Fault Isolation for Device Drivers

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum
Dept. of Computer Science, VU University Amsterdam, The Netherlands
E-mail: {jnherder, herbertb, beng, philip, ast}@cs.vu.nl

Abstract

This work explores the principles and practice of isolating low-level device drivers in order to improve OS dependability. In particular, we explore the operations drivers can perform and how fault propagation in the event a bug is triggered can be prevented. We have prototyped our ideas in an open-source multiserver OS (MINIX 3) that isolates drivers by strictly enforcing least authority and iteratively refined our isolation techniques using a pragmatic approach based on extensive software-implemented fault-injection (SWIFI) testing. In the end, out of 3,400,000 common faults injected randomly into 4 different Ethernet drivers using both programmed I/O and DMA, no fault was able to break our protection mechanisms and crash the OS. In total, we experienced only one hang, but this appears to be caused by buggy hardware.

Keywords: Operating Systems, Device Drivers, Bugs, Dependability, Fault Isolation, SWIFI Testing

"Have no fear of perfection—you'll never reach it."
Salvador Dalí (1904–1989)

1 INTRODUCTION

Despite recent research advances, commodity operating systems still fail to meet public demand for dependability. Studies seem to indicate that unplanned downtime is mainly due to faulty system software [13, 37]. A survey across many languages found well-written software to have 6 faults/KLoC; with 1 fault/KLoC as a lower bound when using the best techniques [16]. In line with this estimate, FreeBSD reportedly has 3.35 post-release faults/KLoC [5], even though this project has strict testing rules and anyone is able to inspect the source code.

It is now beyond a doubt that extensions, such as device drivers, are responsible for the majority of OS crashes. Even though extensions typically comprise up to two-thirds of the OS code base, they are generally provided by untrusted third parties and have a reported error rate of 3–7 times higher than other code [3]. Indeed, Windows XP crash dumps showed that 65–83% of all crashes can be attributed to extensions and drivers in particular [10, 26].

The reason that these crashes can occur is the close integration of (untrusted) extensions with the (trusted) core kernel. This violates the *principle of least authority* by granting excessive power to potentially buggy components. As a consequence, a malfunctioning device driver can, for example, wipe out kernel data structures or overwrite servers and drivers. Not surprisingly, memory corruption was found to be one of the main OS crash causes [35].

Fixing buggy drivers is infeasible since configurations are continuously changing with, for example, 88 new drivers per day in 2004 [26]. On top of this, maintainability of existing drivers is hard due to changing kernel interfaces and growth of the code base [29]. Our analysis of the Linux 2.6 kernel shows a sustained growth in LoC of about 5.5% every 6 months, as shown in Fig. 1. Over the past 4.5 years, the kernel has grown 49.2% and now surpasses 5.1M lines of executable code—largely due to device drivers, comprising 57.6% of the kernel or 3.0M lines of code.

While there is a consensus that drivers need to be isolated, e.g. [19, 20, 21, 36], the issue to be addressed in each approach is "Who can do what and how can this be done safely?" We strongly believe that least authority should be the guiding principle in any dependable design. "Every program ... should operate using the least set of privileges necessary to complete its job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs ... so that unintentional, unwanted, or improper uses of privilege are less likely to occur [31]."

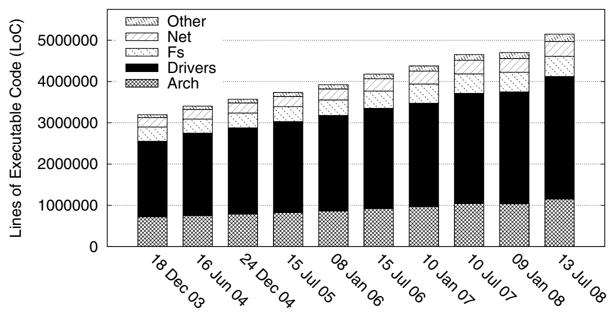

Figure 1: Growth of the Linux 2.6 kernel since its release.

1.1 Contribution and Paper Outline

In contrast to earlier work [17], this study addresses the fundamental issue of fault isolation for device drivers. The main contributions are (i) a classification of driver operations that are root causes of fault propagation, and (ii) a set of isolation techniques to curtail these powers in the face of bugs. We believe this analysis as well as the isolation techniques proposed to be an important result for any effort to isolate faults in drivers, in any OS. A secondary contribution consists of the full integration of our isolation techniques in a freely available open-source OS, MINIX 3.

MINIX 3 strictly adheres to least authority. As a baseline, each driver is run in a separate user-mode UNIX process with a private (IO)MMU-protected address space. This takes away all privileges and renders each driver harmless. Next, because this protection is too coarse-grained, we have provided various fine-grained mechanisms to grant selective access to resources needed by the driver to do its job. Different per-driver policies can be defined by the administrator. The kernel and trusted OS servers act as a reference monitor and mediate all accesses to privileged resources such as CPU, device I/O, memory, and system services. This design is illustrated in Fig. 2.

Rather than proving isolation formally [7], we have taken a pragmatic, empirical approach and iteratively refined our isolation techniques using software-implemented fault injection (SWIFI). After several design iterations, MINIX 3 is now able to withstand millions of faults representative for system code. Even though we injected 3,400,000 faults, not a single fault was able to break the driver's isolation or corrupt other parts of the OS. We did experience one hang, but this appears to be caused by buggy hardware.

This paper continues as follows. First, we relate our work to other approaches (Sec 2) and discuss assumptions and limitations (Sec. 3). Next, we introduce isolation techniques based on a classification of privileged driver operations (Sec. 4) and illustrate our ideas with a case study (Sec. 5). Then, we describe the experimental setup (Sec. 6) and the results of our SWIFI tests (Sec 7). Finally, we discuss lessons learned (Sec. 8) and conclude (Sec. 9).

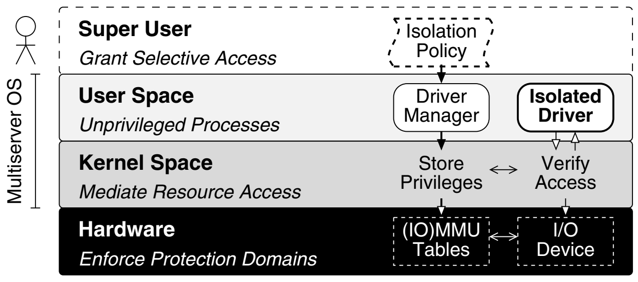

Figure 2: MINIX 3 isolates drivers in unprivileged processes.

2 RELATED WORK

Several other approaches that try to improve dependability by isolating drivers have been proposed recently. Below we survey four different approaches in a spectrum ranging from legacy to novel isolation techniques.

First, wrapping and interposition are used to run safely untrusted drivers inside the OS kernel. For example, Nooks [36] combines in-kernel wrapping and hardware-enforced protection domains to trap common faults and permit recovery. SafeDrive [38] uses wrappers to enforce type-safety constraints and system invariants for extensions written in C. Software fault isolation (SFI) as in VINO [32] instruments driver binaries and uses sandboxing to prevent memory references outside their logical protection domain. XFI [8] combines static verification with run-time guards for memory access control and system state integrity.

Second, virtualization can be used to run services in separate hardware-enforced protection domains. Examples of virtual machine (VM) approaches include VMware [34] and Xen [9]. However, running the entire OS in one virtual machine is not enough, since driver faults can still propagate and crash the core OS. Instead, a multiserver-like approach is required whereby each driver runs in a paravirtualized OS in a dedicated VM [21]. The client OS runs in a separate VM and typically accesses its devices by issuing virtual interrupts to the driver OS. This breaks VM isolation by introducing new, ad-hoc communication channels.

Third, language-based protection and formal verification can also be used to isolate drivers. For example, OKE [1] uses a customized Cyclone compiler to instrument an extension's object code according to a policy corresponding to the user's privileges. Singularity [19] combines type-safe languages with protocol verification and seals processes after loading. The seL4 project [7] aims at a formally verified microkernel by mapping the design onto a provably correct implementation. Devil [24] is a device IDL that enables consistency checking and low-level code generation. Dingo [30] simplifies interaction between drivers and the OS by reducing concurrency and formalizing protocols.

Finally, multiserver systems like MINIX 3 encapsulate untrusted drivers in user-mode processes with a private address space. For example, Mach [12] experimented with user-mode drivers directly linked into the application. L^{4}Linux [14] runs drivers in a paravirtualized Linux server. SawMill Linux [11] is multiserver OS, but focuses on performance rather than driver isolation. NIZZA [15] supports safe reuse of legacy extensions for security-sensitive applications. In recent years, user-mode drivers were also used in commodity systems such as Linux [20] and Windows [25], but we are not aware of efforts to isolate drivers based on least authority and believe that these systems could benefit from the ideas presented in this work.

3 ASSUMPTIONS AND LIMITATIONS

In our research, we explore the limits on software isolation, rather than proposing hardware changes. Unfortunately, older PC hardware has various shortcomings that make it virtually impossible to build a system where drivers run in full isolation. However, now that modern hardware with support for isolating drivers is increasingly common—although sometimes not yet perfect—we believe the time has come to revisit design choices made in the past. For example, the following three hardware improvements enable building more dependable operating systems:

(1) To start with, older PCs have no means to protect against memory corruption by unauthorized direct memory access (DMA). Our solution is to rely on IOMMU support. Like a traditional MMU, which provides memory protection for CPU-visible addresses, the IOMMU provides memory protection for device-visible addresses. If a driver wants to use DMA, a trusted party validates the request and mediates setting up the IOMMU tables for the driver's device. We have used AMD's Device Exclusion Vector (DEV), but IOMMUs are now common on many platforms.

(2) Furthermore, the PCI standard mandates shared, level-triggered IRQ lines that lead to inter-driver dependencies, since a driver that fails to acknowledge a device-specific interrupt may block an IRQ line that is shared with other devices. We avoided this problem by using dedicated IRQ lines, but the PCI Express (PCI-E) bus provides a structural solution based on virtual message-signaled interrupts that can be made unique for each device.

(3) Finally, all PCI devices on the standard PCI bus talk over the same communication channel, which may lead to conflicts. PCI-E uses a point-to-point bus design so that devices can be properly isolated. However, hardware limitations still exist, as PCI-E is known to be still susceptible to PCI-bus hangs if a malfunctioning device claims an I/O request but never puts the completion signal on the bus.

In addition to improved hardware dependability, performance has increased to the point where software techniques that previously were infeasible or too costly have become practical. We build on the premise that computing power is no longer a scarce resource (which is generally true on desktops nowadays) and that most end users would be willing sacrifice some performance for improved dependability. Preliminary measurements comparing MINIX 3 against Linux and FreeBSD show an overhead of roughly 10–25%, but the performance can no doubt be improved through careful analysis and removal of bottlenecks. Independent studies have already addressed this issue and shown that the overhead incurred by modular designs can be limited to 5–10% [11, 14, 20, 22]. However, instead of focusing on performance, the issue we have tried to address is isolating untrusted drivers that threaten OS dependability.

4 ENFORCING LEAST AUTHORITY

This section first classifies the privileged operations drivers need and then presents per class the isolation techniques MINIX 3 employs to enforce least authority.

4.1 Classification of Driver Privileges

The starting point for our discussion is the classification of potentially dangerous driver operations shown in Fig. 3. At the lowest level, CPU usage should be controlled in order to prevent bypassing higher-level protection mechanisms. For example, consider kernel-mode CPU instructions that can be used to reset page tables or excessive use of CPU time by a driver that winds up in an infinite loop.

Unauthorized memory access is an important threat with drivers that commonly exchange data with other parts of the system and may engage in direct memory access (DMA). Indeed, field research has shown that memory corruption is one of the most important causes (27%) of system outages [35]. In 15% of the crashes the corruption is so severe that the underlying cause cannot be deduced [28].

It is important to restrict access to I/O ports and registers and device memory in order to prevent unauthorized access and resource conflicts. Programming device hardware is complex due to its low-level interactions and lack of documentation [30]. Especially the asynchronous nature of interrupt handling can be hard to get correct, as evidenced by the error IRQL_NOT_LESS_OR_EQUAL that was found to cause 26% of all Windows XP crashes [10].

Interprocess communication (IPC) allows servers and drivers running in separate protection domains to cooperate, but dealing with unreliable and potentially hostile senders and receivers is a challenge [18]. A related power built on top of the IPC infrastructure, which routes requests through the system, is requesting (privileged) OS services.

Privileges	Isolation Techniques
(Class I) CPU Usage	**See Sec. 4.2.1**
+ *Privileged instructions*	→ *User-mode processes*
+ *CPU time*	→ *Feedback-queue scheduler*
(Class II) Memory access	**See Sec. 4.2.2**
+ *Memory references*	→ *Address-space separation*
+ *Copying and sharing*	→ *Run-time memory granting*
+ *Direct memory access*	→ *IOMMU protection*
(Class III) Device I/O	**See Sec. 4.2.3**
+ *Device access*	→ *Per-driver I/O policy*
+ *Interrupt handling*	→ *User-level IRQ handling*
(Class IV) System services	**See Sec. 4.2.4**
+ *Low-level IPC*	→ *Per-driver IPC policy*
+ *OS services*	→ *Per-driver call policy*

Figure 3: Classification of privileged operations needed by low-level drivers and summary of MINIX 3's defense mechanisms.

4.2 Per-Class Isolation Techniques

We now describe how MINIX 3 isolates drivers. In short, each driver is run in an unprivileged UNIX process, but based on the driver's needs, we can selectively grant fine-grained access to each of the privileged resources in Fig. 3. We believe that UNIX processes are attractive, since they are lightweight, well-understood, and have proven to be an effective model for encapsulating untrusted code.

4.2.1 Class-I Restrictions—CPU Usage

Privileged Instructions All drivers are runs in an ordinary UNIX process with *user-mode* CPU privileges, just like normal application programs. This prevents drivers from executing privileged CPU instructions such as changing memory maps, performing I/O, or halting the CPU. Only a tiny microkernel runs with *kernel-mode* CPU privileges and a small set of kernel calls is exported to allow access to privileged services in a controlled manner.

CPU Time With drivers running as UNIX processes, normal process scheduling techniques can be used to prevent CPU hogging. In particular, we use a multilevel-feedback-queue scheduler (MLFQ). Processes with the same priority reside in the same queue and are scheduled round-robin. Starvation of low-priority processes is prevented by degrading a process' priority after it consumes a full quantum. Since CPU-bound processes are penalized more often, interactive applications have good response times. Periodically, all priorities are increased if not at their initial value.

Two additional protection mechanisms exist. First, the driver manager can be configured to periodically check the driver's state and start a fresh copy if it does not respond to heartbeat requests, for example, if it winds up in an infinite loop [17]. Second, a resource reservation framework is provided in order to provide more stringent temporal protection for processes with real-time requirements [23].

4.2.2 Class-II Restrictions—Memory Access

Memory References We use MMU-hardware protection to enforce strict *address-space separation*. Each driver has a private, virtual address space with a fixed size depending on the driver's requirements. The MMU translates CPU-visible addresses to physical addresses using the MMU tables controlled by the kernel. Unauthorized memory references outside of the driver's address space result in an MMU exception and cause the driver to be killed.

Drivers that want to exchange data could potentially use page sharing, but, although efficient, with page sizes starting at 4 KB the protection is too coarse-grained to share safely small data structures. Therefore, we developed the fine-grained authorization mechanism discussed next.

Copying and Sharing We allow safe data exchange by means of fine-grained, delegatable *memory grants*. Each grant defines a memory area with byte granularity and gives a specific other process permission to read and/or write the specified data. A process that wants to grant another process access to its address space must create a grant table to store the memory grants. On first use, the kernel must be informed about the location and size of the grant table. After creating a memory grant it can be made available to another process by sending an IPC message that contains an index into the table, known as a grant ID. The grant then is uniquely identified by the grantor's process ID plus grant ID. The receiver, say, B of a grant from A can refine and transfer its access rights to a third process C by means of an *indirect grant*. This results in a hierarchical structure as shown in Fig. 4. This resembles recursive address spaces [22], but memory grants are different in their purpose, granularity, and usage—since grants protect data structures rather than build process address spaces.

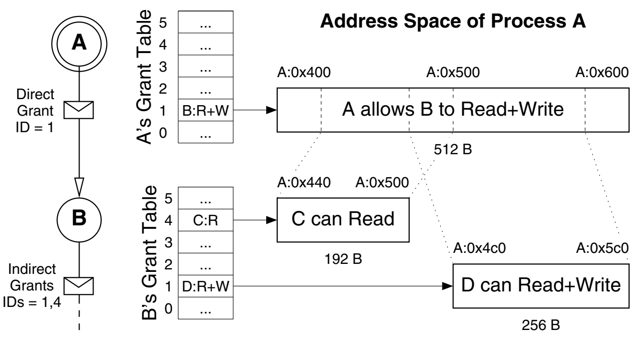

Figure 4: Hierarchical structure of memory grants. Process A directly grants B access to a part of its memory; C can access subparts of A's memory through indirect grants created by B.

The SAFECOPY kernel call is provided to copy between a driver's local address space and a memory area granted by another process. Upon receiving the request message, the kernel extracts the grant ID and process ID, looks up the corresponding memory grant, and verifies that the caller is indeed listed as the grantee. Indirect grants are processed using a recursive lookup of the original, direct grant. The overhead of these steps is small, since the kernel can directly access all physical memory to read from the grant tables; no context switching is needed to follow the chain. The request is checked against the minimal access rights found in the path to the direct grant. If access is granted, the kernel calculates the physical source and destination addresses and copies the requested amount of data. This design allows granting a specific driver access to a *precisely* defined memory region with perfect safety. If needed, certain non-copying page-level performance optimizations are possible for large pieces of memory.

Direct Memory Access DMA from I/O devices can be restricted in various ways. One way to prevent invalid DMA is to restrict a driver's I/O capabilities to deny access to the motherboard's DMA controller used by ISA devices and have a trusted DMA driver mediate all access attempts. However, this approach is impractical for PCI devices using bus-mastering DMA, since it requires each PCI device to be checked for DMA capabilities. Therefore, we relied on modern hardware where the peripheral bus is equipped with an IOMMU that controls all DMA attempts. Rejected DMA writes are simply not executed, whereas rejected DMA reads fill the device buffer with ones.

A driver that wants to use DMA needs to send a SETIOMMU request the trusted IOMMU driver in order to program the IOMMU. Only DMA into the driver's own address space is allowed. Before setting up the IOMMU tables the IOMMU driver verifies this requirement by checking the driver's memory map through the UMAP kernel call. It also ensures that the memory is pinned. When the DMA transfer completes, the driver can copy the data from its own address space into the address space of its client using the memory-grant scheme discussed above. An extension outside the scope of this paper is to use memory grants to program the IOMMU. This improves flexibility and performance, since a driver could safely perform DMA directly into a buffer in another process' address space.

4.2.3 Class-III Restrictions—Device I/O

Device Access Since each driver typically has different requirements, we associated each driver with an isolation policy that grants fine-grained access to the exact resources needed. Policies are stored in simple text files defined by the administrator. Upon loading a driver the driver manager reads the policy from disk and informs the kernel and trusted OS servers, so that the restrictions can be enforced at run-time. As an example, Fig. 5 shows the complete isolation policy of the Realtek RTL8139 Ethernet driver. Below we focus on device I/O (pci device), whereas access to system services (ipc and kernel) is discussed in Sec. 4.2.4.

```
driver rtl8139                    # ISOLATION POLICY
{
    pci device   10ec/8139
                 ;
    ipc          KERNEL PM DS RS
                 INET PCI IOMMU TTY
                 ;
    kernel       DEVIO IRQCTL UMAP MAPDMA
                 SETGRANT SAFECOPY
                 TIMES SETALARM GETINFO
                 ;
};
```

Figure 5: Per-driver policy definition is done using simple text files. This is the *complete* isolation policy for the RTL8139 driver.

The specification of I/O resources is different for PCI and ISA devices. For PCI devices, the keys pci device and pci class grant access to one specific PCI device or a class of PCI devices, respectively. Upon loading a driver the driver manager reports these keys to the trusted PCI-bus driver, which dynamically determines the permissible I/O resources by querying the PCI device's configuration space initialized by the BIOS. For ISA devices, the keys io and irq statically configure the I/O resources by explicitly listing the permissible I/O ports and IRQ lines in the policy. In both cases, the kernel is informed about the I/O resources using the PRIVCTL kernel call and stores the privileges in the process table before the driver gets to run.

If a driver requests I/O, the kernel first verifies that the operation is permitted. For devices with memory-mapped I/O, the driver can request to map device-specific memory persistently into a its address space using the MEMMAP kernel call. Before setting up the mapping, however, the kernel performs a single check against the I/O resources reported through PRIVCTL. For devices with programmed I/O, fine-grained access control to device ports and registers is implemented in the DEVIO kernel call and the vectored variant VDEVIO. If the call is permitted, the kernel performs the actual I/O instruction(s) and returns the result(s) in the reply message. While this introduces some kernel-call overhead, the I/O permission bitmap on x86 CPUs was not considered a viable alternative, because the 8-KB per-driver bitmaps would impose a much higher demand on memory and make context switching more expensive. In addition, I/O permission bitmaps do not exist on other architectures, which would complicate porting.

Interrupt Handling Although the lowest-level interrupt handling must be done by the kernel, all device-specific processing is done local to each driver in user space. This is important because programming the hardware and interrupt handling in particular are difficult and relatively error-prone [10]. Unfortunately, PCI devices with shared IRQ lines can still introduce inter-driver dependencies that violate least authority, as described in Sec. 3.

A user-space driver can register for interrupt notifications for a specific IRQ line through the IRQCTL kernel call. Before setting up the association, however, the kernel verifies the driver's access rights by inspecting the policy installed by the driver manager or the PCI bus driver. If an interrupt occurs, a minimal, generic kernel-level handler disables interrupts, masks the IRQ line that interrupted, notifies the registered driver(s) with an asynchronous HWINT message, and finally reenables the interrupt controller. This process takes about a microsecond and the complexity of reentrant interrupts is avoided. Once the device-specific processing is done, the driver(s) can acknowledge the interrupt using IRQCTL in order to unmask the IRQ line.

4.2.4 Class-IV Restrictions—System Services

Low-level IPC With servers and drivers running in independent UNIX processes, they can no longer make direct function calls to request system services. Instead, MINIX 3 offers IPC facilities based on message passing. By default, drivers are not allowed to use IPC, but selective access can be granted using the key ipc in the isolation policy. For example, the policy in Fig. 5 enables IPC to the kernel, process manager, name server, driver manager, network server, PCI bus driver, IOMMU driver, and terminal driver. The IPC destinations are listed using human-readable identifiers, but the driver manager retrieves the process IDs from the name server upon loading a driver. Then it informs the kernel about the IPC privileges granted using PRIVCTL, just like is done for I/O resources. The kernel stores the driver's IPC privileges in the process table and enforces them at run-time using simple bitmap operations.

As an aside, the use of IPC poses various other challenges [18]. Most notable is the risk of blockage when synchronous IPC is used in asymmetric trust relationships that occur when (trusted) system servers call (untrusted) drivers. MINIX 3 uses asynchronous and nonblocking IPC in order to prevent blockage due to unresponsive drivers. In addition, the driver manager periodically pings each driver to see if it still responds to IPC, as discussed in Sec. 4.2.1.

OS Services Because the kernel is concerned only with passing messages from one process to another and does not inspect the message contents, restrictions on the exact request types allowed must be enforced by the IPC targets themselves. This problem is most critical at the kernel task, which provides a plethora of sensitive operations, such as managing processes, setting up memory maps, and configuring driver privileges. Therefore, the last key of the policy shown in Fig. 5, kernel, restricts access to individual kernel calls. In line with least authority, the driver is granted only those services needed to do its job: perform device I/O, manage interrupt lines, request DMA services, make safe memory copies, set timers, and retrieve system information. Again, the driver manager fetches the calls granted upon loading the driver and reports them to the kernel using PRIVCTL. The kernel inspects the table with authorized calls each time the driver requests service.

Finally, the use of services from the user-space OS servers is restricted using ordinary POSIX mechanisms. Incoming calls are vetted based on the caller's user ID and the request parameters. For example, administrator-level requests to the driver manager will be denied because all drivers run with an unprivileged user ID. Since the OS servers perform sanity checks on all input, request may also be rejected due to invalid or unexpected parameters, just like is done for ordinary POSIX calls.

5 DRIVER ISOLATION CASE STUDY

We have prototyped our ideas in the MINIX 3 operating system. As a case study, we now discuss the working of the Realtek RTL8139 PCI driver, as sketched in Fig. 6. The driver's life cycle starts when the administrator requests the driver to be loaded, using the isolation policy shown in Fig. 5. The driver manager creates a new process and informs the kernel about the IPC targets and kernel calls allowed using the PRIVCTL call. It sends the PCI device ID to the PCI bus driver, which looks up the I/O resources of the RTL8139 device and also informs the kernel. Finally, only once the execution environment has been properly isolated, the driver manager executes the driver binary.

During initialization, the RTL8139 driver contacts the PCI bus driver to retrieve the I/O resources of the RTL8139 device and registers for interrupt notifications with the kernel using IRQCTL. Only the I/O resources in the isolation policy are made accessible though. Since the RTL8139 device uses bus-mastering DMA, the driver also allocates a local buffer for use with DMA and requests the IOMMU driver to program the IOMMU accordingly using SETIOMMU. This allows the device to perform DMA into only the driver's address space and protects the system against arbitrary memory corruption by invalid DMA requests.

During normal operation, the driver executes a main loop that repeatedly receives a message and processes it. Requests from the network server, INET, contain a memory grant that can be used with the SAFECOPY kernel call in order to read from or write to only the message buffers and nothing else. Writing garbage into INET's buffers results in messages with an invalid checksum, which will simply be discarded. The RTL8139 driver can program the network card using the DEVIO kernel call. The completion interrupt of the DMA transfer is caught by the kernel's generic handler and forwarded to the RTL8139 driver. The interrupt is handled in user space and acknowledged using IRQCTL. In this way, the driver can safely perform its task without being able to disrupt any other services.

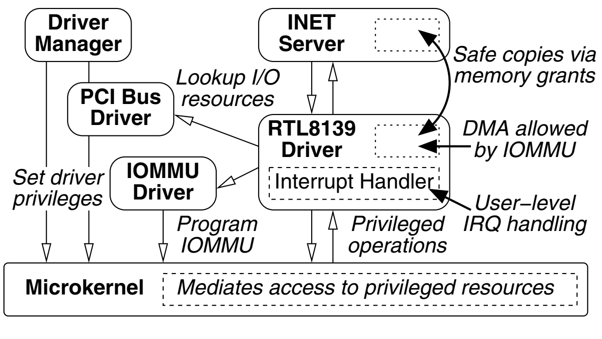

Figure 6: Interactions between an isolated RTL8139 PCI driver and the outside world in MINIX 3.

6 EXPERIMENTAL SETUP

We used software-implemented fault injection (SWIFI) to assess and iteratively refine MINIX 3's isolation techniques. The goal of our experiments is to show that faults occurring in an isolated driver cannot propagate and damage other parts of the system.

6.1 SWIFI Test Methodology

We have emulated a variety of problems underlying OS crashes by injecting selected machine-code mutations representative for both (i) low-level hardware faults and (ii) typical programming errors. In particular, we used 8 fault types from an existing fault injector [27, 36], as discussed in Sec. 6.2. Process tracing is used to control execution of the targeted driver and corrupt its program text at run-time. For each fault injection, the code to be mutated is found by calculating a random offset in the text segment and finding the closest suitable address for the desired fault type. This is done by reading the binary code and passing it through a disassembler to inspect the instructions' properties.

Each test run is defined by the following parameters: fault type to be used, number of SWIFI trials, number of faults injected per trial, driver targeted, and the workload. After starting the driver, the test suite repeatedly injects the specified number of faults into the driver's text segment, sleeping 1 second between each SWIFI trial so that the targeted driver can service the workload given. A driver crash triggers the test suite to sleep for 10 seconds, allowing the driver manager to restart the driver transparently to application programs and end users [17]. When the test suite awakens, it looks up the PID of the (restarted) driver, and continues injecting faults until the experiment finishes.

We iteratively refined our design by verifying that the driver could successfully execute its workload during each test run and inspecting the system logs for anomalies afterwards. While complete coverage of all possible problems cannot be guaranteed, we injected increasingly larger numbers of faults into different driver configurations. As described in Sec. 7.1, the system can now survive even millions of fault injections. This result strengthens our trust in the effectiveness of MINIX 3's isolation techniques.

Fault Type	Affected Program Text	Code Mutation
BINARY	randomly selected address	flip one random bit
POINTER	use of in-memory operand	corrupt address
SOURCE	assignment statement	corrupt right hand
DESTINATION	assignment statement	corrupt left hand
CONTROL	loop or branch instruction	change control flow
PARAMETER	operand loaded from stack	replace with NOPs
OMISSION	random instruction	replace with NOPs
RANDOM	selected from above types	one of the above

Figure 7: Fault types and code mutations used for SWIFI testing.

6.2 Fault Types and Test Coverage

Our test suite injected a meaningful subset of all fault types supported by the fault injector [27, 36]. For example, faults targeting dynamic memory allocation were left out because this is not used by our drivers. This selection process led to 8 suitable fault types, as summarized in Fig. 7. To start with, BINARY faults flip a bit in the program text to emulate hardware faults. The other fault types approximate a range of C-level programming errors commonly found in system code. For example, POINTER faults emulate pointer management errors, which were found to be a major cause of system outages [35]. Likewise, SOURCE and DESTINATION faults emulate assignment errors; CONTROL faults are checking errors; PARAMETER faults represent interface errors; and OMISSION faults can underly a wide variety of errors due to missing statements [2].

Although our fault injector could not emulate all possible (internal) error conditions [4, 6], we believe that the real issue is exercising the (external) isolation techniques that confine the test target. In this respect, the SWIFI tests proved to be very effective and pinpointed various shortcomings in our design. Analysis of the results also indicates that we obtained a good test coverage, since the SWIFI tests stressed each of the isolation techniques presented in Sec. 4.

6.3 Driver Configurations and Workload

We have experimented with different kinds of drivers, but decided to focus on MINIX 3's networking stack after we found that networking is by far the largest driver subsystem in Linux 2.6: 660 KLoC or 13% of the kernel's code base. In particular, we used the following configurations:

1. Emulated NE2000 (Bochs v2.2.6)
2. NE2000 ISA (Pentium III 700 MHz)
3. Realtek RTL8139 PCI (AMD Athlon64 X2 3800+)
4. Intel PRO/100 PCI (AMD Athlon64 X2 3800+)

The workload used during the SWIFI tests caused a continuous stream of network I/O requests in order to exercise the drivers' full functionality. In particular, we maintained a TCP connection to a remote *daytime* server, but this is transparent to the working of the drivers, since they simply put INET's message buffers on the wire (and vice versa) without inspecting the actual data transferred.

Although each of the drivers consists of at most thousands of lines of code, more important is the driver's interaction with the surrounding software and hardware. For example, the NE2000 driver uses programmed I/O, whereas the RTL8139 and PRO/100 drivers use DMA and require IOMMU support. Moreover, all drivers heavily interact with the INET server, PCI-bus driver, and kernel. Therefore, we believe that we have picked a realistic test target and covered a representative set of complex interactions.

7 RESULTS OF SWIFI TESTING

We now present the results of the final SWIFI tests performed after iterative refinement of the isolation techniques. The following sections discuss the robustness against failures, unauthorized access attempts, availability under faults, and problems encountered.

7.1 Robustness against Failures

The first and most important experiment was designed to stress test our isolation techniques by inducing driver failures with high probability. We conducted 32 series of 1000 SWIFI trials injecting 100 faults each—adding up to a total of 3,200,000 faults—targeting each of the 4 driver configurations for each of the 8 fault types discussed in Sec. 6. As expected, the drivers repeatedly crashed and had to be restarted by the driver manager. (The crash reasons are investigated in Sec. 7.2.) Fig. 8 gives a histogram with the number of failures per fault type and driver. For example, for RANDOM faults injected into the Emulated NE2000, NE2000, RTL8139, and PRO/100 driver we observed 826, 552, 819, and 931 failures, respectively. Although the fault injection induced a total of 24,883 driver failures, never did the damage (noticeably) spread beyond the driver's protection domain and affect the rest of the OS.

The figure also shows that different fault types affected the drivers in different ways. For example, SOURCE and DESTINATION faults more consistently caused failures than OMISSION faults. In addition, we also observed some differences between the drivers themselves, as is clearly visible for POINTER and CONTROL faults. This seems logical for the RTL8139 and PRO/100 cards that have different drivers, but the effect is also present for the two NE2000 configurations that use the same driver. We were unable to trace the exact reasons from the logs, but speculate that this can be attributed to the different driver-execution paths as well as the exact timing of the fault injection.

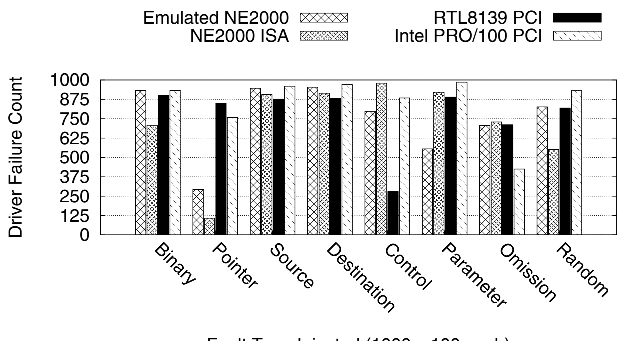

Figure 8: Number of driver failures per fault type. In total, this experiment injected 3,200,000 faults and caused 24,883 failures.

7.2 Unauthorized Access Attempts

Next, we analyzed the nature and frequency of unauthorized access attempts and correlated the results to the classification in Fig. 3. While MINIX 3 has many sanity checks in the system libraries linked into the driver, we focused on the logs from the kernel and driver manager, since their checks cannot be circumvented. Below, we report on an experiment with the RTL8139 driver that conducted 100,000 SWIFI trials injecting 1 RANDOM fault each.

In total, the driver manager detected 5887 failures that caused the RTL8139 driver to be replaced: 3,738 (63.5%) exits due to internal panics, 1,870 (31.8%) crashes due to exceptions, and 279 (4.7%) kills due to missing heartbeats. However, since not all error conditions were immediately fatal, the number of unauthorized access attempts logged by the kernel could be up to three orders of magnitude higher, as shown in Fig. 9. For example, we found 1,754,886 unauthorized DEVIO calls attempting to access device registers that do not belong to the RTL8139 PCI card. Code inspection confirmed that the driver repeatedly retried failed operations before giving up with an internal panic or causing an exception due to subsequent fault injections.

Each type of violation maps onto one or more classes of powers listed in Figure 3. For instance, CPU exceptions are a Class I violation that is caught by the corresponding Class I restrictions. Likewise, invalid memory grants and MMU exceptions fall in Class II, unauthorized device I/O matches Class III, and unauthorized IPC and kernel calls are examples of Class IV. While not all subclasses are represented in Fig. 9, the logs showed that our isolation techniques were indeed effective in all subclasses.

Unauthorized Access	Count	Percentage
1. Unauthorized device I/O	1,754,886	81.2%
2. Unauthorized kernel call	322,005	14.9%
3. Unauthorized IPC call	66,375	3.1%
4. Invalid memory grant	17,008	0.8%
5. CPU or MMU exception	1,780	0.1%
Total violations detected	**2,162,054**	**100.0%**

Figure 9: Top five unauthorized access attempts by the RTL8139 PCI driver for a test run with 100,000 randomly injected faults.

7.3 Availability under Faults

We also measured how many faults—injected one after another—it takes to disrupt the driver and how many more are needed for a crash. Disruption means that the driver can no longer successfully handle network I/O requests, but has not yet failed in a way detectable by the driver manager. Injected faults do not always cause an error, since the faults might not be on the path executed. As described in Sec. 6.3, a connection to a remote server was used to keep the driver busy and check for availability after each trial.

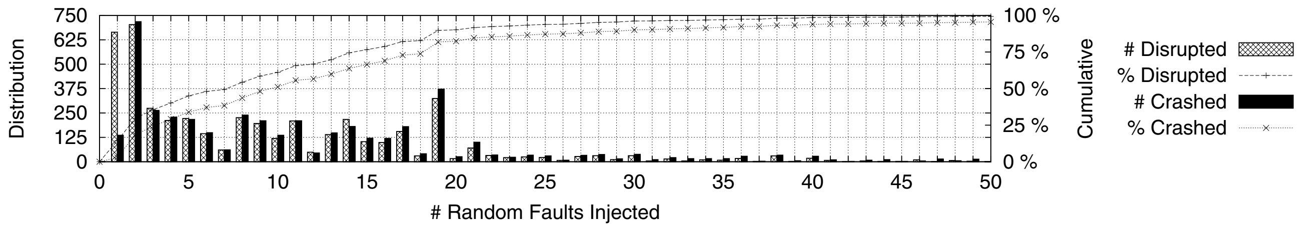

Figure 10: Number of faults needed to disrupt and crash the NE2000 ISA driver, based on 100,000 randomly injected faults. We observed 664 disruptions and 136 crashes after 1 fault. Crashes show a long tail to the right and surpass 99% only after 250 faults.

Fig. 10 shows the distribution of the number of faults needed to disrupt and crash the NE2000 driver for 100,000 SWIFI trials injecting 1 RANDOM fault each. Disruption usually happens after only a few faults, but the number of faults needed to induce a crash can be high. For example, we observed 664 disruptions after 1 fault, whereas one run required 2484 faults before the driver crashed. On average, the driver failed after 7 faults and crashed after 10 faults.

7.4 Problems Encountered

As mentioned above, we have taken a pragmatic approach toward dependability and went through several design iterations before we arrived at the final system. In order to underline this point, Fig. 11 briefly summarizes some of the problems that we encountered (and subsequently fixed) during the SWIFI testing of MINIX 3. Interestingly, we found many rare bugs even though the system was already designed for dependability [17], which illustrates the usefulness of extensive fault injection.

- Kernel stuck in infinite loop in load update due to inconsistent scheduling queues (bug in scheduler)
- Driver causes process manager to hang by not receiving synchronous reply (all IPC to untrusted code now is asynchronous)
- Driver request to perform SENDREC with nonblocking flag goes undetected and fails (bug in IPC subsystem)
- IPC call to SENDREC with target ANY not detected and kept pending forever (bug in IPC subsystem)
- Illegal IPC destination (ANY) for NOTIFY call caused kernel panic rather than erroneous return (bug in IPC subsystem)
- Kernel panic due to exception caused by uninitialized struct priv pointer in system task (bug in kernel call handler)
- Network driver went into silent mode due to bad restart parameters
- Infinite loop in driver not detected because driver manager's priority was set too low to ping driver and check its heartbeat
- System-wide starvation due to excessive kernel debug messages
- Isolation policy allowed arbitrary memory copies, which corrupted INET (isolation policy violated least authority)
- Driver reprogrammed RTL8139 hardware's PCI device ID (code was present in driver, now removed)
- Wrong IOMMU setting caused legitimate DMA read by the disk controller to fail and corrupt the file system

Figure 11: Bugs found during SWIFI testing of MINIX 3.

8 LESSONS LEARNED

Our experiments resulted in several insights that are worth mentioning. To start with, the fault injection proved very helpful in finding programming bugs, as shown in Fig. 11. An interesting observation, however, is that some hard-to-trigger bugs showed up only after several design iterations and injecting many millions of faults. In the past, similar efforts often limited their tests to a few thousands of fault injections, which may not be enough to trigger rare faults. For example, Nooks [36] and Safedrive [38] reported only 2000 and 44 fault-injection trials, respectively.

Although this work focuses on mechanisms rather than policies, *policy definition* is a hard problem. At some point, the driver's policy accidentally granted access to a kernel call for copying arbitrary memory without grants, causing memory corruption in the network server. We 'manually' reduced the privileges granted, but techniques such as formalized interfaces [30] and compiler-generated manifest [33] may be helpful to define correct policies.

Furthermore, while our design makes the system as a whole more robust, availability of individual services cannot be guaranteed due to *hardware limitations*. In a very small number of cases, less than 0,1% of all NE2000 ISA driver crashes, the NE2000 ISA card was put in an unrecoverable state and could not be reinitialized by the driver. Instead, a low-level BIOS reset was needed. If the card had a 'master reset' command, the driver could have solved the problem, but our card did not have this.

Finally, we had to abandon one experiment due to an insurmountable hardware limitation: tests with a driver for the Realtek RTL8029 PCI card caused the entire system to freeze. We narrowed down the problem to writing a specific (unexpected) value to an (allowed) control register of the device—presumably causing a PCI bus hang. We believe this to be a peculiarity of the specific device or weakness of the PCI bus rather than a shortcoming of our design.

In summary, however, the results show that fault isolation and failure resilience [17] indeed help to survive bugs and enable on-the-fly recovery. While we have used MINIX 3, many of our ideas are generally applicable and may also bring improved dependability to other systems.

9 SUMMARY & CONCLUSION

This paper investigates the privileged operations that low-level device drivers need to perform and that, unless properly restricted, are root causes of fault propagation. We showed how MINIX 3 systematically restricts drivers according to the principle of least authority in order to limit the damage that can result from bugs. In particular, fault isolation is achieved through a combination of structural constraints imposed by a multiserver design, fine-grained per-driver isolation policies, and run-time memory granting. We believe that many of these techniques are generally applicable and can be ported to other systems.

We have taken an empirical approach toward dependability and have iteratively refined our isolation techniques using software-implemented fault-injection (SWIFI) testing. We targeted 4 different Ethernet driver configurations using both programmed I/O and DMA. While we had to work around certain hardware limitations, the resulting design was able to withstand 100% of 3,400,000 randomly injected faults that were shown to be representative for typical programming errors. The targeted drivers repeatedly failed, but the rest of the OS was never affected.

ACKNOWLEDGMENTS

Supported by Netherlands Organization for Scientific Research (NWO) under grant 612-060-420.

REFERENCES

[1] H. Bos and B. Samwel. Safe Kernel Programming in the OKE. 2002.

[2] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong. Orthogonal Defect Classification-A Concept for In-Process Measurements. *IEEE TSE*, 18(11):943–956, 1992.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proc. 18th SOSP*, 2001.

[4] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults–Based on Field Data. In *Proc. 26th FTCS*, 1996.

[5] T. Dinh-Trong and J. M. Bieman. Open Source Software Development: A Case Study of FreeBSD. In *Proc. 10th Int'l Symp. on Software Metrics*, 2004.

[6] J. Duraes and H. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE TSE*, 32(11):849–867, 2006.

[7] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a Practical, Verified Kernel. In *Proc. 11th HotOS*, 2007.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proc. 7th OSDI*, 2006.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proc. 1st OASIS*, 2004.

[10] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. 20th LISA*, 2006.

[11] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. 9th ACM SIGOPS European Workshop*, 2000.

[12] D. B. Golub, G. G. Sotomayor, Jr, and F. L. Rawson III. An Architecture for Device Drivers Executing as User-Level Tasks. In *Proc. USENIX Mach III Symp.*, 1993.

[13] J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Proc. 5th SRDS*, 1986.

[14] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. 6th SOSP*, 1997.

[15] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proc. 1st Int'l Conf. on Collaborative Computing*, 2005.

[16] L. Hatton. Reexamining the Fault Density-Component Size Connection. *IEEE Software*, 14(2), 1997.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure Resilience for Device Drivers. In *Proc. 37th DSN*, 2007.

[18] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Countering IPC Threats in Multiserver Operating Systems. In *Proc. 14th PRDC*, 2008.

[19] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. Sealing OS Processes to Improve Dependability and Safety. In *Proc. 2nd EuroSys*, 2007.

[20] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Comp. Science and Techn.*, 20(5), 2005.

[21] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th OSDI*, 2004.

[22] J. Liedtke. On μ-Kernel Construction. In *Proc. 15th SOSP*, 1995.

[23] A. Mancina, G. Lipari, J. N. Herder, B. Gras, and A. S. Tanenbaum. Enhancing a Dependable Multiserver OS with Temporal Protection via Resource Reservations. In *Proc. 16th RTNS*, 2008.

[24] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *Proc. 4th OSDI*, 2000.

[25] Microsoft Corporation. Architecture of the User-Mode Driver Framework. In *Proc. 15th WinHEC*, 2006.

[26] B. Murphy. Automating Software Failure Reporting. *ACM Queue*, 2 (8), 2004.

[27] W. T. Ng and P. M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proc. 29th FTCS*, 1999.

[28] V. Orgovan. Online Crash Analysis - Higher Quality At Lower Cost. In *Presented at 13th WinHEC*, 2004.

[29] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding Collateral Evolution in Linux Device Drivers. In *Proc. 1st EuroSys*, 2006.

[30] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming Device Drivers. In *Proc. 4th EuroSys Conf.*, 2009.

[31] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proc. of the IEEE*, 63(9), 1975.

[32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. 2nd OSDI*, 1996.

[33] M. Spear, T. Roeder, O. Hodson, G. Hunt, and S. Levi. Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. In *Proc. 1st EuroSys*, 2006.

[34] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. USENIX'01*, 2001.

[35] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems. In *Proc. 21st FTCS*, 1991.

[36] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *ACM TOCS*, 23(1), 2005.

[37] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proc. 6th PRDC*, 1999.

[38] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proc. 7th OSDI*, 2006.