A Survey and Comparison of Fault Isolation Approaches for Operating System Kernel Extensions

Bachelor-Thesis by Claudius Cremer
September 2010
Declaration of Authorship

I hereby confirm that I have authored this bachelor thesis independently and without use of others than the indicated resources.

Darmstadt, September 3rd, 2010

(C. Cremer)
## Contents

1 Introduction 6

2 Functionality and Problems of Device Drivers 6
   2.1 Functionality ................................................. 7
   2.2 Common Bugs ................................................ 8
   2.3 Malicious Attacks via Device Drivers ........................ 8

3 Taxonomy and Criteria of the fault isolation techniques 8
   3.1 Taxonomy ..................................................... 9
   3.2 Criteria ...................................................... 9

4 Grouping and Summary of the isolation techniques 10
   4.1 Sandbox-based isolation .................................... 11
      4.1.1 Nooks .................................................. 11
      4.1.2 Microdrivers ........................................... 12
      4.1.3 BGI ..................................................... 13
      4.1.4 Nexus .................................................... 14
   4.2 Virtual Machine-based isolation ............................ 15
      4.2.1 iKernel .................................................. 15
      4.2.2 LeVasseur ............................................... 16
      4.2.3 Transparent Fault Tolerance ........................... 17
      4.2.4 VEXE'DD ............................................... 19
   4.3 Hardware-based isolation .................................... 20
      4.3.1 RingCycle ............................................... 20
   4.4 Language-based isolation .................................... 21
      4.4.1 Singularity .............................................. 21
      4.4.2 SafeDrive ............................................... 22

5 Evaluation and Comparison 23
   5.1 Sandbox-based Isolation .................................... 23
   5.2 Virtual Machine-based Isolation ............................ 24
   5.3 Hardware-based Isolation .................................... 26
   5.4 Language-based Isolation .................................... 27
   5.5 Overview of the Approaches ................................ 28

6 Conclusion and Future Work 35
# List of Figures

1. Results of Coverity and Microsoft ................................................. 6
2. I/O Model of the x86 architecture .................................................... 7
3. Taxonomy of isolation techniques ..................................................... 11
4. The iKernel Architecture ................................................................ 16
5. I/O architecture of Xen 3.0 .............................................................. 18
6. I/O descriptor ring in Xen ............................................................... 19
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amount and Size of Drivers in Commodity Operating Systems</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Ratings of sandbox–based isolation approaches</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>Ratings of virtual machine–based isolation approaches</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Ratings of hardware–based isolation approaches</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Ratings of sandbox–based isolation approaches</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Overview of the Approaches</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Summary of the Ratings</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Overview of OS and Modifications</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>Overview of Fault Detection and Recovery</td>
<td>31</td>
</tr>
<tr>
<td>10</td>
<td>Results of Network Performance Tests</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>Results of Storage Performance Tests</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>Results of other Hardware Performance Tests</td>
<td>34</td>
</tr>
</tbody>
</table>
Abstract

Over the years many researchers have investigated the impact of device drivers on the reliability of computer systems and identified drivers as a major cause for system failures [52, 8]. They have searched for solutions that stabilize systems and at the same time do not reduce performance. In this thesis several different approaches to prevent driver failures from affecting overall system reliability are being evaluated and compared. They are organized according to a taxonomy proposed in a survey of isolation techniques at University of Southern California[63]. For an expedient rating six evaluation criteria covering performance, intrusiveness, recovery support, fault isolation, extensibility, and portability are proposed in this thesis. Our results show that the driver problem is not yet solved satisfactorily and given reliability is still only achieved at the expense of performance in present fault isolation mechanisms for commodity operating systems.
1 Introduction

Device drivers are essential components of operating systems: By acting as an interface between kernel and hardware they provide simplified communication and flexibility. However, in today's commodity operating systems with Monolithic kernels like Windows and Linux or with Mach Kernels like Mac OS X, drivers operate as kernel modules with high privileges. Therefore, an erroneous driver can crash the whole system easily. Studies show that drivers are the most prominent cause for operating system failures: A Stanford University study from 2001 describes, that drivers from Linux systems are the most buggy in terms of error rates, having three to seven times higher error rates than the rest of the kernel [8]. Also, Coverity [12], a source code defect detection and analysis system, discovered in 2004 by analyzing the Linux kernel 2.6.9 that 53 percent of bugs were found in device drivers [11]. A study of Microsoft's online crash analysis from 2003 shows that in Windows XP 89% of system crashes are caused by drivers [52]. The results of Coverity and Microsoft's online crash analysis are shown in Figure 1.

In Microkernel OS architectures drivers are moved out of the kernel into the lower-privileged user mode. This protects the kernel, but introduces a significant performance penalty [33]. Changing commercial operating systems like Unix, Linux and Windows, which are historically based on the Monolithic kernel structure, to this kernel structure would cause an enormous reengineering effort. By now, Linux and Windows both support user mode device drivers [38, 16], but only some types of drivers are supported and they perform poorly [62].

Device manufacturers mostly write drivers without having extensive kernel programming experience or access to documents of kernel modules and functions. Compared to user-mode programming there are fewer tools for debugging and development of kernel code available. Hence, drivers are often created by copying and editing code from existing drivers. This leads to the persistence of old errors or to the introduction of new bugs caused by a lacking understanding of the existing code. Another reason for the increasing number of faulty drivers is the continuous advent of new devices and device types on the market and the steady growth of the driver code required to exploit their sophisticated functionality. Testing an operating system for reliability with all possible combinations of existing drivers is obviously impossible. Because of the high market share of operating systems like Linux and Windows and the huge amount of device drivers that they support, a framework, which monitors device drivers and contains driver faults, makes more sense than approaches which demand a complete change of the kernel structure or rewriting of existing drivers. Besides the costs of downtime and the repair of a failed system, failures can have a great impact on embedded or mobile systems. A failure of such systems would be fatal as they are mostly unmanaged or difficult to reach. Therefore, a software isolation framework that provides a reliable system by fault detection, isolation and recovery is an appropriate solution.

There are a lot of approaches for ensuring software fault isolation and all authors argue that approaches are exception-ally effective and performant using more or less specific evaluation criteria. In this thesis I discuss several fault isolation frameworks, classify them by their fundamental isolation mechanisms they are using and attempt to develop a set of evaluation criteria that allow for an objective comparison. In the next chapter I will describe the functionality of device drivers and will introduce problems addressed by most approaches. Afterwards, in chapter three, I will introduce the taxonomy of the approaches and will describe the six criteria. On basis of the taxonomy I will introduce the approaches and summarize their basic components and evaluation proceedings in chapter four. Then in chapter five I will evaluate, compare, and rate these approaches on the basis of the criteria introduced in chapter three. Afterwards I will give a recommendation and discuss, in how far the goal - reliability with low overhead on existing operating systems and hardware - is achieved by the investigated approaches to fault isolation.

2 Functionality and Problems of Device Drivers

In order to motivate the isolation of drivers from the kernel, I explain the operating mode of device drivers and discuss faults and possible malicious attacks related to a lack of isolation in this chapter.
2.1 Functionality

Device drivers control the interaction between the operating system and hardware devices. In an operating system with a monolithic kernel structure, all operating system modules and device drivers are consolidated using the kernel's address space and running with kernel privileges. Having also linked device drivers to the kernel, Exokernels are also affected [69]. This is due to historical and performance reasons: Historically, because when monolithic kernel structures were established, operating systems and device drivers were programmed solely by computer system vendors. And because of the interaction between hardware components and kernel functions, the integration of device drivers into the kernel has a great performance benefit compared to running them with insufficient privileges in user-space [33].

How device drivers are installed and initialized depends on the system platform’s I/O model. As all approaches considered in this thesis are based on the x86 architecture model, it is introduced here. At boot time of the operating system attached hardware is identified and its type, capabilities, and resource requirements are queried. After the discovery, interrupt lines, which can be shared by multiple devices, are assigned, memory space is reserved, and relevant drivers are linked to the kernel. If a user program needs a hardware resource, the kernel passes this request to the corresponding device driver for execution. Two methods are used for communication between kernel and drivers: Shared memory and special I/O functions. When the device fulfilled the task, it sends an interrupt to the kernel. Interrupts are used by the device to signal a tasks completion or if an error occurs. The kernel then calls the device driver again, which performs the respective actions and returns status information like error-reporting. The driver also confirms the interrupt to the device otherwise the device will continue to send the interrupt. To reduce processor consumption when transferring large data, a mechanism called direct memory access (DMA) is frequently applied. With direct memory access the device driver can directly write data to or from a reserved memory region. When the kernel issues a request to the device driver, the DMA controller is set up with the identity of the device, the memory address and the amount of bytes which will be transferred. The device driver typically writes a pointer containing the DMA address range to the device's register. After the DMA transfer is finished and no error occurred, it sends an interrupt to the kernel, which informs the corresponding driver. Again the driver then returns status information and the processed data to the application. If the device operates asynchronously, the device driver can block the processor until the device sends an interrupt. When shutting down the system or a driver is unloaded, it releases the assigned memory space and unregisters its interrupt handler. Figure 2 shows a simplified overview of the I/O Model of the x86 architecture.

![I/O Model of Device Drivers](image)

Figure 2: I/O Model of the x86 architecture

There also exist driver hierarchies: Some busses have controller drivers which coordinate communication and control of drivers attached peripherals. The controller driver takes over I/O operations, interrupt handling, and DMA transfers. Besides device drivers, loadable firmware, which resides on some devices like network cards or graphic cards, can also manipulate the functionality of its device [69].

To isolate a driver in this architecture, a fault isolation framework should interpose between kernel and driver and between driver and its hardware device. It ought to verify communication of the two interfaces, restrict the driver to its assigned memory space, and observe the compliance of DMA operations and interrupt handling. If these conditions are fulfilled, this fault isolation approach provides strong fault isolation.
2.2 Common Bugs

As described in the introduction, device drivers are almost supplied by third party vendors and not by the operating system developers. Being a software product, bugs are inevitable: In reference [22], studies show that there are 1-16 found bugs per 1000 lines of code. Although with each update the number of bugs decreases exponentially, it never reaches the zero mark.[49] Also, when including new features through driver updates, new bugs are unavoidable. The trustworthiness of device drivers also depends on the quality assurance efforts of the driver developers. If the device driver is an open source project, the quality-assurance-related process can be worse than in commercial products. The following weaknesses were addressed in the isolation frameworks as common problems:

- **Pointer exceptions:** Being written in languages with pointer arithmetic, pointer exceptions like null pointer exceptions are often a rich source of bugs[22, 5]. Therefore these bugs were included by nearly all approaches when conducting fault detection tests.

- **Buffer overruns:** Having full access privileges to the memory, there are no restrictions for a driver to write beyond its assigned address space.

- **Read/write to wrong address space:** Similar to the buffer overrun illicit memory access can corrupt memory partitions, which are not assigned to the specified driver. Thus, a flawed driver can write any code to sensitive kernel memory locations. Because DMA can access any physical memory location its wrong usage can have similar consequences.

- **Deadlocks:** Deadlock errors can be caused by incorrect interrupt handling. For example, if a driver does not confirm an interrupt, the interrupt is repeated by the device. This can lead to locking of the device and because hardware interrupts have high-priority, to starvation of other processes [69].

Nearly all of fault isolation approaches discussed in this thesis consider these described weaknesses.

2.3 Malicious Attacks via Device Drivers

Being executed with kernel privilege and having full access to kernel data structures, device drivers attract attackers to misuse drivers and to exploit vulnerabilities. The compromising of wireless device drivers in Windows XP [6] and Mac OS X [35] show the potential exploitation of security vulnerabilities in drivers. Only a few approaches take malicious attacks into consideration [5, 69]. However, as many attacks are based on the bugs described in the section before, the majority of attacks are also blocked by the prevention of such invalid operations. The following weak points used by attackers were concerned by some fault isolation techniques:

- **Buffer overflows:** By exploiting a buffer overflow and injecting code into a device driver an attacker can overtake a system. For example, at the Black Hat Briefings computer security conference in 2005 a buffer overflow for Windows USB stack which would enable an adversary to gain full control over the Windows machine was demonstrated [13]. In [6] this could even be achieved remotely via a wireless LAN adapter. The authors of Microdrivers [5] found several buffer overrun vulnerabilities concerning device drivers in the MITRE vulnerability database [58].

- **Function pointers:** By modifying function pointers in the driver an attacker can hijack the kernel control flow. For this an attacker sets the pointer to injected code which will then executed by the kernel. A study from 2007 by Petroni et al. shows that 22 of 25 Linux rootkits used this attack to hijack the system [50].

- **DMA:** Williams et al. describe in their RVM paper [69] that an Ethernet adapter can send any memory location read by DMA to a remote host or overwrite any memory address with a received packet. They also construct an example with DMA writing to another device: by overwriting the video memory it would be possible to manipulate the screen. Also, with DMA the system can be easily crashed by overwriting kernel memory. DMA weaknesses are still a problem in modern operating systems: In 2010 Researchers of the European Security Expertise Center demonstrated how to gain access to Windows 7 via DMA [3].

At last, the loadable firmware can also be a problem: In July 2010 a hardware vendor shipped replacement mainboards which contained a trojan in its embedded systems management firmware [14]. However, none of the approaches support monitoring loadable firmware and the authors of Nooks and RVM state that it is not possible to verify loadable firmware by fault isolation techniques.

3 Taxonomy and Criteria of the Fault Isolation Techniques

This chapter introduces the taxonomy of fault isolation techniques and the criteria for evaluation and comparison.
3.1 Taxonomy

To classify the fault isolation approaches compared in this thesis, four categories proposed in a survey of isolation techniques at the University of Southern California[63] are being used. These categories denote the basic isolation concepts of the discussed approaches and will be used to provide a well-structured discussion in the next chapter. Fault Isolation techniques can be categorized as follows:

- **Sandbox-based isolation:** Driver interactions with other system components are restricted by moving the drivers into encapsulated areas, where all communication with the kernel or hardware is monitored and verified at runtime. This can be achieved by applying access control mechanisms or by modifying or wrapping function calls by the driver.

- **Virtual-Machine-based isolation:** Virtual Machines emulate physical hardware and thus isolate between the emulated environment and the real machine. A virtual machine solution can be installed as an underlying platform with guest operating systems running on it or it is installed on top of an existing operating system, the host. The virtual machine monitor or so-called hypervisor manages the hardware resources among the installed guest VMs. To lower the overhead, hardware vendors provide new hardware features like the Intel VT-x [61] and AMD-V [2] technology to improve virtualization in terms of performance and isolation. Fault isolation approaches take advantage of the isolation that virtual machines provide by moving device drivers to virtual machines. However, a dysfunction of the virtual machine monitor or hypervisor can affect the whole system.

- **Hardware-based isolation:** In hardware-based isolation special devices like the I/O Memory Management Unit (MMU) or the processor enforce isolation by restricting access to certain resources, such as memory. The I/O Memory Management Unit, which translates a device's Direct Memory Access (DMA) addresses to physical addresses, can prevent write access to arbitrary memory addresses. Although several of the evaluated approaches include hardware mechanisms to verify driver behavior, only one approach was found which is solely based on hardware.

- **Language-based isolation:** Language-based fault isolation uses designated programming languages and compilers to ensure that the driver binaries operate appropriately and accesses only explicitly allocated memory regions. The language-based fault isolation techniques discussed here make use of type safety, an extent of programming languages to prevent data type errors. Type safety constraints can be static, catching potential errors at compile time, or dynamic, observing values at run time, or a combination of both.

3.2 Criteria

Although the introduced solutions have some techniques in common, like for example using function call wrappers, they are not easily comparable for a number of reasons:

- The reported results of performance and recovery tests were obtained using very diverse hardware.

- Different operating systems were specifically targeted by different approaches. In total, Four different operating systems were used as target systems.

- As a consequence of using differing hardware and operating system platforms, different benchmark utilities and different driver implementations were used.

- Results were reported in different units. For example, network throughput measurements are being diversely reported in megabytes per second or in a normalized manner. The throughput differences between native and isolated drivers are either reported in percent or as relative factor to native.

- Fault injections for testing the prototypes were performed applying different fault models and injection tools.

A precise and objective comparison would only be possible, if all investigated frameworks had been evaluated on identical platforms using identical benchmarks. To nonetheless approximate an objective comparison as well as possible, six fundamental criteria are chosen for the comparison in chapter 5. For none of the introduced approaches all six criteria have been discussed explicitly before. It is therefore considered a comparatively fair set, covering both emphasized strengths and implicit trade-offs.

- **Performance:** Most software implemented fault isolation approaches have the bad reputation of operating at the expense of performance. Therefore all approaches attribute great importance to this criterion and most of them report extensive performance tests. A good fault isolation solution is therefore not only apt to make operating systems more reliable but also does not impose significant operational overhead. Peaks constitute a crucial aspect
in this rating, because the CPU overhead is additive when executing more drivers and can lead to system timeouts or hangs. Besides the CPU performance overhead, memory overhead (e.g. the size of virtual machines) is also considered. To estimate the size of loaded device drivers, I installed three commodity operating systems on a virtual machine using VMware Workstation 7. The operating systems and virtual environment were installed or configured with default configurations. The driver information was gathered using the commands `driverquery` (Windows XP), `kextstat` (MacOSX) and `lspci` plus `lsmod` (Linux). The results are shown in following table:

Table 1: Amount and Size of Drivers in Commodity Operating Systems

<table>
<thead>
<tr>
<th>OS</th>
<th>Windows XP SP3</th>
<th>MacOSX 10.6.4</th>
<th>Linux Debian 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount Drivers</td>
<td>92</td>
<td>81</td>
<td>45</td>
</tr>
<tr>
<td>Average size</td>
<td>25,782 KB</td>
<td>43,438 KB</td>
<td>28,631 KB</td>
</tr>
<tr>
<td>Size Of Drivers:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sound driver</td>
<td>7,040 KB</td>
<td>16,384 KB</td>
<td>19,072 KB</td>
</tr>
<tr>
<td>Network driver</td>
<td>16,384 KB</td>
<td>118,784 KB</td>
<td>18,100 KB</td>
</tr>
<tr>
<td>VGA driver</td>
<td>25,344 KB</td>
<td>32,768 KB</td>
<td>3,230 KB</td>
</tr>
</tbody>
</table>

- **Intrusiveness**: The amount of driver or kernel code that needs to be modified for integrating the respective fault isolation framework. This criterion also takes into account if a potentially required manipulation is possible at all, for example if source code access restrictions apply. Of course, the less code needs to be changed the better the solution can be integrated into an existing system. Relative to the kernel code which consists of millions of lines of code [59], the addition of a few thousand lines to the kernel is a small effort and must only be done once. However, the problem is that not all commodity operating systems’ kernel code is publicly available (e.g. Microsoft Windows), so that these solutions have limited applicability. As there is a vast number of drivers - for example over 35000 for Windows XP [48] and 70% of Linux’s 2.4.1 code base are for device drivers [8] - , modifications to the driver code also constitute an applicability limitation. Furthermore, just like for kernel code, many companies do not publish the source code of their drivers. That is why the rating gets better the less intrusive a framework is. Table 1 shows the amount of loaded drivers when evaluating three default operating system installations. However, most of the drivers counted at the above-mentioned evaluation were implemented by the operating system developers and their code is therefore trusted. But, because the vendor information is missing mostly, it is hard to separate between trusted operating system drivers and untrusted third party drivers. Thus, an absolute number cannot be stated.

- **Fault isolation**: Fault isolation is the most important criterion, since it is the major aim of the described solutions. This measure examines how many previously undetected failures are detected when the discussed approaches are being applied.

- **Recovery**: The recovery criterion shows how successful the recovery mechanism is resuming a driver’s operation after it has failed.

- **Extensibility**: It gives information on how easily new hardware and its corresponding drivers can be integrated in the system. A report of 2004 [42] showed that there are 88 new driver versions being released daily. If including a new driver into the isolated environment imposes great overhead, then the approach is not very well suited for updating drivers or installing new hardware.

- **Portability**: This criterion deals with how much effort is required for porting the framework to other operating systems. Since many commodity operating systems have similar architectures that suffer from the same problems in terms of isolation, approaches that are transferable from one commodity OS to another one are reasonable. As shown in table 1, the size of drivers, which control the same hardware, vary significantly (e.g. the network driver ranges from 16 Megabytes to 119 Megabytes) on the three evaluated operating systems. Therefore, low overhead is a suitable condition for portability. However, none of the investigated approaches implemented and tested their fault isolation framework on more than one operating system.

4 Grouping and Summary of the isolation techniques

In this section the approaches discussed in this thesis are introduced and classified according to the taxonomy presented in the previous chapter. I restrict myself to describing only basic principles of the approaches and testing procedures
of performance as well as recovery. A more detailed discussion of these approaches according to a set of isolation quality metrics is presented in the following chapter. Eleven approaches regarding fault isolation of device drivers will be introduced in the next subsections and their categorization is shown in figure 3.

![Isolation Techniques Diagram]

### Figure 3: Taxonomy of isolation techniques

#### 4.1 Sandbox-based isolation

In this subsection I will introduce four approaches which provides isolation by encapsulating drivers in execution environments and additionally check each inbound and outbound communication. Nooks [56] suggests a protection and isolation environment for driver execution. Microdrivers [19] splits the driver into a kernel level component and a user level component and monitors the communication between the two components. BGI [7] divides the driver into separate protection domains, which use access control lists (ACL) to access virtual memory and an interposition library for communications with the kernel. Based on a microkernel operating system, Nexus [69] executes drivers in user space and verifies their operation with a reference validation mechanism.

##### 4.1.1 Nooks

Acting as a layer between kernel and drivers without changing their code, Nooks is categorized as Sandbox-based approach. A Nooks means a hidden or secluded spot [45] and its author uses the notion to describe a lightweight kernel protection domain for preventing device drivers from crashing an operating system.

Introduced in 2002, the Nooks architecture targets existing operating systems and drivers without modifying kernel or driver code. However, modifications to the kernel must be made when integrating Nooks with the operating system. A Nook can contain multiple drivers for performance reasons and not all drivers need to run in a Nook protection environment. Drivers are executed within these domains allowing them to read but preventing them from writing to memory outside their protection environment.

An isolation mechanism, which creates, manipulates and maintains domains, controls the inter-domain or domain-kernel communication using wrapper stubs and remote procedure calls like Extension Procedure Calls (XPC) [55].
Shared kernel resources are duplicated inside the domain and maintained by an object tracker. If procedures like network packet delivery do not need to perform synchronously, they are performed as a single batch after the driver has completed execution.

Two types of recovery are supported: Full restart and rollback. Full restart unloads and restarts the driver. Rollback allows retrying the failed operation by restoring the driver to a previous state from maintained shadow copies of driver states. This recovery mechanism is termed Shadow Driver.

As a testing platform, two computers were equipped identically with a Pentium 4 1.7 GHz processor, one gigabyte of memory and the Linux v2.4.10 operating system. Additionally two Gigabit Ethernet adapters were installed on both systems. To test the impact of the driver performance, the network drivers on one machine were isolated with Nooks. The prototype implementation prevents writing of data and verifies incoming and outgoing function calls from device drivers by wrapping all calls into and out of device drivers. Although drivers were not executed in a separate Nook by the prototype implementation, the costs of different levels of protection were emulated.

For performance tests, the network benchmark Netperf [27] was used. Sixteen drivers comprising network adapter, sound card, hard disk, file system and in-kernel web server drivers were manipulated with faults and loaded to a Linux 2.4.18 kernel. Three different Linux kernels were used: an unchanged kernel, a modified kernel with and one without recovery mechanism. These modified operating systems were installed in a VMware Virtual Machine.

The faults were automatically placed by a ported and adapted fault injection tool [45] used for the Rio File Cache [44]. Failures were categorized as system crash failures and non-fatal failures. When system crash failures occur, the operating system freezes, reboots or throws an exception. Non-fatal failures are all other failures that are not affecting the operating system. 517 faults that lead to system crashes in an operating system without Nooks and 417 errors that were non-fatal to an unmodified operating system were injected. While the detection of system-crash-inducing faults was successful for almost all injections, 40% of non-fatal faults were not spotted. The author states that the failure detectors need to be more specialized to the behavior of some classes of drivers.

In addition, ten manual bugs were injected or checks of objects in the driver code were removed, all of which were detected or prohibited by Nooks.

Three manipulated drivers were tested on real machines with the recovery mechanism by observing the behavior of the applications using them. The behavior was categorized as crash, malfunction, or fully functional.

4.1.2 Microdrivers

Microdrivers were introduced in 2008 [19] and were extended in 2009 [5]. In this approach device drivers are split into a trusted kernel-level partition (k-driver) and an untrusted user-level partition (u-driver) to improve fault isolation. Also, driver development for the user-level partition is eased by supporting user mode debugging tools. By isolating and monitoring the user-level partition Microdrivers is categorized as a Sandbox-based isolation technique.

Performance-critical or high priority functions such as interrupt handling are implemented in the kernel-level partition and executed in kernel-mode. The remaining code, such as device initialization and configuration, is moved into the user-level partition and executed outside the kernel in a separate process. Both partitions can communicate with the physical device.

The k-driver functions as communication medium between the kernel and u-driver partition. It forwards requests from the kernel to the u-driver and receives invocations of privileged procedures or kernel functions from the u-driver. Since the u-driver is untrusted, all messages from the u-driver are monitored by the RPC Monitor, which will be described later. For efficiency, the u-driver partition may be shared between multiple drivers.

Because existing driver code needs modifications in order to support the Microdriver approach, the tool DriverSlicer is introduced. DriverSlicer automatically splits driver code into the two partitions depending on typical function prototypes in the code. The tool also replaces method calls to allocator/deallocator and lock/unlock functions with wrappers. If the automatic determination of code delegation fails, the programmer needs to write marshaling annotations in the code.

Furthermore, the tool optimizes the code for data communication between the two components by analyzing complex data structures and allowing only updated fields to cross the kernel/user boundary. An optimal boundary should contain only a few functions, should be crossed infrequently and pass little data. The authors point out that operating systems like Windows Vista or Mac OS X, in which some drivers already are moved completely into the user mode, are lacking this optimization.

On both sides of the boundary runtime layers let function calls appear to be local, although they are implemented as remote procedure calls (RPC). They also invoke a run time library, which contains an object tracker that synchronizes the data on both sides. The object tracker again consists of a bidirectional table that stores and translates pointers of both sides similar to the Nooks framework. The communication of both sides is controlled by a RPC monitor. The RPC monitor can detect data corruption, but cannot prevent the effects or recover from a system crash. The RPC monitor runs as a kernel module and verifies all k-driver function calls invoked by the u-driver via control transfer policies. While these policies are being checked, the RPC monitor copies the data received to a vault area in the kernel’s address space. If the data satisfies the set of policy invariants, it is synchronized with the corresponding kernel data structure and removed.
from the vault. If it fails, a recovery mechanism can be triggered. Further, the RPC monitor verifies data structure invariants that enforce the integrity of updates to kernel data structures by the u-driver. These integrity constraints are automatically extracted from the code with a modified version of Daikon [18], an invariant inference tool for user-space applications. The authors recommend specifying the invariants in a training phase during driver development.

A new lock primitive called Combolock prevents deadlocks or handle so-called Spinlocks that are frequently used in the kernel. For example a spinlock may fail, if a u-driver process acquired a spin-lock on a kernel object and is descheduled. Kernel threads then have to wait until the u-driver is scheduled again.

Besides controlling the communication and synchronizing data between the two components, the runtime also supports recovery of the u-driver. However, no recovery mechanism is implemented in the prototype. The authors suggest, that Microdrivers could be extended with failure detection and recovery mechanisms such as Shadow Drivers [55] or SafeDrive [70] to improve system reliability.

To evaluate the Microdrivers approach, two Ethernet adapters were tested using a Pentium D 3.0 GHz, one gigabyte memory and the operating system CentOS 4.2, which is based on a Red Hat Linux. Additionally an USB host and a sound card were tested on this configuration. Further, an Nvidia nForce Ethernet adapter was tested on an AMD Opteron 2.2GHz, also with one gigabyte memory and CentOS 4.4. To measure the performance, modified drivers were compared to unchanged original drivers using the benchmark utilities vmstat [68] and Netperf [27]. Besides measuring the throughput and CPU utilization, the amount of manual annotations, and the memory usage of unmodified device drivers and the corresponding Microdrivers were examined. Tests of the newly introduced RPC monitor included four drivers: Two network cards, a RealTek RTL-8139 and a 8139C+, and a USB host controller were tested using QEMU 0.9.1. An Ensoniq soundcard driver was tested using VMWare workstation 6. QEMU and VMWare were running on an unmodified Linux-2.6.18.1 kernel. All test results were compared to the ones for their native drivers.

The prototype of the first paper had only rudimentary mechanisms of detecting and recovering faults in a u-driver: If the user-level component crashed, control returned to the kernel. Also only null pointer dereferences were injected into the user-mode parts. The null pointer injections didn’t lead to an unstable operational state of the operating system, but the application using the driver hang and the device was unusable. Later the RPC monitor was evaluated in more detail using simulated attacks, fault injection and false positives of overly specific invariants (serendipitous invariants). Simulated attacks included buffer overflow attacks, control hijacking via injected downcalls as well as via modified function pointers and non-control data attacks.

Before the RPC monitor was tested, each u-driver was used with training workload and training configurations to obtain a set of invariants. Alsamixer [31], a Unix soundcard utility, and the Unix Ethernet configuration command ethtool were used for the training phase. To test overly specific invariants, the SafeDrive fault injector was used to inject 400 random faults into the u-driver. Because of the limitations of the prototype, only the Ethernet adapter drivers were tested. As the implemented RPC monitor cannot prevent system crashes, system logs were evaluated to examine wether the RPC monitor detected a crash.

The performance overhead of the Microdrivers approach was assessed by measuring the throughput and CPU utilization of the two Ethernet card drivers and the USB driver. Netperf [27] was used to evaluate send and receive network packets from a client machine. The USB driver test implied a 140 megabytes file copy to a USB disk. To test the CPU utilization of the soundcard driver, a 256 Kbps MP3 file was played. However, the duration of the MP3 file was not mentioned. Each test was run ten times and the average was reported as the result.

4.1.3 BGI

Byte-Granularity Isolation (BGI) [7] by Castro et al. was introduced in 2009 as a fault isolation technique, which consists of a compiler plug-in and an interposition library. Being an isolation technique that implements memory protection through access control, BGI is categorized as a sandbox-based isolation framework. Although BGI is designed to contain driver faults and to prevent buffer overflows and underflows, the authors confess that effects of malicious code cannot be prevented: calling host functions in the wrong order, non-atomic inline access checks, unchecked memory reads, and preventing malicious code from programming device hardware are currently not implemented for performance reasons. Similar to Nooks, device drivers run in separate protection domains. An access control list specifies the access rights of the domains to individual bytes of virtual memory. An interposition library checks, grants and revokes access rights according to the invoked operation.

Four data structures are used to store access control lists at runtime and are shared by all domains: a kernel table, a driver table per process, a kernel conflict table, and a driver conflict table per process. The kernel table and driver tables are large arrays of access rights, whereas the conflict tables contain entries where more than one domain owns access rights. For performance reasons, BGI tries to avoid accesses to conflict tables and uses spin locks and atomic-swaps ([7], Section 6.5) for synchronization. For memory allocations and for the protection of these tables from being overwritten, the kernel has to be modified to reserve virtual address space and preallocate physical pages.

The compiler plug-in inserts function primitives to manipulate ACLs and links the device driver with the interposition library. The compiler modifies all direct and indirect calls to kernel functions and addresses of the driver code to use
the corresponding wrappers in the interposition library. Furthermore, function calls to use the ACLs are inserted into the
driver code. Similar to the DriverSlicer tool used in the Microdriver approach the compiler uses source code annotations
for automated code generation. The interposition library checks the access rights and objects passed in cross-domain
calls. Furthermore, the interposition library consists of kernel wrappers and extension wrappers. Kernel wrappers check,
grant or revoke rights to the arguments or objects supplied by the driver and calls the wrapped kernel function. The
extension wrappers call the driver functions and verify, grant, or revoke arguments returned by the driver. The wrappers
have a similar purpose like the Nooks architecture, but do not maintain translation tables or replicated resources.

In addition to read and write rights, access rights needed to be extended by icall rights for implementing control flow
integrity assurance, type rights for object type safety, and ownership rights for allocation/deallocation of memory. Icall
rights are granted when functions are called indirectly or passed in kernel calls that expect function pointers. Type rights are
granted when drivers receive an object from the kernel that they need to manipulate. Ownership rights are used to
keep track of allocated objects within a domain. For optimal use of dynamic memory allocators in x86 Windows the
ACL data structures are aligned to 8-byte slots. Adopted from WIT [1], local and global variables are aligned to 4 byte
boundaries and surrounded by guard slots to prevent overflows.

BGI also supports recovery from driver failures. Driver wrappers call driver functions within a try clause. If the
wrapper detects a failure or the domain is already in a recovery state, an exception is thrown. The wrapper then cleans
up the ACL-tables to release kernel objects assigned to the respective driver’s domain. Then the Windows Plug and Play
(PnP) Manager is used to unload and restart all domains that returned an exception error. Therefore, device vendors are
expected to provide the appropriate functions for the PnP manager and only drivers supporting the Windows Plug and
Play mechanism are supported.

Like in Microdrivers, recovery is not fully implemented in the prototype, as the driver wrapper releases not all kernel
objects. However, what types of kernel objects cannot be released is not explained in detail. All tests were run on an Intel
Core2 Duo processor with four gigabyte of memory running Windows Vista Enterprise SP1. Sixteen drivers, including an
Intel PRO/100 Ethernet adapter, the FAT file system and an USB host driver, were tested by compiling the drivers with
and without BGI instrumentation. For the hard disk, file system and USB drivers, the benchmark PostMark [28] was used.
The network Throughput was measured with the ttcp utility [51] and kernel CPU time with kernrate [37]. The FAT file
system drivers and Intel Pro/100 drivers were exposed to injected with fault injection, which studies [54] indicated to be
most common in operating systems. 429 out of 675 manipulated buggy drivers failed and their failures were categorized
into blue screen errors, freeze of the operating system and freeze of the application. Faults resulting in a blue screen
were tested again in a separate BGI domain and investigated. Fifty manipulated drivers, which resulted in blue screen
failures that BGI can detect, and 140 drivers, which led to operating system or application freezes were injected to test
the recovery mechanism of the BGI prototype. After the original driver was loaded by the recovery process, a test was
repeated and evaluated. Because the recovery was not fully implemented in the prototype and some failures could not
be detected, recovery did not work for all drivers.

4.1.4 Nexus

In order to prohibit error propagation at the OS/driver interface, the Nexus operating system [10] implements a specific
isolation mechanism [69]. Nexus is a Microkernel based operating system and therefore executes device drivers in user-
space. The authors state that it is nevertheless possible to implement the proposed mechanism on process isolating
operating system like Windows and Linux. Because the implemented reference validation mechanism (RVM) isolates
drivers and associated devices from the kernel, this approach is categorized as sandbox-based.

The reference validation mechanism also monitors direct memory access (DMA) as well as interrupt handling and
interferes at invalid read and writes and misuses of interrupts. When installing the driver, the RVM connects the driver
with the appropriate device and allocates I/O-resources to the driver. The RVM also invokes a device-specific reference
monitor which in turn uses a device-specific specification (DSS). Similar to manifests in the Singularity approach (cf.
Section 2.1, [53]), DSS are declarative and open for auditing. The authors estimate one to five days to develop a DSS to
a well documented existing driver depending on the familiarity with the language used for DSS.

A provided compiler translates the domain specific specifications to an executable reference monitor. Drivers can be
unloaded and restarted, but their corresponding reference monitors persist. Acting as a state machine, the reference
monitor accepts all allowed transitions of the observed device driver and blocks operations that lead to illicit transitions.
A state machine transition is defined by Boolean expressions over state variables paired with program fragments that
change them. State variables are made up from a history of operations and events by the device. If the reference monitor
detects an illegal state, the driver is terminated and the corresponding device is reset by the RVM.

As monitored memory reads and writes are limiting performance, unmonitored memory irrelevant to the DSS can be
specified. Also rate limits by using counters and timers can be assigned to prevent starvation of processes or hardware.
Rate limits also work with devices on a shared interrupt line. If a driver wants to allocate memory, performs I/O, or
exits, the reference validation mechanism sends corresponding events to the reference monitor which checks it’s DSS for
permission. Interrupts sent by devices are treated similarly.
To test the performance of the framework, drivers from Linux 2.4.22 were ported to Nexus user space drivers and tested on both systems. Additional changes to the driver source code were necessary to work with DSS efficiently. Device drivers of an Intel i810 sound card, Intel e1000 network card, USB UHCI controllers, USB mice, and USB discs were implemented on the Nexus operating system. Nexus and Linux 2.4.22 were installed as dual boot on a Pentium 4 3.0 GHz. Network tests included an Athlon X2 2.4 GHz system running Linux 2.6.22 as remote host. The network card driver and the sound card driver were tested in five different configurations: As in-kernel driver on both Linux and Nexus and as user-mode driver on Nexus without reference monitor, with trivial reference monitor, and with the fully functional reference monitor.

Benchmarks included network performance, interrupt handling times, CPU utilization and basic I/O operations. Network performance was measured by sending UDP packets at constant rates from and to the remote host using the network benchmark tool Iperf [15]. Also network round-trip time was tested using the ping command. To tweak network performance, the network driver was optimized by merging several operations into a single system call. The i810 sound card was used to measure the latency of the interrupt handling. CPU utilization was tested with a 30 sec. video stream involving both Ethernet card and sound card driver. The video averaged 1071 Kbps and was played with mplayer [20]. Tests of basic I/O operations included unmonitored memory, monitored memory, port I/O, interrupts and memory mapped I/O. The counting of unmonitored memory operations of the network driver stalled the performance, therefore this result was estimated. The USB UHCI controller, USB mouse, and USB disc were added to this test.

In order to test robustness two procedures were used: First, malicious drivers were simulated by randomly interfering communication between the driver and the reference validation mechanism. There were 31 perturbation tests with the trivial reference monitor and 1200 with the fully functional reference monitor. Tests were aborted when the device could be damaged. Second, drivers with manipulated code of six known attacks were installed. The attacks included interrupt handling, direct kernel communication and DMA.

### 4.2 Virtual Machine-based isolation

In the following we are looking at four different approaches that utilize virtualization to establish strong isolation among system components. In the iKernel architecture [57] untrusted device drivers run in their own virtual machine and uses lightweight communication stubs and shared memory for communication. LeVasseur uses the microkernel-based L4 to isolate untrusted drivers in virtual machines and uses many optimization techniques to keep the resource overhead at the minimum. Using the virtual machine architecture Xen Transparent Fault Tolerance moves drivers to Xens isolated driver domains and extends Xen with a driver fault detection and recovery mechanism. VEXE'DD uses Microsoft’s Virtual PC to isolate untrusted drivers in virtual machines, but was resigned when attempting to implement the demands of this approach.

#### 4.2.1 iKernel

Published in 2007 iKernel [57], which stands for isolation Kernel, is another approach which isolates drivers in dedicated virtual machines. With this approach no modification or recompilation of device drivers is required. Furthermore, it takes advantage of Intel’s VT-x technology [61], a hardware support for virtualization. Intel’s VT-x enforces isolation between the VMs and the host operating system and allows direct communication between the device driver inside the VM and the device. VT-x does not support DMA and interrupt handling, but its successor VT-d, which was not yet available by the time iKernel was developed. Each driver can be moved to its virtual machine or several can be collocated in a single one.

As shown in figure 4 there are two different kernel types in the iKernel architecture: the host kernel, which hosts the standard operating environment and the driver kernel, a minimized kernel hosting the driver in the virtual machine. Communication between the host kernel and the driver kernels is done by shared memory, which is strictly isolated from the system memory. For this purpose, the host kernel and the virtual machines are extended with lightweight communication stubs. The host and the virtualized drivers individually set the communication format how requests and responses are written to the shared memory. However, though the authors give an example of the communication format used by their implemented LED device, the format and interoperation is not further described. Merely the communication format is described as a common messaging format, which the VM and the host driver stubs agree on to interoperate.

When the driver is installed, it is moved to a newly built virtual machine. Its driver kernel allocates the shared page as a statically allocated memory region. Whenever the host driver stub needs to access a hardware resource, it writes its request to the shared memory region in above mentioned agreed common messaging format. The driver kernel periodically checks for requests and translates them to the real functions of the hosted driver.

The prototype uses the KVM virtual machine [21] supplied with a Linux 2.6.20 Ubuntu distribution that is used as host as well as guest operating system. Further, the guest kernel is stripped down to a minimum to reduce resource consumption of each VM. As a module of KVM, QEMU [34] is used for hardware emulation. Furthermore, the KVM guest mode has been modified to allow direct hardware access within a guest operating system. An Intel Core2Duo machine, a Dell Precision 390, was used to measure the latency of requests from the host. To test this concept, a parallel port...
driver was moved to a VM and communicated with an attached LED device. Unfortunately, the authors thought only of lowering the latency instead of testing the isolation with faulty or malicious driver scenarios.

4.2.2 LeVasseur

Another virtual machine-based isolation has been introduced in 2004 by LeVasseur et al. [32]. Their approach moves untrusted device drivers to separate virtual machines executed in user mode. Each driver can be moved to an individual single VM or grouped with others in a dedicated VM.

The hypervisor hosts the operating system and user programs which communicate with the VM-isolated drivers. It also shares and protects the memory and I/O ports between the VMs. The virtual machine monitor (VMM) is responsible for resource management and establishing the virtualization layer. It can reside with full privileges inside the hypervisor or outside with no privileges communicating through a specialized interface.

Within the virtual machine (VM) the driver and its native operating system (subsequently referred to as DD/OS) is executed. The driver does not need any modification to run in a DD/OS. Each DD/OS is only aware of itself and its corresponding device. The DD/OS is extended with an interface to allow the driver to communicate with its device. Also, the DD/OS hosts a server-like interface module for receiving and responding to external requests. The interface module is an abstraction layer that emulates functions of the kernel as well as the device driver and translates client requests. Clients can be the hypervisor, user programs, or drivers. If properly implemented, the interface module can be reused for multiple drivers. Shared resources like the PCI bus are accessed only by one DD/OS which communicates with all other DD/Os via its server module. Additionally the interface module manages memory pages and uses interrupts like hardware devices.

The DD/OS is highly optimized to keep the resource overhead at a minimum. Besides page sharing and swapping unused pages to disk, the DD/OS are further enhanced by memory ballooning, special treatment of zero pages, and compression for inactive pages. Memory ballooning is a special technique where the VMM claims physical memory back from the guest operating systems. It dynamically inflates or deflates the guest VMs memory space forcing the VM to swap its memory to disc when inflating [64].

Another problem of hosting multiple VMs are time critical operations and preemption. To avoid time-outs or malfunctioning of the device the VMM manages time scheduling and preemption to preserve time assumptions of the guest operating system. For DMA access, the VMs are categorized into three levels of trust: the client of the DD/OS is untrusted, both DD/OS as well as the client are untrusted and DD/OS and client distrust each other. All levels require that page translations have to stay constant and DD/OS memory must be statically allocated. Therefore, the aforementioned page swapping and memory compression is limited for machines with DMA hardware that can handle page faults. Otherwise, a DMA operation could overwrite invalid memory segments. The latter two levels have further DMA restrictions enforced by the hypervisor. If the DD/OS is untrusted, the hypervisor gives DMA permissions and monitors them. When DD/OS and the client distrust each other, the client reserves its own memory. Then, after a successful validation by the DD/OS as well as the hypervisor the DD/OS safely performs the DMA operation.

However, a wrongly implemented DD/OS can bypass the memory protection by performing DMA to physical memory. The authors state that with the help of special hardware like memory management units (IO-MMU) this can be prohibited. Since I/O-MMUs do not support multiple address contexts or simultaneous use of memory mappings, time multiplexing
for selected drivers has been implemented. To reduce the bus bandwidth overhead and delivery latency, time multiplexing was enhanced by dynamic adaptions of the bus utilization and device activity.

In case a VM needs another VM’s resource, it raises a communication interrupt in the other VM and signals a completion interrupt when done. When a user program needs access to an isolated driver, the VMM undertakes the communication interrupts. If a driver malfunction is detected, the virtual machine is rebooted and resets the driver to its initial state. Provided the DD/OS is still in a function, it is possible to negotiate a soft reboot, enabling the completion of any open tasks by the VM and the completion or abortion of outstanding DMA operations. A driver may be not able to recover its device, if a hardware reset of the device is needed. The virtual machine manager caches client requests until the driver is restarted or returns an error message.

To test the prototype, the paravirtualization component of the L4 microkernel platform is used. Two Linux versions, 2.4.22 and 2.6.8.1, were ported to L4. To use the L4 mechanisms, like receiving interrupts and scheduling, the kernel of the 2.4 version was ported almost automatically by the L4Linux adaption [46]. In contrast, 3000 lines of code were manually added to the kernel of the 2.6 version.

The VMM is executed as L4 user-level task. To compare performance and resource consumption, both Linux versions were installed on identical machines as native operating systems without virtualization. For CPU utilization, network and disk performance tests, an Intel Pentium 4 2.8 GHz with an Intel 82540 Gigabit Ethernet adapter and a Maxtor SATA disk with 120 gigabyte and 7200 rpm was used. To evaluate IO-MMU time multiplexing, an AMD Opteron 1.6 GHz with 8111 chipset was used. The size of each disk and network DD/OS was measured by uncompressing a kernel source tree to disk and by sending and receiving tests with the ttcp command [51]. DD/OS instances of the PCI bus, IDE controller and an Intel e1000 Ethernet card were implemented on the Linux 2.6 kernel test machine to test the memory optimizations like ballooning and page compression.

The PostMark benchmark [28] measured sharing files over the network and snapshots of all three DD/OS instances were analyzed. The Snapshots were also checked for uncompressed as well as compressed page duplicates to examine memory sharing. The hardware memory management unit (IO-MMU) was included to test scheduling and context switching. The performance of drivers consolidated in one DD/OS versus drivers hosted in individual DD/OS versus original, native device driver environments was investigated. This was performed for the disk and network drivers and additionally to the ttcp network tests the Netperf benchmark [27] was used. Disk performance was tested with a self-designed streaming benchmark which directly accessed the disk partition.

To test CPU utilization the number of DD/OS instances was increased. Except for the first DD/OS containing the PCI bus driver, the additional DD/OSs were idle and hosted the Intel e1000 network driver. The performance of applications was studied by using the Postmark benchmark [28]. An e-mail server was emulated on the client machine using file storage on a network files system (NFS) server. The client machine running the benchmark was an Intel Pentium 4 1.4 GHz with 256 megabyte memory, Intel 82540 Gigabit Ethernet adapter. The NFS server used the isolated driver framework and was configured as in the above mentioned performance tests. One thousand files ranging from 500 to 1000 bytes size and 10000 file transactions were configured for this benchmark. Besides the comprehensive performance testing, no fault tests were applied.

4.2.3 Transparent Fault Tolerance

Published in 2010, this technique [25] is the currently most to date virtual machine based approach. It is based on the virtual machine software Xen 3.0 [9] where the hypervisor is set between the hardware and the virtual machines called domains. In Xen, the first domain called domain0 contains an interface for controlling the hypervisor and guest domains. Version 3.0 introduced special guest domains called isolated device driver domains (IDD) which are shown shown in figure 5. IDDs allow hosting of unmodified drivers and direct communication with their corresponding devices. The guest domains, which host the operating system and user processes, can directly send requests to and receive responses or from the IDDs. Each IDD is based on a commodity operating system like Windows or Linux.

Besides the unmodified native device drivers, IDDs also host backend drivers which process and translate requests from the guest domains. Backend drivers communicate via an I/O descriptor ring with a corresponding frontend driver inside the guest domain. The I/O descriptor ring is a circular data structure with four location pointer indices for requests and corresponding responses. The backend driver as well as the frontend driver have each one pointer to insert (producer pointer) and one pointer to detach (consumer pointer) data into or from the I/O descriptor ring. The backend driver, If the frontend driver receives a response, it assumes that the backend driver as well as the native driver processed the request correctly. Since shared memory is not allowed except for the I/O ring, the grand table mechanism is used to manage permissions to shared memory.

Besides the I/O ring, guest domains and IDDs are connected with event channels for asynchronous notifications. They are used to tell the backend driver that a request is waiting to be processed, or to tell a frontend driver that there is a response waiting. A database called Xenstore contains all backend drivers information including configuration and status. When a frontend driver is loaded, it queries the Xenstore for its corresponding backend driver.
Kernel crashes and malfunctions constitute the considered IDD fault types. IDD crashes are detected and cleaned up by the hypervisor via domain scheduling mechanisms. To spot IDD malfunctions a detection mechanism called Driver VM Monitor (DVM) was introduced in this approach. The DVM resides in the hypervisor and checks the state of an IDD periodically. This period is adapted to the number of how many times an IDD is scheduled. At every check the I/O ring and the physical interrupt are logged and stored at the hypervisor stack. By comparing the current state with the logged information, the DVM inspects if the IDD is malfunctioning.

To avoid the misinterpretation of a busy IDD, not responding in a timely manner, as being in a failed state, a corresponding threshold value can be adjusted. As the workload of guest domains continuously changes, a dynamic threshold formula has been designed to take this into consideration. The physical interrupt state is additionally logged and included as an IDD load indicator in this formula. Also the I/O ring pointer integrity of the frontend driver is checked by the DVM.

Because only I/O Ring information is checked by the DVM, asynchronous I/O operations like receiving a network packet cannot be detected by the DVM. As these I/O operations are triggered from the outside of the machine, the DVM cannot determine whether an IDD is in a fault state or has no incoming data. Therefore, since most of device accesses involve interrupts, the detection formula was extended to include interrupt requests. However, if a fault corrupts a data page of a network packet, the DVM is not able to detect such a fault. The authors admit that the DVM can theoretically be extended by respective integrity checks, but only at the cost of high overhead.

A Driver VM Hand-off (DVH) mechanism was implemented to perform recovery from IDD failures. The DVH seamlessly switches from a failed IDD to a working backup IDD by preparing the I/O ring pointers and redirecting them to the backend driver of the working IDD. After the hand-over process the unfinished I/O requests are resubmitted to the working IDD. Besides the DVH two additional interfaces allow triggering the DVH process: one is used by the domain0 or the hypervisor; the other one is used by the guest domain allowing third party monitoring tools to trigger the DVH process.

The prototype was installed on an Intel Core2 Duo 2.33 GHz processor with two gigabyte of memory using Xen 3.0.4.1 and a paravirtualized Linux 2.6.16.33 kernel. The memory space for IDDs and guest operating systems were set to 256 megabyte each. The evaluation tests are focused on a network driver Intel gigabit Ethernet card solely. For this network IDD two I/O rings were used: one for transmitting and one for receiving. Several servers with an Intel Core2 Quad 2.4 GHz processor with four gigabyte memory were used for network communication.

Tests were aimed to examine the performance overhead and correctness of the DVM fault detection mechanism as well as latency and data loss when using the DVH hand-over mechanism. The audit function of the proxy software httperf [40] was used to determine the detection frequency of the DVM and to evaluate the delay of a fault state detection. For fault detection tests the authors used the same injection tool previously used for the evaluation of Nooks [56] and the RIO file cache [44]. 27 types of fault injections are supported by this tool. Additionally, the IDD kernel and the native device driver were manually modified with twelve fault types to better see the behavior of the detection mechanism. Though false negative cases did not occur, the DVM can result in false positives. The iperf benchmark [15] was used to evaluate the dynamic threshold formula in terms of false positives.

To test the recovery delay, DVH-induced delays were compared to latencies caused by the complete reloading of the backend driver and two different configurations of the Linux bonding driver [60]. The Linux bonding driver sums up several installed Ethernet driver and displays them as single device to the upper layer, thereby enabling individual driver
failures to be transparently masked using other driver instances. For each recovery method, a self-designed network benchmark streamed 12000 TCP packets per second and 21000 UDP packets per second. The data size of the packets was configured to eight bytes. Besides the measurement of the recovery delay, the loss of packets during recovery was also monitored for all tests.

The system was tested with a realistic workload using online transaction processing (OLTP) with MySQL [47], a relational database management system (RDBMS) software, and the performance benchmark tool sysbench [30], which requested about 130 database transactions per second. Additionally, the Xen kernel source was copied via SSH, NFS and downloaded via FTP. These workloads were executed in a guest domain with a network IDD communicating with a native server. Also, the execution time for each workload was logged and compared by the diff command. Furthermore, the number of guest domains was increased to assess the influence of the number of VMs on the execution time of OLTP. Finally, the detection and recovery mechanisms were evaluated using fault injection. To examine the overhead of DVM and DVH, the I/O, CPU, and memory overhead were investigated using several benchmarks. The results were compared to using a guest domain not subject to fault injections. Benchmarks included Httperf [40], the grep command for I/O tests, and the stream benchmark [36] for stressing the processor.

4.2.4 VEXE’DD

In 2005 Erlingson et al. published an approach called virtual extension environments for device drivers (VEXE’DD) [17]. VEXE’DD is a derivative of VEXE which generally isolates extensions from host applications. Using Microsoft’s Virtual PC, this architecture isolates untrusted device drivers in virtual machine and is therefore classified as virtual machine-based isolation. Because of difficulties with implementing required changes and the release of other approaches for virtual machine-based isolation, the paper has been issued as an incomplete report on the lessons learned during the implementation attempt and is therefore excluded from the evaluation in the next chapter. For the sake of completeness I have chosen to include VEXE’DD in this work.

A modified version of Virtual PC allowing device drivers to access hardware and to use direct memory access and other related I/O functionality was created for isolation. Also, optimizations for running multiple instances of virtual machines without occupying too much system resources were applied. Because of the lack of exclusive hardware control and the difficulties to apply the modifications, the authors admit that Virtual PC is not an appropriate platform for device driver isolation.

The VEXE’DD architecture uses containers to isolate the device drivers from the kernel and applications. The master container holds the kernel and the user programs and virtual containers are executing the device drivers. Virtual containers also hold copies of the kernel currently running in the master container. It is also possible to execute other operating system kernels depending on the communication interface between the containers and the compatibility of the drivers’ features. Drivers do not need to be rewritten or recompiled to run in a container.

In the master container the device driver is replaced by a safe mediator, which maintains system invariants and communicates with the virtual container. The safe mediator makes use of the container contract, a state machine, and
allows only calls, which trigger transitions that don’t lead to system crashes. It also checks any responses from the virtual container for safety. Inspired by [66], interrupts are caught within the master container and the relevant virtual container’s memory is mapped by the safe mediator. However, DMA is not monitored in this solution, because the driver communicates DMA commands in an incomprehensible way. To recognize this incomprehensible driver communication the authors suggest expanding VEXE to standard interface libraries or even to user-mode applications.

When a new device is detected by the master container operating system’s hardware detection mechanism, VEXE-DD creates a safe mediator emulating the driver instantiation and creates a virtual container for the untrusted driver. The identified resources are excluded from the master container and mapped to the virtual container. The shim, the counterpart of the mediator, receives requests from the master container and translates them to the isolated device driver. The virtual container also contains a contract state machine which is based on the kernel running inside the container. Container contracts are therefore constructed once for a given operating system version.

If the safe mediator detects a device driver failure or the virtual container is corrupted, its virtual container is restarted. Furthermore, the safe mediator can re-initialize the driver to a known state before the failure occurrence. Although the authors mention that they tested a floppy disk driver and used Windows XP, results or benchmarks of a prototype were not included in the paper.

### 4.3 Hardware-based isolation

RingCycle isolates drivers by using the ring model of the x86 architecture and is the only hardware-based approach found.

#### 4.3.1 RingCycle

This approach [29] enforces isolation by moving drivers from the kernel ring to lower privilege rings of the x86 architecture. Violations are detected by hardware and therefore the approach is labeled as hardware-based isolation. The ring model of the x86 architecture features four numbered privilege domains called protection rings. Ring 0, where the kernel is running, has most privileges and ring 3 holding the user programs the least. Rings 1 and 2 are not used in commodity operating systems like Linux and Windows.

If a program wants to access a certain resource, its privilege level is checked by the hardware. If the privilege level is not sufficient, a general protection fault exception is thrown. Privilege checks are required on memory access, I/O port access, control transfers and special instructions. RingCycle transfers device drivers from ring 0 to the unused rings 1 and 2. Ring 1 can contain trusted drivers that have control over the processor interrupt flags whereas drivers in ring 2 are fully untrusted. To support virtual memory isolation, a segment is taken from the user memory space and used for the middle rings. Hence, drivers cannot access kernel space anymore.

In the ring model code running can only return to its ring level and higher. To return to lower ring level like ring 0, drivers are run in separate threads called driver threads. Driver threads are similar user-mode threads, but are executed in the middle rings, use the middle rings address space and inherit the capabilities of the user process which called the driver routine. Because creating and deleting threads cost time and performance, a driver thread pool maintains driver threads. When a driver is installed, its thread is added to the pool until the driver is unloaded or the driver is requested by another user process. In case a driver is needed, its driver thread is pulled from the pool to execute the necessary routine, and upon completion it is sent back to the pool. The kernel manages the driver threads by removing the driver thread from the pool, marking it runnable and putting it back to the pool after the driver sends a system call back. Also communication between the user program and the driver is handled by the kernel. To call a code segment on a higher privileged level, RingCycle uses call gates [24] as function wrappers, a special instruction of the x86 architecture for this purpose.

RingCycle uses ten call gates, where each call gate has a specific number of parameters. Depending of the amount of parameters of a function, a call gate with the same count is used. To avoid the rewriting of drivers to use call gates, a driver API (DAPI) is provided that renders calls into the kernel in a header file. The DAPI consists of many pre-processor macros to reduce the amount of redundant code. To pass on requests from user programs the kernel formerly pointed to data structures in kernel space. Modifications of the kernel are needed to use the driver’s virtual memory segment to map the corresponding page of the kernel structure. In the prototype these pages could be modified by the driver, but the author states that these pages can be set to read-only and all writes could be verified by the kernel at the expense of performance.

Because these memory sharing operations are causing many page faults, a memory map caching algorithm was implemented to increase performance. If the kernel tries to map a page into the driver memory segment, the algorithm checks whether it is already mapped. For this purpose, a simple linked list of mapped pages and which drivers use a mapped page are maintained by the kernel.

If the memory isolation is violated in RingCycle, the kernel kills the causing thread and removes it from the pool, displays an error message, and informs the user program. Given the possibility that multiple driver threads of the same driver are killed, one thread is marked as reserved and is used for proper unloading. Depending on proper exiting and the quality of the driver, it is possible to reload the driver.
If the unloading of the faulty driver also fails, the kernel forces it to be unloaded and to remain non-reloadable until a reboot.

The prototype was installed on a Pentium II 330 MHz and on a Pentium 4 3.2 GHz machine. The real time clock driver of Linux was used to test the RingCycle framework and analyze the control flow. Micro-benchmarks specialized in monitoring assembler instructions were used to measure the time of interrupts and call gates. A dummy interrupt handler, a dummy call gate entry and a special driver taking time stamps and calling the dummy functions were implemented. The tests were run several times and the average was taken as result.

Macro-benchmarks were used to measure the overhead and the time for activating and deactivating driver threads. A special user program was implemented to interact with a fake device and taking time stamps. For each test the dummy driver was moved to another ring. The tests involved only the Pentium 4 machine and were run with and without the memory map caching algorithm.

4.4 Language-based isolation

In this section two language-based approaches will be introduced: Singularity [53] and SafeDrive [70]. Both approaches use type safety to establish strong isolation.

4.4.1 Singularity

This approach [53] is the basis for isolation in the Singularity operating system, which is a prototype of Microsoft Research focusing on type-safe languages and strong process isolation [39]. It was published in 2006. Except for the hardware abstraction layer (HAL) the operating system relies on type safe languages for isolation and therefore this technique is categorized as language-based isolation. C# was extended to Sing# to support type safety and static analysis. Code that does not fulfill or attempts to circumvent type safety is not allowed. Furthermore, the language was extended by I/O and resource requirement declarations so that Singularity can detect the hardware and software preconditions the driver requires. The language extensions also allow for the detection of resource conflicts, an ordering of the drivers at boot time and the provision of guarantees for resources used by the driver. Because of this information, device drivers are called self-describing artifacts.

Singularity executes each driver in a separate software isolated process (SIP). SIPs are isolated domains inside the kernel space which are protected by type safety. Drivers inside SIPs can be stopped and restarted without affecting other parts of the operating system. When installing a driver, the driver's code is compiled from Microsoft Intermediate Language (MSIL) to native code by the Bartok compiler [23]. While compiling, type safety rules and static analysis are applied to enforce system policies.

When installed, a type safe verified runtime of each driver is statically linked as trusted runtime and executed with full privileges. This runtime handles interrupts, memory and direct memory access channels, and other communications with the hardware through an abstraction environment. All processes in Singularity communicate via bidirectional channels. Messages on a channel are defined value types and the formats and protocols of the messages are determined in a contract. The contract is also used to verify valid sequences of sent messages. Each channel endpoint owns a public name which is managed by global namespace server so that clients can easily find and connect to channels.

Singularity was extended to manage the installation of device drivers: To allow the operating system to check provided guarantees and to determine priorities, an abstraction for special treatment of drivers was created. An abstraction consists of a set of resources like namespaces and channels and of a set of declarative policies regarding these resources. The abstraction is represented by a manifest and a protected tree folder: The manifest is an extended markup language (XML) file that lists all resources and policies related to the driver and can be generated automatically by using MSIL custom attributes. The authors implemented an extra compiler step that checks and verifies those attributes, but they admit that stricter rules would be required to prohibit accidentally placed attributes. The protected tree folder consists of the driver's executable code.

When a driver is installed, the system checks it's compatibility with the system, verifies by its signature information that its code was not manipulated except by the installer and compiles it to native code. At boot time, Singularity only enables drivers that can be satisfied with their resource needs. If a conflict is detected, the driver will not be added to the system or similar rules apply as described in system policies which are described in a set of structured XML declarations. Although the driver cannot communicate with the hardware except for using standardized objects, incorrect use of direct memory access can't be prevented due to hardware limitations.

The approach was tested on an AMD Athlon 64 3000+ with one gigabyte of memory. Only the average sequential disk read performance was tested and compared to Singularity without implemented checks, Windows XP SP2, Fedora Core 4 Linux 2.6.11 and FreeBSD Unix Version 5.3. Using a WD2500JD SATA hard disk with a capacity of 250 gigabyte, 7200 rpm and command queuing disabled, 512 megabyte of sequential data was read single threaded on the same partition. The results are reported as averages of seven conducted tests. Neither the isolation of drivers inside software isolated processes nor the detection of fault injections were tested.
4.4.2 SafeDrive

Also published in 2006, SafeDrive [70] is another approach, which uses language based techniques for detecting and recovering software extensions. SafeDrive improves existing device drivers written in the C programming language with type checking and restart capabilities. For the detection of memory and type errors, a system called Deputy is introduced. Deputy uses type annotations in header files and shared structures to enforce memory and type safety. Based on CCured [43], another type safety approach for C, the Deputy compiler analyzes pointers, local and global variables, and other structs and automatically creates annotations. If the compiler cannot identify a variable, manual annotations by the programmer are required. After the annotation phase, the Deputy compiler tests and optimizes the code.

Because of significant invasion to extensions the current prototype does not verify operations like memory allocation, deallocation, and mapping operations. Therefore, SafeDrive aims not to protect against malicious exploitation attempts. Also type casts and dependencies are sometimes misinterpreted by Deputy. Although the authors experienced only about one percent of such casts, Deputy offers a mechanism to suppress any errors for particular casts or assignments.

Wrappers use the run-time checks inserted by the compiler to verify data structures and kernel calls for correct usage. All kernel functions that require update tracking are wrapped by SafeDrive. To return successfully from a driver function, the programmer needs to identify entry points, which are then extended with wrapper functions. These wrapper functions manage the failure handling by returning a pre-specified error message and set a failed state flag. While the flag is set, the error message is returned immediately at any call to the driver.

An update tracking module maintains a linked list of all updates made by the driver to the kernel state. Each stored update is linked with compensation information and is indexed with a hash table. The compensation information contains a pointer to a function and data needed to undo the update. The kernel needed modifications to allow tracking of events not directly reported by the driver.

Two separate pools store updates recorded by the tracking module. One pool stores updates for the driver, the other stores current processor and control path states like spinlock acquirements. To properly tolerate a failed driver, the re-entry points of call backs form kernel functions and their context is recorded. After the driver fails, control jumps directly to the recorded re-entry points allowing to execute any kernel code still on the stack. Similar to the handling of domain termination in lightweight remote procedure calls (LRPC) [4], this technique allows kernel functions to finish executing while skipping any driver code still on the stack. If a run-time check fails, it calls the SafeDrive recovery mechanism. After the error details are logged for debugging, SafeDrive cleans up the driver module, reverses updates to the kernel state, restores important kernel invariants and optionally restarts the failed extension. Because the driver is completely restarted, restoring local invariants of the driver is not necessary.

The prototype did not support a failed driver’s state and session reconstruction. Thus, a failed driver is restarted to its initial state. The authors recommend the ShadowDriver approach of the Nooks project to extend their recovery mechanism. All tests were executed using the Linux 2.6.15.5 kernel. Drivers of the following six peripherals were used to test the recovery mechanism of SafeDrive: An Intel e1000 Ethernet card, a Broadcom Tigon3 Ethernet card, an USB mass-storage driver with a Sandisk 256 megabyte flash drive, an Intel 8x0 onboard soundcard, a Creative Audigy2 sound card and an NVidia GeForce4 Ti 4200.

First, the recovery mechanism was tested with randomly injected faults created by an optional extension tool of the Deputy compiler. Based on empirical studies of kernel code, this tool injects seven categories of faults. Because the Deputy error detection catches errors at compile time, the authors opted out the use of binary fault injection tools like the one used in the evaluation of the Rio file cache [44]. The Intel Ethernet adapter was used solely for the evaluation of the fault detection mechanism. The test included downloading and verifying an 89 megabyte file and executing 20 iterations of each of the seven categories with and without SafeDrive.

Several benchmarks were used to evaluate the performance cost of run-time checks, update tracking and capturing of the re-entry points. The benchmark results were then compared with native drivers. For network performance tests, the Intel and Broadcom Ethernet adapters were tested on a dual Intel Xeon 2.4 Ghz machine. The benchmark netperf [27] was used to test network throughput and sar [67], a system monitoring tool included in many Unix distributions, measured the CPU utilization. The USB-storage driver was tested on an IBM Thinkpad T42 notebook equipped with an Intel Pentium-M 1.6 Ghz processor. The tar command [41] was used to extract a Linux 2.2.26 kernel source code tar archive and to measure the CPU utilization at the same time.

The two sound cards were tested on two different machines: The Intel 8x0 sound card was tested as onboard peripheral on the IBM Thinkpad T42 notebook. The Creative sound card was built in an Intel Pentium II 450Mhz machine. While a thirty minute 44.1 Khz wave file was playing, the hardware performance measuring tool oprofile [26] captured how much kernel time on behalf of the sound card drivers was spent. The application aplay and the library als a [31] sound library were used to playback the wave file. At last, the Nvidi a graphics card was tested on an Intel Pentium 4 2.4 GHz machine. Again oprofile was used to measure the CPU usage while initializing and closing an X Window session.
5 Evaluation and Comparison

In this section I will evaluate and compare the approaches summarized in chapter 4 and on the basis of the criteria introduced in chapter 3. Each subsection begins with a table providing a simplified overview on the literature-based evaluation. The table also includes the average rate for each approach, which were derived from the results of the six criteria. (−) is given in situations, where the respective prototype or implementation’s behavior is inadequate (with the respect of the given criterion), (o) means an adequate behavior, (+) is used to rate an above average behavior and (++) marks outstanding behavior of the prototype. The results will also be discussed in the following subsections. The last subsection gives an overview and summarizes the results of the published performance tests.

5.1 Sandbox-based Isolation

The results of the comparative investigation for the sandbox-based approaches are shown in the following table 2:

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Intrusiveness</th>
<th>Fault isolation</th>
<th>Recovery</th>
<th>Extensibility</th>
<th>Portability</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGI</td>
<td>o</td>
<td>0</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>o</td>
</tr>
<tr>
<td>Microdrivers</td>
<td>++</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>o</td>
</tr>
<tr>
<td>Nexus</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>o</td>
</tr>
<tr>
<td>Nooks</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

- **Performance** As all four solutions included tests of Ethernet adapters, the main focus lies on the results of their benchmark tests. Although the network throughput does not change significantly, the processor activity rises approx. 12 percent with BGI and even 29 percent with the Nooks architecture. Although, other driver tests with Nooks had an acceptable average CPU overhead increase of 1.2 percent (with BGI an average of 6.4 percent was measured), Nooks is rated (−) and BGI (o), because of the directly comparable results.

The network CPU utilization of Nexus was evaluated while performing the streaming video test and increased from 0.3 percent to 0.8 percent. Including the results of the sound card and other kernel operations, the cpu time is only 1.9 percent higher. However, the throughput of UDP send benchmark dropped by up to fifty percent at a packet size below 800 bytes. Also the average roundtrip time of the network round-trip test took half the time longer with the isolated driver. Thus, Nexus is rated (+). With a very slight CPU performance increase, the Microdrivers approach scores best with respect to this criterion. The slight performance increase in throughput by Nooks and Microdrivers is attributed to the batching of the Nooks’ object tracker and the optimizations done by the Microdrivers compiler.

- **Intrusiveness**

None of the investigated solutions is working without any changes to the kernel or driver code, so none of the approaches gets a (+++) rating. The Nooks prototype changes about one thousand lines of the kernel code. And only drivers that use kernel data structures directly, for example by returning and updating kernel pointers, must be changed. Because of the comparatively small changes to the kernel and the driver code, Nooks is ranked (+).

The Microdrivers prototype requires driver source code to transform the driver into a Microdriver. Therefore it uses a code generator which automatically adds an average of 4791 lines of code for the k-driver components and 1959 lines of code for the u-driver components, but can generate up to 37900 additional lines of code for driver transformation. Furthermore, 1 to 6 percent lines of code require manual annotations. This is a small percentage, but it means that a fully automated generation is impossible.

The authors of BGI only mention that code changes to the kernel and the drivers are necessary. Unfortunately, these changes and the amount of manual annotations are not reported. Furthermore, the mentioned kernel modifications were performed on a closed-source kernel. For these reasons Microdrivers and BGI are rated (o).

Nexus does not need any modifications to the kernel. But, when ported from Linux to Nexus, the tested drivers in Nexus have 0.45 to 5.2 percent of lines added or changed to support interrupt handling and DMA operations. Also the expense of adapting the drivers is not further discussed. Therefore Nexus is rated (+)

- **Fault isolation**

Except for Nexus, none of the other frameworks detected all fault injections. Nooks detected 99 percent of all injected faults (512 out of 517). BGI successfully detected and contained 96 percent of injected faults when
testing the FAT file system and only 40 percent within the network tests. MicroDrivers detected 84 percent when evaluating the 8139 gigabit network card and 61 percent were detected with the 8139C+ Ethernet card. According to the detection success Nexus and Nooks were rated (++). Because of the varying results the other approaches were rated (o).

- **Recovery support**

  Microdrivers didn’t implement a recovery mechanism in the prototype, so that Microdrivers is ranked (-). Currently, the RPC monitor can only detect but not prevent data corruption. However, the authors state, that an adoption of ShadowDrivers or Safedrive would improve the reliability provided by Microdrivers considerably.

  Nooks’s recovery mechanism restarts all tested drivers, which were manually injected, successfully. The applications using the driver even worked without malfunctioning after the online recovery of the driver. Also 94 percent of the automatically detected failures were recovered. This gives Nooks a high score for this criterion.

  For BGI 50 faulty drivers were injected to test its recovery capabilities. 21 drivers that raised a BGI exception were recovered successfully except for two. The authors state that these two could not be recovered due to limitations of the prototype. Also, the recovery mechanism currently supports only PnP drivers. Although Nexus is able to unload and load device drivers, the recovery mechanism was not tested. Like Microdrivers Nexus is therefore rated (-).

- **Extensibility**

  Because of the code modifications applied to the kernel or driver code, the extensibility of the approaches is limited. Executing similar drivers in the same Nook can facilitate adding new devices to the system. However, eleven of the sixteen tested drivers needed wrappers that were unique for the respective driver. Although the wrappers can be created automatically and shared for driver classes, an integration of a new driver needs to be examined by the programmer for individual or new functions.

  The Microdrivers approach needs to adapt the device drivers, but the provided tool DriverSlicer helps to automate the transformation. Even though only a small amount of code needs manual annotations, an incorrect or missing annotation could crash the Microdriver. Despite a comparatively high degree of automation, Microdrivers’s transformation process therefore still relies fundamentally on a (small) set of manual modifications for every considered driver. Multiple drivers can access the same u-driver component, which can be helpful for the integration of new drivers. Additionally, u-drivers can be written and debugged with common user level programming tools, which arguably supports the implementation of new drivers.

  BGI needs modifications to the kernel and the driver code. Similarly to Microdrivers, a tool helps to transform existing driver code to compatible code but equally entails the same risks of manual annotations. Because of the use of the Windows Plug and Play (PnP) manager, the applicability of the framework is limited to drivers with PnP support and therefore to Windows. Similar to Nooks, drivers need to be adapted for the Nexus approach. Also the authors state that writing a DSS based on an existing driver is tempting and estimate the effort for developing a DSS to take one to five days depending on the documentation and familiarity with the DSS language.

- **Portability**

  Except Nexus, all frameworks require modifications to the kernel and are therefore ranked (-). Although this needs to be done only once, it is not possible to transfer a framework to all commodity operating systems, as the kernel code in many cases is not published. Except for Nooks, the amount of changes and which parts of the kernel need modifications are not mentioned so that the expense of portability actually cannot be ranked. The authors of Nooks plan to utilize hardware memory protection techniques, which may improve performance but may require more complex modifications in order to support different hardware architectures. Though Nexus is a microkernel-based operating system, the RVM mechanism could be ported to any operating system which supports hardware-implemented process isolation according to the authors. However, because the effort of porting cannot be estimated, Nexus is ranked (o).

### 5.2 Virtual Machine-based Isolation

The results of the comparative investigation for the virtual Machine-based approaches are shown in the table 3.

- **Performance**

  iKernel only evaluated the latency of sending a request to the device. Also, only a self-build parallel port LED device was evaluated with this approach. Though the authors are satisfied with the measured latency, a wider evaluation
Table 3: Ratings of virtual machine–based isolation approaches

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Intrusiveness</th>
<th>Fault isolation</th>
<th>Recovery support</th>
<th>Extensibility</th>
<th>Portability</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>iKernel</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>LeVasseur</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>o</td>
<td>o</td>
<td>-</td>
<td>o</td>
</tr>
<tr>
<td>TFT</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

with usual hardware peripherals would be more useful. Because of the insufficient performance tests, iKernel is rated lowest in the comparison. Further, iKernel’s memory consumption per virtual machine is 27 megabytes, which, compared to the other approaches, is not adequate.

In contrast to iKernel, LeVasseur reports extensive performance tests. LeVasseur’s approach performed adequately regarding the disk and network throughput. However, the high CPU utilization for network and disk operations gave a rather bad result: Network CPU utilization ranges between 60 and 100 percent, disk CPU utilization between 20 and 90 percent. Admittedly, the fixed CPU utilization cost of executing a steady state DD/OS are very low. They start at 0.6 percent and any additional DD/OS constantly cost only 0.2 percent. Also, the extensive optimizations of memory consumption achieve very good results: The memory consumption of three DD/OS were reduced from 52 megabyte to only 6 megabyte. However, because of the very high CPU utilization, LeVasseur is rated (o).

The performance of Transparent Fault Tolerance was also tested extensively. The results of the performance and resource consumption tests show that the overhead is negligible: When evaluating the I/O workload, the DVM caused only two percent of performance overhead. The cumulated memory consumption for each VM takes only 120 bytes. The tested network and disk throughput also make no significant difference. Unfortunately, the results were given normalized, so that a comparison of this benchmark to other approaches is not possible. Furthermore the authors focused on the execution time for realistic workloads and multiple guest domains: while OLTP needs about 300 milliseconds execution time difference when executed in one virtual machine, every added VM adds 360 milliseconds in average. The copy operation via SSH takes more than 700 milliseconds execution time difference, marking the peak of the realistic workload test. Because of the moderate execution latency, Transparent Fault Tolerance did not receive the highest score.

• Intrusiveness

By using wrapper stubs, iKernel does not need to modify the kernel or device drivers. Only the virtual machine implementation KVM was modified to allow direct hardware access by the driver. Therefore, iKernel is rated (+++) with respect to this criterion. To deploy Transparent Fault Tolerance, no kernel and driver modifications are needed. Like iKernel, this approach is rated with the highest score.

The authors of the LeVasseur approach stated that device drivers need to be recompiled to let them control its device via a DD/OS pass-through mechanism. But how the recompilation is executed is not further discussed. Furthermore, the Linux versions used to implement this approach needed modifications: 3000 additional lines of code were added to the Linux 2.6 kernel to make use of the L4 mechanisms. For these reasons LeVasseur is rated (-).

• Fault isolation

iKernel and LeVasseur did not test the fault detection. Also, the communication between the device driver and its device are not monitored by these approaches. iKernel refers to future incorporation of hardware support like Intel’s VT-d to monitor device communication and DMA. Although LeVasseur successfully isolated DMA operations by using the I/O-MMU, the effectiveness of the fault isolation was not tested. Because of the missing robustness tests both frameworks are rated (-). Transparent Fault Tolerance successfully detected all 270 fault injections which resulted to system crashes or driver malfunctions. This is rewarded with the highest grade.

• Recovery support

The LeVasseur and iKernel solutions recover a faulted driver via rebooting the driver’s virtual machine. However, drivers that rely on a hardware reset to reinitialize their devices may not be able to recover their devices and a reboot of the machine or the peripheral may be required. Like for the fault isolation, the recovery mechanisms were not tested. Because of the missing recovery evaluation, both approaches got the lowest rating.

With a very low recovery delay, no packet loss with TCP and barely any packet loss with UDP, Transparent Fault Tolerance successfully recovered the network driver. However, this recovery design requires that a second VM
offering the same driver functionality is running to perform the I/O ring hand-over. The authors recommend a fast VM instantiation technique [65] to instantly clone a backup VM, if a running backup VM is not possible for performance reasons. This clone mechanism would take 300 milliseconds delay to initiate a guest domain. Because of this restriction, Transparent Fault Tolerance is rated (+).

• Extensibility
Because drivers do not need any modifications to be isolated in the iKernel and Transparent Fault Tolerance approaches, a simple integration is assumed. However, the effort to create communication stubs within iKernel and among backend and frontend drivers in Transparent Fault Tolerance are not discussed by the authors. In LeVasseur, the effort related to the recompilation of a driver is not further detailed. Thus, the expense to extend all three approaches is hard to judge and this missing information leads to a low rating.

• Portability
Because all three approaches rely on their virtualization solutions, portability depends on how commonly these are supported. The iKernel architecture is based on a Linux kernel running KVM and minimal Linux kernels running in the driver's virtual machines, so iKernel's portability is rated average. As Xen does support a wide range of operating systems, Transparent Fault Tolerance is rated (+). Also, iKernel and Transparent Fault Tolerance are restricted to virtualization software incorporating hardware virtualization support. However, no modifications to the kernel and device drivers are needed, which are good conditions for porting this solution even to closed source code operating systems and drivers. This saves iKernel and Transparent Fault Tolerance from a lower rating. LeVasseur's is based on its virtualization solution is based on the L4 paravirtualization environment. Although a L4Linux adaption [46] is continuously being updated, other operating systems are not yet supported. Also, because of the kernel modifications, LeVasseur's approach is rated (-).

5.3 Hardware-based Isolation
The results of the comparative investigation for the hardware-based approaches are shown in the table 4.

<table>
<thead>
<tr>
<th>Table 4: Ratings of hardware–based isolation approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
</tr>
<tr>
<td>Ringcycle</td>
</tr>
</tbody>
</table>

• Performance
RingCycle performs very well: Only two microseconds of overhead is measured compared to the native driver. However, only a real time clock driver was tested for performance overhead. Like the iKernel approach, a wider evaluation with common hardware peripherals would be more useful. The author promises that in future work support for more existing drivers will be added. But since the initial publication in 2006 no updates were found.

• Intrusiveness
This approach needs to modify the kernel and to recompile the driver. Unfortunately, the amount of lines changed or added is not mentioned. Although recompilation of a driver is almost done automatically by the compiler, kernel modifications must be done manually. Consequently RingCycle is rated with (-).

• Fault Isolation
Because RingCycle only enforces memory isolation regarding the ring privilege level, it is rated (o). Although the kernel memory is protected, drivers can corrupt memory pages inside their ring or any lower ring. Also, RingCycles interrupt handling may invoke deadlocks. At last, detection and robustness tests of this approach by injecting bugs or executing faulty drivers were conducted. Yet here, too, the author is aware of the limitations and refers to future work.

• Recovery support
Recovery support is not implemented in this approach and is therefore marked with the lowest grade. If a violation is detected, the driver is unloaded. Like Microdrivers, the author recommends a combination with another
approach like Nooks to improve the detection of faulty drivers and their recovery. Because of the modifications needed to be added to the kernel and the drivers source code must be available for recompilation, the extensibility of RingCycle is assessed as very low. Original drivers still work within the RingCycle approach and are executed with full privileges in Ring 0, but they don't benefit from the isolation.

- **Portability**
  Because of the kernel modifications, this approach is limited to open source operating systems. Also, because of the ring protection layer, this approach is limited to operating systems which support the x86 hardware architecture. However, because the x86 architecture is widely spread, this limitation is reasonable for commodity operating systems.

### 5.4 Language-based Isolation

The results of the comparative investigation for the hardware-based approaches are shown in the table 5.

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Intrusiveness</th>
<th>Fault isolation</th>
<th>Recovery support</th>
<th>Extensibility</th>
<th>Portability</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SafeDrive</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>Singularity</td>
<td>+</td>
<td>o</td>
<td>+</td>
<td>-</td>
<td>o</td>
<td>not rated</td>
<td>o</td>
</tr>
</tbody>
</table>

- **Performance**
  An overhead of only 0.6 percent was measured for Singularity compared to Singularity without this framework when evaluating the performance impact by testing the average sequential read disk performance. Furthermore, compared to the other tested operating systems Windows XP, Linux, and FreeBSD, the disk throughput of this approach can be judged as competitive. This gives Singularity a high mark for this criterion. Although more hardware is mentioned in the paper like a NVidia nForce4 Ethernet card and a S3Trio graphics card, such further benchmarks have not been published. Because of this, Singularity misses the highest score.

  In contrast to Singularity, SafeDrive tested the performance of its framework intensively. But the results are not sufficiently convincing to get a good rating. The average throughput of the ethernet cards and the usb storage driver results in -4.3 percent and the throughput reaches a maximum of -17 percent. The CPU utilization resulted in 9.3 percent in average, reaching its maximum at 23 percent. Therefore, SafeDrive is rated (o).

- **Intrusiveness**
  To adapt drivers to SafeDrive and Singularity, the source code of drivers is required and intense manipulations need to be made. The Singularity approach also extended the operating system to treat device drivers with a higher priority as well as to provide resource requirements and guarantees. Unfortunately, the amount of code changes have not been reported so that it is difficult to judge the intrusiveness. Driver developers must invest additional time to create the manifests and resource declarations. But, because of the support of the Bartok compiler, the usage of the high-level language SING# as well as the usage of the XML standard for the Manifest programmers can use a wide variety of validation and development utilities. However, this does not save Singularity from the lowest score, as changes to the kernel and the driver code are significant in this rating.

  SafeDrive also needs to make modifications to the kernel and to device drivers resulting in the same score as Singularity. Although kernel code changes are penalized, only 1084 total lines of kernel code needed to be changed. Changes are required for approximately one to four percent of existing driver code. Although this is a small amount, many annotations, wrapper functions, and recovery-related changes must be made manually. The authors intend to automatize this process in future work.

- **Fault isolation**
  Singularity prohibits the installation of drivers that cannot initialize successfully or have not declared required hardware resources. Additionally, software isolated processes (SIP) prevent drivers from manipulating kernel or other memory pages. Assembly-level instructions, like interrupt handling and direct memory access, are handled in a trusted runtime with hardware and memory isolation. Drivers can only communicate with their hardware through standardized channel objects. However, Singularity cannot prevent the incorrect use of DMA, which can
unintendedly overwrite the wrong memory partitions. The authors consider that future work and updates of Singularity might solve this problem. Unfortunately, error detection and type safety violations during run-time were not inspected, therefore Singularity is rated (o).

That type safety mechanisms are not prohibiting faulty drivers at run-time is also shown by SafeDrive’s error detection tests: out of 140 runs with injected faults, 54 were detected by SafeDrive. Thirteen of the detected problems were discovered by the Deputy compiler at compile time. Although more than half of the injected faults did not crash the driver or result in an unexpected behavior, a higher detection rate would be desirable and the approach is rated (o) in this criterion.

- **Recovery support**

  Singularity lacks recovery support and is therefore rated (-). SafeDrive at least recovers to the driver’s initial state. Two out of twenty one non-crash driver malfunctions and five out of ten innocuous errors triggered the recovery subsystem and recovered the driver successfully. It is not clearly reported if the recovery process was necessary when triggered. However, because of the partial success of recovery, SafeDrive is rated (o).

- **Extensibility**

  As stated in the intrusiveness criterion discussion, the driver’s source code and additional work is needed to adapt a driver to the Singularity approach. However, the support of generic drivers saves this approach from obtaining the lowest score. If a specialized driver can not be loaded for any reason, the peripheral can still be used by a generic driver. However, generic drivers do not provide as many features and hardware optimizations as specialized drivers. Because each driver must be manipulated manually, as stated in the intrusiveness criterion, this approach is rated (-).

- **Portability**

  Because the Singularity isolation approach only suits the Singularity operating system and is not intended to be ported this approach is not rated for this criterion. SafeDrive is restricted to drivers written in C. According to the authors all drivers in Linux were written in this language and hence porting SafeDrive is not an obstacle in the Linux / Unix world. Nevertheless, code changes to the kernel and to the drivers makes porting of this approach to closed source operating systems like Windows very difficult or even impossible. SafeDrive is therefore rated (o).

5.5 Overview of the Approaches

In this subsection an overview of the approaches as well as their fault detection and recovery is given. Also the published results of the performance tests are summarized as far as they were reported, adaptable or comparable. Table 6 shows the year of publishing, current website and taxonomy of the discussed approaches. Table 7 summarizes the ratings of all evaluated approaches. The used operating systems and modifications to drivers and kernels are listed in table 8. Table 9 shows the an overview of the fault detection and recovery. The tables 10, 11 and 12 show the results of performance tests.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Publishing year</th>
<th>Website</th>
<th>Taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGI</td>
<td>2009</td>
<td><a href="http://research.microsoft.com/">http://research.microsoft.com/</a></td>
<td>Sandbox-based</td>
</tr>
<tr>
<td>MicroDrivers</td>
<td>2009</td>
<td>-</td>
<td>Sandbox-based</td>
</tr>
<tr>
<td>Nexus</td>
<td>2008</td>
<td><a href="http://www.cs.cornell.edu/People/egs/nexus/">http://www.cs.cornell.edu/People/egs/nexus/</a></td>
<td>Sandbox-based</td>
</tr>
<tr>
<td>Nooks</td>
<td>2002</td>
<td><a href="http://pages.cs.wisc.edu/~swift/nooks/">http://pages.cs.wisc.edu/~swift/nooks/</a></td>
<td>Sandbox-based</td>
</tr>
<tr>
<td>iKernel</td>
<td>2007</td>
<td>-</td>
<td>Virtual-machine based</td>
</tr>
<tr>
<td>LeVasseur</td>
<td>2004</td>
<td><a href="http://l4ka.org/">http://l4ka.org/</a></td>
<td>Virtual-machine based</td>
</tr>
<tr>
<td>TFT</td>
<td>2010</td>
<td>-</td>
<td>Virtual-machine based</td>
</tr>
<tr>
<td>RingCycle</td>
<td>2006</td>
<td><a href="http://www.acm.uiuc.edu/projects/RingCycle/">http://www.acm.uiuc.edu/projects/RingCycle/</a></td>
<td>Hardware-based</td>
</tr>
<tr>
<td>SafeDrive</td>
<td>2006</td>
<td><a href="http://ivy.cs.berkeley.edu/safedrive/">http://ivy.cs.berkeley.edu/safedrive/</a></td>
<td>Language-based</td>
</tr>
<tr>
<td>Singularity</td>
<td>2006</td>
<td><a href="http://research.microsoft.com/">http://research.microsoft.com/</a></td>
<td>Language-based</td>
</tr>
</tbody>
</table>
Table 7: Summary of the Ratings

<table>
<thead>
<tr>
<th>Approach</th>
<th>Performance</th>
<th>Intrusiveness</th>
<th>Fault isolation</th>
<th>Recovery support</th>
<th>Extensibility</th>
<th>Portability</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGI</td>
<td>0</td>
<td>0</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Microdrivers</td>
<td>++</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nexus</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Nooks</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>iKernel</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LeVasseur</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>TFT</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RingCycle</td>
<td>+</td>
<td>-</td>
<td>o</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>SafeDrive</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Singularity</td>
<td>+</td>
<td>0</td>
<td>+</td>
<td>-</td>
<td>0</td>
<td>not rated</td>
<td>0</td>
</tr>
<tr>
<td>Approach</td>
<td>OS/VM</td>
<td>Driver needs modification?</td>
<td>Driver lines added/changed</td>
<td>Kernel needs modification?</td>
<td>Kernel lines added/changed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------</td>
<td>----------------------------</td>
<td>----------------------------</td>
<td>----------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGI</td>
<td>Windows Vista Enterprise SP1</td>
<td>yes</td>
<td>n/a</td>
<td>yes</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MicroDrivers</td>
<td>Linux 2.6.18.1</td>
<td>yes</td>
<td>up to 37900 LoC</td>
<td>yes</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nexus</td>
<td>Nexus</td>
<td>yes</td>
<td>0.45 - 5.2%</td>
<td>no</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nooks</td>
<td>Linux 2.4.10</td>
<td>yes</td>
<td>-</td>
<td>yes</td>
<td>924 lines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iKernel</td>
<td>Linux 2.6.20/KVM with QEMU</td>
<td>no</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeVasseur</td>
<td>Linux 2.6.8.1/L4</td>
<td>yes (recompilation)</td>
<td>n/a</td>
<td>yes</td>
<td>3000 (Linux 2.6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeVasseur</td>
<td>Linux 2.4.22/L4</td>
<td>yes</td>
<td>n/a</td>
<td>yes</td>
<td>3000 (Linux 2.6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFT</td>
<td>Linux 2.6.16.33/Xen 3.0.4.1</td>
<td>no</td>
<td>-</td>
<td>no</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RingCycle</td>
<td>Linux 2.6.7</td>
<td>yes (recompilation)</td>
<td>n/a</td>
<td>yes</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SafeDrive</td>
<td>Linux 2.6.15.5</td>
<td>Yes</td>
<td>1 - 4 %</td>
<td>yes</td>
<td>1084 lines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singularity</td>
<td>Singularity</td>
<td>yes</td>
<td>n/a</td>
<td>yes</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approach</td>
<td>Fault Detection (Percent)</td>
<td>Recovery success (Percent)</td>
<td>Recovery mechanism</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------</td>
<td>---------------------------</td>
<td>-----------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGI</td>
<td>39.8% (102 of 256 faults) intelpro</td>
<td>90% (19 out of 21)</td>
<td>Recovers only PnP drivers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>96 % (166 of 173 faults) fat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MicroDrivers</td>
<td>84% (95 of 113 faults) 8139too</td>
<td>-</td>
<td>Not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>61% (64 of 105 faults) 8139cp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nexus</td>
<td>100% (1200 of 1200 faults)</td>
<td>Not tested</td>
<td>Reloads Driver</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Noooks</td>
<td>99% (512 of 517 faults)</td>
<td>94% (127 out of 135)</td>
<td>Last known good state</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iKernel</td>
<td>Not tested</td>
<td>Not tested</td>
<td>Restarts VM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeVasseur</td>
<td>Not tested</td>
<td>Not tested</td>
<td>Restarts VM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFT</td>
<td>100% (270 of 270 faults)</td>
<td>100% (1 out of 1)</td>
<td>Last known good state</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RingCycle</td>
<td>Not tested</td>
<td>-</td>
<td>Not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SafeDrive</td>
<td>83 % (54 of 65 faults)</td>
<td>9.5% (2 out of 21)</td>
<td>Initial state</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Detected by Compiler 24% (13 of 54)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singularity</td>
<td>Not tested</td>
<td>-</td>
<td>Not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Overview of Fault Detection and Recovery
### Table 10: Results of Network Performance Tests

<table>
<thead>
<tr>
<th>Approach</th>
<th>Benchmark-Tool</th>
<th>Ethernet card</th>
<th>Configuration</th>
<th>Network Performance Test Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Benchmark</td>
</tr>
<tr>
<td>BGI</td>
<td>ttcp</td>
<td>Intel Pro/100</td>
<td>TCP configuration</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Message size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>socket buffer</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>packet size</td>
<td>Benchmark</td>
</tr>
<tr>
<td>Notentor xframe</td>
<td></td>
<td></td>
<td>TCP configuration</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Message size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>socket buffer</td>
<td>Benchmark</td>
</tr>
<tr>
<td>MicroDrivers</td>
<td>netperf</td>
<td>RealTek RTL-8139 Gigabit</td>
<td>TCP configuration</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Message size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>packet size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>packet size</td>
<td>Benchmark</td>
</tr>
<tr>
<td>Nexus</td>
<td>iperf</td>
<td>Intel Pro/1000 Gigabit</td>
<td>TCP configuration</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Message size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>socket buffer</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>packet size</td>
<td>Benchmark</td>
</tr>
<tr>
<td>LeVasseur</td>
<td>ttcp</td>
<td>Intel 82540m Gigabit</td>
<td>TCP configuration</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Message size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>socket buffer</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>packet size</td>
<td>Benchmark</td>
</tr>
<tr>
<td>Safedisk</td>
<td>iperf</td>
<td>Intel e1000-Gigabit</td>
<td>TCP configuration</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Message size</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>socket buffer</td>
<td>Benchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>packet size</td>
<td>Benchmark</td>
</tr>
</tbody>
</table>

**Notes:**
- CPU Utilization is relative to native.
### Table 11: Results of Storage Performance Tests

<table>
<thead>
<tr>
<th>Approach</th>
<th>Hardware</th>
<th>File description</th>
<th>Filesize</th>
<th>Benchmark</th>
<th>native</th>
<th>isolated</th>
<th>Throughput</th>
<th>native</th>
<th>isolated</th>
<th>CPU Utilization (%)</th>
<th>relative to native</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGI</td>
<td>Seagate Barracuda ST3100825AS 100GB 7200rpm SATA Disk</td>
<td>10000 files (FAT file system test)</td>
<td>n/a</td>
<td>Postmark, kernrate</td>
<td>n/a</td>
<td>n/a (-12.3%)</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a (+10.01%)</td>
<td>1.10x</td>
<td></td>
</tr>
<tr>
<td>pDisk</td>
<td>SanDisk 2GB USB2 0.0 flash drive</td>
<td>100 files</td>
<td>n/a</td>
<td>Postmark, kernrate</td>
<td>n/a</td>
<td>n/a (+0%)</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a (+0.91%)</td>
<td>1.01x</td>
<td></td>
</tr>
<tr>
<td>Microwriter</td>
<td>USB Disk</td>
<td>Sequential file</td>
<td>140 MB</td>
<td>n/a</td>
<td>n/a</td>
<td>585.44 KB/s</td>
<td>578.95 KB/s (-1.1%)</td>
<td>4.92%</td>
<td>7.01% (+42%)</td>
<td>1.42x</td>
<td></td>
</tr>
<tr>
<td>Nools</td>
<td>80MB 7200 rpm IDE Disk</td>
<td>788 C source code files</td>
<td>n/a</td>
<td>make/gcc compiler</td>
<td>n/a</td>
<td>n/a (-2%)</td>
<td>99%</td>
<td>100%</td>
<td>(+1%)</td>
<td>1.01x</td>
<td></td>
</tr>
<tr>
<td>LeVasseur</td>
<td>Maxtor 6Y120MB 130GB 7200rpm 5/4A disk</td>
<td>Sequential file</td>
<td>n/a</td>
<td>self build streaming benchmark</td>
<td>n/a</td>
<td>90.75 MB/s</td>
<td>n/a (nearly identical)</td>
<td>99%</td>
<td>100%</td>
<td>(+1%)</td>
<td>1.01x</td>
</tr>
<tr>
<td>Safedisk</td>
<td>SanDisk 256 MB USB flash drive</td>
<td>Linera 2.2, 26 source code tar ball</td>
<td>82 MB</td>
<td>untar</td>
<td>n/a</td>
<td>1.64 MB/s</td>
<td>1.64 MB/s</td>
<td>5.50%</td>
<td>6.8% (+24%)</td>
<td>1.28x</td>
<td></td>
</tr>
<tr>
<td>Singularity</td>
<td>Western Digital WD2500JD 250GB 7200rpm 5/4A disk</td>
<td>Sequential file</td>
<td>512 MB</td>
<td>average sequential disk read performance</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a (+20% to +90%)</td>
<td>1.26x</td>
<td>1.9x</td>
<td></td>
<td>1.06x</td>
</tr>
</tbody>
</table>
Table 12: Results of other Hardware Performance Tests

<table>
<thead>
<tr>
<th>Approach</th>
<th>Hardware</th>
<th>Benchmark</th>
<th>CPU Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>native</td>
<td>isolated</td>
</tr>
<tr>
<td>Sound cards</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SafeDrive</td>
<td>Intel 8x0</td>
<td>oprofile</td>
<td>3.79</td>
</tr>
<tr>
<td></td>
<td>Creative Audigy 2</td>
<td>44.1 Khz wave file</td>
<td>9.1</td>
</tr>
<tr>
<td>Nexus</td>
<td>Intel i810</td>
<td>n/a</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>1071 Kbps video file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graftic cards</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SafeDrive</td>
<td>Nvidia GeForce4 Ti 4200</td>
<td>oprofile</td>
<td>12.13</td>
</tr>
<tr>
<td>Other Hardware</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ringcycle</td>
<td>Real time clock driver</td>
<td></td>
<td>n/a</td>
</tr>
<tr>
<td>iKernel</td>
<td>Parallel port LED device</td>
<td></td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6 Conclusion and Future Work

The discussion of all considered approaches according to the presented criteria leads to the result, that none of them isolates driver faults completely, but all help to improve system reliability to some extend. As current monolithic operating systems do not feature any isolation for kernel extensions, any of the discussed techniques can help improving system reliability. Except for Nooks and Transparent Fault Tolerance, all approaches have an average score. In sandbox-based approaches, the Microdrivers approach achieves good performance, Nexus scores best on fault detection and Nooks has the best recovery mechanism. But Microdrivers does not isolate the kernel extension completely (there is still an unisolated fraction residing in the kernel) and Nexus does not include a recovery mechanism. Therefore combining Microdrivers, Nexus and Nooks may extend the missing properties of each framework. Although a more intense testing of the recovery mechanism would be desirable, Transparent Fault Tolerance scores best in the category of virtual machine-based approaches. The hardware-based and language-based approaches score badly, because of missing fault isolation tests, missing recovery mechanism implementations or poor operational performance. Also, iKernel and LeVasseur focussed more on improving performance than evaluating fault isolation. It is hard to understand, why the authors neglected fault isolation, as it is the main reason for developing this approach. Therefore the system’s reliability should have higher importance than performance.

Additionally to providing fault isolation, some approaches can help to develop and debug drivers: by porting uncritical function calls to the user-mode, Microdrivers allows programmers to take advantage of user-mode development tools. By using a high level language, this also applies to the Singularity approach. BGI found twenty-eight bugs like incorrect uses of the kernel interface, although these drivers have been tested and analyzed extensively by verification tools. SafeDrive’s compiler detected 24% of injected faults, of which none were detected by GCC, helping to prevent bugs at runtime. Nooks’ and theoretically Nexus’ recovery mechanism can also be used to update drivers to new versions without restarting the system.

Summed up, there is no fully satisfying solution to the problem of fault isolation. Since Transparent Fault Tolerance performs best in the ranking with respect to the chosen criteria, this framework has to be recommended, although the recovery mechanism needs to be improved. In addition, Transparent Fault Tolerance is ranked as strong isolation besides Nexus as they fully isolate and monitor the driver including DMA.

To improve results of an evaluation and comparison in future work, I consider a test environment with the same drivers, operating system and hardware platform to provide objectively comparable results. Because of the fast growth and development of computer hardware, it is not possible to agree on a standard hardware environment. For sandbox-based and language-based approaches it is therefore advisable to use appropriate virtualization platforms to emulate hardware as done in the evaluation of Nooks for example. However, this is not suitable if hardware isolation mechanisms are used.

Furthermore, the development of a benchmark, which is taking the introduced criteria into account would be useful for the evaluation of future approaches. Also the binary injection tools used by Nooks, Transparent Fault Tolerance, and SafeDrive as well as the studies of most common bugs referred to by BGI may help to work out a useful fault detection benchmark. The benchmark PostMark, which was used by BGI, Transparent Fault Tolerance, and LeVasseur to evaluate real system behavior, may serve as a model for a fault isolation performance benchmark.
References


