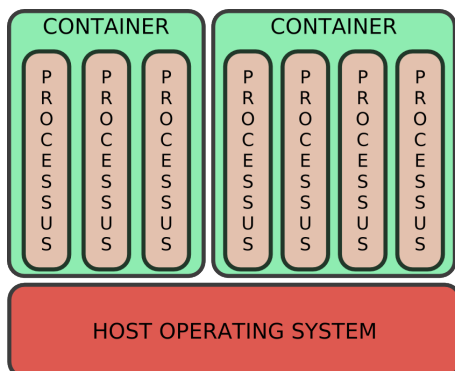


Figure 3: Architecture of a container based virtualized operating system.

### 3 Related Work

#### 3.1 Virtual Machines Based Monitoring

Virtual machines have already been used for monitoring purposes. For example the CoVirt group in the University of Michigan has used type II VMM for different purposes: logging, intrusion detection and recovery.

Revirt [6] is a virtual machine based system logger, which can be used to replay some action too. BackTracker [9] uses virtual machines to find the entry point of an attacker in a system. Finally, Plato [10] is a general purpose tool for virtual machine introspection and interposition. The three projects use either UMLinux or a derivative (FAU Machine).

Garfinkel and Rosenblum [7] use VmWare Workstation (type II VMM) to build a virtual machine monitoring interface dedicated to intrusion detection. However, using a modified type II VMM as a building block for a monitoring system makes the system less reliable and is less efficient. The first reason is that the monitoring application will run at the same level as the virtual machine monitor, so there is no way to protect the VMM from a bug in the monitoring system. Secondly, a type I VMM has better performance, and since the monitoring software will run in a virtual machine, it can be isolated for resource allocation too.

Overall, we believe a type I VMM would be a better base for building a monitoring backdoor.

#### 3.2 RDMA Based Backdoor

The original Backdoor architecture was built by the DiscoLab team [3] using intelligent networking cards (I-NIC) with remote direct memory access (RDMA). The backdoor software ran on the processors embedded in Myrinet cards. The DiscoLab team was able to use the backdoor system for fault

detection and recovery.

The main advantage of the I-NIC approach is to provide a fault tolerant backdoor architecture. Because the backdoor software is running on a completely separate device, it can survive to a hardware failure.

However the I-NIC backdoor implementation requires extra and costly hardware (High speed networking interfaces) that would be more useful used by the main operating system for high performance computing for example.

## 4 Goals

The backdoor mechanism we are building serves several goals:

- debugging
- fault tolerance and recovery
- management (e.g. load balancing)

### 4.1 Debugging

The backdoor architecture can be used during the execution of a program to check data invariants defined by the programmer. For example, for a distributed shared memory system, one could check that the sum of all the memory pages in all nodes is equal to the amount of pages of the system. In this way, it is ensured that no pages lost during the process migration.

Since the backdoor continuously runs after the kernel crashed, the developer can use it as a post mortem analysis tool to read any location from the guest kernel memory.

### 4.2 Recovery

Similarly to the way shown in [3], the backdoor can be although used to extract lightweight snapshot from the kernel subsystem or from processes. Those snapshots can later be used to restart a service in case of failure.

Since the backdoor supports remote memory write operations, it can be used to perform recovery actions. For example, the backdoor system can reset a particular subsystem, or kill a deadlocked program.

### 4.3 Management

A backdoor can help to automate management of resources. It makes it possible to detect the number of resource used in the system, either locally or globally for a set of nodes. Having this information, it possible to rebalance the load of the system. For example, if a node is overloaded (high-latency between the time when the request is issued and the time it is fully processed), a signal can be sent to the system or logged in order to trigger

load rebalancing. To reduce the load, the signal could a reduction of the frequency of the requests.

The backdoor architecture can be although used to detect malicious changes. For example, the backdoor can help executing simple tasks like checksumming the kernel memory code to check that there was no intrusion in the operating system (rootkits attacks).

## 5 Backdoor Over a Virtual Machine

We want to build a backdoor architecture using a virtual machine as base system. We had to make a choice between the two majors open source virtual machines monitors Xen and User Mode Linux.

We choose to develop our backdoor mechanism on the Xen virtual machine monitor for three reasons. The first reason is that Xen is a type I VMM and has a lower overhead on the virtual machines than type II VMM. Since Xen is paravirtualized, it does not need to `ptrace` or insert breakpoint in the guests kernels and it permits Xen to give more CPU to the virtual machines compared to User Mode Linux.

Second, we believe that there is less chance for a bug to compromise the isolation of the virtual machines than in a hosted environment because the layer that runs on the hardware in the type I virtual machine monitor is much thinner.

Third, the advantage of Xen compared to User Mode Linux is the ability to have driver domains (domains who manage a device), which can make the whole system more reliable in case of faulty drivers.

Finally, we believe that Xen has a promising future since it can take advantage the virtualization technology included in the latest x86 processors, which allows to run unmodified guests.

### 5.1 Xen Virtual Machine Monitor

In the following we use this terminology:

- the *hypervisor* is the virtual machine monitor in a type I virtual machine.
- a *domain* is a guest kernel hosted by the hypervisor. In Xen, we distinguish two different domains: dom0 and domU's, dom0 is a privileged guest used for administration purpose. For example, dom0 is used to launch the domU's (unprivileged domains). The main "privileged" feature is to have access to any part of any domain memory. The setup of new domains is done by domain 0.

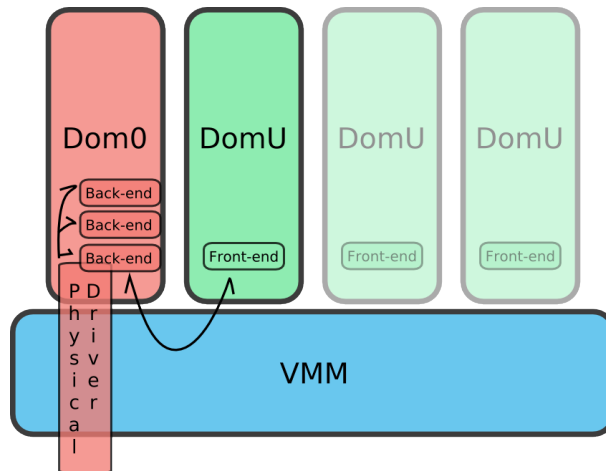


Figure 4: Architecture of the Xen driver. On the left the domain zero which manages all the physical devices and does their virtualization via the back-end/front-end infrastructure.

### 5.1.1 Device Management

The particularity of Xen is that the hypervisor has no driver for the physical devices. The Xen virtual machine monitor only manages the isolation of the address space, the isolation for the access to the physical devices and the CPU resources allocation.

For physical devices access, the devices are bound to a specific domain. If the device is bound to a non privileged domain, the domain will have exclusive access to it. Otherwise, if it is bound to dom0, dom0 will usually virtualize it and share it between all the domains. This is done with the communication infrastructure provided by the hypervisor.

### 5.1.2 Memory Management

In the memory layout of a virtual machine, we distinguish the physical memory (P) which is the virtualized memory seen by a guest OS, and the machine memory (M) which is the entire memory from the computer, i.e. what the virtual machine monitor sees.

We first describe the usual memory layout on Linux x86. The first three gigabytes of the virtual address space belongs to the userspace program. The last gigabyte belongs to the kernel.

In a paravirtualized environment (e.g. Xen), there is an another extra region: the upper 16 MB, which is used for the hypervisor. The reason behind this layout is the fact that TLB flushes should be avoided for performance reason (for example during a system call).



For building our backdoor, we assume the following:

- the different kernels will not be paged out (the virtual addresses of a guest kernel will always correspond to a page in the physical memory).
- we have a way to map a page from a "remote" kernel in the privileged kernel (dom0) context.
- we can resolve a virtual address from another guest.

Three different methods are possible for the virtualization of the virtual memory addresses:

- $P = M$ , each guest is allocated a subset of the machine memory and accesses it directly. The hypervisor restricts the access to the other part of the memory.
- $P \neq M$ , the hypervisor gives  $n$  pages to the guest and the guest has the knowledge that they are different from the real addresses.
- $VP$ , virtual physical mode, which is like  $P \neq M$  but the guest does not know that the address it manipulates are not the real ones. In that case, the page table is usually in shadow mode, which means that the hypervisor will catch any attempt from the guest to update the page table, and translate the physical address to a machine address. Obviously, this doesn't work if the guest manages devices doing DMA (direct memory access), since it needs to pass the real address to the device.

The Xen x86 architecture currently uses  $P \neq M$ , which means that the guest keeps translation table in memory, and does the translation when doing an memory mapping update.

## 5.2 Soft Memory Management Unit

While designing the backdoor system, the first problem we had to solve is to translate a virtual address of a different guest into the corresponding machine address. To achieve this, we have to create a software memory management unit (MMU). On x86 hardware, a 32 bit virtual address is translated to a physical address by the MMU in the following way:

The address is divided in three fields:

- the first 10 bits represent the offset in the PGD (page global directory), the PGD located at the value of the control register CR3.
- the value obtained from the PGD points to the page table entry, the following 10 bits represents the offset in the page table entry (PTE) that points to the actual physical page.

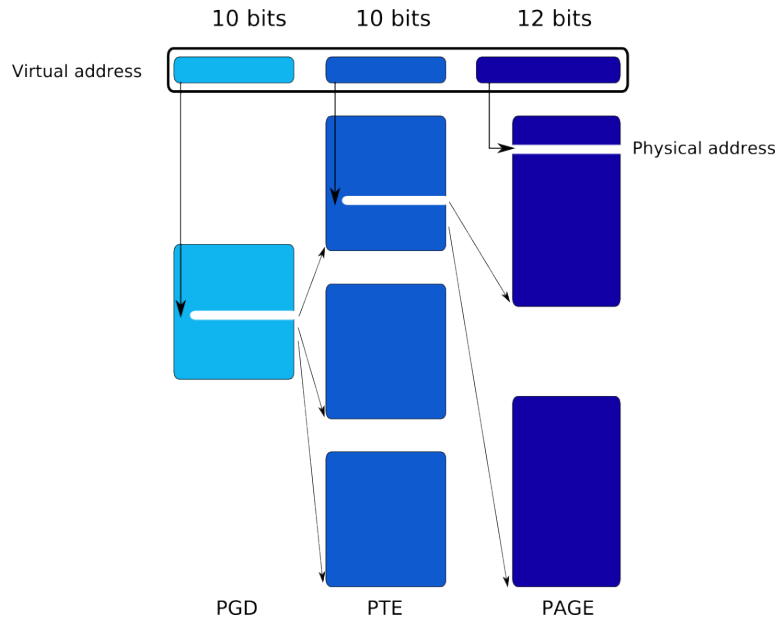


Figure 7: Resolution of a virtual address by the memory management unit.

- finally, the physical address is found in the physical page at the offset which has the last 12 bits for value.

The figure 7 shows a summary of how the virtual address resolution is done.

Usually, this translation is done by the hardware, but in our case we have to do it in the software. The first step is to retrieve the value of the CR3 from the other domain, which is done using a privileged operation to get the state of the virtual CPU. Then, we map the PGD and follow the page table until we find the physical page. If it is not possible to map a page, or if the page does not belong to the target domain, the Xen hypervisor will raise an error.

To summarize, we need three memory management calls to the hypervisor in order to read from a virtual address for another domain.

### 5.3 Backdoor Interface

To provide the access to the memory of another guest kernel, we have built a driver that exposes a character device to userspace. This device has the same semantics as the Linux character device `/dev/kmem`, except that it exposes the virtual memory address space for another guest operating system process instead of the virtual memory address space for the current process.

The device supports the following operations: `open`, `lseek`, `read`, `write`

and `close`. Before issuing a read or a write, the process should call `lseek` to specify the starting address from the virtual memory.

When some addresses are accessed a lot, it is more efficient to use the `mmap` call, which maps a portion of the address space from a guest kernel in the process address space. After the call, accessing the newly mapped memory returns the value from the guest kernel virtual memory. A library was built to make the access to the device easier.

We modified the build process of the Linux kernel to generate an object file that is not linked with the final kernel image but which includes all the possible debug informations about the structures used in the kernel. It includes all the types globally defined in the kernel: structures, enumeration and unions, as well as typedef definitions.

Then we have although modified the ELF object code analyzer `objdump` to generate the library from the object file debug section. Instead of printing the informations, our modified ELF parser prints the library code corresponding to the types defined in the object file. We can use this code as the base for our library.

The generated library contains the code to access any structure and resolve a member of the structure from a file interface to the virtual memory. It allows to know the size of any given type defined in the kernel.

The virtual address are found by parsing the file `System.map`, which contains the mapping between symbols and virtual addresses for all the symbols defined globally in the kernel image.

## 5.4 Example

As an example, we describe the different steps involved when reading the list of the processus of domain 1:

- Lookup the address of the init process in `System.map`.
- Lookup the size of `struct task_struct`, the offset of the member `tasks` and the offset of the member `comm` in this struct.
- Open `/dev/domain.mem` our pseudo-device.
- Use `ioctl` to set the target domain to domain 1.
- Read the `struct task_struct` of the init process.
- Use the `tasks` member to read the next structure until the end of the linked list.
- For each struct, print the process name using the `comm` field.

## 6 Implementation Issues

We implemented and tested the backdoor mechanisms with Xen version 3.0.2 (based on Linux 2.6.16). We used computers with Pentium III processors running at 500MHz with 512 Mb of RAM.

### 6.1 Performance Issues

Currently, the backdoor device needs 5 ms to complete a read. It could be greatly improved by reducing the calls to the hypervisor. Actually, we need to map two pages to be able to resolve a virtual address, and then we issue the mapping of the needed page. Instead of doing the translation ourself, we could add a new interface to the hypervisor to translate a virtual address from any given domain. Otherwise, a caching mechanism could be implemented to avoid resolving address we translated earlier. Special care should be taken to track invalid mapping, so it probably makes more sense to call the hypervisor to do the translation for us.

Since dom0 is privileged, the resolution of the monitoring will affect the performance of the other domains. The degradation of the performance in a guest domain is proportional to the resolution of the monitoring. For example, with a read every 100 milliseconds, there will be a 10% degradation for the overall CPU time available for a guest domain, with a resolution of 10 milliseconds, the CPU time left to the guest domain will be cut by half.

### 6.2 x86 Specifics

Our implementation relies on certain x86 specifics that makes it hard to port to other architectures.

Unlike the x86 case, where the page table has the structure of a tree, other architectures like PowerPC use a hash structure. In that case, when a hash lookup fails, the hardware ask the kernel to resolve it. So, we do not have a simple method to lookup a virtual address from the privileged domain.

A first possible solution is to have an intimate knowledge of the way the other guest kernel store the different mapping and how it maps the physical memory in its virtual address space. The other solution is to have a new hypercall or privileged operation that resolves virtual address for a given domain.

### 6.3 Atomicity and Races

While doing the monitoring application, we must be aware that the guest domain could update the memory during our read. We are in the same situation as a read in an SMP context without locking. The solutions that could be used are the following:

- locate and use the guest kernel locks and spinlocks. It needs lot of work and may be unsafe for a crashed kernel. Furthermore, we would need to write in the guest kernel memory.
- only read scalar values (they are atomic on x86)

## 7 Event Based Monitoring

### 7.1 Backdoor Informations Exchanges

For the communications between several backdoors, we use the XML-RPC protocol [1]. The XML-RPC protocol is a standard protocol used to make functions calls over a network (remote procedure calls). It encodes the arguments and the returned object using XML and uses HTTP as the transport. Although this is not the most efficient protocol we could have used, the XML-RPC was chosen because it is powerful and simple to use due to the existing libraries.

During the monitoring, a backdoor is acting both as a client and as a server at the same time. It pushes the information it retrieves in the system, and receives the information from the other nodes.

We manage the global knowledge by having indexed dictionaries of all the collected data in the system in each node. When a backdoor needs data from other nodes, it will issue a call to retrieve the data for a predefined key. At the same time, it will send what it collected locally.

The update mechanism can be described by the following pseudo code:

```

knowledge = {} # no initial knowledge
peers = [...] # list of peers to connect to
server = start_server(knowledge) # the local server

while 1:
    update_local_knowledge(knowledge)
    update_server(server, knowledge)
    update_remote_knowledge(peers, knowledge)

```

## 8 Experiments

We used two scenarios to test our backdoor system. The first one is a single node setup that involves a unique backdoor. The second scenario uses multiples backdoors and "sensors" from different nodes and computes a distributed state.

## 8.1 Single Node

Our first setup uses the backdoor in a single node context. The monitor triggers an alarm when the number of runnable threads in the system is above a given level.

This way, we can detect a forkbomb attack even when the system is too loaded to respond. Xen uses a real-time scheduling algorithm (sEDF, earliest deadline first), so it is ensured that the monitoring domain will receive enough CPU time even if the monitored domain is heavily loaded.

The number of running processes is found by counting the number of structures in the `task_struct` list using the method described in 5.4. We do not count the structures that points to processes which already exited (but were not deallocated) by checking the value of the `state` field.

## 8.2 Distributed Monitoring

To test a distributed monitoring setup, we created a simple scenario consisting of multiple nodes using the network block device [2] driver. The network block driver makes a remote file or physical block device (a disk partition for example) appear as if it was a local disk partition. The kernel module acts as the client, whereas a userspace process acts as the server on the remote node.

For this experiment, we added a scalar in the client (on the kernel side) which is incremented each time a request is sent to a remote server. For the server, we added some code to export the number of requests processed to a XML-RPC client/server. Then, we created multiple network block device on a node and connected them to different server nodes.

The monitor collects the data from the servers and from the client. It can compute an approximation of the load by taking the delay between a send from a client and the end of the processing of the request in the server.

# 9 Future Work

## 9.1 Distributed Operating System

We want to use our backdoor system in a distributed operating system. For this, the first milestone would be to have the single system image operating system Kerrighed running over the Xen hypervisor. Then, we should be able to use the backdoor mechanism as a monitoring and recovery tool for some subsystems of Kerrighed. A first objective could be to have a recoverable version of KerFS the distributed filesystem.

Even if having a single system image (SSI) operating system running over a virtual machine would be slower, we believe that the gain in manageability would be sufficient to make the system attractive.

For example, a large organisation could run a SSI virtual machine on the desktops, the SSI operating system would only use the spare cycle. Or, on a grid environment, an application could be deployed by using virtual machines, which makes migration or checkpointing more easy.

## 9.2 Aggregation of Data

The second interesting problem for the backdoor system is to have a scalable design for the aggregation of data from a large number of nodes (for example in a grid operating system context). A hierarchy should probably be built to aggregate the information without losing too much information for the global management of the system.

## 10 Conclusions

We have presented a backdoor architecture based on type I virtual machine monitor. Associated with a library that extract debug information from a kernel, our backdoor can be used to monitor a virtual machine and to exchange information gathered from other backdoors monitoring different virtual machines. This cooperative behaviour can be used for load balancing or fault detection in a cluster of machine or any distributed system.

Our backdoor architecture can although be integrated with a distributed operating system such as the single system image Kerrighed. It could be used as a base to implement reliable components in Kerrighed.

## Acknowledgments

This internship was done in the Distributed Computing laboratory at the Rutgers University, and in the PARIS team from IRISA/INRIA in Rennes. This work was funded by the associated team Phenix.

## References

- [1] <http://www.xmlrpc.com/>.
- [2] <http://nbd.sourceforge.net/>.
- [3] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote repair of operating system state using backdoors. In *International Conference on Autonomic Computing (ICAC-04)*, New-York, NY, May 2004. Initial version published as Technical Report, Rutgers University DCS-TR-543.

- [4] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.
- [5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [6] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. Re-Virt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the 2002 Symposium on Operating Systems Design and Implementation*, Dec 2002.
- [7] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.
- [8] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the ACM Workshop on Virtual Computer Systems*, pages 74–112, 1973.
- [9] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, 2003.
- [10] Samuel T. King, George W. Dunlap, and Peter M. Chen. Plato: A platform for virtual machine services. Technical Report CSE-TR-498-04, Electrical Engineering and Computer Science Department, University of Michigan, 2004.
- [11] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [12] J. Robin and C. Irvine. Analysis of the intel pentium’s ability to support a secure virtual machine monitor. In *Proc. of the 9th USENIX Security Symposium*, 2000.
- [13] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. May 2005.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Virtual Machines</b>	<b>2</b>
<b>3</b>	<b>Related Work</b>	<b>4</b>
3.1	Virtual Machines Based Monitoring . . . . .	4
3.2	RDMA Based Backdoor . . . . .	4
<b>4</b>	<b>Goals</b>	<b>5</b>
4.1	Debugging . . . . .	5
4.2	Recovery . . . . .	5
4.3	Management . . . . .	5
<b>5</b>	<b>Backdoor Over a Virtual Machine</b>	<b>6</b>
5.1	Xen Virtual Machine Monitor . . . . .	6
5.1.1	Device Management . . . . .	7
5.1.2	Memory Management . . . . .	7
5.2	Soft Memory Management Unit . . . . .	9
5.3	Backdoor Interface . . . . .	10
5.4	Example . . . . .	11
<b>6</b>	<b>Implementation Issues</b>	<b>12</b>
6.1	Performance Issues . . . . .	12
6.2	x86 Specifics . . . . .	12
6.3	Atomicity and Races . . . . .	12
<b>7</b>	<b>Event Based Monitoring</b>	<b>13</b>
7.1	Backdoor Informations Exchanges . . . . .	13
<b>8</b>	<b>Experiments</b>	<b>13</b>
8.1	Single Node . . . . .	14
8.2	Distributed Monitoring . . . . .	14
<b>9</b>	<b>Future Work</b>	<b>14</b>
9.1	Distributed Operating System . . . . .	14
9.2	Aggregation of Data . . . . .	15
<b>10</b>	<b>Conclusions</b>	<b>15</b>