BUILDING A DEPENDABLE OPERATING SYSTEM: FAULT TOLERANCE IN MINIX 3

VRIJE UNIVERSITEIT

BUILDING A DEPENDABLE OPERATING SYSTEM: FAULT TOLERANCE IN MINIX 3

ACADEMISCH PROEFSCHRIFT

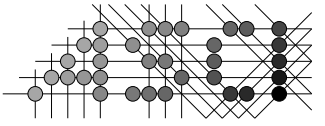
ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 9 september 2010 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

JORRIT NIEK HERDER

geboren te Alkmaar

promotor: prof.dr. A.S. Tanenbaum
copromotor: dr.ir. H.J. Bos



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 208.

This research was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 612-060-420.

"We're getting bloated and huge. Yes, it's a problem.
. . . I'd like to say we have a plan."

Linus Torvalds on the Linux kernel, 2009

ISBN 978-94-6108-058-5

Parts of this thesis have been published before:

ACM SIGOPS Operating System Review, 40(1) and 40(3)
USENIX ;login:, 31(2), 32(1), and 35(3)
IEEE Computer, 39(5)
Springer Lecture Notes in Computer Science, 4186
Springer Real-Time Systems, 43(2)

Proc. 6th European Dependable Computing Conf. (EDCC'06)
Proc. 37th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN'07)
Proc. 14th IEEE Pacific Rim Int'l Symp. on Dependable Computing (PRDC'08)
Proc. 39th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN'09)
Proc. 4th Latin-American Symp. on Dependable Computing (LADC'09)

Contents

List of Figures

Acknowledgements

This thesis marks the end of a remarkable period in my life for which I am indebted to many people. First and foremost, I would like to thank my mentors, Andy Tanenbaum and Herbert Bos, who were always there for me and shaped my research in many ways. Andy's strong focus on elegance and simplicity, critical attitude toward virtually anything, and persistence and determination have been very inspiring. I am particularly grateful for all the freedom he gave me and his seemingly never-ending trust in whatever I came up with. Herbert's ability to balance Andy's outspoken beliefs, considerate and thoughtful approach to research, and mindfulness for social needs has helped me to put things in perspective. I have also enjoyed his enthusiasm for traveling the world and keenly tried to follow his example. Having had two such wonderful mentors has been an absolute delight and made my Ph.D. experience not only very rewarding, but also truly enjoyable.

Next, I would like to thank the members of my thesis committee, including George Candea (Dependable Systems Laboratory, École Polytechnique Fédérale de Lausanne), Bruno Crispo (Department of Computer Science, Vrije Universiteit Amsterdam), Michael Hohmuth (Operating System Research Center, AMD), Galen Hunt (Operating Systems Group, Microsoft Research), and Michael Swift (Computer Sciences Department, University of Wisconsin-Madison), for reviewing this thesis. It has been an honor to get direct feedback from some of the leading researchers in the field. Their extensive and constructive comments have been helpful to improve the text and make it what it is now.

I also would like to express my gratitude for the opportunities I have been offered to give seminar presentations and guest lectures at various labs all over the world. In particular, I appreciate the hospitality provided by Bruno Crispo (Università degli Studi di Trento), Marko van Eekelen (Radboud Universiteit Nijmegen), Steven Hand (University of Cambridge), Hermann Härtig (Technische Universität Dresden), Gernot Heiser (National ICT Australia), Giuseppe Lipari (Scuola Superiore Sant'Anna), and Andrew Warfield (University of British Columbia). Being able to share my ideas and getting early, critical feedback have helped to advance my research and shape my view of the field.

I was fortunate enough to do various internships during the course of my Ph.D. Halfway through, I left for a research internship at Microsoft Research Silicon Valley

under the supervision of Úlfar Erlingsson and Martín Abadi. Being pampered with corporate housing in San Francisco, an excursion to the Monterey Bay Aquarium, and a BBQ at Bill Gates' house in Medina, WA was a great diversion from university life. Next year, I did a software engineering internship at Google in New York and had the pleasure to work with Michael Harm, Norris Boyd, and Brian Kernighan. The Google experience and lunches on the roof terrace overlooking the New York City skyline again made for a very enjoyable summer. Not much later, I flew out to Sydney for a visit to the embedded systems group led by Gernot Heiser at National ICT Australia. The relaxed atmosphere and top-notch L4 research team made for another pleasant and inspiring stay. I would like to thank my former colleagues and all others who contributed to these experiences.

Most of my time was spent at Vrije Universiteit Amsterdam though. Special mention goes the MINIX team. In the first place, credit is due to Ben Gras and Philip Homburg, who did most of the hard programming while I was plotting crazy little ideas. I am happy to have them as my paranymphs. Arun Thomas and Thomas Veerman later joined the team in the same spirit. It also has been a great pleasure to work with—and be challenged by—my fellow Ph.D. students: Mischa Geldermans, David van Moolenbroek, Raja Appuswamy, Cristiano Giuffrida, Tomáš Hrubý, and Erik van der Kouwe. I appreciate the team's proofreading effort and have great memories of our all-too-frequent coffee breaks, movie nights, and team dinners. Next, I would like to thank the rest of the Department of Computer Science. Although many contributed to the pleasant research environment, I particularly would like to thank my predecessors and peers outside the MINIX team, including Spyros Voulgaris, Swaminathan Sivasubramanian, Melanie Rieback, Srijith K. Nair, Willem de Bruijn, Georgios Portokalidis, Asia Slowinska, Remco de Boer, and Rik Farenhorst, for showing me the way while I was still growing up and our interesting discussions about various random topics.

Several 'external' contributors to MINIX 3 deserve mention too. To start with, I would like to thank Antonio Mancina (Scuola Superiore Sant'Anna), who visited the MINIX group in 2007 and brought real-time support to MINIX 3. Next, I am grateful to Bingzheng Wu (Chinese Academy of Sciences), who participated in Google Summer of Code 2008 and 2009 and worked on MINIX 3's storage stack and memory grants. Parts of this work are described in this thesis. I also want to acknowledge the numerous local students who did a term project on MINIX 3 or contributed as a research assistant, including, among others, Balázs Gerőfi, Luke Huang, Jens de Smit, Anton Kuijsten, Pieter Hijma, and Niek Linnenbank.

Last but not least, I would like to thank my family and friends who supported me throughout this endeavor and provided the necessary distraction from work. In particular, I wish to thank my wife, Eva Marinus, for her love and support, tolerating my idiosyncrasies, and all the adventures we have had together.

Jorrit N. Herder
Sydney, Australia, July 2010

Samenvatting

Dit hoofdstuk biedt een beknopte Nederlandse samenvatting van dit academisch proefschrift met als titel *"Het bouwen van een betrouwbaar besturingssysteem: Foutbestendigheid in MINIX 3"*. Hoofdstuk 7 bevat een uitgebreidere Engelse samenvatting en gaat dieper in op de onderzoeksbijdragen.

Een van de grootste problemen met computers is dat ze niet voldoen aan de verwachtingen van gebruikers ten aanzien van betrouwbaarheid, beschikbaarheid, veiligheid, etc. Een onderzoek onder Windows-gebruikers liet bijvoorbeeld zien dat 77% van de klanten 1 tot 5 fatale fouten per maand ondervindt en de overige 23% van de klanten maandelijks zelfs meer dan 5 fatale fouten ervaart. De oorzaak van deze problemen ligt in het besturingssysteem ("operating system") dat een centrale rol heeft in vrijwel elk computersysteem. De meeste fouten zijn terug te leiden tot stuurprogramma's voor randapparatuur ("device drivers") die relatief foutgevoelig zijn. Dergelijke stuurprogramma's zijn nauw geïntegreerd in het besturingssysteem, waardoor fouten zich gemakkelijk kunnen verspreiden en het hele besturingssysteem kunnen ontregelen. Dit probleem doet zich niet alleen voor bij standaard besturingssystemen voor de PC, zoals Windows, Linux, FreeBSD en MacOS. Besturingssystemen voor mobiele apparatuur (bijvoorbeeld telefoons, PDAs, fotocamera's, etc.) en ingebedde computers (bijvoorbeeld in auto's, pinautomaten, medische apparatuur, etc.) zijn veelal gebaseerd op een vergelijkbaar ontwerp waardoor ze soortgelijke problemen kennen.

Onbetrouwbare besturingssystemen veroorzaken niet alleen persoonlijke frustraties, maar hebben ook grote maatschappelijke consequenties zoals economische schade en veiligheidsrisico's. Dit onderzoek heeft zich daarom tot doel gesteld om een uitermate betrouwbaar besturingssysteem te bouwen dat fouten in stuurprogramma's kan weerstaan en herstellen. Deze doelstellingen zijn gerealiseerd door het besturingssysteem foutbestendig ("fault tolerant") te maken zodat het normaal kan blijven functioneren ondanks het optreden van veel voorkomende problemen. Hierbij hebben we voortgebouwd op recente technologische vooruitgang en ingespeeld op de veranderende eisen en wensen van gebruikers. Aan de ene kant biedt moderne computer hardware betere ondersteuning voor het afschermen van foutgevoelige stuurprogramma's. Aan de andere kant is de rekenkracht van computers zodanig toegenomen dat technieken die voorheen te kostbaar waren nu praktisch

toepasbaar zijn. Daarnaast is de snelheid van desktop PCs tegenwoordig ruim voldoende en leggen steeds meer gebruikers de nadruk op betrouwbaarheid.

In dit onderzoek promoten we het gebruik van een modulair besturingssysteem ("multiserver operating system") dat compatibiliteit met UNIX waarborgt, maar stuurprogramma's en systeemtoepassingen net als gewone gebruikerstoepassingen in een onafhankelijk proces uitvoert. Dit model combineert hardware-bescherming met software-technieken om stuurprogramma's te isoleren, zodat getriggerde fouten minder schade kunnen aanrichten. In ons ontwerp hebben we twee strategieën toegepast om de foutbestendigheid verder te verhogen: (1) fout isolatie ("fault isolation") om de tijd tussen fatale fouten te vergroten ("mean time to failure") en (2) foutherstellend vermogen ("failure resilience") om de benodigde tijd voor het repareren van fouten te verkleinen ("mean time to recover"). Beide aspecten zijn in gelijke mate van belang voor het verhogen van de beschikbaarheid van het besturingssysteem. Naast de hogere foutbestendigheid bieden modulaire besturingssystemen ook vele andere voordelen: een korte ontwikkelingscyclus, een vertrouwd programmeermodel en eenvoudig systeembeheer.

Om onze ideeën te kunnen testen, hebben we van het open-source besturingssysteem MINIX 3 gebruikt. MINIX 3 voert stuurprogramma's, systeemtoepassingen en gebruikerstoepassingen uit in onafhankelijke processen. Slechts een klein deel van het besturingssysteem, bestaande uit zo'n 7500 regels programmacode, draait met alle rechten van de computer en controleert de rest van het systeem. Communicatie tussen de verschillende onderdelen van het besturingssysteem is alleen mogelijk door berichten van proces naar proces te kopiëren. Wanneer een gebruikerstoepassing bijvoorbeeld een bestand van de harde schijf wil lezen, moet deze een bericht sturen naar de systeemtoepassing voor het bestandssysteem, dat vervolgens een bericht stuurt naar het stuurprogramma voor de harde schijf. Eén van de uitbreidingen op MINIX 3 is een systeemtoepassing die alle stuurprogramma's beheert ("driver manager"). Deze component maakt het mogelijk om systeemtoepassingen en stuurprogramma's te starten en te stoppen zonder de computer opnieuw te hoeven starten. Het zorgt er tevens voor dat de stuurprogramma's strikt van elkaar en van de rest van het besturingssysteem worden afgeschermd en het kan veel voorkomende fouten in stuurprogramma's detecteren en automatisch herstellen.

Hoewel veel van de gebruikte ideeën en technieken op zich niet nieuw zijn, was hun gecombineerde potentieel om de betrouwbaarheid van besturingssystemen te verbeteren nog niet voldoende onderzocht en overtuigend aangetoond. Door dit te doen met behulp van MINIX 3 levert dit proefschrift de volgende wetenschappelijke en praktische bijdragen:

- Het laat zien hoe de betrouwbaarheid van besturingssystemen kan worden verbeterd met behoud van het vertrouwde UNIX-programmeermodel. In tegenstelling tot aanverwant onderzoek, is alleen het binnenwerk van het besturingssysteem vernieuwd. Hierdoor kan compatibiliteit met bestaande software worden behouden en is praktische toepassing stapgewijs mogelijk.

- Het classificeert de geprivilegieerde verrichtingen van stuurprogramma's die ten grondslag liggen aan het verspreiden van fouten en bespreekt voor elke klasse een reeks fout-isolatie technieken om de schade die fouten kunnen veroorzaken te beperken. Dit resultaat is van belang voor elke poging om stuurprogramma's af te zonderen ongeacht het besturingssysteem.

- Het introduceert een ontwerp dat het besturingssysteem in staat stelt om een breed scala aan fouten in belangrijke componenten automatisch te detecteren en te repareren zonder gebruikerstoepassingen te onderbreken en zonder tussenkomst van de gebruiker. Veel van deze ideeën zijn van toepassing in een bredere context dan besturingssystemen alleen.

- Het evalueert de effectiviteit van het gepresenteerde ontwerp door middel van uitgebreide tests met door software nagebootste fouten. In tegenstelling tot eerdere projecten, zijn letterlijk miljoenen fouten nagebootst, waardoor ook veel sporadisch voorkomende fouten opgespoord konden worden en verbeterde betrouwbaarheid met een hoge mate van zekerheid is aangetoond.

- Het beschrijft hoe recente hardware virtualisatie technieken gebruikt kunnen worden om beperkingen van bestaande fout-isolatie technieken te overwinnen. Tegelijkertijd bespreekt het enkele resterende tekortkomingen in huidige PC-hardware waardoor zelfs volledig afgezonderde stuurprogramma's het besturingssysteem nog steeds kunnen laten vastlopen.

- Tot slot heeft dit onderzoek niet alleen geleid tot een ontwerp, maar is dit ontwerp ook geïmplementeerd, met als resultaat het besturingssysteem MINIX 3 dat publiek beschikbaar is via de officiële website http://www.minix3.org/. Dit foutbestendige besturingssysteem maakt duidelijk dat de voorgestelde aanpak praktisch toepasbaar is.

Samenvattend kunnen we concluderen dat met dit onderzoek naar het bouwen van een foutbestendig besturingssysteem, dat bestand is tegen de gevaren van de foutgevoelige stuurprogramma's, een stap is gezet in de richting van meer betrouwbare besturingssystemen.

Chapter 1

General Introduction

In spite of modern society's increasingly widespread dependence on computers, one of the biggest problems with using computers is that they do not meet user expectations regarding reliability, availability, safety, security, maintainability, etc. While these properties are, in fact, different concepts, from the users' point of view they are closely related and together delineate a system's *dependability* [Avižienis et al., 2004]. The users' mental model of how an electronic device should work is based on their experience with TV sets and video recorders: you buy it, plug it in, and it works perfectly for the next 10 years. No crashes, no monthly software updates, no unneeded reboots and downtime, no newspaper stories about the most recent in an endless string of viruses, and so on. To make computers more like TV sets, the goal of our research is to build a dependable computing platform, starting with the operating system (OS) that is at the heart of it.

Our research focuses on the dependability needs of ordinary PCs (including desktops, notebooks, and server machines), moderately well-equipped mobile devices (such as cell phones, PDAs, and photo cameras), and embedded systems (as found in cars, ATMs, medical appliances, etc.). Dependability problems with commodity PC operating systems (OSes), including Windows, Linux, FreeBSD, and MacOS, pose a real threat, and hanging or crashing systems are commonplace. For example, Windows' infamous blue screens are a well-known problem [Boutin, 2004]. However, the same problems are showing up on mobile devices and embedded systems now that they become more powerful and start to run full-fledged OSes, such as Windows Mobile, Embedded Linux, Symbian, and Palm OS. The design of these systems is not fundamentally different from PC OSes, which means that the dependability threats and challenges for mobile devices and embedded systems are similar to those encountered for ordinary PCs.

Our aim is to improve OS dependability by starting from scratch with a new, lightweight design, without giving up the UNIX [Ritchie and Thompson, 1974] look and feel and without sacrificing backward compatibility with existing applications. We realize that software is not always perfect and can contain bugs, but want faults

and failures to be masked from the end user, so that the system can continue running all the time. In particular, we address the problem of buggy device drivers, which run closely integrated with the core OS and responsible for the majority of all crashes. We are willing to make trade-offs that help to protect the system at the expense of a small amount of performance. Our design metric was always: how does this module, algorithm, data structure, property, or feature affect the system's dependability?

This thesis describes how we have realized this goal and built a dependable OS, MINIX 3, which is freely available along with all the source code. The remainder of this chapter is organized as follows. To begin with, Sec. 1.1 further motivates the need for improved OS dependability. Next, Sec. 1.2 describes the threats posed by device drivers and Sec. 1.3 investigates the principles underlying OS crashes. Then, Sec. 1.4 introduces our solution to the problem and Sec. 1.5 briefly previews related work. Finally, Sec. 1.6 defines the exact research focus and Sec. 1.7 lays out the organization of the rest of this thesis.

1.1 The Need for Dependability

Perhaps the need for dependability is best illustrated by looking at the potentially far-reaching consequences of software failures. For example, between 1985 and 1987, a type of bug known as a race condition led to six accidents with the Therac-25 radiation-therapy machine involving a radiation overdose that caused serious injury and three fatalities [Leveson and Turner, 1993]. In 1991, arithmetic errors in the Patriot missile-defense system used during the Gulf War prevented intercepting an Iraqi Scud missile killing 28 American soldiers and injuring around 100 other people [Blair et al., 1992]. In 1996, an integer overflow caused the Ariane-5 satellite to be destroyed shortly after launch by its automated self-destruct system resulting in loss of reputation and damages of at least US$ 370 million [Dowson, 1997]. In 2003, an infinite loop prevented alarms from showing on FirstEnergy's control system and seriously hampered the ability to respond to a widespread power outage affecting over 50 million people in the U.S. and Canada [Poulsen, 2004]. These events as well as many others [e.g. Garfinkel, 2005], illustrate the general need for dependable computing platforms.

Narrowing our focus, OS dependability is of particular importance because of the fundamental role the OS plays in almost any computer system. The OS is the lowest level of system software, commonly referred to as the *kernel*, that interfaces between the computer hardware and applications run by the end user. In our view, the OS has three key responsibilities:

- The OS mediates access to the central processing unit (CPU), memory, and peripheral devices, so that applications can interoperate with the computer.

- It acts as a kind of virtual machine by offering a convenient application programming interface (API) at a level higher than the raw hardware.

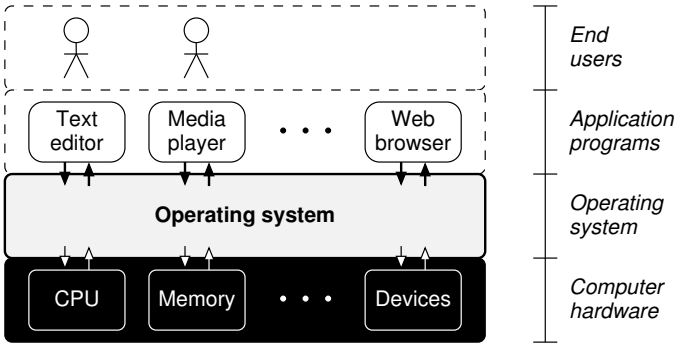


Figure 1.1: The fundamental role of the OS in a computer system: the OS is the lowest level of system software that controls the hardware, manages resources, and interfaces with applications.

- Finally, the OS controls and manages the system's software and hardware resources in order to prevent conflicts and enforce security policies.

We do not consider the system utilities and application programs shipped with the OS to be part of the OS. This design is illustrated in Fig. 1.1. Because of this important role of the OS at the heart of the system any problem with the OS will have immediate repercussions on the dependability of the system as a whole.

Despite its important role, OS dependability is still lacking. Studies seem to indicate that most unplanned downtime can be attributed to faulty system software rather than hardware failures [Gray, 1990; Lee and Iyer, 1993; Thakur et al., 1995; Xu et al., 1999], and within this class OS failures deserve special attention because of their severity. A January-2003 survey among home and small business users showed that all participants experienced Windows crashes or serious bugs; 77% of the customers experienced 1–5 crashes per month, whereas 23% of the customers experienced more than 5 monthly crashes [Orgovan and Dykstra, 2004]. Although application crashes are more common, OS crashes have a broader scope and more severe consequences [Ganapathi and Patterson, 2005]. In contrast to crashed applications that can be restarted immediately, an OS crash takes down all running applications, closes all network sessions, destroys all unsaved user data, and requires a full, time-consuming reboot of the machine.

Rebooting after an OS crash is not only frustrating, but also a real problem to end users. First, computer crashes are unacceptable for ordinary users who may not be willing or used to deal with failures. Anecdotal evidence is provided by support calls from friends or family who have some computer problem to be fixed. Next, OS crashes are a serious issue for data centers and server farms where monthly OS reboots translate to many daily failures due to the large number of machines running. Even if individual OSes have 99.9% uptime, achieving an overall service level of 'three nines' or 99.9% availability is a big challenge. Hence, data centers usually employ additional measures, such as replication and preventive reboots, to guarantee

availability. Finally, there is a host of safety-critical systems that are simply too important to fail. Safety-critical systems are found in, for example, aerospace, aviation, automotive, military, medical, telecommunications, banking, and public-service applications. Therefore, OS dependability is crucial.

1.2 The Problem with Device Drivers

In order to improve OS dependability we have primarily focused on OS extensions. In a typical design, the core OS implements common functionality, such as process scheduling and programming the memory management unit (MMU), and provides a set of kernel interfaces for enhancing or extending this functionality. OS extensions use these hooks to integrate seamlessly with the core OS and extend its functionality with a specific service, such as a device driver, virus scanner, or protocol stack. In this way, the OS can support different configurations without having to include all possible anticipated functionality at the time of shipping; new extensions can be installed after the OS has been deployed.

We are particularly interested in low-level device drivers that control the computer's peripheral devices, such as printers, video adapters, audio cards, network cards, storage devices, etc. Drivers separate concerns by providing an abstraction layer between the device hardware and the OS modules that use it. For example, a network server defines a generic, high-level network protocol and uses a network driver to do the low-level input/output (I/O) from and to a specific network card. Likewise, a file server implements a file-system format and relies on a disk driver to read or write the actual data from or to the controller of the disk drive. In this way, the core OS can remain hardware-independent and call upon its drivers to perform the complex, hardware-dependent device interaction.

Drivers are important not only qualitatively, but also quantitatively, and can comprise up to two-thirds of the OS code base [Chou et al., 2001]. This is not surprising since the number of different hardware devices in 2004 is quoted at 800,000 with 1500 new devices per day [Murphy, 2004]. Even though many devices have the same chipset and some drivers can control multiple devices, literally tens of thousands of drivers exists. In the same year, Windows reportedly had 31,000 unique drivers and up to 88 drivers were being added every day. A more recent report mentions an average of 25 new and 100 revised drivers per day [Glerum et al., 2009]. While not all these drivers are present in any system, each individual driver runs closely integrated with the core OS and can potentially crash the entire system.

It is now beyond a doubt that OS extensions and drivers in particular are responsible for the majority of OS crashes. To start with, analysis of failure reports for the Tandem NonStop-UX OS showed that device drivers contribute the greatest number of faults [Thakur et al., 1995]. A static compiler analysis of Linux also found that driver code is most buggy, both in terms of absolute bug counts and in terms of error rate. Drivers had most bugs for all error classes distinguished and ac-

counted for 70%–90% of the bugs in most classes. The error rate corrected for code size in drivers also is 3–7 times higher than for the rest of the kernel [Chou et al., 2001]. Finally, studies of Windows' automatically generated crash dumps again pinpoint drivers as a major problem. For example, crash analysis shows that 70% of all OS crashes is caused by third-party code, whereas 15% is unknown because of severe memory corruption [Orgovan and Dykstra, 2004]. Another independent study showed that 65% of all OS crashes are due to device drivers [Ganapathi et al., 2006].

Looking at the driver development cycle we believe that driver quality is not likely to improve on a large scale any time soon. Although we did not investigate in detail why drivers are so error-prone, several plausible explanations exist:

- First, driver writing is relatively difficult because drivers are complex state machines that must handle application-level requests, low-level device interaction, and system events such as switching to a lower power state.

- Second, drivers are often written by third parties, such as the device manufacturer or a volunteer from the open-source community, who may be ignorant of system rules and accidentally violate interface preconditions.

- Third, OS and hardware documentation is frequently lacking, incomprehensible, or even incorrect, causing programmers to (reverse) engineer the driver until it works—without correctness guarantees [Ryzhyk et al., 2009a].

- Fourth, driver writers sometimes incorrectly assume the hardware to work correctly, as evidenced by, for example, infinite polling or lack of input validation, which may hang or crash the OS [Kadav et al., 2009].

- Fifth, relating to the previous points, driver writers often are new to the task and lack experience. By a huge margin, the average third-party driver writer is writing his first (and only) device driver [Hunt, pers. comm., 2010].

- Finally, in contrast to the core OS that typically remains stable and is more heavily tested, drivers come and go with new hardware and may have more poorly tested code paths due to rare system configurations [Merlo et al., 2002].

In addition, we believe that fixing buggy drivers is infeasible because of the overwhelmingly large number of extensions and continuously changing configurations. Although Microsoft's error reporting revealed that a small number of organizations are responsible for 75% of all driver crashes, a heavy tail indicates it is extremely hard to eliminate the remaining 25% of the crashes [Ganapathi et al., 2006]. Moreover, the number of problems that can be resolved using automatic checkers fluctuates because software and hardware configurations are continuously changing [Murphy, 2004]. This leads to a highly complex, dynamic configuration space that is extremely hard, if not impossible, to check for correctness. Finally, even if bugs can be fixed, bug fixes not infrequently introduce new problems.

1.3 Why do Systems Crash?

Having established device drivers as a main dependability threat, we now study the more fundamental principles that lead to OS crashes. Although OS crashes can be caused by bugs in hardware, firmware, and software [Glerum et al., 2009], this thesis primarily focuses on the latter because software bugs are more prevalent and can be addressed by the OS itself. Buggy driver software is unavoidable due to an excess of complex system code and hardware with bizarre programming interfaces. The reason underlying OS crashes, however, are a series of design flaws that allow the bugs to spread and cause damage. Below, we investigate these issues in turn.

1.3.1 Software Complexity

Large software systems seem to be buggy by nature. Studies for many programming languages reveal a similar pattern, leading to the conclusion that any well-written code can be expected to have a fault density of at least 0.5–6 faults per 1000 lines of executable code (LoC) [Hatton, 1997]. For example, the 86,000-LoC IBM DOS/VS OS displayed 6 faults/KLoC during testing [Endres, 1975]. Next, a 90,000-LoC satellite planning system contained 6–16 faults/KLoC throughout the software life cycle [Basili and Perricone, 1984]. Furthermore, a 500,000-LoC AT&T inventory tracking system had 2–75 faults/KLoC [Ostrand and Weyuker, 2002]. In line with these estimates, the multimillion-line FreeBSD OS was found to have 1.89 'post-feature test' faults/KLoC [Dinh-Trong and Bieman, 2005], even though this project has strict testing rules and anyone is able to inspect the source code. The 'post-feature test' fault density includes all faults found in completed features during the system test stage [Mockus et al., 2002]. However, we believe that this fault density may even be an underestimate because only bugs that were ultimately found and filed as a bug report were counted. For example, static analysis of 126 well-tested and widely used Windows drivers revealed significant numbers of latent bugs [Ball et al., 2006]. A more recent study took 6 popular Windows drivers and found 14 new bugs with relatively little effort [Kuznetsov et al., 2010].

There are various reasons for all these bugs. Module size and complexity seem to be poor predictors for the number of faults, but new code was generally found to contain more bugs. Attempts to relate fault density to module size and complexity have found different results and thus are inconclusive: studies found a negative correlation [Basili and Perricone, 1984; Ostrand and Weyuker, 2002], a U-shaped correlation [Hatton, 1997], a positive correlation [Chou et al., 2001], or no correlation at all [Fenton and Ohlsson, 2000]. However, code maturity seems to be a viable metric to predict faults. In Linux, 40%–60% of the bugs were introduced in the previous year [Chou et al., 2001]. In the AT&T inventory tracking system, new files more frequently contained faults and had higher fault densities [Ostrand and Weyuker, 2002]. Interestingly, however, the overall fault density stabilized after many releases and did not go asymptotically to zero [Ostrand et al., 2005].

In addition to bugs in new drivers, maintainability of existing code is complicated by software aging due to maintenance and redesign [Gray and Siewiorek, 1991]. In particular, changes in OS interfaces may trigger changes in dependent drivers. Such changes are known as *collateral evolution* and can account for up to 35% of the number of lines modified from one OS version to the next. Many collateral evolutions are quite pervasive, with one change in a library function affecting around 1000 sites in Linux 2.6 [Padioleau et al., 2006]. We expect this situation to deteriorate because dependencies between the core OS and extensions seem to increase with each release [Schach et al., 2002]. Therefore, even if a driver appears to be bug-free in one version, changes to the rest of the OS may silently break it in the next release because of *regression faults*, that is, faults present in the new version of the system that were not present prior to modification.

In order to assess what these findings mean for an actual OS, we analyzed the Linux 2.6 kernel during a 5-year period since its initial release. Fig. 1.2 shows the code evolution with 6-month deltas for 11 versions ranging from Linux 2.6.0 to 2.6.27.11. The figure is indicative for the evolution of Linux' enormously complex code base, which poses a serious threat to its dependability. The */drivers* subsystem is by far the largest subsystem and more than doubled in a period of just 5 years, comprising 51.9% of the kernel or 2.7 MLoC. The entire kernel now surpasses 5.3 MLoC. Linux' creator, Linus Torvalds, also acknowledges this fact and recently called the kernel 'bloated and huge' [Modine, 2009]. Using a conservative estimate of 1.89 faults/KLoC found for FreeBSD [Dinh-Trong and Bieman, 2005], this means that the 5.3-MLoC Linux kernel may contain over 10,000 bugs, some of which may eventually be triggered and crash the OS. While we acknowledge that

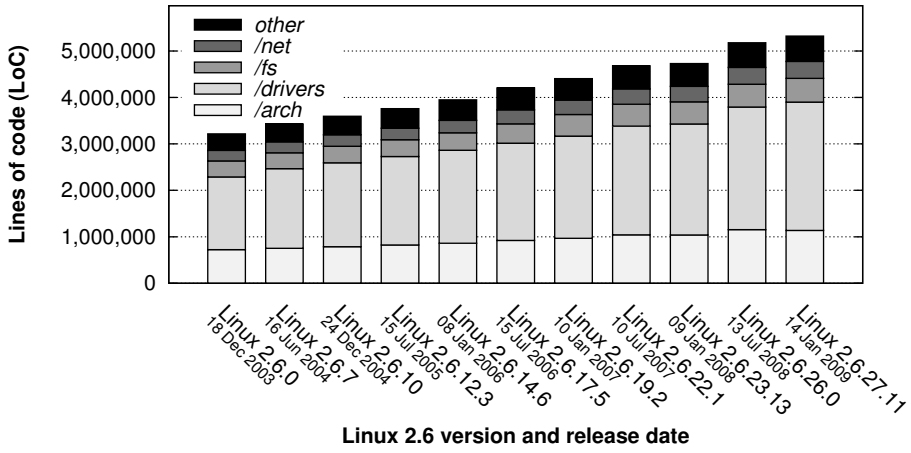


Figure 1.2: Growth of the Linux 2.6 kernel and its major subsystems in lines of executable code (excluding comments and white space) for a 5-year period with 6-month deltas.

only a subset of the code is active on any given system, test coverage for new or infrequently used drivers may be poor and deteriorate over time.

The obvious conclusion of these studies is: more code means more bugs means more dependability problems. As software develops, each new release tends to acquire more features and is often less dependable than its predecessor. There seems to be no end to the increasing size and complexity of software, leading to a *perpetual software crisis* [Myhrvold, 1997]. Moreover, with multimillion-line systems, no person will ever read the complete source code and fully understand it. Of course, different people will understand different parts, but because components interact in complex ways, with nobody understanding the whole system, it is unlikely that all the bugs will ever be found. On top of this, bug triage is sometimes postponed for months or even delayed indefinitely [Guo and Engler, 2009] and administrators are generally slow to apply bug fixes [Rescorla, 2003]. Therefore, this thesis aims to improve dependability by anticipating buggy code.

1.3.2 Design Flaws

While it may not be possible to prevent bugs, we strongly believe it is the responsibility of the OS to tolerate them to a certain extent. In our view, a bug local to a device driver or other OS extension should never be able to bring down the OS as a whole. In the worst case, such an event may lead to a local failure and gracefully degrade the OS by shutting down only the malfunctioning module. If a bug in, say, an audio driver is triggered, it is bad enough if the audio card stops working, but a full OS crash is completely unacceptable. Unfortunately, current OSes do not do very well according to our metric.

The fact that current OSes are susceptible to bugs can be traced back to two fundamental design flaws. The main problem is lack of isolation between the OS modules due to the use of a *monolithic design*. Virtually all OSes, including Windows, Linux, and FreeBSD, consist of many modules linked together to form a single binary program known as the *kernel*. Even though there is a clear logical structure with a basic executive layer (the core OS) and extensions that provide added functionality, all the code runs in a single protection domain. Since kernel code is allowed to execute privileged CPU instructions, drivers can accidentally change the memory maps, issue I/O calls for random devices, or halt the CPU. In addition, all the code runs in a single address space, meaning that an invalid pointer or buffer overflow in any module can easily trash critical data structures in another module and spread the damage. This design is illustrated in Fig. 1.3.

A related problem is violating a design principle known as the *principle of least authority* (POLA) [Saltzer and Schroeder, 1975] by granting excessive privileges to untrusted (driver) code. Most sophisticated users normally would never knowingly allow a third party to insert unknown, foreign code into the heart of their OS, yet when they buy a new peripheral device and install the driver, this is precisely what they are doing. In addition, many drivers come preinstalled with the OS distribu-

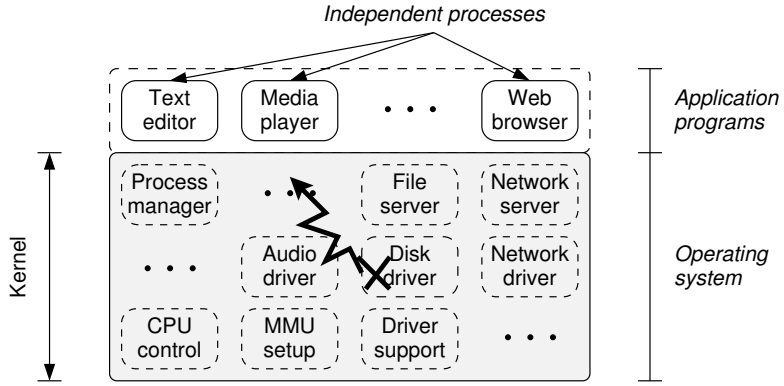


Figure 1.3: A monolithic design runs the entire OS in the kernel without protection barriers between the OS modules. A single driver fault can potentially crash the entire OS.

tion, which results in an unwarranted feeling of trust. However, because drivers are executed in the kernel they can circumvent all security policies and the slightest malfunctioning can take down the OS. Effectively, drivers are granted full control over the machine—even though this is not required for drivers to do their job.

The reason for running the entire OS in the kernel with no protection between the modules seems to be historically motivated. Early OSes were relatively small and simple and, therefore, still manageable by a few experts. More importantly, computers were far less powerful and imposed severe constraints upon the OS with respect to resource usage. Linking all the modules together did not waste memory and resulted in the best performance. The use of a monolithic design thus seemed logical at that time. Today, however, the landscape has completely changed: current OSes tend to be much larger and far more complex and (desktop) computing power is no longer a scarce resource. These software and hardware advances mean that the design choices of the past may no longer be appropriate.

1.4 Improving OS Dependability

Because we do not believe that bug-free code is likely to appear soon, we have designed our OS in such a way that certain major faults can be tolerated. The idea underlying our approach is to exploit *modularity*, as is commonly done in other engineering disciplines. For example, the hull of a modern ship is compartmentalized such that, if one compartment springs a leak, only that compartment is flooded, but not the entire hull, and the ship can seek a safe harbor. Likewise, computer hardware is componentized such that, if the DVD-RW drive breaks down, writing DVDs is temporarily not possible, but the computer works fine otherwise, and the broken unit may be replaced with a new one. This kind of modularity is key to dependability and forms the basis underlying our dependable OS design.

1.4.1 A Modular OS Design

For decades, the proven technique for handling untrusted code, such as application programs, has been to run it as an independent process with a private memory address space and in an unprivileged CPU mode. Such an unprivileged process is known as a *user process*. Hardware protection provided by the CPU and MMU are used to set up a protection domain in which the process can run safely, isolated from the kernel and other user processes. If a user process attempts to execute a privileged instruction or dereference a pointer outside of its address space, a CPU or MMU hardware exception is raised and control is transferred back to the OS via a trap. The user process may be suspended or aborted, but, more importantly, the rest of the system is unaffected and can continue to operate normally.

We took this idea to the extreme with a modular design that fully compartmentalizes the OS, just like is done for application programs. Moving the entire OS into a single user process [Härtig et al., 1997] makes rebooting the operating system after a crash faster, but does not address the degree of driver isolation. What is required instead is a *multiserver design* that splits the OS functionality into multiple independent user-level processes, and very tightly controlling what each process can do. Only the most basic functionality that cannot be realized at the user level, such as programming the CPU and MMU hardware, is allowed to remain in the kernel. Because the kernel is reduced to the absolute minimum it is now called a *microkernel*. While servers and drivers require more privileges than applications, each is run as an ordinary process with the same hardware-enforced protection model. The structure of a microkernel-based multiserver design is illustrated in Fig. 1.4.

We are not arguing that moving most of the OS to user level reduces the total number of bugs present. However, we are arguing that, by converting a kernel-level bug into a user-level bug, the effects when a bug is triggered will be less devastating. Monolithic designs are like ships before compartmentalization was invented: every leak can sink the ship. In contrast, multiserver designs are like modern ships with isolated compartments that can survive local problems. If a bug is triggered, the problem is isolated in the driver and can no longer bring down the entire OS. In addition, the multiserver design changes the manifestation of certain bugs. For example, a classification of driver failures on Windows XP shows that the use of user-level drivers would structurally eliminate the top five contenders, responsible for 1690 (67%) of the 2528 OS crashes analyzed [Ganapathi et al., 2006]. The following driver failures from this analysis illustrate this point:

- Just under a quarter of these failures (383) are caused by driver traps or exceptions that cannot be handled by the kernel. A trap is usually caused by a CPU hardware exception such as division by zero, arithmetic overflow, or invalid memory access. When a trap occurs the CPU stops execution of the current process, looks up the corresponding trap handler, and switches context to the kernel in order to handle the exception. However, upon a trap caused by a kernel-level driver it may be impossible to switch to another kernel context,

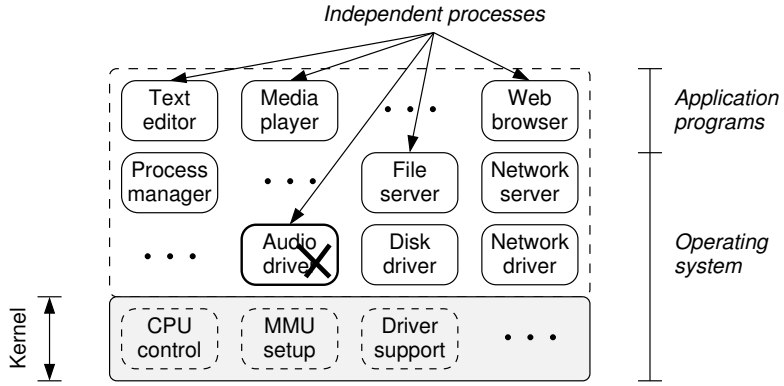


Figure 1.4: A multiserver design runs each server and driver as an independent, hardware-protected user process. This design prevents propagation of faults between the OS modules.

and the OS crashes. In contrast, traps or exceptions caused by a user-level driver can be dealt with using the OS' normal exception handling mechanisms that already are in place for application programs.

- About a fifth of these failures (327) are caused by infinite loops in drivers, that is, pieces of code that lack a functional exit and repeat indefinitely. A driver may wind up in an infinite loop, for example, if it repeatedly retries a failed operation or busy waits for the hardware to become ready, without checking against a maximum number of retry attempts or scheduling a time-out alarm, respectively. Because a kernel-level driver runs with full privileges, such a bug may consume all CPU resources and hang the entire system. In contrast, a user-level driver runs under the control of the OS scheduler, just like normal application programs do, and can only waste the CPU resources assigned to it. The rest of the OS still gets a chance to run, and corrective measures such as lowering the driver's priority can be taken.

There are two potential downsides to the use of a multiserver OS design. First, because device drivers often depend on the core OS and need to perform sensitive operations, it is not possible to maintain full backward compatibility and directly run a kernel driver as a user process. Instead, minor modifications to the driver code are generally required in order to mediate the use of privileged functionality [Herder, 2005]. Nevertheless, we believe that the level of compatibility provided by multiserver systems is high enough to be of practical use. For example, it is generally possible to port existing drivers from other OSes. Second, because user-level drivers need to call other parts of the OS to perform operations they could formerly do themselves, a small performance overhead due to context switching and data copying is expected. In practice, this overhead is limited to 0%–25%, depending on the system's workload, and should not pose a problem for most use cases.

1.4.2 Fault-tolerance Strategies

The fine-grained process-based encapsulation provided by a multiserver design allowed us to explore the idea of *fault tolerance*, that is, the ability to continue to operate normally in the presence of faults and failures [Nelson, 1990]. If a driver fault affects the rest of the OS at all, the quality of service may be gracefully degraded proportional to the problem's severity, but the fault may never lead to a system-wide failure. Our design applies two fault-tolerance strategies, namely, *fault isolation* and *failure resilience*. Our fault-isolation mechanisms prevent local driver faults from damaging the rest of the system. This cannot prevent a faulty driver from failing, however. Therefore, our failure-resilience mechanisms attempt to recover from a broad range of driver failures. We further introduce these notions below:

- Fault isolation means that the damage caused by a bug cannot propagate and spread beyond the protection domain of the component in which the bug is contained. By splitting the OS into small, independent components we can establish protection barriers across which faults cannot propagate, resulting in a more robust system. Bugs in user-level drivers have much less opportunity to trash kernel data structures and cannot touch hardware devices they have no business touching. While a kernel crash is always fatal, a crash of a user process rarely is. Although drivers need more privileges than ordinary applications, we have attempted to minimize the risks by restricting driver privileges according to the principle of least authority

- Failure resilience literally refers to the ability to recover quickly from a failure condition. Once driver faults are properly isolated from the core OS, on-the-fly recovery of certain driver failures may be possible. By giving a special component known as the *driver manager* the power to monitor and control all drivers at run time, we can detect and repair a wide range of failures, including unresponsive, misbehaving, and crashing drivers. When a failure is detected, the driver manager looks up the associated policy, and can automatically replace the malfunctioning component with a fresh copy. In some cases, full recovery is possible, transparent to applications and without user intervention.

These fault-tolerance strategies map onto two important dependability metrics. First, *mean time to failure* (MTTF) characterizes a system's uptime, that is, the time until a failure occurs. Second, *mean time to recover* (MTTR) characterizes the time needed to recover from a failure, that is, the time needed for defect detection and repair. Together, these metrics define a system's *availability* (A):

$$A = \frac{MTTF}{MTTF + MTTR}$$

with an availability of A $= 1$ (i.e. MTTF $>>$ MTTR) corresponding to the ideal of zero downtime. This formula shows that availability can be increased either by

maximizing MTTF or by minimizing MTTR. Our fault tolerance techniques work from both sides. Fault isolation increases MTTF by making the system as a whole more robust against failures. However, since bugs are a fact of life, infinite uptime may not be realistic. Therefore, failure resilience reduces MTTR by restarting failed drivers on the fly rather than requiring a full reboot. Although we do not attempt to quantify exactly MTTF and MTTR, the point we want to make is that reducing MTTR may be as effective as increasing MTTF [Gray and Siewiorek, 1991].

1.4.3 Other Benefits of Modularity

In addition to dealing with buggy drivers, the use of modularity helps improving dependability in a broader context. While there are some restrictions, we believe that a modular OS environment supports both programmers and administrators throughout the development cycle and may lead to higher productivity, improved code quality, and better manageability. We discuss some of these benefits below.

Short Development Cycle The huge difference between monolithic and multiserver OSes immediately becomes clear when looking at the development cycle of OS components. System programming on a monolithic OS generally involves editing, compiling, rebuilding the kernel, and rebooting to test the new component. A subsequent crash requires another reboot, and tedious, low-level debugging usually follows. In contrast, the development cycle on a multiserver OS is much shorter. Typically, the steps are limited to editing, compiling, testing, and debugging—just like is done for application programs. Because OS development at the user level is easier, the programmer can get the job done faster.

Normal Programming Model Since drivers are ordinary user processes, the normal programming model applies. Drivers can use system libraries and, in some cases, even make ordinary system calls, just like applications. This strongly contrasts to the rigid programming environment of monolithic kernels. For example, in Windows, kernel-level code running at high a priority level must be careful not to access pageable memory, because page faults may not be handled. Likewise, the normal way to acquire kernel locks may not be used in the lowest-level driver code, such as interrupt handlers. All these kernel-level constraints make it easy for programmers to make mistakes. In essence, working at the user level makes programming easier and leads to simpler code and, therefore, reduces the chance of bugs.

Easy Debugging Debugging a device driver in a monolithic kernel is a real challenge. Often the system just halts and the programmer does not have a clue what went wrong. Using an emulator usually is of no use because typically the device being driven is new and not supported by the emulator. On Windows platforms debugging a kernel-level driver is normally done using two machines: one for the driver and a remote machine running the debugger. In contrast, if a driver runs as

a user process, a crash leaves behind a core dump that can be subjected to post-mortem analysis using all the normal debugging tools. Furthermore, as soon as the programmer has inspected the core dump and system logs and has updated the code, it is possible to test the new driver without a full reboot.

Good Accountability When a user-level driver crashes, it is completely obvious which one it is, because the driver manager can tell which process exited. This makes it much easier than in monolithic kernels to pin down whose fault a crash was, and possibly who is legally liable for the damage done. Holding hardware manufacturers and software vendors liable for their errors, in the same way as the producers of tires, medicines, and other products are held accountable, may be an incentive to improve software quality. Although some software providers are willing to remedy problems brought to their attention, the number of vendors participating in error reporting programs is still very small [Glerum et al., 2009]. With drivers running at the user level it will become easier to set up error reporting infrastructure, which may help to get software providers in the loop.

Simple Maintenance Our modular design makes system administration and maintenance easier. Since OS modules are just processes, it is relatively easy to add or remove servers and drivers. It becomes easier to configure the OS by mixing and matching modules. Furthermore, if a module needs to be patched, this can usually be done in a process known as a *dynamic update*, that is, component replacement or patching without loss of service or a time-consuming reboot [Baumann et al., 2007]. This is important since reboots due to maintenance cause a large fraction (24%) of system downtime [Xu et al., 1999]. In contrast to monolithic kernels, module substitution is relatively easy in modular systems and often can be done on the fly.

1.5 Preview of Related Work

Recently, several other projects have also acknowledged the problem of buggy device drivers and attempted to make them more manageable and prevent them from crashing the entire OS. In our discussion, we are primarily interested in how different run-time systems deal with untrusted code and what trade-offs they pose. Although we also cover some ways to improve driver quality, prevention of bugs is an orthogonal problem and outside the scope of this work. For the purpose of this thesis, we have classified the related work into four categories:

- In-kernel sandboxing.
- Virtualization techniques.
- Formal methods.
- User-level frameworks.

The work described here classifies as a user-level framework. Interestingly, many of the ideas used in each of these classes have been around for decades, but have been revisited recently in order to improve OS dependability.

Without going into all the details or concrete systems, which we will do in Chap. 6, we briefly preview each approach below and contrast them to our approach. First, in-kernel sandboxing tries to isolate drivers inside the kernel by providing a restricted execution environment for the untrusted code. This is called a sandbox. One approach is to wrap each driver in a layer of software that controls the driver's interaction with the kernel. One particular benefit of this approach is that it allows to retrofit dependability into commodity OSes. In addition, interposition allows catching a wide variety of fault types, since the wrapper code can be aware of driver protocols. However, in-kernel drivers can sometimes still execute dangerous CPU instructions and additional protection mechanisms are required for untrusted execution. Although drivers often do not have to be modified, new support infrastructure is typically needed in the kernel. This adds additional complexity to the kernel and introduces new maintenance problems.

Second, virtualization techniques present a virtual, hardware-protected execution environment that exports only a subset of the computer's resources to the client. Privileged operations are intercepted and vetted before execution. Virtualization is generally used to run multiple OSes in isolation, for example, Windows next to Linux, FreeBSD, or MacOS, but cannot prevent drivers running inside the OS from crashing their execution environment. Although the crashed OS can be restarted without affecting other virtual machines, all running applications and unsaved user data are still lost. Isolation can be achieved by running untrusted drivers in separate virtual machines, but this requires the OS hosting the driver as well as the virtual execution environment to be modified in order to let driver communication go in and out. Such modifications break the original intent of virtualization and require protection measures similar to those found in user-level frameworks in order to guarantee proper isolation. Furthermore, resource management may not scale if each and every OS extension needs to be isolated separately.

Third, formal methods exploit advances in safe languages and verification techniques in order to evaluate novel OS designs. Current OSes are commonly written in low-level languages like C or C++, which use error-prone memory pointers all the time. In contrast, safe, high-level languages structurally eliminate many problems, because the compiler refuses to generate 'dangerous' code and the run-time system automates memory management. In addition, it becomes possible to perform static analysis on driver code and verify that system invariants are not violated. Driver synthesis also seems attractive, but writing a formal specification of the device interface requires substantial manual effort. A downside of these approaches is that it completely overhauls the traditional, well-known development cycle and often is incompatible with existing code. Furthermore, formal verification of the entire OS is infeasible, and hardware support is still required to isolate untrusted code and protect against memory corruption by incorrectly programmed devices.

Fourth, user-level frameworks run drivers as user processes protected by the CPU and MMU hardware. Running the entire OS in a single process is of little help because drivers still run closely integrated with the core OS. Instead, each individual driver must be encapsulated in an independent process. This model is a simple, well-understood and proven technique for handling untrusted code. Although driver modifications are generally required, the changes tend to be limited to the code that uses privileged functionality. This basic idea has been around for a few decades, but was never fully explored because it incurs a small performance overhead. Although modular designs have been tested before, many projects focused on performance [e.g. Liedtke, 1993; Härtig et al., 1997] and security [e.g. Singaravelu et al., 2006] rather than dependability. Not until recently were user-level drivers introduced in commodity OSes, but a fully compartmentalized OS is still a rarity.

From the user's point of view, a multiserver OS is in the middle of the spectrum ranging from legacy to novel isolation techniques. For example, in-kernel sandboxing works with commodity OSes and existing drivers, but introduces additional complexity into the kernel to work around rather than fix a flawed design. Virtualization provides only a partial solution to the problem and cannot elegantly deal with individual driver failures. Formal methods potentially can provide a very high degree of isolation, but often throw away all legacy by starting from scratch with a design that is not backward compatible. In contrast, user-level frameworks balance these factors by redesigning the OS internals in order to provide hard safety guarantees, while keeping the UNIX look and feel for both developers and end users, and maintaining compatibility with existing applications.

An important benefit of our modular design is that it can be combined with other isolation techniques in order to achieve the best of all worlds. For example, language-based protection can be exploited on a small scale, because drivers running as ordinary user processes can be implemented in a programming language of choice. Likewise, sandboxing techniques such as wrapping and interposition may also be applied to monitor the working of selected drivers more closely. Because our design runs all OS services as independent user processes, the protection model can be tightened incrementally, starting with the most critical components.

1.6 Focus of this Thesis

Having introduced the general problem area as well as our high-level solution to OS dependability, the time has come to define the exact focus and highlight the main contribution of this thesis. We first position the research by explicitly stating what we did and did not do.

- The focus of this thesis is dependability rather than *security* [Avižienis et al., 2004]. Dependability refers to a system's reliability, availability, safety, integrity, and maintainability, whereas security deals with availability, integrity, and confidentiality. Nevertheless, problems in each of these domains often

have the same root cause: bugs in the software. For example, a buffer overrun in a driver error can cause an OS crash, but it can also allow a cleverly written virus or worm to take over the computer [Lemos, 2005; Ou, 2006]. Bugs that can be exploited are known as *vulnerabilities* and have a density several orders of magnitude lower than ordinary bugs [Ozment and Schechter, 2006]. Our design is intended to curtail the consequences of bugs that may accidentally cause service failures under normal usage, but does not aim to protect against vulnerabilities that can be actively exploited by a malicious party to infiltrate or damage a computer system. Nevertheless, since confinement of bugs may prevent them from becoming a vulnerability, our efforts to improve dependability may also help to improve security [Karp, 2003]. Even if a driver is compromised, the OS restrictions ensure that the attacker gains no more privileges than the driver already had.

- Another aspect that we do not address is *performance*. While we are the first to admit that some of the techniques presented in this thesis incur a small performance overhead, performance is an issue orthogonal to dependability. Measurements on a prototype implementation seem to indicate a performance overhead of up to 25%, but we never designed the OS for high performance, and the performance can no doubt be improved through careful analysis and removal of bottlenecks. In fact, several independent studies have already addressed the issue of performance in modular OS designs [Liedtke, 1993, 1995; Härtig et al., 1997; Gefflaut et al., 2000; Haeberlen et al., 2000; Leslie et al., 2005a] and have shown that the overhead can be limited to 5%–10%. Furthermore, we build on the premise that computing power is no longer a scarce resource, which is generally true on desktop PCs nowadays. Hardware performance has increased to the point where software techniques that previously were infeasible or too costly have become practical. Moreover, the proposed modular design can potentially exploit the increasing levels of parallelism on multicore CPUs [Larus, 2009]. Finally, the performance-versus-dependability trade-off has changed. We believe that most end users are more than willing to sacrifice some performance for improved dependability.

Now that we have clearly positioned our work, we provide a preliminary view on the main contribution of this thesis. In particular, this work improves OS dependability by tolerating faults and failures caused by buggy device drivers. While many of the techniques used, such as user-level drivers, fault isolation, and failure resilience, are not completely new, to the best of our knowledge we are the first to put all the pieces together to build a flexible, fully modular UNIX clone that is specifically designed to be dependable, with only a limited performance penalty. At the same time we have kept an eye on the system's usability. For instance, the fault-tolerance mechanisms are fully transparent to end users and the programming interface is easy to use for system programmers. In order to realize this goal we had to face numerous challenges, which are the subject of the rest of this thesis.

1.7 Outline of this Thesis

The rest of this thesis is organized as follows. Chap. 2 starts out by giving an introduction to the MINIX 3 OS. We present the high-level architecture of our fault-tolerant design and discuss the important role of the driver manager herein. We also make explicit the assumptions underlying our design and point out the limitations of fault isolation and failure resilience.

Chap. 3 investigates the privileged operations that low-level device drivers need to perform and that, unless properly restricted, are root causes of fault propagation. We show 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, we present fault-isolation techniques for each of the privileged driver operations. We also illustrate our ideas with a case study.

Chap. 4 explains how MINIX 3 can detect defects and repair them on-the-fly. We introduce the working of our defect-detection mechanism, the policy-driven recovery procedure, and post-restart reintegration of the components. Next, we discuss the concrete steps taken to recover from driver failures in the network stack, storage stack, and character-device stack and describe the consequences for the rest of the system. We also present two case studies.

Chap. 5 evaluates our design along three different axes. First and foremost, we present the results of extensive software-implemented fault-injection (SWIFI) experiments that demonstrate the effectiveness of our design. Second, we discuss performance measurements to assess the costs of our fault-tolerance mechanisms. Third, we briefly describe a source-code analysis of MINIX 3.

Chap. 6 puts this work in context by comparing it to related efforts to improve OS dependability and highlighting the different trade-offs posed. We survey a range of concrete systems and present case studies for each of the approaches that we distinguished, including in-kernel sandboxing, virtualization techniques, formal methods, and user-level frameworks.

Finally, Chap. 7 summarizes the main results of this thesis and points out the most important lessons that we learned. We conclude by stating the main contributions of this thesis, discussing the applicability of our findings, and suggesting possible directions for future work.

Chapter 2

Architectural Overview

This chapter provides a high-level overview of our fault-tolerant OS architecture. We briefly introduce the platform that we used for our research, MINIX 3, and describe the design changes that we made in order make drivers more manageable. In addition, we summarize the most important fault-isolation and failure-resilience mechanisms. We also discuss the rationale behind our design.

The remainder of this chapter is organized as follows. To start with, Sec. 2.1 provides the necessary background about the MINIX OS on which this research is based. Sec. 2.2 presents the new infrastructure for managing drivers. Next, Sec. 2.3 describes how driver faults can be isolated from the core OS and Sec. 2.4 explains how certain failures can be detected and repaired. Finally, Sec. 2.5 describes the fault and failure model assumed in our research.

2.1 The MINIX Operating System

This section introduces the MINIX OS, which we took as a starting point for our research. All ideas described in this thesis have been prototyped in MINIX and eventually should become part of the mainstream OS. Some symbols were renamed in this thesis for the purpose of readability, however. Below, we briefly discuss the history of MINIX and give a high-level overview of its architecture.

2.1.1 Historical Perspective

The MINIX OS has a long and rich history; some episodes of which are worth mentioning. When the UNIX OS was developed by AT&T Bell Laboratories in the 1960s and released in the mid-1970s [Ritchie and Thompson, 1974], universities worldwide soon adopted UNIX to teach OS classes. Many courses were based on a commentary on Version 6 UNIX [Lions, 1977]. However, when AT&T realized the commercial value of UNIX in the late 1970s, the company changed the Version 7 UNIX license agreement to prohibit classroom use of source code. The lack of a

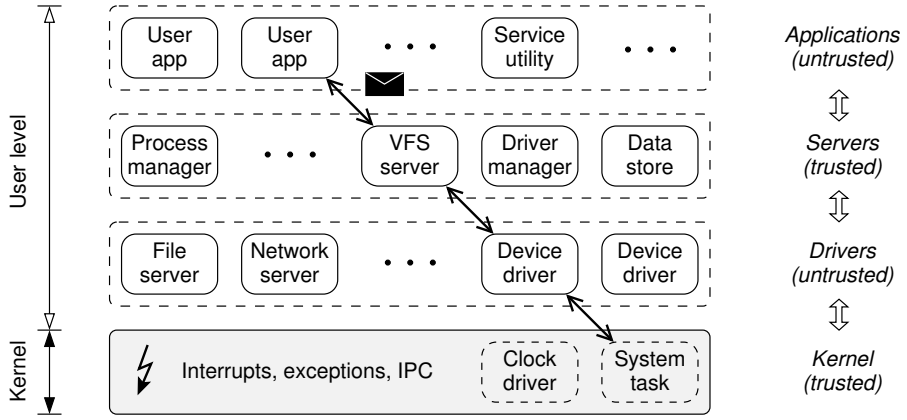


Figure 2.1: Multiserver design of the MINIX 3 OS. Each application, server, and driver runs as an independent user process. Interprocess communication (IPC) is based on message passing.

small and simple OS suitable for teaching initiated the development of a new UNIX clone, MINIX, which was released in 1987 [Tanenbaum, 1987]. MINIX was functionally compatible with UNIX, but had a more modular structure and implemented parts of the OS functionality outside the kernel. In particular, the process manager and file server were run as independent user processes, but all drivers remained in the kernel. With a companion book and all the sources available for classroom use MINIX soon became, and still is, widely used for teaching.

Although the MINIX source code was available, modification and redistribution were restricted, thereby creating a niche for the Linux OS in 1991 [Torvalds, 1991; Torvalds and Diamond, 2001]. In strong contrast to the design of MINIX, however, the Linux kernel reverted to a monolithic structure, which sparked a famous debate on the Usenet newsgroup *comp.os.minix* in 1992 [DiBona and Ockman, 1999, Appendix A]. Linux did not change the design and still has a monolithic kernel, and has grown enormously ever since, as evidenced by Fig. 1.2 in Sec. 1.3.1. MINIX, in contrast, stayed small and simple. The most notable releases include MINIX 1.5, which was ported to several architectures, and MINIX 2, which added POSIX compliance [Institute of Electrical and Electronics Engineers, 1990], but the overall design stayed mostly the same.

The research reported in this thesis builds on MINIX 3, which was released when all drivers were removed from the kernel and transformed into independent user processes in 2005 [Herder, 2005]. MINIX 3 is the first fully modular version of MINIX that runs each server and driver in a separate process, as sketched in Fig. 2.1. The system is still used for teaching [Tanenbaum and Woodhull, 2006], but also provides new research opportunities. In this thesis, we have taken the process-based encapsulation as a starting point for improving the system's fault tolerance. As discussed in Sec. 1.4.2, we focus on fault isolation and failure resilience in particular.

2.1.2 Multiserver OS Structure

Before we continue we define some more formal terminology. Like virtually any modern OS, MINIX 3 runs in the 32-bit *protected mode* of the x86 (IA-32) architecture so that it can use hardware protection features, such as virtual memory and protection rings. The memory space and privileged CPU mode associated with the kernel are called *kernel space* and *kernel mode* (ring 0), respectively. Likewise, the memory space and unprivileged CPU mode associated with user processes are called *user space* and *user mode* (ring 3), respectively. A modular user-level OS design is commonly referred to as a *multiserver OS*, whereas the kernel-level part is known as the *microkernel*. A pure microkernel design is minimalist and strictly separates policies and mechanism: the kernel implements only the most basic mechanisms that need kernel-level privileges, whereas all policies are provided by the user-level servers and drivers [Liedtke, 1995]. The MINIX 3 OS fits this definition.

We now briefly introduce the structure of MINIX 3. Although all processes are treated equally, a logical layering can be distinguished, as shown in Fig. 2.1. At the lowest level, a small microkernel of about 7500 lines of executable code (LoC) performs privileged operations. The kernel intercepts hardware interrupts, catches exceptions in user processes, handles communication between processes, and programs the CPU and MMU in order to run processes. It also contains two tasks that are compiled into kernel space, but otherwise scheduled as normal processes. First, the clock driver handles clock interrupts, keeps track of system time, performs process scheduling, and manages timers and alarms. While some of these basic functions could be implemented at the user level, it would be very inefficient to do so. However, higher-level clock functionality such as real-time scheduling is realized at the user level [Mancina et al., 2009]. Second, the system task offers a small set of kernel calls to support authorized user-level servers and drivers in doing their job. In principle, it provides only privileged functionality that cannot be realized at the user level. Sec. 3.2 discusses these operations in more detail.

The next level up contains the drivers. There is one device driver (sometimes referred to as a function driver) for each major device, including drivers for storage, network, printer, video, audio, and so on. In addition, the driver layer contains protocol drivers for file systems and network protocols, such as the *file server* and *network server*. Each driver is a user process protected by the CPU and MMU the same way ordinary user processes are protected. They are special only in the sense that they are allowed to make a small number of kernel calls to perform privileged operations. Typical examples include setting up interrupt handlers, reading from or writing to I/O devices, and copying memory between address spaces. This design introduces a small kernel-call overhead, but also brings more fine-grained control because the kernel mediates all accesses. A bitmap in the kernel's process table controls which calls each driver (and server) can make. Also, the kernel maintains data structures that define which I/O devices a driver may use and copying is allowed only with explicit permission from all parties involved.

On top of the driver layer is the server layer. The *process manager* and *virtual file system* implement the POSIX API and provide process and memory management and virtual-file-system services, respectively. User processes make POSIX system calls by sending a message to one of these servers, which then carries out the call. If the call cannot be handled by the server, it may delegate part of the work to a lower layer. New in MINIX 3 and an important contribution of this thesis is the *driver manager*, which manages all the other servers and drivers. The driver manager can start new drivers and restart failing or failed ones on-the-fly. The *data store* provides naming services and can be used to backup and restore state. Several other servers also exists, for example, the *information server*, which provides debug dumps of system data structures.

Finally, located above the server layer are ordinary, unprivileged application programs, such as text editors, media players, and web browsers. When the system comes up, *init* is the first application process to run and forks off *getty* processes, which execute the *shell* on a successful login. The shell allows other applications to be started, for example, the *service utility* that allows the administrator to request services from the driver manager. The only difference between this and other UNIX systems is that the library procedures for making system calls work by sending messages to the server layer. While message passing is used under the hood, the system libraries offer the normal POSIX API to the programmer.

2.1.3 Interprocess Communication

Since MINIX 3 runs all servers and drivers as independent processes, they can no longer directly access each other's functions or data structures. Instead, processes must make a *remote procedure call* (RPC) when they want to cooperate. In particular, the kernel provides *interprocess communication* (IPC) services based on *message passing*. If two processes need to communicate, the sender constructs a message in its address space and requests an IPC call to send it to the other party. The stub code linked with the application puts the IPC parameters on the stack or in CPU registers and executes a trap instruction, causing a software interrupt that puts the kernel's IPC subsystem in control. The kernel then checks the IPC parameters and executes the corresponding IPC handler. Messages are never buffered in the kernel, but always directly copied or mapped from sender to receiver, speeding up IPC and eliminating the possibility of running out of buffers.

Several different IPC interaction modes can be distinguished. First, *synchronous IPC* is a two-way interaction where the initiating process is blocked by the kernel until the other party becomes ready. When both parties are ready, the message is copied or mapped from the sender to the receiver and both parties may resume execution. The term *IPC roundtrip* is used if the sender synchronously awaits the reply after sending a request. Second, *asynchronous IPC* means that the caller can continue immediately without being blocked by the kernel. The IPC subsystem takes care of buffering the message and delivers it at the first opportunity on behalf of the caller.

Primitive	Semantics	Storage	Mode	Blocking
SEND	Block until message is sent	Caller	Synchronous	Blocking
RECEIVE	Block until message arrives	-	Synchronous	Blocking
SENDREC	Send request and await reply	Caller	Synchronous	Blocking
NBSEND	Send iff peer is receiving	Caller	Synchronous	Nonblocking
ASEND	Buffered delivery by kernel	Caller	Asynchronous	Nonblocking
NOTIFY	Event signaling mechanism	Kernel	Asynchronous	Nonblocking

Figure 2.2: The synchronous, asynchronous, and nonblocking IPC primitives implemented by the kernel's IPC subsystem. All nonblocking calls were added to MINIX 3.

Both synchronous and asynchronous IPC may be combined with a time-out to abort the IPC call if it did not succeed within the specified interval. However, since finding sensible time-out values is nontrivial, zero or infinite time-outs are commonly used. The former is referred to as *nonblocking IPC*: delivery is tried once and the status is immediately returned. The latter is referred to as *blocking IPC* and corresponds to a normal synchronous IPC interaction.

The IPC primitives and message format used by the MINIX 3 IPC subsystem are shown in Figs. 2.2 and 2.3, respectively. The most basic primitives are the synchronous, blocking calls SEND and RECEIVE. Arguments to these calls are the IPC endpoint of the destination or source process and a pointer to a message buffer. No buffering is required because the caller is blocked until the message has been copied from the sender to the receiver. The SENDREC primitive combines these primitives in a single call, doing a synchronous SEND following by an implicit RECEIVE. This not only saves one (costly) kernel trap, but also has different semantics: it prevents a race condition if the recipient sends the reply using nonblocking IPC. The remaining IPC primitives are nonblocking and are new in MINIX 3. NBSEND is a nonblocking variant of the synchronous SEND primitive that returns an error if the other party is not ready at the time of the call. ASEND supports asynchronous IPC with buffering of messages local to the caller. Arguments are a pointer to and size of a table with message buffers to be sent. Each slot in the table contains all the information needed to deliver the message: delivery flags, the IPC destination, the actual message, and a status word. The caller is not blocked, but immediately returns, and the kernel scans the table with messages to be sent, promising to deliver the messages as soon as possible. Finally, the asynchronous NOTIFY primitive supports servers and drivers in

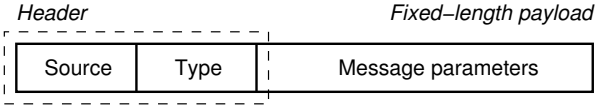


Figure 2.3: Format of fixed-length IPC messages in MINIX 3: the message header describes the sender and message type and is followed by a payload that depends on the type.

signaling events. The call is nonblocking and if the notification cannot be delivered directly, the kernel marks it pending in a statically allocated bitmap in the destination's process structure. Pending notifications of the same type are merged and will be delivered only once.

For reasons of simplicity, MINIX 3 uses only small, *fixed-length* messages. A MINIX 3 message is a structure with a message header containing the message source and type, followed by a union of different payload formats containing the message arguments. This structure is shown in Fig. 2.3. The message size depends on the CPU architecture and is determined at compile time as the largest of all types in the union. The message type and payload can be freely set by the sender, but the kernel reliably patches the sender's IPC endpoint into the message's source field upon delivery. In this way, the receiver can always find out who called.

2.2 Driver Management

Since all servers and drivers are normal user processes, they can be controlled and managed like ordinary applications. Although device drivers were initially part of the MINIX 3 boot image, we added support for starting and stopping drivers on the fly. In particular, the *driver manager* is used to coordinate this procedure; it manages all servers and drivers in the system. The system administrator can request services from the driver manager, such as starting a new driver, using the *service utility*. For example, the most basic command to start a new driver is:

```
$ service up <driver binary> –dev <device node>
```

Likewise, restarting or stopping a driver can be done with the commands service refresh and service down, respectively. There is also support to update a driver with a new version while it is still running. This procedure is called a *dynamic update* and can be requested via the command service update.

In addition to the –dev parameter shown above, the service utility supports several more advanced parameters to configure the system. For example, the parameters for starting a new driver with service up are listed in Fig. 2.4. The first few parameters control the dynamics of starting a new server or driver. Briefly, –args allows passing an argument vector to the component to be started, just like passing command line parameters to an application. The –dev parameter is used for drivers only and causes the VFS server to be informed about the associated device node. With –label a custom name can be specified. Because each process has a unique, kernel-generated IPC endpoint, system processes cannot easily find each other. Therefore, we introduced stable identifiers consisting of a human-readable name plus an optional number that are published in the data store.

The other parameters can be used to set a custom policy for the system's fault-tolerance mechanisms. To start with, drivers are associated with an isolation policy

Parameter	Default	Explanation
–args <argument string>	No arguments	Arguments passed upon executing the driver
–dev <device node>	No device	Device node to be associated with the driver
–isolation <file name>	*drivers.conf*	Configuration file with isolation policy
–label <identifier>	Binary name	Stable name published in the data store
–mirror <boolean>	False	True if binary should be mirrored in memory
–period <time in seconds>	5 seconds	Period between driver heartbeat requests
–recovery <file name>	Direct restart	Shell script governing the recovery procedure

Figure 2.4: Parameters supported by the service up call to configure a newly started component.

that controls which privileged operations they may perform. By default driver privileges are listed in */etc/drivers.conf*, but a different configuration file can be specified using –isolation. An example of a concrete isolation policy is described in Sec. 3.4. In order to check liveness of all drivers, the driver manager can periodically request a heartbeat message. The –period parameter allows fine-tuning the frequency of checking. The driver manager can automatically restart failed components, but if more flexibility is needed, a shell script with a more advanced recovery policy can be specified with –recovery. Finally, the –mirror flag tells the driver manager whether it should make an in-memory copy of the driver binary. This feature is used, for example, for recovering failed disk drivers. How these fault-tolerance mechanisms work is the subject of this thesis.

The procedure to start a driver is a strictly defined sequence of events, as illustrated in Fig. 2.5. The steps are as follows: (1) the administrator decides on a driver policy and calls the service utility, which performs some sanity checks and (2) forwards the request to the driver manager. The driver manager verifies that the caller is

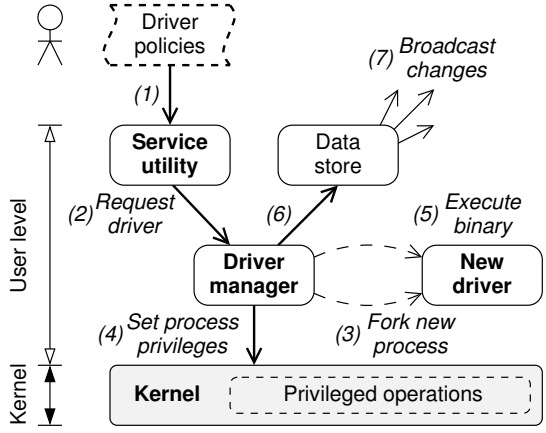


Figure 2.5: The administrator uses the service utility to request drivers and set isolation and recovery policies. The driver manager starts the driver and publishes its details in the data store.

authorized and stores the policy. Then the driver is started by the driver manager: (3) it creates a new process for the driver and looks up the IPC endpoint that uniquely identifies the process, (4) it informs other parts of the system about the new process and the privileges granted by the isolation policy, and (5) it executes the driver binary once the process has been properly isolated. Finally, (6) the details of the newly started driver are published in the data store and (7) dependent components are notified about the new system configuration.

The *data store* makes it possible to configure the system dynamically without hardcoding all dependencies. In its essence, the data store is a database server that allows system components to store and retrieve integer values, character strings, or even entire memory regions by name. System processes can store data either privately or publicly. A special feature of the data store is that it provides publish-subscribe semantics for any data that is stored publicly. This feature helps to reduce dependencies by decoupling producers and consumers. Components can subscribe to selected events by specifying the identifiers or regular expressions they are interested in. Whenever a piece of data is published or updated the data store automatically broadcasts notifications to all subscribed components.

These properties make the data store very suitable as a name server. Upon loading a new server or driver, the driver manager publishes the stable name and IPC endpoint in the data store, so that all dependent components are automatically notified. As an example, the network server is subscribed to the key 'eth.*' in order to receive updates about the system's network drivers. Therefore, the network server is immediately alerted whenever a network driver is started or restarted, and can start initialization or recovery, respectively.

2.3 Isolating Faulty Drivers

We now present an overview of the MINIX 3 isolation architecture. We use the term *isolation architecture* to indicate that the trusted parts of the system enforce certain restrictions upon the untrusted parts. In particular, as discussed in Sec. 1.4.2, we require drivers to be constrained according to the principle of least authority (POLA). While we primarily focus on the OS software, it is important to realize that some of our protection techniques also rely on hardware support. Altogether this is defined as the *trusted computing base* (TCB): "the totality of protection mechanisms within a computer system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy [Department of Defense, 1985]." Below, we first introduce our isolation architecture and then briefly discuss how this interacts with the hardware.

2.3.1 Isolation Architecture

The MINIX 3 isolation architecture is realized by combining several building blocks. In particular, drivers are isolated in three different ways:

- System-wide structural constraints.
- Static per-driver isolation policies.
- Dynamic access-control mechanisms.

As a baseline, each driver is run as an independent user process. This takes away all privileges and renders each driver harmless. However, because this protection is too coarse-grained, we have also provided static mechanisms to grant fine-grained access to resources needed by the driver. In addition, we have developed dynamic mechanisms that support safe run-time data exchange.

System-wide Structural Constraints

The use of a microkernel-based multiserver design that compartmentalizes the OS brings several dependability benefits. To begin with, minimizing the kernel reduces its complexity, makes it more manageable, and lowers the number of bugs it is likely to contain. At about 7500 LoC the kernel is sufficiently small that a single person can understand all of it, greatly enhancing the chance that in the course of time all the bugs can be found. The small size of the kernel may even make it practical to verify the code either manually or using formal verification [Klein, 2009]. This provides a solid foundation to build an OS upon.

Next, the fact that each application, server, and driver is encapsulated in a separate user-mode process with a private address space is crucial to isolate faults. First, because drivers no longer run with kernel-mode CPU privileges, they cannot directly execute potentially dangerous instructions and cannot circumvent the restriction mechanisms implemented by the rest of the OS. Second, because the MMU hardware enforces strict, process-based, *address-space separation*, many problems relating to memory corruption are structurally prevented. OSes are usually written in C and C++, which tend to suffer from bad pointer errors. If an offending process causes a CPU or MMU exception, it will be killed by the process manager and given a core dump for future debugging, just like any other user process. Third, normal UNIX protection mechanisms also apply. For example, all drivers run with an unprivileged user ID in order to restrict POSIX system calls.

At the user level, the OS processes are restricted in what they can do. In order to support the servers and drivers in doing their job, the kernel exports a number of kernel calls that allow performing privileged operations in a controlled manner, as described below. In addition, servers and drivers can request services from each other. For example, device drivers no longer have privileges to perform I/O directly, but must request the kernel to do the work on their behalf. Likewise, memory allocation is done by sending a request to the process manager. While these mechanisms still allow drivers to use privileged functionality, the use of no-privilege defaults makes access control more manageable. Effectively, the system provides multiple levels of defense with increasingly finer granularity.

Static Per-driver Isolation Policies

Since each server and driver is likely to have different requirements, we associated each with an isolation policy. Different per-driver policies that grant fine-grained access to the exact resources needed can be defined by the administrator. For example, access to I/O resources is assigned when drivers are started. In this way, if, say, the printer driver tries to write to the disk controller's I/O ports, the kernel will deny the access. Fig. 2.6 lists various other resources, such as IPC and kernel calls, that can be restricted in MINIX 3. Isolation policies are stored in simple text-based configuration files, such as */etc/drivers.conf*. Each driver has a separate entry in the configuration file, listing the exact resources granted. Sec 3.4 gives a case study of a network driver and provides an example configuration file in Fig. 3.7.

Although this research does not pertain to specific policies—it focuses on enforcement mechanisms instead—an interesting question is who is responsible for policy definition. Precautions are needed to prevent driver writers from circumventing the system's protection mechanisms by demanding broader access than required. In our current implementation, the system administrator is responsible for policy definition or inspection. We feel that the MINIX 3 policies are sufficiently simple to do so, but the system can be augmented with other approaches, if need be. Possible extensions include having a trusted third-party vet and sign policies or automating policy generation based on source-code annotations.

Policy enforcement is done by the both the driver manager and the trusted servers and drivers in the OS. As mentioned in Sec. 2.2, loading a driver and installing its isolation policy is done in three steps. First, the driver manager forks a new process for the driver and looks up the child's unique, kernel-generated IPC endpoint. The endpoint is used as a handle to identify the new process. Second, the driver manager informs the kernel and selected servers and drivers about the isolation policy so that it can be enforced at run time. This is done by passing the IPC endpoint and the resources granted to the components listed in Fig. 2.6. Finally, with the isolation policy in place, the child process is made runnable and can safely execute the driver

Resource key	Policy enforcement	Explanation
ipc calls	IPC subsystem	Restrict IPC primitives that may be used
ipc targets	IPC subsystem	Restrict IPC destinations that may be called
ipc kernel	Kernel task	Restrict access to individual kernel calls
driver	Driver manager	Control if driver can manage other drivers
isa io	Kernel task	Mediate legacy ISA device input and output
isa irq	Kernel task	Mediate legacy ISA device interrupt-line control
isa mem	Kernel task	Map legacy ISA device memory into a driver
pci device	PCI-bus driver	Restrict access to a single PCI device
pci class	PCI-bus driver	Restrict access to a class of PCI devices

Figure 2.6: Resources that can be configured via per-driver isolation policies in MINIX 3. By default all access is denied. Fine-grained access can be granted by adding resources to a driver's policy.

binary. When the driver attempts to make a privileged call, the callee looks up the policy installed by the driver manager and verifies that the caller is authorized before servicing the request.

Dynamic Access-control Mechanisms

Finally, we have added two dynamic access-control mechanisms that support drivers in exchanging data. Strict, process-based, address-space separation provided by the MMU hardware is too restrictive, since applications, servers, drivers, and hardware devices often need to access parts of each other's memory. For example, the application that wants to store a file on the hard disk must exchange data with the file server, which buffers disk blocks in its cache, and the disk driver, which is responsible for the actual I/O. However, because memory allocation typically involves dynamically allocated memory ranges, these kinds of interactions cannot be handled by structural constraints and isolation policies. Instead, we developed two run-time mechanisms that allow for safe, fine-grained memory access.

First, in order to support safe interprocess memory access, we have developed a new delegatable *memory grant* mechanism that enables byte-granular memory sharing without compromising the address-space separation offered by the MMU. A process that wants to grant selective access to its memory needs to create a capability listing the grantee's IPC endpoint, the precise memory area, and access rights. The grant is stored in a table known to the kernel and can be made available to another process by sending the grant's index into the table. The grantee then can copy to or from the granter using a kernel call that takes the address of a local buffer and the memory grant. The kernel looks up the memory grant in the grant table in order to verify that access is permitted and makes the actual copy with perfect safety. Zero-copy protocols are also supported through grant-based memory mappings.

Second, we rely on hardware support to protect against peripheral devices that use *direct memory access* (DMA). DMA is a powerful I/O construct that allows devices to operate directly on main memory, bypassing the protection offered by CPU and MMU hardware. DMA can be controlled, however, using an *I/O memory management unit* (IOMMU) that keeps tables with memory ranges accessible from the device layer. The IOMMU works similar to the traditional MMU, with the distinction that the MMU provides memory protection for CPU-visible addresses, whereas the IOMMU provides memory protection for device-visible addresses. If a driver wants to use DMA, it must request the trusted IOMMU driver to set up the access rights before initiating the DMA transfer. The IOMMU driver validates the request and, if access is allowed, sets up the IOMMU tables for the driver's device. Only access into the driver's own address space is allowed. Nevertheless, it is still possible to perform DMA directly to or from the address space of the end consumer by setting up a safe, grant-based memory mapping. This design protects both the OS and user applications against memory corruption. The next section gives further background on the IOMMU's working.

2.3.2 Hardware Considerations

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 that the time has come to revisit design choices made in the past.

Support for Isolation

As a first example, older PCs have no means to protect against memory corruption due to unauthorized DMA operations. As mentioned above, our solution is to rely on IOMMU support [Intel Corp., 2008; Advanced Micro Devices, Inc., 2009]. In particular, we have implemented support for AMD's Device Exclusion Vector (DEV). The IOMMU logically sits between the device bus and main memory, as shown in Fig. 2.7, and works mostly the same as a normal MMU. The IOMMU intercepts all memory access attempts from devices, looks up the I/O page table associated with the device, determines whether the access is permitted, and translates the I/O address requested to the physical memory address to be accessed. The IOMMU protects main memory against untrusted devices, just like the MMU protects memory against untrusted programs. However, in contrast to normal MMUs, which raise an MMU exception upon an unauthorized access attempt, current-generation IOMMUs do not provide a direct indication to the I/O device if a translation fails. Instead, rejected DMA writes are simply not executed, whereas rejected DMA reads typically cause the IOMMU to set all bits in the response to 1. Exceptions can be detected though by reading out the IOMMU's status register. Alternatively, the IOMMU can be configured to write the error to an in-memory event log and signal an interrupt in order to put the OS in control.

Another example relates to the working of the peripheral bus that connects peripheral devices to the CPU. In particular, we identified a problem relating to in-

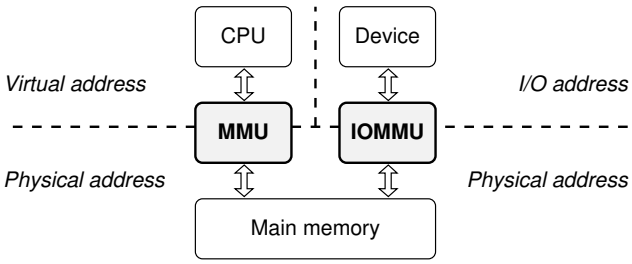


Figure 2.7: The MMU hardware provides memory protection for CPU-visible addresses, whereas the IOMMU hardware provides memory protection for device-visible addresses.

terrupt handling on the Peripheral Component Interconnect (PCI) bus that is in widespread use. Interrupts support drivers in handling I/O efficiently: instead of repeatedly polling the device, the driver can wait for a hardware *interrupt request* (IRQ) signaling that the device needs service. Because the number of IRQ lines on the interrupt controller is limited, devices sometimes have to share a single IRQ line, which may cause driver interdependencies. The PCI standard mandates level-triggered IRQ lines, which means that an interrupt is signaled by asserting the IRQ line to its active level, and holding it at that level until serviced. Therefore, a driver that fails to acknowledge an IRQ on a shared IRQ line effectively blocks the IRQ line for other drivers, since the level-triggered nature makes it impossible to detect status changes in other devices. The IRQ line is freed only if the driver takes away the reason for interrupting, which requires device-specific interrupt handling at the driver level and cannot be solved in a generic way by the OS. The newer PCI Express (PCI-E) standard that replaces PCI provides a structural solution based on message-signaled interrupts (MSI). A device that needs service writes a message into a special memory area and the chipset inspects the message to trigger the corresponding CPU interrupt. MSI alleviate the problem of shared interrupt lines because the interrupt consists of a short message rather than a continuous condition: a single driver failure can no longer block interrupts from devices that share the IRQ line. Still, sharing is unwanted because an IRQ from a single device triggers the interrupt service routines of all drivers associated with the IRQ line. Therefore, we avoided the problem altogether by using a dedicated IRQ line for each device.

Performance Perspective

In addition to improved hardware dependability, computing performance per unit cost has increased to the point where software techniques that previously were infeasible or too costly have become practical. For example, with modular designs the costs of context switching when control is transferred from one system process to another is one of the main performance bottlenecks. While the relative costs can still be significant, the absolute costs of context switching has gone down steadily with improved CPU speeds. For example, the costs of an IPC roundtrip between two processes, that is, two IPC messages, was reduced from 800 μs for Amoeba running on a 16-MHz Motorola 68020 processor [Renesse et al., 1988] to only 10 μs for L3 running on a 50-MHz Intel 486-DX processor [Liedtke, 1993]. Our own measurements showed that the costs of an IPC roundtrip for MINIX 3 running on a 2.2-GHz AMD Athlon 64 3200+ processor is 1 μs. A program executing 10,000 system calls/sec thus wastes only 1% of the CPU on context switching. With workloads ranging from 290 kernel-driver interactions/sec for low-bandwidth disk I/O up to 45,000 packet transmissions/sec for high-bandwidth gigabit Ethernet [Swift et al., 2006], a small overhead is still expected for certain applications. Nevertheless, these data points show that the increase in computing performance has dramatically reduced the user-perceived overhead of modular designs.

These results should be attributed to performance improvements made possible by Moore's law, which postulates that CPU transistor counts double about every two years [Moore, 1965]. Although MINIX 3 has the lowest absolute IPC roundtrip overhead, when taking processor speeds into account, the relative performance of MINIX 3 is the worst of the three examples we studied above. A rough estimate shows that the MINIX 3 IPC implementation might be 10–100 times slower than the L4 IPC implementation. However, MINIX 3 has never been optimized for performance, and the performance can no doubt be improved through careful analysis and removal of bottlenecks [e.g. Liedtke, 1995]. Instead, we have built on the premise that computing power is no longer a scarce resource, which is generally true on desktops nowadays, and tried to address the issue of untrusted drivers that pose a threat to OS dependability.

2.4 Recovering Failed Drivers

Building on the isolation architecture introduced above, we have attempted to recover failed drivers transparently to applications and without user intervention. In many other areas, both in hardware and software, such failure-resilient designs are common. For example, RAIDs are disk arrays that continue functioning even in the face of drive failures. The TCP protocol provides reliable data transport, even in the face of lost, misordered, or garbled packets. DNS can transparently deal with crashed root servers. Finally, *init* automatically respawns crashed daemons in the application layer of some UNIX variants. In all these cases, the underlying failure is masked, allowing the system to continue as though no errors had occurred. In this thesis, we have extended this idea to the OS.

2.4.1 Defect Detection and Repair

While a human user detects a driver crash when the system freezes, the OS needs different techniques. Therefore, the driver manager monitors all drivers at run time and takes corrective measures if a problem is detected. In many cases, failures can be handled internal to the OS with no application-visible retries, weird signals, reconnection requests, chance of data loss, performance hiccups, and so on.

Run-time Defect Detection

The driver manager uses three orthogonal defect detection techniques. First, the driver manager can detect component crashes because it is the parent process of all drivers and servers. If a server or driver crashes or otherwise exits, it becomes a zombie process until the driver manager collects it, the same way all UNIX systems allow parent processes to collect child processes that have exited. Second, the driver manager can periodically check the status of selected drivers. Currently, this is done using heartbeat messages that can be configured on a per-driver basis via the service

utility. If no reply is received within the time-out interval, further action is taken. More advanced techniques would be a straightforward extension. Third, the driver manager can be explicitly instructed to replace a malfunctioning component with a new one. Explicit updates are done if the administrator requests a dynamic update or if a trusted OS component files a complaint about a subordinate process.

Policy-driven Recovery

The basic idea underlying our design is that restarting a failed component may take away the root cause of the failure and thereby solve the problem [Gray, 1986; Chou, 1997]. As discussed in Sec. 2.5, this hypothesis indeed holds for a range of transient physical faults, interactions faults, and elusive development faults that remain after testing. The high-level recovery procedure is illustrated in Fig. 2.8: (1) an application requests the virtual file system (VFS) to perform an I/O operation, (2) VFS forwards the request to the corresponding driver, and (3) the driver starts processing the request, but crashes before it can send the reply. Since the driver manager monitors all drivers, it detects the failure and initiates the recovery procedure: (4) the driver manager replaces the failed driver with a new copy and (5) informs VFS about the new configuration. Then, VFS scans its tables and notices the pending I/O operation: (6) the request is resubmitted to the restarted driver and, finally, (7) the I/O operation can be successfully completed. While all this is happening, the system continues to run normally and no processes need to be terminated.

If a problem is detected, the driver manager fetches the recovery policy of the failed component from its tables in order to determine what to do. By default failed components are replaced with a fresh copy, but if special recovery steps are needed

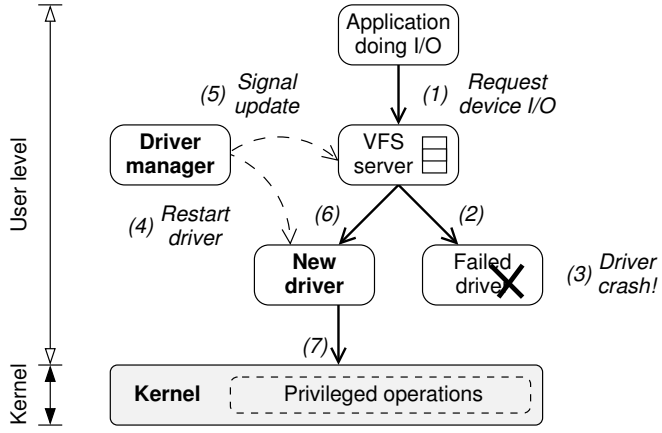


Figure 2.8: Basic idea underlying our failure-resilient OS. If the driver manager detects a driver failure, it looks up the driver's recovery policy, and may restart the driver and notify dependent components so that the system can continue to operate normally.

or wanted, the administrator can provide a shell script that governs the recovery procedure. The recovery script can be specified via the service utility upon loading the driver. The script may, for example, record the event in a log file, move the core dump of the dead process to a special directory for subsequent debugging, send an email to a remote system administrator, or even notify the driver's manufacturer. This design brings full flexibility and helps to automate system administration. Concrete examples of recovery scripts are given in Secs. 4.2 and 4.5.

2.4.2 Assumptions and Limitations

One of the main challenges with restarting failed components is that it is sometimes impossible to recover *internal state* lost during a failure. Examples of internal state include device configuration parameters, information about pending I/O requests, and data buffers. Fortunately, state management turns out to be a minor problem for many drivers. In our experience with the MINIX 3 RAM-disk, floppy-disk, and hard-disk drivers as well as a range of network-device drivers, drivers contained no or very limited internal state, and recovery of state was straightforward. Therefore, the assumption underlying our failure-resilience mechanisms is that drivers that are stateless or contain only limited state that is updated infrequently.

Nevertheless, our design also provides limited support for recovering stateful servers and drivers. In principle, lost state can be retrieved from the data store, which allows drivers to store privately data that should persist between crashes. For example, an audio driver could store the sound card's mixer settings in the data store and the RAM-disk driver could store the base address and size of the RAM disk's memory. However, the data store cannot ensure data integrity and cannot tell which part an outstanding operation has already completed and which part has not. In order to do so, operations that update internal state may have to implement some form of checksumming and should be remodelled as transactions that are either committed or aborted as one logical unit. We did not investigate this option though, because relatively heavy driver modifications may be required in order to achieve optimal results. Sec. 4.2.3 provides more information on MINIX 3's support for state management as well as its current shortcomings.

In addition, there are certain inherent limitations that prevent recovery transparent to applications and end users for certain classes of I/O. The two requirements for effective recovery are: (1) the I/O stack must guarantee data integrity and (2) I/O operations must be idempotent. The former means that data corruption can be detected. The latter means that I/O operations can be reissued safely with the same final outcome. For example, transparent recovery is possible for network drivers, because the TCP protocol can detect garbled and lost packets and safely retransmit the corrupted data. In contrast, partial recovery is supported for character-device drivers, because the I/O is not idempotent and an I/O stream interruption is likely to cause data loss. For example, with streaming audio and video applications the user may experience hiccups when a driver fails and needs to be recovered.

2.5 Fault and Failure Model

Although we aim to improve OS dependability by tolerating faults and failures in device drivers, we are obviously aware that we cannot cure all bugs. Therefore, this section zooms in on the exact fault and failure types our system is designed for. We first want to clarify the causal relationship between faults and failures using the *fault → error → failure model*: a fault that is triggered during execution may lead to an erroneous system state that appears as failure if it propagates to the module's interface. For example, a memory defect is a fault that leads to an error if program data is corrupted due to bit flips. This may go undetected until the erroneous data is used and triggers an exception that causes a component failure.

Our system is primarily designed to deal with device-driver failures caused by soft *intermittent faults* [Avižienis et al., 2004]. Such faults were found to be a common crash cause and represent a main source of downtime [Gray, 1986; Chou, 1997]. Intermittent faults include, for instance, transient physical faults, interaction faults, and elusive development faults or residual faults that remain after testing because their activation conditions depend on complex combinations of internal state, external requests, and run-time environment. For example, drivers may crash as a response to application-level requests, device interactions such as I/O and interrupt handling, or kernel events such as switching to a lower power state or swapping out memory pages. Bugs in this category are sometimes referred to as *Heisenbugs* (named after the Heisenberg uncertainty principle), since they disappear or manifest differently when an attempt is made to study them.

The Heisenbug hypothesis may be exploited to improve software fault tolerance, since retrying a failed operation may take away the root cause of the failure and thereby solve the problem [Gray, 1986]. As an example, consider resource leaks, which represent 25.7% of all defects found in a study of over 250 open-source projects with a combined code base exceeding 55 MLoC [Coverity, Inc., 2008]. A resource leak, such as not releasing memory buffers or file handles no longer needed, may trigger an unexpected failure, but tends to go away after a restart. Other examples of intermittent problems include:

- CPU and MMU exceptions triggered by unexpected user or device input.
- Failed attempts to exploit a vulnerability, such as a buffer overflow attack.
- Race conditions due to unexpected software or hardware timing issues.
- Infinite loops caused by unresponsive hardware or an internal inconsistency.
- Aging bugs, such as resource leaks, that cause a driver to fail over time.
- Memory bit flips that disrupt the execution path and trigger an exception.
- Temporary device failures that require hardware reinitialization.

While hard to track down, these bugs illustrate that many problems can be cured by replacing a failing or failed component with a fresh instance, even if the exact underlying causes are unknown. However, this strategy requires that failures are *fail-stop*, that is, failures must be detected before they can propagate and corrupt the

rest of the OS [Schlichting and Schneider, 1983]. Hence, drivers must be properly isolated to confine the problem in the first place.

Previous projects have also attempted to improve OS dependability by retrying failed operations in a slightly different execution environment. For example, a case study on the Tandem NonStop system showed that 131 out of 132 bugs were intermittent and could be solved by reissuing the failed operation [Gray, 1986]. Likewise, a study of the IBM MVS/XA OS showed that up to 64% of the errors in critical jobs could be remedied through a retry [Mourad and Andrews, 1987]. Next, the Tandem GUARDIAN system obtained a level of 75% software fault tolerance by performing backup execution in a different process [Lee and Iyer, 1995]. Furthermore, Linux shadow drivers were able to recover automatically 65% of 390 driver failures induced by fault-injection experiments [Swift et al., 2006]. Finally, automatic restarts in the Choices OS kernel resulted in recovery 78% of the time [David and Campbell, 2007]. The working of our design in MINIX 3 is based on the same ideas: malfunctioning drivers are replaced with a fresh copy and failed operations are retried in the new execution environment.

In order to give a balanced viewpoint, we also list a number of known limitations. First, our design cannot cure hard permanent faults, such as algorithmic and deterministic failures that repeat after a component restart. Bugs in this category are sometimes referred to as *Bohrbugs* (named after the Bohr atom model), since they manifest consistently under the same conditions. However, recurring problems can be tracked down more easily, and once the bug has been found, MINIX 3 supports a dynamic update with a new or patched version of the component. Next, we cannot deal with Byzantine failures, including random or malicious behavior where a driver perfectly adheres to the specified system behavior but fails to do its job. Such bugs are virtually impossible to catch in any system. Furthermore, we cannot deal with timing failures, for instance, if a deadline of a real-time schedule is not met, although the use of heartbeat messages helps to detect unresponsive components. Finally, our system cannot recover from permanent physical failures, for example, if the hardware is broken or cannot be reinitialized by a restarted driver. It may be possible, however, to perform a hardware test and switch to a redundant hardware interface, if available [Bartlett, 1981]. We did not investigate this option, though.

In the remainder of this thesis, we focus on confinement and recovery of intermittent faults and failures, which, as argued above, represent an important area where our design helps to improve OS dependability.

Chapter 3

Fault Isolation

Perhaps someday software will be bugfree, but for the moment all software contains bugs and we had better learn to deal with them. In most OSes, faults and failures can disrupt normal operation. For example, commodity OSes such as Windows, FreeBSD and Linux use a monolithic design where a single driver fault can easily propagate and potentially lead to a system-wide failure, requiring a reboot of the machine. We believe that this brittleness is unacceptable to end users and businesses alike, and have investigated techniques to improve OS robustness.

The remainder of this chapter is organized as follows. Sec. 3.1 discusses general isolation principles and classifies privileged driver operations that, unless properly restricted, are root causes of fault propagation. Next, Sec. 3.2 introduces the MINIX 3 user-level driver framework and Sec. 3.3 details the techniques used to enforce least authority. Finally, Sec. 3.4 illustrates driver isolation with a case study of the MINIX 3 networking stack.

3.1 Isolation Principles

Below, we introduce the design principle underlying our design and show how it can be applied to drivers. We present a classification of operations that are root causes of fault propagation as well as a set of general rules for isolating drivers.

3.1.1 The Principle of Least Authority

While there is a broad consensus among researchers that drivers need to be isolated, the central issue to be addressed always is "Who can do what and how can this be done safely?" Whether the isolation of untrusted drivers is based on in-kernel sandboxing, virtualization techniques, formal methods, or user-level frameworks, the same question arises in each approach.

We strongly believe that *least authority* or *least privilege* should be the guiding principle in any dependable design. In short, this design principle states that each

component should be able to access only those resources needed for its legitimate purpose. A more complete definition reads: "Every program and every user of the system should operate using the least set of privileges necessary to complete the 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 to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur [Saltzer and Schroeder, 1975]." This chapter studies how this idea can be applied to drivers in the OS.

3.1.2 Classification of Privileged Operations

This section classifies the privileged operations needed by OS extensions and drivers in particular. We identified four orthogonal classes that map onto the core components of any computer systems: CPU, memory, peripheral devices, and system software. Fig. 3.1 summarizes the privileged operations. Drivers are special in that they perform device I/O, but the other classes equally apply to other kinds of OS extensions. We briefly introduce each class and the threats posed by it below. In Sec. 3.3, we will discuss how our model deals with each one.

Class I: CPU Usage A process that runs in kernel mode has access to the full set of privileged CPU instructions that can be used to bypass higher-level protection mechanisms. A kernel-mode driver can, for example, reset the page tables, perform I/O, disable interrupts, or halt the processor. These functions are vital to the correct operation of the OS and cannot be exposed to untrusted code without putting the system at risk. Nevertheless, because of the low-level nature of device drivers, they must be able to perform certain privileged operations that are not normally available to user-level application processes.

Class	Privileged operation	Explanation
CPU usage	CPU instructions	Use of kernel-mode CPU instructions
	CPU time	Scheduling and CPU time consumption
Memory access	Memory references	Access to process address space
	Copying and sharing	Data exchange between processes
	Direct memory access	DMA operation from driver's device
Device I/O	Device access	Access to peripheral devices
	Interrupt handling	Access to the device's IRQ line
IPC	IPC primitives	Safety of low-level IPC subsystem
	Process interaction	Asymmetric trust relationships
	System services	Requesting privileged services

Figure 3.1: Classification of privileged driver operations. The classes map onto the core components of computer systems: CPU, memory, peripheral devices, and system software.

A related problem is using excessive CPU time or CPU hogging, which may lead to performance problems or even bring down the system. For example, consider a device driver that winds up in an infinite loop and keeps executing the same code over and over again. Since low-level driver code, such as the device-specific interrupt handler, often does not run under the control of the process scheduler, it may hang the system if it does not run to completion. An analysis of Windows XP crashes found that the error condition THREAD_STUCK_IN_DEVICE_DRIVER ranked as the second most-frequent (13%) crash cause [Ganapathi et al., 2006]. Another study of Linux drivers found 860 cases of infinite device polling that may cause the OS to hang due to misplaced trust in the hardware [Kadav et al., 2009].

Class II: Memory Access Since drivers often need to exchange data with system servers and application programs, a particularly important threat is memory corruption due to unauthorized memory access. For example, drivers typically perform device input and output and need to copy data to and from buffers in different address spaces. A pointer provided by the caller cannot be used directly by the driver and the device, but needs to be translated to a physical address before it can be used. A recent study found that 9 out of 11 pointer bugs were in device drivers [Johnson and Wagner, 2004]. Translation errors or copying more data than the buffer can hold may cause memory corruption. Indeed, a study of field failures in OSes has shown that memory corruption is one of the most important crash causes [Sullivan and Chillarege, 1991]. In 15% of the Windows crashes, the memory corruption is so severe that crash dump analysis cannot pinpoint the bug(s) or even the driver responsible for the crash [Orgovan and Dykstra, 2004].

Direct memory access (DMA) is a special case of device I/O that must be restricted to prevent corruption of arbitrary memory. A device that supports DMA can directly transfer data to an arbitrary location in physical memory without intervention of the CPU and MMU, bypassing both software and hardware memory protection mechanisms. Legacy ISA devices typically rely on the on-board DMA controller, whereas PCI devices may have built-in DMA capabilities. The term *bus-mastering DMA* is used if a device can take control of the bus and initiate the DMA transfer itself. The kernel is not in control during the DMA transfer and cannot verify whether the operation requested can be executed safely. In addition, the I/O address used by the device is not checked by the MMU hardware, as shown in Fig. 2.7. Therefore, a buggy or malicious driver whose device is capable of DMA can potentially overwrite any part of physical memory by using an incorrect I/O address.

Class III: Device I/O It is important to restrict access to I/O ports and registers and device memory in order to prevent unauthorized access. For example, the network driver should be able to touch only the network interface card, and may not access, say, the PCI bus or disk controller. However, with kernel-level drivers, nothing prevents a driver from interfering with other peripheral devices. If multiple drivers simultaneously operate on the same device, resource conflicts are likely and may

result in data corruption or even cause the system to stop functioning. Unfortunately, programming device hardware is error-prone due to its low-level interactions and lack of documentation [Ryzhyk et al., 2009a].

Furthermore, interrupt handling poses several problems because interrupts are inherently asynchronous and have to be handled at the lowest level for performance reasons. When the device needs service, it raises an interrupt request (IRQ) to put the kernel in control. The kernel, in turn, will look up the associated driver to perform the device-specific interrupt handling. Interrupt handlers often run at a high priority level and have to meet special (OS-specific) constraints, which makes interrupt handling error-prone. For example, the error condition IRQL_NOT_LESS_OR_EQUAL was found to be responsible for more (26%) Windows XP crashes than any other single error [Ganapathi et al., 2006].

Class IV: IPC Finally, interprocess communication (IPC) poses various threats relating to the system software. Although IPC is often associated with multiserver designs, IPC is also important for other approaches, since a means of communication between extensions and the core OS is always required. To illustrate the importance of IPC, measurements on MacOS X and OpenDarwin, which use the Mach IPC mechanism, reported 102,885 and 29,895 messages from system boot until the shell is available, respectively [Wong, 2003]. We conducted the same measurement on MINIX 3 and obtained a similar number of 61,331 messages that were exchanged between 33 independent components: 3 kernel tasks, 5 system servers, 15 potentially unreliable drivers, init, and 9 daemons.

Because the IPC subsystem is so heavily used by the rest of the OS, it must be designed for maximum robustness. With untrusted system code making many thousands of IPC calls per second, erroneous invocations or call parameters, such as invalid IPC endpoints or bad message buffers, cannot be prevented. Furthermore, because both trusted and untrusted parts of the system rely on IPC, unauthorized access attempts may occur. For example, untrusted drivers should not be allowed to use kernel calls for process management. Finally, the use of global resources might lead to resource exhaustion when one or several clients collectively perform too many IPC requests. If message buffers are dynamically allocated, the IPC subsystem may run out of memory and no longer be able to serve new calls.

Even if the IPC subsystem itself works reliably, unexpected interactions between senders and receivers can potentially disrupt the system. For example, consider a buggy driver that corrupts the message contents, sends the reply to the wrong party, causes a deadlock due to a cyclic dependency, or simply does not respond to a request. In particular, *asymmetric trust* relationships between senders and receivers introduce several problems when synchronous IPC is used. For example, an untrusted client may block a server if it does not receive the server's reply [Shapiro, 2003]. We identified two further problems, shown in Fig. 3.2, where a driver acting as an untrusted server can block its client(s): the caller is blocked if the driver does not receive or reply to an incoming IPC call.

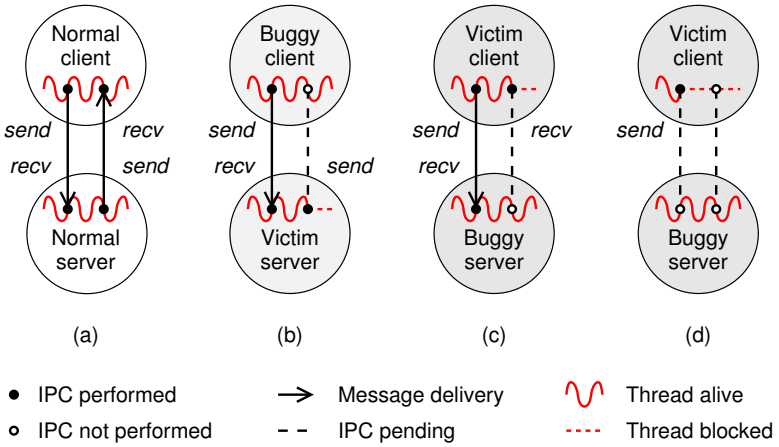


Figure 3.2: Asymmetric trust and vulnerabilities in synchronous IPC: (a) normal client-server roundtrip, (b) untrusted client blocks server, and (c) and (d) untrusted server blocks client.

A related power built on top of the IPC infrastructure, which routes requests through the system, is requesting privileged operations. If a driver is not allowed to perform a given operation directly, it should also be prevented from requesting another (privileged) process to do the work on its behalf. For example, isolating a driver in a private address space is not worth much if it can still ask the kernel to access arbitrary memory and the kernel blindly complies. Likewise, even though user-level drivers may not directly perform I/O, the driver's restrictions may be bypassed if the kernel exports a call to perform I/O on behalf of the driver.

3.1.3 General Rules for Isolation

We now briefly discuss in what sense these potentially dangerous privileged operations should be curtailed, rather than (just) how we have done it. As discussed above, the guiding principle is always to enforce strictly least authority upon untrusted code. This principle leads to the following general rules for isolation:

(I) Drivers may not have access to the full CPU. In particular, access to privileged CPU instructions must be denied or mediated by the kernel to prevent bypassing other protection mechanisms. Furthermore, drivers may not directly use CPU time, but must run under the scheduler's control.

(II) Drivers may not access main memory unless needed by the driver to do its job. Besides access to the memory associated with the driver process, there must be a mechanism to exchange data safely with the components it needs to interact with. DMA from the device level must be permitted only to the driver's own memory or a memory region granted to the driver.

(III) Drivers may not access I/O resources except those belonging to the device controlled by the driver. Any other I/O, including generic ones such as querying the PCI bus to look up a device's I/O resources or programming the IOMMU to allow DMA access, must be mediated by trusted drivers.

(IV) Drivers may not directly perform IPC to access arbitrary services. Instead, both the IPC subsystem and service provider must provide mechanisms to restrict communication and grant selective access to privileged services that are performed on behalf of the driver.

In sum, these rules enforce *no-privilege defaults*: every driver operation is denied unless authorization explicitly granted.

Any system that wants to isolate faults in device drivers should implement the above set of rules. In the following sections, we describe how we have implemented these rules in MINIX 3.

3.2 User-level Driver Framework

Because the kernel runs with all the privileges of the machine, we started out by removing all the drivers from the kernel and transforming them into independent user-level processes [Herder, 2005]. 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. With the exception of the clock task, which is very simple and remains in the kernel to facilitate process scheduling, all drivers have been removed from the kernel. This section briefly discusses how we moved drivers out of the kernel.

3.2.1 Moving Drivers to User Level

In order to transform kernel-level drivers into user-level drivers we analyzed their dependencies on the core OS and each other. The analysis showed that dependencies occur because of various reasons, including device I/O and interrupt handling, copying data from and to the rest of the OS, access to kernel information, debug dumps, and assertions and panics. Interestingly, some dependencies were caused by bad design, for example, when variables that were really local were declared global. Fortunately, these dependencies were easily resolved.

We were able to group the interdependencies into five categories based on who depends on whom or what. For each category, a different approach in removing the dependencies was needed:

(A) Driver-kernel Dependencies Many drivers touch kernel symbols (both functions and variables), for example, to copy data to and from user processes. The solution is to add new kernel calls to support the user-level drivers.

(B) Driver-driver Dependencies Sometimes one driver needs support from another. For example, the console driver may be needed to output diagnostics. Like above, new message types have been defined to request services from each other.

(C) Kernel-driver Dependencies The kernel can depend on a driver symbol, for example, to call a driver's watchdog function when a timer expires. Kernel events are now communicated to the user level using nonblocking notification messages.

(D) Interrupt-handling Dependencies Some interrupt handlers directly touch data structures of in-kernel device drivers. The solution is to mask the interrupt at the kernel level and notify to the corresponding user-level driver to perform the device-specific interrupt handling.

(E) Device-I/O Dependencies All drivers interact with the I/O hardware, which they cannot always do directly in a safe way at the user level. Therefore, several new kernel calls relating to I/O have been provided.

3.2.2 Supporting User-level Drivers

Since user-level drivers cannot perform privileged operations directly, the core OS provides support functionality to perform I/O, memory copying, and the like. This functionality is implemented by the kernel as well as various user-level servers and drivers that are part of the trusted computing base (TCB), such as the process manager, IOMMU driver, and PCI-bus driver. If a user-level driver needs to perform a privileged operation, it can no longer directly execute the operation by itself. Instead, the driver needs to request a more privileged, trusted party to perform the operation on its behalf. Because the driver manager informs the system about the permissible operations upon starting each driver, the TCB can control access to privileged resources: requests are carefully vetted against the policy installed, and executed only if the driver is permitted to make the call.

The kernel exports a range of kernel calls for privileged operations that may only be performed by the kernel. Fig. 3.3 summarizes the most important kernel calls added to support user-level drivers. All calls are handled by the kernel's system task, which is programmed as a main loop that repeatedly receives an IPC message, looks up the kernel call's request type, verifies that the caller is authorized, calls the associated handler function, and returns the result. For example, a driver can read from its device by sending to the kernel a message of type VDEVIO, including the I/O port(s) to be read as part of the message payload. The kernel will receive the message, verify that the driver is authorized to make the calls and that the I/O ports belong to the driver's device, perform the device I/O on behalf of the driver, and return the value(s) read in the reply message. Sec. 3.3 provides more details about the most important kernel calls and the way in which authorization works.

Kernel call	Purpose
SYS_VDEVIO	Read or write a given I/O port or vector of I/O ports (programmed I/O)
SYS_MEMMAP	Map device memory into caller's address space (memory-mapped I/O)
SYS_MAPDMA	Ensure DMA buffer is contiguous and pinned (direct memory access)
SYS_IRQCTL	Set or reset a hardware interrupt policy for a given interrupt line
SYS_SETALARM	Schedule a watchdog timer that causes a notification message
SYS_SETGRANT	Inform the kernel about the location and size of the memory grant table
SYS_SAFECOPY	Copy a capability-protected memory region between address spaces
SYS_SAFEMAP	Map a capability-protected memory region into caller's address space
SYS_GETINFO	Retrieve a copy of a kernel data structure or other system information
SYS_SYSCTL	Forward diagnostic output to the primary console and system log
SYS_PRIVCTL	Report a driver's permissible operations (used by the driver manager)

Figure 3.3: Overview of new kernel calls for device drivers. These kernel calls allow unprivileged drivers to request privileged operations that can only be done by the kernel.

In addition to kernel-level support, drivers are supported by a number of user-level servers and drivers. Besides general OS support from the POSIX servers, the IOMMU driver and PCI-bus driver are of particular importance for restricting untrusted drivers. The IOMMU driver and PCI-bus driver are similar to other drivers in the system, but their respective isolation policies allow them to access special hardware. Therefore, these drivers are considered part of the TCB. Privileged operations can be requested by sending a request message, just as is done for kernel calls. The IOMMU driver mediates access to the IOMMU hardware: it allows a driver to set up a memory map for use with DMA. Likewise, the PCI-bus driver mediates access to the PCI bus: it allows a driver to query the configuration space of its associated PCI device in order to look up the PCI device's I/O resources. The protection mechanisms used to ensure that untrusted drivers cannot request broader access than required are further detailed below.

3.3 Isolation Techniques

We now describe in detail 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 privileged resource. Our discussion follows the classification of privileged operations given in Sec. 3.1.2; the isolation techniques for each class are described in a separate subsection.

3.3.1 Restricting CPU Usage

CPU usage is restricted by the structural constraints imposed by a multiserver design. All drivers (except the clock task) have been removed from the kernel and are now run as independent user-level processes that are scheduled sequentially. This reduces both access to privileged instructions as well as CPU time.

Privileged Instructions

Although the x86 (IA-32) architecture begins executing in *real mode* when it is powered on, MINIX 3 switches to *protected mode* early during boot time so that it can use hardware protection features, such as virtual memory and protection rings. The kernel's bootstrap code then sets up a restricted execution environment for the OS. Only a few tasks that are part of the microkernel of about 7500 lines of code (LoC) are run with *kernel-mode* (ring 0) CPU privileges. All drivers are run in an ordinary UNIX process with *user-mode* (ring 3) 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.

Attempts by unprivileged code to access privileged instructions are denied or mediated by the kernel. If a user-mode process attempts to execute directly a privileged CPU instruction, the CPU raises an exception and puts the kernel in control. The kernel then checks which process caused the exception and sends it a POSIX signal, which forces a process exit if no signal handler has been installed. As discussed in Sec. 3.2.2, a small set of kernel calls is exported so that drivers can request privileged services in a controlled manner. The kernel checks whether the driver is authorized and performs the privileged operations on behalf of the driver.

CPU Time

With drivers running as UNIX processes, normal process scheduling techniques can be used to prevent CPU hogging. In particular, we have used a *multilevel-feedback-queue* (MLFQ) scheduler. Processes with the same priority reside in the same queue and are scheduled round-robin. When a process is scheduled, its quantum is decreased every clock tick until it reaches zero and the scheduler gets to run again. Starvation of low-priority processes is prevented by degrading a process' priority after it consumes a full quantum. This prevents drivers that wind up in an infinite loop from hogging the CPU. Moreover, since CPU-bound processes are penalized more often, interactive applications generally have good response times. Periodically, all priorities not at their initial value are increased so that processes with changing scheduling characteristics are not penalized unnecessarily.

There is an additional protection mechanism to deal with drivers that are 'stuck,' for example, due to an infinite loop. As discussed in Sec. 2.2, the driver manager can be configured to check periodically the driver's state. If the driver does not respond to a heartbeat request, the driver manager can replace it with a fresh copy or take another action depending on the driver's recovery script. This defect-detection technique is discussed in more detail in Sec. 4.1.

3.3.2 Restricting Memory Access

Memory access is restricted using a combination of software and hardware protection. Each process has a private address space that is protected by the MMU and

IOMMU hardware. In order to support data exchange between processes, either through copying or mapping, we have provided a scheme for safe run-time memory granting enforced by the kernel.

Memory References

We rely on MMU-hardware protection to enforce strict *address-space separation*. Each driver has a private, virtual address space depending on the driver's compile-time and run-time requirements. The address space contains the driver's program text, global and static data, heap, and execution stack. Upon starting a process the boundaries are determined by the process manager, which requests the kernel to program the MMU accordingly. The MMU translates CPU-visible addresses to physical addresses using the MMU tables programmed by the kernel. Page faults within the allowed ranges, for instance due to an unallocated stack page, are caught by the kernel and serviced transparently to the process. However, an unauthorized memory reference outside of the driver's address space results in an MMU exception that causes the driver to be killed.

Drivers that want to exchange data with other system processes could potentially use page sharing as provided by, for example, System V IPC and POSIX Shared Memory, but these models do not provide the flexibility and fine-grained protection needed to isolate low-level drivers. We identified several shortcomings. First, protection is based on group ID and user ID instead of individual drivers. Second, page sharing uses coarse-grained pages while byte-granular protection is needed for small data structures. Third, delegation of access rights is not supported. Fourth, access rights are not automatically invalidated if a process sharing memory crashes. 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*. A memory grant can be seen as a *capability* that can be transferred to another party in order to grant fine-grained access. In contrast, an *access control list* (ACL) does generally not support delegation and is more coarse-grained. Each grant defines a memory region 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 using the SETGRANT kernel call. Memory grants can be made available to another process by sending the grant's index into the table, known as the *grant ID*. The grant then is uniquely identified by the grantor's IPC endpoint and grant ID, and can be passed in a kernel call to perform privileged memory operations.

The structure of a memory grant is shown in Fig. 3.4. The grant's flags indicate whether the grant is in use, the grant's type, and the kind of access allowed. A *direct*

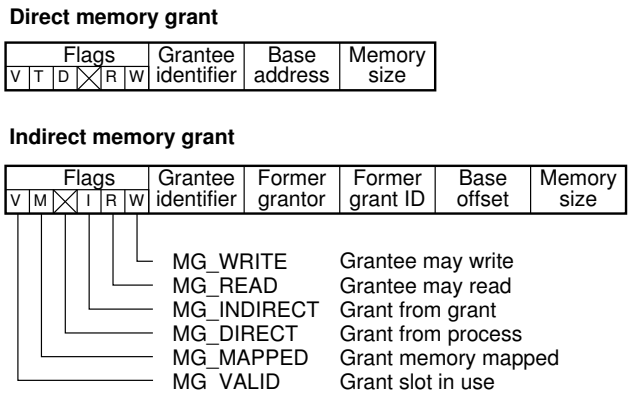


Figure 3.4: Structure of direct and indirect memory grants. Overview of memory grant flags.

grant (MG_DIRECT) means that a process A grants another process B limited access to a memory region in its own address space. The memory region is specified by a base address and size. The receiver of a direct grant, say, B, can refine and transfer its access rights to a third process C by means of an *indirect grant* (MG_INDIRECT). The memory region covered by an indirect grant is relative to the previous grant, and is specified by an offset and size. However, the target memory area is always in the address space of the process at the root of the grant chain. Finally, the R/W flags define the access type that is granted: read, write, or both.

Delegation of memory grants is supported via indirect grants and results in a hierarchical structure as shown in Fig. 3.5. This structure resembles recursive address spaces [Liedtke, 1995], but memory grants are different in their purpose, granularity, and usage—since grants protect data structures rather than build process address spaces. In the figure, the grantor creates a direct grant with read-write access to a 512-byte memory region starting at virtual address 0x400 and extending up to but not including address 0x600. The grant is stored in the grantor's grant table and has grant ID 2. The memory grant then is passed to the grantee by sending the grantor's IPC endpoint and grant ID using the system's normal IPC mechanisms. The figure also shows how the grantee creates two indirect grants. For example, the indirect grant with grant ID 4 allows read-only access to 256 bytes starting at an offset of 64 bytes relative to the original grant. Note that the target memory for an indirect grant always belongs to the *root* grantor process. The indirect grant contains the previous grantor's IPC endpoint, so that it is possible to follow to chain to the root and determine the precise access rights. In this case, the indirect grant gives access to the memory range starting at virtual address 0x440 and ending at 0x540.

When a process wants to access a memory area it has been granted, it needs to call the kernel, which verifies the grant's validity and performs the operation requested. 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

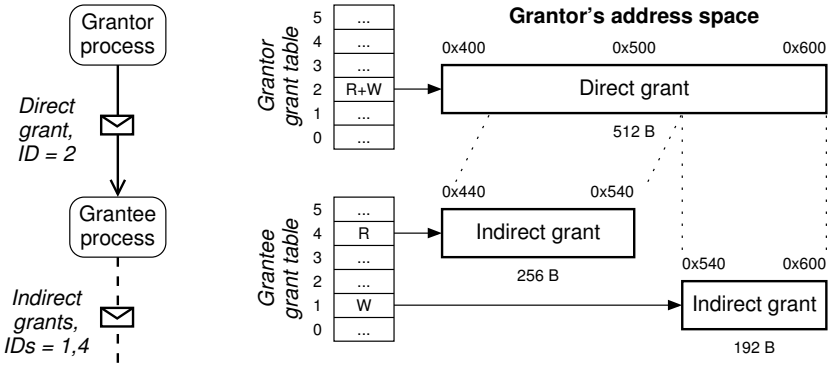


Figure 3.5: Hierarchical structure of memory grants. A direct grant gives access to part of the grantor's memory. An indirect grant gives access to a subpart of the *root* grantor's memory.

request message, the kernel extracts the IPC endpoint and grant 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 and read from the grant tables; no context switching is needed to follow the chain. The copy request is checked against the minimal access rights found in the path to the direct grant. Finally, 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.

Grants can also be used to set up memory mappings. The SAFEMAP kernel call allows a process to map the memory indicated by a memory grant into its own address space. In order to prevent unintended memory disclosure, only grants that are page-aligned and span entire pages can be memory mapped. The granularity of the protection thus depends on the hardware. The kernel also verifies that the grant's access modifiers match the page protection requested. If the request is allowed, the kernel forwards the request to the virtual memory (VM) subsystem, which sets up the mapping and updates its administration for future clean-up. Finally, the kernel sets the memory grant flag MG_MAPPED in the memory grant(s) used in order to indicate that additional work is needed during grant revocation.

The memory grant model supports immediate revocation of all access rights at the grantor's discretion. If the grant is not used in a memory mapping, revocation is simply done by unsetting the flag MG_VALID in the memory grant. If the grant is mapped, an additional kernel call, SAFEREVOKE, is needed in order to undo the memory mapping: the pages involved need to be marked 'copy-on-write' by the VM subsystem. The details of checking the MG_MAPPED flag and making the kernel call are conveniently hidden in a system library. Implicit revocation due to an exiting grantor process is automatically detected by the clean-up routines in the VM

subsystem. In all cases, revocation is permanent and cannot be circumvented by other processes, since the kernel always validates all grants in the chain leading to the root grantor before executing new grant operations.

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 device's DMA controller and have a trusted DMA driver mediate all access attempts. Although this would be a one-time effort for ISA devices that use the motherboard's central DMA controller, the approach is impractical for bus-mastering PCI devices that come with their own DMA controller: each PCI device needs to be checked for DMA capabilities and a specialized helper driver must be written for each different DMA engine. Therefore, we rely on modern hardware where the peripheral bus is equipped with an IOMMU that controls all DMA attempts. As discussed in Sec. 2.3.2, the IOMMU intercepts DMA operations from peripheral devices and validates the memory access using the information stored in the IOMMU tables. In particular, we implemented support for AMD's Device Exclusion Vector (DEV).

Access to the IOMMU is mediated by a trusted IOMMU driver, consisting of under 500 LoC in the case of AMD's DEV. A driver that wants to use DMA first needs to allocate a range of contiguous physical memory using the MAPDMA kernel call. Then it requests the IOMMU driver to program the IOMMU. The protection enforced is based on a very simple rule: only DMA into the driver's own virtual address space is allowed. Before setting up the IOMMU tables the IOMMU driver verifies this requirement through the MEMMAP kernel call, which also returns the physical address. It also verifies that the page protection matches the DMA operation requested and that all memory pages involved are pinned. Because each DMA protection domain is associated with a specific hardware device rather than a software driver, the IOMMU driver must verify that the driver has permission to access the device, which is identified by the combination of peripheral bus, device number, and device function. This check can be done by sending a request to the driver manager or PCI-bus driver. Finally, the IOMMU driver programs the IOMMU in order to allow access. Where possible the driver uses the same DMA buffer during its lifetime in order to increase performance [Willmann et al., 2008].

Because the actual DMA operation is done asynchronously (at the device's discretion), revocation of access rights due to unexpected process exits must be handled with care: if the physical memory associated with a driver that used DMA would be reallocated, a DMA operation done after the driver exit could cause unexpected memory corruption. Therefore, the IOMMU driver reports to the process manager all memory ranges programmed into the IOMMU. If a driver involved in DMA exits, the process manager sends an exit notification to the IOMMU driver in order to clean up. Only once the memory of the exiting process is removed from the IOMMU tables, can it be safely returned to the free list.

3.3.3 Restricting Device I/O

Device access and interrupt handling is restricted using per-driver isolation policies. As discussed in Sec. 2.2, policies are stored in simple text files defined by the administrator. Upon loading a driver the driver manager informs the kernel and trusted OS servers, so that the restrictions can be enforced at run time.

Device Access

The driver manager uses isolation policies in order to ensure that each driver can only access its own device. Upon loading a new driver the driver manager first compares the isolation policy against the policies of running drivers. If the new driver's device is already in use, the launch process is aborted. The policies are suitable for both statically and dynamically configured I/O devices on the ISA bus and PCI bus, respectively. For ISA devices, the keys isa io and isa irq statically configure the device's I/O resources by explicitly listing the I/O ports and IRQ lines in the policy, respectively. ISA plug-and-play (PnP) devices are not supported by MINIX 3. For PCI devices, the keys pci device and pci class grant access to one specific PCI device or a class of PCI devices, respectively. The driver manager reports the device or device class 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 computer's basic input/output system (BIOS). 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.

When a driver requests I/O, the kernel always verifies that the operation is permitted by checking the request against the I/O resources reported through PRIVCTL. For devices with memory-mapped I/O, the driver can request to map device-specific memory persistently into its address space using the MEMMAP kernel call. For devices with programmed I/O, fine-grained access control for device ports and registers is implemented in the VDEVIO kernel call. 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 (IA-32) architectures was not considered a viable alternative, because the 8-KB per-driver bitmaps would impose a higher demand on kernel 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

When a device needs service, it asserts its interrupt line in order to raise an interrupt request (IRQ) and put the kernel in control. Although the lowest-level interrupt handling must be done by the kernel, all device-specific processing is done local to each user-level driver. The kernel implements support for the Intel-8259-compatible *programmable interrupt controller* (PIC). A generic kernel-level interrupt handler

catches all IRQs (except clock IRQs) and forwards them to the associated user-level driver using a nonblocking notification message. The code to do so consists of only a few lines and is the same for all drivers. In contrast, the device-specific interrupt-handling code in the driver is generally much more complex. In this way, bugs triggered during interrupt handling are isolated in the user-level driver process.

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 checks the driver's policy installed by the driver manager or PCI-bus driver. If an interrupt occurs, the generic kernel-level handler disables interrupts, masks the IRQ line that interrupted, asynchronously notifies the registered driver(s), and, finally, reenables the interrupt controller. This process takes about a microsecond and the complexity of reentrant interrupts is avoided. Interrupt notifications use the IPC notification mechanism, which allows the handler to set a bit in the driver's 'pending events' bitmap and then continue without blocking. When the driver is ready to receive the interrupt, the kernel turns it into a normal IPC message of type HWINT. Once the device-specific processing is done, the driver(s) can acknowledge the interrupt using IRQCTL in order to unmask the IRQ line.

3.3.4 Restricting IPC

IPC is restricted through careful design of the IPC subsystem as well as per-driver isolation policies. The IPC subsystem provides a set of reliable communication primitives, introduced in Sec. 2.1.3, as well as mechanisms to restrict their use. In addition, trusted system servers use well-defined IPC protocols to safeguard communication with untrusted drivers.

Low-level Primitives

Dependability of the IPC subsystem is realized because the kernel fully controls what happens during an IPC call. The following IPC properties can be safely assumed: atomicity of IPC calls, reliable message delivery, and isolation between IPC calls. First, atomicity is trivially met since the kernel simply does not return control to the caller until it is done. Second, reliable delivery is achieved because the kernel copies or maps the entire message to the destination process. Message integrity is automatically preserved. Resource exhaustion is structurally prevented since the IPC subsystem uses only statically allocated resources and message buffers local to the caller. Third, isolation is guaranteed because multiple IPC calls are handled independently and snooping on other processes' IPC traffic is not possible. These well-defined semantics allow servers and drivers to set up reliable communication channels and do their work without external interference.

The IPC subsystem also provides mechanisms to control the use of IPC and force IPC patterns onto untrusted processes. First, we restrict the set of IPC primitives (SEND, ASEND, etc.) available to each process. Second, we restrict which services a

process can send to using *send masks*. In principle, send masks can be used to restrict the possible destinations for each individual IPC primitive, but policy definition in the face of multiple, per-primitive IPC send masks proved impractical. Therefore, send masks restrict the allowed IPC destinations regardless of the primitive that is used. Furthermore, send masks are defined as a symmetric relation: if A is allowed to send a request to B, B's send mask is automatically updated such that B is allowed to send the response to A. Receiving is not protected, since it is meaningful only if an authorized process sends a message.

A final protection mechanism is the use of unique IPC endpoints. In order to disambiguate between processes that may (over time) occupy the same slot in the kernel's process table, IPC endpoints contain a 16-bit generation number that is incremented every time a process reuses a process table slot. Slot allocation is done round robin in order to maximize the time before endpoint reuse. This design ensures that IPC directed to an exited process cannot end up at a process that reuses a slot. (Note that we solely focus on buggy drivers and do not protect against malicious drivers attempting to overtake an IPC endpoint using a brute-force attack that quickly cycles through the generation number space.) Moreover, in the event that a system server exits, the data store's publish-subscribe mechanism immediately notifies all dependent processes about the invalidated endpoint.

Interaction Patterns

By default, drivers are not allowed to use IPC, but selective access can be granted on a per-driver basis using isolation policies. The keys ipc calls and ipc targets determine the permissible IPC primitives and IPC destinations, respectively. Upon loading a driver the driver manager informs the kernel about the IPC privileges granted using PRIVCTL, just as is done for I/O resources. The kernel stores the driver's IPC privileges in the process table and the IPC subsystem enforces them at run time using simple bitmap operations. In this way, driver communication can be restricted to only those system processes drivers need to talk to.

Protection against caller blockage due to deadlocks and asymmetric trust relationships can be implemented in various ways, each of which comes with a certain complexity. First, time-outs help to detect failing IPC calls, but are hard to get correct for programmers—arbitrary or overly conservative time-out values are not uncommon—and may lead to periods of blockage. Second, multithreading allows spawning a separate thread for handling untrusted IPC interactions, but requires a more complex thread-aware IPC subsystem. Third, asynchronous and nonblocking IPC prevents blocking on untrusted IPC targets, but comes with a state-machine-driven programming model. The last option seemed most suitable in the context of MINIX 3, for two reasons. First, it required relatively little programming effort because it affected only two trusted system servers: the virtual file system (VFS) and the network server (INET). Second, a state-machine-based approach also facilitates recovery after a driver crash, because pending requests can be replayed from

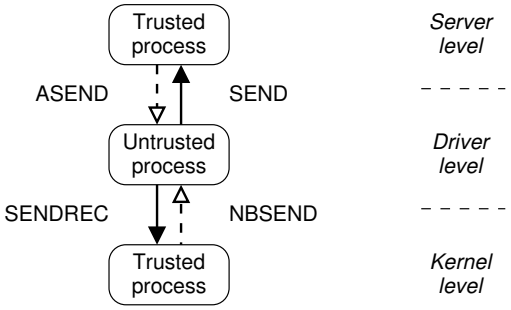


Figure 3.6: IPC patterns to deal with asymmetric trust. Trusted processes use asynchronous or nonblocking IPC (dashed lines) for sending to untrusted processes, which use synchronous IPC (solid lines) to prevent race conditions that may cause messages to be dropped.

the work queue. Therefore, the IPC subsystem provides both synchronous, asynchronous, and nonblocking IPC primitives.

In MINIX 3, asynchronous and nonblocking IPC is used only at a few well-defined decoupling points where (trusted) system servers have to communicate with (untrusted) device drivers. The IPC patterns that we use in these cases are summarized in Fig. 3.6. First, in order to deal with untrusted clients (also see Fig. 3.2(b)), trusted servers use the nonblocking NBSEND to send driver replies. Second, in order to deal with drivers acting as an untrusted server (also see Fig. 3.2(c,d)), trusted clients use the asynchronous ASEND to send driver requests and do not block waiting for the reply. These two simple rules ensure that the core system servers can never be blocked by a driver that does not do the corresponding RECEIVE. The design consequently forces drivers to use synchronous IPC in order to prevent race conditions that may cause messages to be dropped. In particular, requests sent using an asynchronous ASEND must be matched by a synchronous SEND, because the driver cannot know if the caller is ready to receive the reply. In a similar vein, drivers must use a synchronous SENDREC to request kernel services, because the kernel runs at a higher priority and simply drops the reply if the driver is not ready to receive it. While this design isolates trusted processes from faults in untrusted drivers, additional mechanisms, such as the driver heartbeat requests described in Chap. 4, are still required in order to detect failures and ensure progress.

System Services

Because the kernel is concerned with only 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 system task in the kernel, which provides a plethora of sensitive operations, such as creating processes, setting up memory maps, and configuring driver privileges. Therefore, the key ipc kernel in the per-driver isolation policies is used

to restrict access to individual kernel calls. In line with least authority, each driver is granted access to only those services needed to do its job, such as safe memory operations and device I/O. Again, the driver manager fetches the calls granted upon loading a driver and reports them to the kernel using PRIVCTL. The kernel task inspects the table with authorized calls each time a driver requests service. It should be noted that multiple levels of defense are used for certain kernel calls. Even if a driver is authorized to use, say, SAFECOPY or VDEVIO, the protection mechanisms described in Secs. 3.3.2 and 3.3.3 are enforced. For example, memory copies require a valid memory grant and device I/O is allowed only for the driver's device, respectively. This ensures the correct granularity of isolation.

Finally, the use of services from the user-level 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 are denied because all drivers run with an unprivileged user ID. Furthermore, since the OS servers perform sanity checks on all input, requests may also be rejected due to invalid or unexpected parameters. This is similar to the sanity checks done for ordinary POSIX system calls from the application layer.

3.4 Case Study: Living in Isolation

As a case study, we now discuss the working of an isolated Realtek RTL8139 network driver. The driver's life cycle starts when the administrator requests the driver to be loaded using the isolation policy shown in Fig. 3.7. The driver is granted access to a single PCI device, defined by the combination of the vendor ID (10ec) and the device ID (8139). The policy enables IPC to the kernel, process manager, data store, driver manager, PCI-bus driver, IOMMU driver, and network server, respectively. The kernel calls granted allow the driver to perform device I/O, manage interrupt lines, request DMA services, make safe memory copies, output diagnostics, set timers, and retrieve system information, respectively.

```
1  driver rtl8139                  # ISOLATION POLICY
2  {
3          pci device              10ec/8139
4                                  ;
5          ipc targets             KERNEL PM DS RS PCI IOMMU INET
6                                  ;
7          ipc kernel              VDEVIO IRQCTL MAPDMA SETGRANT SAFECOPY
8                                  SYSCTL TIMES SETALARM GETINFO
9                                  ;
10 };
```

Figure 3.7: Per-driver policy definition is done using simple text files. This is the *complete* isolation policy for the RTL8139 driver as found in */etc/drivers.conf*.

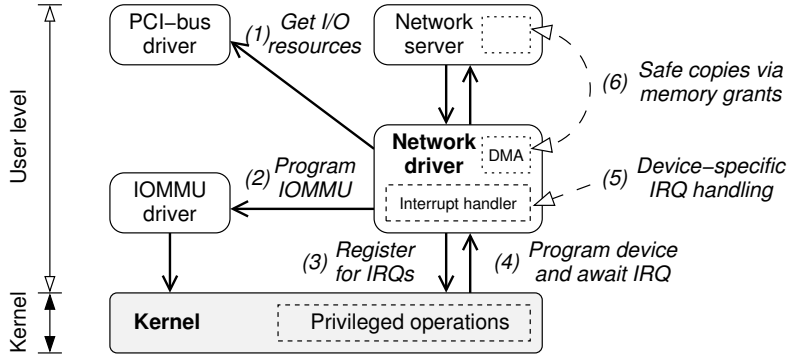


Figure 3.8: Interactions between an isolated network-device driver and the rest of the OS. Access to privileged resources is mediated by the PCI-bus driver, IOMMU driver, and the kernel.

After verifying that the caller is authorized to start new drivers, the driver manager stores the policy and creates a new process. The new process is uniquely identified by its IPC endpoint. The driver manager queries the data store for the IPC endpoints of the allowed IPC targets and maps the allowed kernel calls onto their call numbers in order to create the actual permission bitmaps. Then it informs the kernel about the IPC targets and kernel calls allowed using the PRIVCTL call. The PCI device ID is sent to the PCI-bus driver, which retrieves the I/O resources belonging to the RTL8139 device from the device's PCI configuration space and, in turn, informs the kernel. Finally, only after the execution environment has been properly isolated, the driver manager executes the driver binary.

The working of the RTL8139 driver in its isolated execution environment is sketched in Fig. 3.8. When the driver gets to run it first executes its initialization routines. In step (1), the RTL8139 driver contacts the PCI-bus driver to retrieve the I/O resources associated with the RTL8139 PCI device. Since the RTL8139 device uses bus-mastering DMA, (2) the driver allocates a local buffer for use with DMA and requests the IOMMU driver to program the IOMMU accordingly. 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. Finally, (3) the RTL8139 driver registers for interrupt notifications using the IRQCTL kernel call. Only IRQ lines reported by the PCI-bus driver are made accessible though.

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 allows the driver to access only the message buffers and nothing else. We now consider a request to read from the network. In step (4), the RTL8139 driver programs the network card using the VDEVIO kernel call. The completion interrupt of the DMA transfer is caught by the kernel's generic handler and (5) forwarded to the RTL8139 driver where the device-specific interrupt handling is done. The interrupt is handled at the user level and acknowledged using the IRQCTL kernel

call. In step (6), the driver makes a SAFECOPY kernel call to transfer the data read to INET. Although we did not implement this, a zero-copy protocol can be realized for performance-critical applications by mapping the memory granted by INET into the driver's address space using the SAFEMAP kernel call. Writing garbage into INET's buffers results in messages with an invalid checksum, which will simply be discarded. In order to prevent race conditions when both INET and the driver try to access the packet data, INET revokes the memory access rights once the driver signals that a packet has arrived. In this way, the driver can safely perform its task without being able to disrupt any other services.

If the driver process crashes or otherwise exits unexpectedly, all the privileges that were granted to the driver are revoked by the OS. The process manager will be the first to notice the driver exit, and notifies all processes involved in the exit procedure, such as the kernel's system task and IOMMU driver. The driver's CPU and memory privileges are automatically revoked because the driver process is no longer scheduled by the kernel. Likewise, the driver's I/O and IPC privileges that were stored at the kernel are reset when the driver's process-table entry is cleaned up. Memory grants that INET or other processes created for the driver are no longer usable because the grants contain the driver's unique IPC endpoint, which is invalidated by the kernel. Memory grants created by the driver itself are automatically cleaned up when the driver's address space is recollected by the OS. Before this can be done, however, the IOMMU driver resets the DMA protection domain that was set up for the driver's device.

Finally, when all privileges have been revoked, the driver manager is notified that one of its children has exited, so that it can perform its local clean-up. The driver manager first updates its local administration and—depending on the policy provided by the administrator—then may attempt to recover the failed driver. Such driver recovery is the subject of the next chapter.

Chapter 4

Failure Resilience

With driver faults properly isolated, we now focus on another technique that is used by MINIX 3 to improve dependability. In particular, we do not claim that our drivers are free of bugs, but we have designed our system such that it can recover from driver failures. After loading a driver, it is constantly guarded in order to ensure continuous operation. If the driver unexpectedly crashes, exits, or misbehaves otherwise, it is automatically restarted. How the driver manager can detect defects and how the recovery procedure works is described in detail below.

The remainder of this chapter is organized as follows. First, Sec. 4.1 explains the defect detection techniques used by the driver manager. Next, Sec. 4.2 discusses the role of recovery scripts and shows how components can be restarted. Then, Sec. 4.3 discusses the effectiveness of server and driver recovery. Finally, Secs. 4.4 and 4.5 present two case studies that further illustrate the working of our design.

4.1 Defect Detection Techniques

While a human user observes driver defects when the system crashes, becomes unresponsive, or behaves in strange ways, the OS needs other ways to detect failures. Therefore, the driver manager monitors the system at run time to find defects. Note that the driver manager can only observe component failures, that is, deviations from the specified service, such as a driver crash or failure to respond to a request. We do not attempt to detect erroneous system states or the exact underlying faults that led to the failure. However, as discussed in Sec. 2.5, this is not a problem in practice since many problems are intermittent and tend to go away after restarting a failing or failed component.

The defect detection techniques used by the driver manager are based on unexpected process exits, explicit update requests, and periodic monitoring of extensions. A classification of the various conditions that can cause the recovery procedure to be initiated is given in Fig. 4.1 and discussed below.

Technique	Defect trigger	In use	Example scenario
Process exit	CPU or MMU exception	Yes	Driver dereferences invalid pointer
	Killed by signal	Yes	User kills driver that misbehaves
	Internal panic	Yes	Driver detects internal inconsistency
Periodic check	Request heartbeat	Yes	Driver winds up in infinite loop
	Correctness proof	No	See examples in Sec. 4.1.2
Explicit request	Dynamic update	Yes	User starts new or patched driver
	Component complaint	Yes	Server detects protocol violation

Figure 4.1: Classification of defect detection techniques and their implementation status in MINIX 3.

4.1.1 Unexpected Process Exits

The most important technique that initiates the recovery procedure is immediate detection of unexpected process exits. As explained in Sec. 2.2, the driver manager starts new drivers by forking a process, setting the child process' privileges, and executing the driver binary. This means that the driver manager is the parent of all system processes, and according to the POSIX specification, it will immediately receive a SIGCHLD signal if a driver crashes, panics or exits for another reason. The crashed process becomes a zombie process until the driver manager collects the pieces using a wait call, which returns the exit status of the exitee and allows the driver manager to figure out what happened.

This mechanism ensures, for example, that a driver killed by the process manager because it dereferenced a bad pointer and caused an MMU exception is replaced instantaneously. Likewise, CPU exceptions such as a division by zero may also cause the driver to be signaled. Since all drivers run as ordinary user processes, they also can be killed by user signals. This allows the administrator to restart malfunctioning components, although the preferred method is sending an update request to the driver manager. Finally, if a system process detects an internal inconsistency, it can simply log the error and exit in order to be automatically replaced with a fresh copy. Our experiments indicate that both exceptions and internal panics are responsible for a large fraction of all restarts.

4.1.2 Periodic Status Monitoring

Next, the driver manager also proactively checks the system's state in order to detect malfunctioning system services. We have implemented periodic heartbeat requests that require an extension to respond within the next period. Failing to respond N consecutive times causes recovery to be initiated. Heartbeats do not protect against malicious code, but help to detect processes that are 'stuck,' for example, because they are deadlocked in a blocking IPC call or wound up in an infinite loop. On a monolithic system this is effectively a denial of service attack, but since all MINIX 3 drivers run as independent processes, the scheduler notices this behavior and grad-

ually lowers the offending process' priority, so that other processes can continue to run normally. Because the MINIX 3 drivers are single-threaded processes with an event loop, we used a conservative heartbeat period of 1–5 second in order to prevent false negatives during heavy workloads. Nevertheless, this setup proved effective in catching unresponsive drivers in our fault-injection experiments.

Although we did not implement it, more elaborate run-time monitoring is also supported by our design. First, while we currently use fixed heartbeat periods, it may be possible to maintain a failure history and dynamically adapt the heartbeat period to the driver's workload. Furthermore, the driver manager could request some kind of proof that the driver still functions correctly. For example, the driver manager could verify that the driver's code segment is unaltered in order to protect against driver exploits [Xu et al., 2004] and bit flips caused by radiation, electromagnetic interference, or electrical noise [Heijmen, 2002]. As another example, higher-level heartbeats could potentially verify end-to-end correctness by performing an I/O request and comparing the result with the expected response [Hunt, pers. comm., 2010]. Finally, it may be possible to restart active OS modules periodically in order to rejuvenate the OS and proactively recover aging bugs, such as memory leaks, before they can cause failures [Huang et al., 1995; Ishikawa et al., 2005].

4.1.3 Explicit Update Requests

Finally, another class of defect detection techniques are explicit update requests. Sometimes faulty behavior can be noticed by the user, for example, if the audio sounds weird or if the network performs badly. In such a case, the system administrator can explicitly instruct the driver manager to restart a driver. Our design also supports replacing the binary of the extension with a new one, so that patches for latent bugs or vulnerabilities can be applied as soon as they are available. In this scenario we speak of a *dynamic update* [Baumann et al., 2007]. Since reboots due to maintenance are responsible for a large fraction (24%) of system downtime [Xu et al., 1999], dynamic updates that allow run-time patching of system components can significantly improve system availability.

The driver manager can also be used as an arbiter in case of 'conflicts,' allowing authorized components to file a complaint about malfunctioning components. For example, a server could request a driver that sends unexpected request or reply messages to be restarted. This is useful because the driver manager is not aware of server-to-driver protocols and cannot inspect messages exchanged. If a complaint is filed, the driver manager kills the bad component and starts a fresh copy. In order to prevent abuse, the isolation-policy field driver informs the driver manager whether a component is allowed to use this functionality and, if so, for which parties. As described in Sec. 4.4, we have experimented with explicit update requests in the storage stack. However, most of the MINIX 3 servers simply return a normal POSIX error code to signal errors. The error is logged, however, so that the administrator can check the system's state.

4.2 On-the-fly Repair

If a defect is detected, the driver manager starts its recovery procedure to repair the system on the fly. The basic idea is to perform a *microreboot* of the failed component [Candea et al., 2004]. Below, we first discuss the use of recovery scripts and then focus on the actual driver restart. While the primary focus is recovering stateless drivers, we also discuss the MINIX 3 mechanisms for state management.

4.2.1 Recovery Scripts

By default, the driver manager directly restarts malfunctioning components, but if more flexibility is wanted, the administrator can set up a *recovery script* that governs the steps to be taken after a failure. This is done using the service utility parameter –recovery, which accepts the path to shell script and an argument list that is passed to the script upon execution. In principle, each server and driver can be configured with its own recovery script, but scripts can also be written in a generic way and shared among extensions. In the event of a failure, the driver manager looks up the associated script and executes it with the following arguments: (1) the component that failed, (2) the event causing the failure, (3) the current failure count for the failed component, and (4) the argument list passed by the administrator. Recovery scripts must be careful not to depend on functionality that is (temporarily) not available, or request a restart of the failed component before attempting to use it.

Using shell scripts provides great flexibility and power for expressing policies. Even if a generic recovery script is used, the precise steps taken may differ per invocation, depending on the information passed by the driver manager. As an example, consider the generic recovery script in Fig. 4.2. Line 1 gives the path to shell executable and lines 2–5 process the driver manager arguments. Then, lines 7–12 implement a binary exponential backoff strategy in restarting repeatedly failing components. The actual restart command is done after an increasingly large delay in order to prevent bogging down the system in the event of repeated failures. The backoff protocol is not used for dynamic updates that are requested explicitly. Finally, lines 14–24 send a failure alert to a remote administrator if the parameter –a and an email address are passed.

The use of policy-driven recovery provides several benefits. Even though full recovery is not always possible, recovery scripts can assist the administrator in handling from failures. For example, crashes of the network server, INET, not only requires restarting INET, but also affect applications that depend on its functionality, such as the DHCP client and X Window System. A dedicated recovery script that was specifically designed to recover from such failures is discussed in Sec. 4.5. As another example, if a critical component cannot be recovered or fails too often, the recovery script may reboot the entire system, which clearly is better than leaving the system in an unusable state. At the very least, the recovery script can log details about the failing component and its execution environment.

```
1  #!/bin/sh                          # GENERIC RECOVERY SCRIPT
2  component=$1                       # failed component binary
3  reason=$2                          # failure reason
4  repetition=$3                      # current failure count
5  shift 3                            # get to script parameters
6
7  # RESTART BINARY EXPONENTIAL BACKOFF
8  if [ ! $reason -eq UPDATE ]; then
9        sleep $((1 << ($repetition - 1)))
10 fi
11 service refresh $component         # request restart
12 status=$?                          # get restart status
13
14 # E-MAIL OPTIONAL FAILURE ALERT
15 while getopts a: option; do
16 case $option in                    # check optional parameters
17 a)
18       cat << END | mail -s "Failure Alert" "$OPTARG"
19            failure details: $component, $reason, $repetition
20            restart status: $status
21 END
22 ;;
23 esac
24 done
```

Figure 4.2: Example of a parameterized, generic recovery script. Binary exponential backoff is used before restarting, except for dynamic updates. If the optional parameter –a is present, a failure alert is emailed to the given address.

4.2.2 Restarting Failed Components

The actual procedure for restarting a failed driver and reintegrating it into the system consists of three phases. First, when the recovery script requests the driver manager to restart the component, its privileges and the privileges of dependent components need to be (re)set. Second, changes in the OS configuration need to be communicated to dependent components in order to initiate further recovery and mask the problem to higher levels. Third, the restarted component may need to do local recovery. This procedure is illustrated in Fig. 4.3 and discussed below.

Restarting Failed Components

The first two steps in Fig. 4.3 correspond to restarting a failed driver. If no recovery script is used, the driver manager automatically attempts to restart the driver, otherwise it will wait for the recovery script's signal. In step (1), the driver manager performs the necessary clean-up, creates a new process, installs its isolation policy, and executes the binary. Changes in the system configuration are disseminated through the data store. In step (2), the driver manager publishes the stable name and

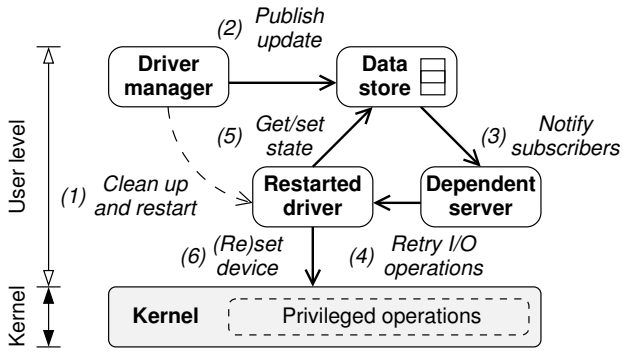


Figure 4.3: Procedure for restarting a failed device driver and reintegrating it into the system. The sequence of steps taken is shown in (roughly) clockwise order.

IPC endpoint of the restarted component in the data store. In this respect, a restart is mostly similar to the steps taken when a component is started through the service utility, as discussed in Sec. 2.2. There are minor differences between starting a new component and restarting a failed component, however.

The driver manager relies on the fact that our design uses temporally unique IPC endpoints in order to identify uniquely a component and grant it the privileges listed in the isolation policy. However, because the IPC endpoint is automatically reset during a restart, requests by the restarted component will be denied. Therefore, dependent components need to be informed about the update, so that they can discard old information and store the component's new IPC endpoint and privileges. All these updates have to be completed before the new component gets to run.

Informing Dependent Components

Next, the dependent components are informed and can start their local recovery procedure. The data store implements a publish-subscribe mechanism, so that subscribed components automatically receive updates about changes in the system configuration. If a driver fails, (3) dependent servers subscribed to the driver's stable name are automatically notified and, if need be, (4) can attempt local recovery, such as retrying failed or pending I/O requests. This design decouples producers and consumers and prevents intricate interaction patterns of components that need to inform each other when the system configuration changes.

In general, we were able to implement recovery at the server level. Application-level recovery is needed only for specific failures that cause data loss or destroy too much internal state. In this case, the normal POSIX error handling mechanisms are used: the application's system call is aborted and an error code is returned. For historical reasons most applications assume that an I/O failure is fatal and give up, but by changing user-space applications we may improve dependability even further. Sec. 4.3 illustrates this point with concrete recovery schemes.

Local Recovery

Finally, the restarted component may need to perform local reinitialization and recovery, such as (5) retrieving internal state that it lost when it crashed and (6) reinitializing its device. The exact steps taken depend on the component. As discussed below, stateful services can retrieve lost state from the data store so that they can properly reinitialize when they are brought up again. Device reinitialization typically is done using the normal device initialization procedures.

4.2.3 State Management

Although this thesis does not focus on stateful recovery, the data store provides a number of mechanisms that support components in storing and retrieving internal state. Below we introduce the working of the data store and highlight several problems relating to state management.

Working of the Data Store

The data store allows components to backup state and restore it after a restart. Fig. 4.4 briefly summarizes (part of) the data store API. First, it is possible to store primitive data types and arrays thereof, such as integer values (DSF_TYPE_U32) or character strings (DSF_TYPE_STR), under a component-specified identifier, known as a *handle*. This mechanism creates a copy of the data to be stored. Retrieval is done by presenting the handle to the data store. A special data type exists for naming information (DSF_TYPE_LABEL), where the handle is a component label and the data

Function	Explanation
ds_publish	Store a piece of typed data (see types below) under the given handle
ds_retrieve	Retrieve a piece of typed data with the given handle or snapshot index
ds_delete	Delete a piece of typed data or a snapshot from the data store
ds_snapshot	Make a copy of a memory-mapped region and get the snapshot index
ds_subscribe	Subscribe to data store entries matching the given regular expression
ds_check	Check which data store entry changed and get its type and handle
Flag	**Explanation**
DSF_PRIVATE	Flag used to store data privately; used when publishing data
DSF_OVERWRITE	Flag used to overwrite the data if an entry with same handle exists
DSF_TYPE_U32	Data type for storing unsigned 32-bit integers (fits in IPC message)
DSF_TYPE_STR	Data type for storing strings up to 16 characters (fits in IPC message)
DSF_TYPE_MEM	Data type for copying grant-based memory regions using SAFECOPY
DSF_TYPE_MAP	Data type for mapping grant-based memory regions using SAFEMAP
DSF_TYPE_LABEL	Data type for publishing labels (identifiers) of system components

Figure 4.4: Summary of the data store API for state management as provided by the system libraries. System-library functions are shown at the top. Flags are shown at the bottom.

stored gives the corresponding IPC endpoint. Only the driver manager is allowed to store naming information. These primitive data types fit in an IPC message and do not require additional copying. Second, it is possible to store entire memory regions specified by memory grants. As discussed in Sec. 3.3.2, memory grants can be used for both copying the data or setting up a memory mapping; a flag tells the data store which mode should be used. With copying (DSF_TYPE_MEM), the data store copies the data once upon request and allows retrieving the same data later on. With mapping (DSF_TYPE_MAP), the data store gets a real-time view of the memory region to be stored, so that the latest (but possibly corrupted) state can always be recovered. The data store's snapshot functionality can be used to checkpoint memory-mapped regions from time to time.

A flag (DSF_PRIVATE) tells the data store to store the data either publicly or privately. The data store's publish-subscribe mechanism allows process to subscribe to public data, such as naming information. In contrast, private data can be retrieved only by the component that stored it. This is enforced by authenticating components with help of the naming information that is also kept in the data store: when data is stored, a reference to the stable name is included in the record, so that the owner can be authenticated through a reverse lookup of the IPC endpoint. This allows servers and drivers to store data privately using simple handles consisting of a logical name rather than a large cryptographic hash. Another benefit of this design is that authentication works between restarts. Although the component's IPC endpoint changes during the restart, the stable name remains the same and the driver manager updates the IPC endpoint associated with the component's stable name as part of the recovery procedure.

These basic mechanisms enable components to backup privately their state and retrieve it after a restart. With memory mapping, the data store gets a real-time view of a memory region in a given process. The data store also can be requested to make a snapshot of the mapped memory region, such that different versions can be maintained. In both cases, if the process crashes, the data store still holds a copy of the data or a reference to the memory region, and allows the restarted process to recover its state by presenting the handle for it. Data is currently not persisted across reboots, but the data store could potentially be extended to do so.

Gaps in State Management

Although these basic mechanisms provide an elegant way to store and retrieve internal state, there are several problems relating to state management that we did not look into. The most important ones are:

- State integrity.
- Transaction support.
- Performance trade-offs.

To start with, while the data store is able to maintain a process' internal state, it cannot prevent a buggy process from corrupting its own state. If a process is allowed to communicate with the data store, a bug may cause it to store accidentally bogus information. A potential solution would be to checksum internal state, so that the restarted process can at least find out about the corruption. Next, it is sometimes impossible to tell which part of an operation was already completed and which part was not. This can be addressed by implementing some form of transaction support and commit only completed transactions to the data store. Finally, there is a trade-off between performance and dependability, because continuous checksumming and checkpointing may be prohibitive for state that changes all the time and is on the performance-critical path. Finding a good balance is a hard problem and may have to be done on a per-component basis.

There are several other, less fundamental issues that may have to be addressed as well. First, quota enforcement may be needed in order to prevent one component from using all the data store's memory. Second, more elaborate access-control mechanisms could support access on a per-process basis rather than storing data either public or private. Third, there should be a policy for garbage collecting data that is no longer needed when a driver exits normally. This kind of functionality should be provided when the data store is more heavily used.

Because of these issues, the data store is primarily used to support dynamic updates where the component writes its state to the data store and exits in a controlled way. For example, an audio driver could use the data store to backup control operations and reinitialize the mixer settings of the sound card after a restart. As another example, a RAM disk driver can store the base address and size of the RAM disk's memory so that it can be restarted on the fly. Upon starting, the drivers can query the data store for previously stored state to determine which initialization method should be followed. If the data store does not know about the requested handle, default initialization is done; if the handle is found, the previous state is restored. Because we did not use checksumming and transactions, we cannot (currently) give hard guarantees about state corruption during an unexpected crash. Instead, we assume *fail-stop* behavior where erroneous state transformations due to a failure do not occur [Schlichting and Schneider, 1983].

In our prototype, state management turns out to be a minor problem for the MINIX 3 device drivers, which are mostly stateless. The effectiveness of driver recovery is presented in detail in Sec. 4.3.1. Recovery of stateful components, such as the file server and network server, is partially supported, as discussed in Sec. 4.3.2.

4.3 Effectiveness of Recovery

In this section, we present MINIX 3's recovery procedure for low-level device drivers and system servers and discuss its effectiveness. Effectiveness can be measured along two axes. First, we say that recovery is *transparent* if it is done without re-

turning an error to application programs and without user intervention. This requires recovery to be done within the realms of the OS. Second, we say that recovery is *lossless* or *full* if no user data is lost or corrupted. This requires that we guarantee 'exactly once' behavior for the outcome of requests. In many cases, our design enables full transparent recovery. However, even if this goal cannot be achieved, the system's failure-resilience mechanisms still help to improve availability by speeding up recovery, as discussed below.

As an aside, the recovery schemes discussed here pertain not only to failures, but as discussed in Sec. 4.1, our design also allows the administrator to update servers and drivers dynamically—even if I/O is in progress. In this case, the driver manager first requests the extension to exit cleanly by sending a SIGTERM signal, giving it a chance to backup its state in the data store. If the extension does not comply within the time-out interval, it will be killed using a SIGKILL signal. The steps that are taken after the extension exits are similar to those for a failure. Most other OSes currently cannot dynamically replace active OS services on the fly as MINIX 3 does.

4.3.1 Recovering Device Drivers

We now focus on low-level device drivers that directly interact with the hardware. For our purposes we distinguish three device-driver classes, namely, the network-device, block-device, and character-device drivers. Each device class has different I/O properties and, therefore, different driver recovery characteristics. In addition, as will become clear from Fig. 4.5, the protocols used by the layers above the driver level also affect the effectiveness of recovery.

The effectiveness of recovery depends on whether I/O requests are *idempotent* and the data stream provides *data integrity*. A request is idempotent if reissuing it does not affect the final outcome. For example, writing a data block to a given disk address is idempotent, but replaying an audio frame is not. In addition, a means to verify data integrity is needed in order to detect data corruption. Typically this is realized by checksumming the data. Because device drivers are not concerned with these properties—they simply accept a request and program the hardware accordingly—the protocols implemented by the higher-level servers and applications determine the effectiveness of recovery [Saltzer et al., 1984].

The different I/O properties for each driver type leads to different recovery paths, as illustrated in Fig. 4.6. As a general rule, recovery is always attempted at the low-

Device Class	**I/O properties**		**Recovery support**	
	Idempotent	*Data integrity*	*Transparent*	*Data loss*
Network	No	Protocol-dependent	Yes	Protocol-dependent
Block	Yes	FS-dependent	Yes	FS-dependent
Character	No	No	Optional	Likely

Figure 4.5: I/O properties of different driver stacks and the extent to which recovery is supported.

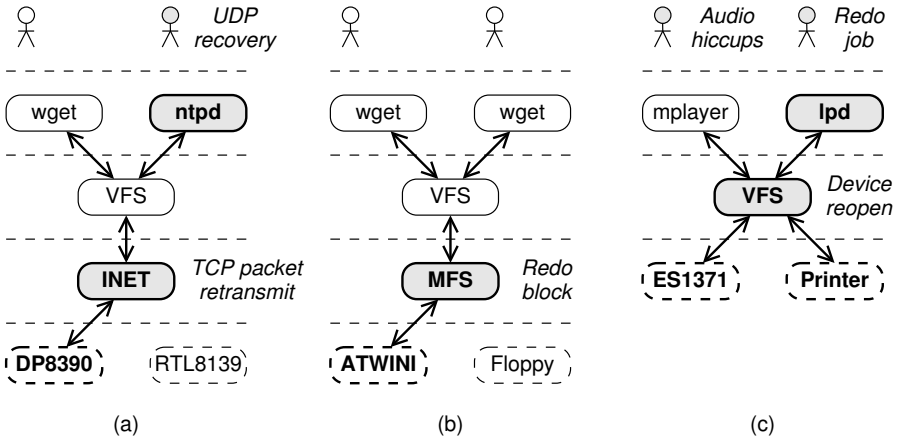


Figure 4.6: Components that have to be aware of driver recovery for different kinds of drivers: (a) network-device driver recovery, (b) block-device driver recovery, and (c) character-device driver recovery. A gray color indicates that a component or the end user is aware of the recovery.

est possible layer so that the rest of the system does not need to be aware of the failure. However, an integrated approach whereby drivers, servers, and applications are involved may be required for optimal results. Ideally, the server level simply reissues failed driver requests and responds normally to the application level, as if nothing special happened. Applications may be blocked during the recovery procedure, but this is not different from a normal POSIX system call that blocks the application until the call has finished. However, if a driver failure cannot be handled at the server level, it has to be reported to the application that issued the I/O request. The application, in turn, may attempt to recover from the I/O failure, but in some cases application-level recovery is not possible, and the end user has to be notified of the problem. The precise recovery procedure for each device class is discussed in more detail in the following subsections.

Recovering Network-device Drivers

If a network-device driver, such as an Ethernet driver, fails full recovery transparent to the application and end user is supported. We have implemented Ethernet driver recovery in MINIX 3's network server, INET. If the application uses a reliable transport protocol, such as TCP, the protocol handles part of the recovery by preventing data corruption and data loss through checksums, sequence numbering, and retransmission timers. From the network server's perspective, an unreliable driver is no different than an unreliable network. Missing or garbled network packets will be detected by the network server (or its peer at the other end of the connection) and can be retransmitted safely. Although network I/O is not idempotent, the higher-level protocol ensures that duplicate packets are filtered out automatically. If an unreliable

transport protocol, such as UDP, is used, loss of data may lead to a degraded quality of service, but is tolerated by the protocol. However, application-level recovery is possible, if need be.

The recovery procedure starts when the process manager informs the driver manager about the exit of one of its children, as discussed in Sec. 4.1. The driver manager looks up the details about the failed driver in its internal tables and runs the associated recovery script to restart it. Because the network server, INET, subscribes to updates about Ethernet drivers by registering the expression 'eth.∗', it is automatically notified by the data store if the driver manager publishes the stable name, say, eth1, and IPC endpoint of the restarted network driver. If INET tries to send data in the short period between the driver crash and the consequent restart, the request fails—because the IPC endpoint is invalidated—and is postponed until the driver is back. Upon notification by the data store, the network server scans its internal tables to find out if the driver update concerns a new driver or a recovered one. In the latter case, INET starts its internal recovery procedure, which closely mimics the steps taken when a driver is first started. The Ethernet driver is reinitialized and pending I/O operations are resumed. The MINIX 3 Ethernet drivers are stateless and do not need to retrieve lost state from the data store.

Recovering Block-device Drivers

If a block-device driver, such as the RAM-disk, hard-disk, CD-ROM, or floppy driver, crashes, full recovery transparent to the application and end user is often possible. We have implemented block-device driver recovery support in the MINIX 3 file server, MFS. Failed I/O operations can be retried safely after the driver has been restarted because block I/O is idempotent. However, since MFS has currently no means to ensure data integrity, there is no way to detect specific failure conditions in which the driver inadvertently acknowledges a failed I/O request or silently corrupts the data. In order to address this situation, we have implemented a filter driver that transparently operates between the file server and block-device driver and provides end-to-end integrity in the storage stack. Without the filter driver incidental data loss can occur; with the filter driver we can provide hard guarantees for both single-driver and single-disk failures. Sec. 4.4 further details the filter driver.

Because a disk-driver failure makes the file system unavailable, the driver manager directly restarts failed disk drivers from a copy in RAM using a modified exec call. In order to do so, disk drivers are started with the service utility flag –mirror set to true, so that the driver manager makes a backup copy of the driver binary in its memory. This approach ensures that driver failures can be handled without relying on file-system access or disk access. We did not provide a similar facility for recovery scripts though. If policy-driven recovery is needed, the system can be configured with a RAM disk to provide trusted storage for crucial data such as the driver binaries, the shell, and recovery scripts. After restarting the driver, the driver manager publishes the new IPC endpoint in the data store, which causes the file server to be

notified. If I/O was in progress at the time of the failure, the IPC rendezvous will be aborted by the kernel, and the file server marks the request as pending. Once the driver is back, the file server updates its device-to-driver mappings, and reinitializes the disk driver by reopening minor devices, if need be. Finally, the file server checks whether there are pending I/O requests, and if so, reissues the failed operations. Like the other drivers in our system, disk drivers are stateless and do not need to retrieve lost state from the data store.

Recovering Character-device Drivers

Recovery of character-device drivers is usually not transparent, since character-device I/O is not idempotent and data may be lost if the I/O stream is interrupted. It is impossible to determine which part of the data stream was successfully processed and which data is missing. If an input stream is interrupted due to a driver crash, input might be lost because it can be read from the controller only once. Likewise, if an output stream is interrupted, there is no way to tell how much data has already been written to the controller, and full recovery may not be possible. Recovery of character-device drivers thus poses an undecidable problem. Therefore, the virtual file system (VFS) generally invalidates all open file descriptors and reports the failure to the application on the next system call. In some cases, application-level recovery is possible, and applications can request VFS to reopen the device associated with the failed driver transparently on their behalf. Nevertheless, the end user may still notice the failure if the I/O stream interruption causes data to be lost.

For historical reasons, most applications assume that a driver failure is fatal and immediately give up, but our design supports continuous operation if applications are made recovery-aware. For this purpose, VFS defines a new error code, ERESTARTED, for signaling an application that the driver temporarily failed but has been successfully recovered. The device's file descriptor is invalidated though. Applications that understand this error code may reopen the device and retry the I/O operation. For example, the MINIX 3 line-printer spooler daemon, *lpd*, works such that it automatically reissues failed print requests without bothering the user. While truly transparent recovery is not possible—partial or duplicate printouts may occur—the user clearly benefits from this approach. Only if the application layer cannot handle the failure, the user needs to be informed. For example, continuing a CD or DVD burn process if the SCSI driver fails will corrupt the disk, so the error must be reported to the user.

In some cases, our design can do even better by handling the character-device driver recovery internal to VFS, transparent to the application. This is supported by introducing a new file descriptor flag, O_REOPEN, which can be set upon opening a device with the POSIX open call. In case of a failure, the flag tells VFS to reassociate transparently open file descriptors with the restarted driver. If I/O was in progress at the time of the failure, the error status EREOPENED is returned to the application in order to signal that data may have been lost, but the application can immediately retry

without the need for reopening the device. If there was no I/O request outstanding, the recovery is fully transparent to the application. We applied this technique to, for example, the *mplayer* media player so that it can continue playing in case of driver failures at the risk of small hiccups.

4.3.2 Recovering System Servers

Our system can not only cope with driver failures, but also recover from failures in certain OS servers. Crashes of the core OS servers, such as the process manager, virtual file system and driver manager, cannot be recovered, since they form an integral part of our failure-resilient design. However, many other servers that extend the base system with optional functionality can potentially be recovered if disaster strikes. The extent to which recovery is supported mainly depends on the amount of state that is lost during a crash. For stateless servers full recovery is generally possible by making minor changes to the dependent components—just like we did for device drivers. However, as described in Sec. 4.2.3, stateful servers are much harder to recover. We illustrate this point with a case study in Sec. 4.5.

As a simple example of stateless server recovery, consider how having a separate *information server* to handle all debug dumps is beneficial to the system's dependability. The information server is responsible for handling all user interaction and the formatting of the debug output. When the user presses a function key, the information server requests a copy of the associated data structures and displays a formatted dump on the console. For example, pressing the F1-key dumps the process table copied from the kernel's system task. Because the information server itself remains stateless and other services do not depend on its functionality, exceptions triggered by the formatting can be transparently recovered via an automated restart or dynamic update. The only thing that needs to be done after a restart is resetting the function key mappings at the keyboard driver, but all other system services are unaffected.

4.4 Case Study: Monitoring Driver Correctness

As discussed in Sec. 4.3.1, MINIX 3 can restart crashed block-device drivers transparently to the file server, but the driver manager lacks the necessary information to detect protocol failures. First, even though block I/O is idempotent and can be retried, the lack of end-to-end integrity means that the file server cannot detect silent corruption of user data. Second, since the MINIX 3 driver manager does not have semantic information about the driver's correct operation and cannot monitor individual driver requests, it cannot detect, for example, a buggy driver that causes legitimate I/O requests to fail. Therefore, we have created a framework that allows installing a *filter driver* between the file server and block-device driver. In this way, the filter driver can transparently implement different protection strategies for safeguarding the user's data. This idea is similar to I/O shepherding [Gunawi et al.,

2007], but the filter driver implements the same interface as the block-device driver so that it can be inserted between the file server and block-device driver without having to modify either of them, or even having them to be aware of the filter. In addition, the filter driver can leverage some of MINIX 3's unique features. For example, if a problem is detected, the filter driver can file a complaint with the driver manager in order to restart the block-device driver.

The filter driver monitors the block-device driver's operation and protects the user's file-system data in two ways. First, it introduces end-to-end integrity by transparently checksumming and optionally mirroring file-system data [Sivathanu et al., 2005; Krioukov et al., 2008]. If both checksumming and mirroring are enabled, file-server read requests are still served from the primary partition, but write requests are transparently duplicated to the backup partition. Writes are read back in order to verify that the data actually made it to the disk. In this way, the filter driver can detect the otherwise silent data corruption, and recover the data from the backup partition, if need be. Second, the filter driver verifies correct driver semantics by monitoring requests and replies for deviations from the driver's specified behavior, for example, if the driver sends an unexpected reply or fails to handle a legitimate request. Such semantics are device-specific. For block devices, we assume that sector-aligned requests that span a sector-multiple and do not exceed the partition's size must succeed, that is, the driver must return OK in the reply.

The working of the filter driver is illustrated in Fig. 4.7. The sequence of actions is as follows: (1) file server requests are transparently intercepted by the filter driver, which (2) copies or maps the data into its address space, computes the checksum for all data blocks, and writes out the checksummed data to disk. Next, the filter driver awaits the reply from the block-device driver and (3) verifies that the result of the I/O operation is as expected. If an anomaly is detected, the filter driver starts

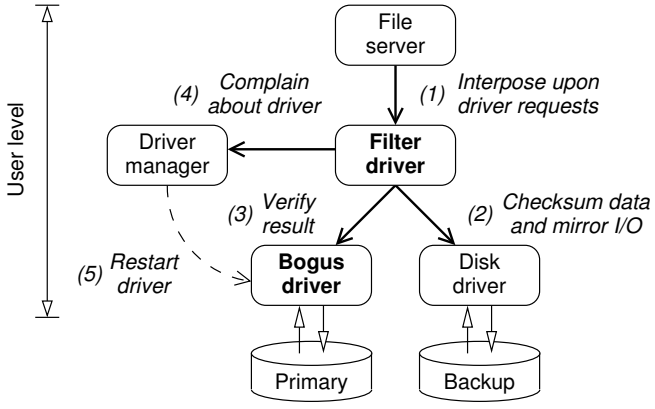


Figure 4.7: A filter driver between the file server and block-device driver can check for driver protocol violations. Different protection strategies based on checksumming and mirroring are supported. If an error is detected, a complaint is filed with the driver manager.

its recovery procedure and (4) files a complaint with the driver manager. Since the filter driver's isolation policy grants the privilege to control block-device drivers, the driver manager acknowledges the complaint and (5) replaces the block-device driver with a fresh copy. All this is done transparently to the rest of the storage stack.

One problem with this approach is that it is impossible to distinguish between controller or drive failures that are faithfully reported by the driver and internal driver failures. While a failed operation may be successfully retried for temporary driver failures, the problem is likely to persist for hardware problems. The recovery strategy acknowledges this fact by checking for similar failure characteristics and giving up after a predefined number of retries.

In order to reserve space for the checksums, the filter driver presents to the file server a virtual disk image smaller than the physical disk. Since the filter driver is not aware of important file-system data structures nor the file-system layout on disk, we checksummed each 512-B data sector independently. We decided to store the checksums close to the data sectors in order to eliminate the overhead of extra disk seeks. In addition, gaps in the on-disk layout of data and checksums should be prevent to maximize the disk's bandwidth and throughput. Therefore, we interspersed data sectors and checksums sectors, as shown in Fig. 4.8. In principle, each checksum sector can contain SECTOR_SIZE / CHECKSUM_SIZE checksums. However, if the checksums for write requests do not cover a whole checksum sector, the checksum sector needs to be read before it can be written—or the checksums of the other data will be lost. Because the optimal layout depends on the file-system block size, we made the number of checksums per checksum sector a filter parameter.

With a single-driver and single-disk configuration, we can give hard guarantees for only detection of data corruption—because a driver can simply wipe the entire disk with no backup to recover from. Nevertheless, two best-effort recovery strategies are possible. First, the filter driver can reissue the failed operation to the block-device driver up to N times. Second, the filter driver can complain about the driver's behavior to have it replaced up to M times. After a total of M restarts $\times$ N retries, the filter has to give up and return an error to the client file server. This strategy can be attempted for either individual operations or the driver's entire life span.

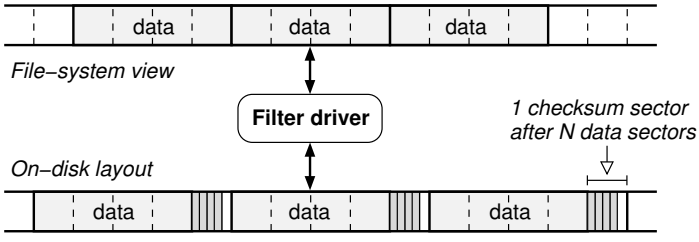


Figure 4.8: The filter driver intersperses 1 checksum sector for every N data sectors. This figure shows the file-system view and on-disk layout for N = 4 checksums per checksum sector.

With a mirrored setup we can give hard guarantees for recovering from single-disk or single-driver failures. Different approaches need to be distinguished for reading and writing. Recovery in case of read failures can be attempted by reading data from the backup partition and bringing the primary into a consistent state. The filter driver can either attempt the above best-effort recovery strategy for the primary partition or directly switch to the backup. Recovery of write failures poses a different problem because mirroring requires all data to be written to both disks. Upon a block-device driver failure, the filter driver can first attempt best-effort recovery and, if the failure persists, gracefully degrade the level of service by shutting down the bad partition and continuing with the remaining one.

4.5 Case Study: Automating Server Recovery

As a second case study we now show how recovery of a stateful server can be automated using recovery scripts. Even though full transparent recovery is not possible, we have implemented support for recovering from failures in the network server, INET. A failure of INET closes all open network connections, including the sockets used by the X Window System. Among other things, recovery requires restarting the DHCP client and rerunning the DHCP exchange. Such a failure generally affects the end user on a desktop machine, but we support the user by automating the recovery process using a shell script that automatically reconfigures the system. This is especially helpful to improve availability of (remotely managed) server machines.

In principle, INET could use the data store to backup its state and restore it after a restart, but we did not investigate this option since INET's state changes on every network operation and hard guarantees cannot be given. As soon as INET acknowledges that it successfully received a TCP packet, it is responsible for preventing data loss. This implies that all network data should be copied to the data store before sending the TCP ACK. In addition, there is a race condition in passing on the data to its clients and updating the internal state such that the data is marked 'successfully processed.' More advanced methods for restarting stateful servers are left as a possible direction for future research.

In order to support the administrator in handling INET failures, we have created a dedicated recovery script that automatically reconfigures the networking daemons, as illustrated in Fig. 4.9. Lines 3–11 and lines 13–23 respectively show the functions abort and daemonize that are used for stopping and starting daemons specified by their binary name. Stopping is done by looking up the daemon's process ID in the process table and killing it. Starting is done by executing the daemon's binary in the background. Lines 25–32 show the actual recovery procedure: the daemons *dhcpd*, *nonamed* and *syslogd* are stopped and INET is restarted along with the daemons. A variant of this script is run on the web server that produces weekly snapshots of MINIX 3. Since this server does not serve any X sessions, INET failures may cause the server to be temporarily unavailable, but generally can be fully recovered.

```
1  #!/bin/sh                          # INET RECOVERY SCRIPT
2
3  # FUNCTION TO STOP A DAEMON
4  abort() {
5      pid=‘ps ax | grep "$1" | grep -v grep | sed ’s,[       ]*([0-9]*).*,1,‘
6      if [ X"$pid" = X ]             # check for daemon pid
7      then
8          return 1                   # no such process
9      fi
10     kill -9 $pid                   # configure daemon down
11 }
12
13 # FUNCTION TO START A DAEMON
14 daemonize() {
15     for dir in $PATH               # search entire path
16     do
17         if [ -f "$dir/$1" ]        # check for executable
18         then
19             "$@" &                 # execute daemon
20             return
21         fi
22     done
23 }
24
25 # START OF ACTUAL RECOVERY SCRIPT
26 abort dhcpd                       # kill networking daemons
27 abort nonamed
28 abort syslogd
29 service restart "$1"               # restart network server (INET)
30 daemonize dhcpd                    # restart networking daemons
31 daemonize nonamed -L
32 daemonize syslogd
```

Figure 4.9: Dedicated recovery script for the network server (INET). Networking daemons are stopped and restarted only after INET has been recovered by the driver manager.

Chapter 5

Experimental Evaluation

Having presented in detail our design, we now discuss how we have evaluated its implementation in MINIX 3. Rather than formally proving our system correct, we have iteratively refined our design using a pragmatic, empirical approach based on extensive *software-implemented fault injection* (SWIFI). Although not the focus of this thesis, we also assess the system's performance and analyze the engineering effort. This chapter presents the raw results of our experiments; we will summarize the lessons learned and draw conclusions in Chap. 7.

The remainder of this chapter is organized as follows. To begin with, Sec. 5.1 describes our SWIFI methodology and presents the test results, including the effectiveness of our fault-isolation and failure-resilience mechanisms. Next, Sec. 5.2 gives insight into the performance of MINIX 3 and compares its performance to FreeBSD and Linux. Finally, Sec. 5.3 quantifies the engineering effort by analyzing the MINIX 3 code base and comparing it to the Linux code base.

5.1 Software-implemented Fault Injection

We have used software-implemented fault injection (SWIFI) to assess two aspects of MINIX 3's design. First, we want to show that common errors in a properly isolated device driver cannot propagate and damage the system. Second, we want to test the effectiveness of our defect detection and recovery mechanisms.

5.1.1 SWIFI Test Methodology

The SWIFI tests emulated a variety of problems underlying OS crashes by injecting selected machine-code mutations representative for both (i) low-level hardware faults and (ii) a range of common programming errors. The fault injection is done at run time and does not require driver modification before the fact. Below, we further introduce the fault-injection procedure, fault types and test coverage, and driver configurations tested.

Fault-injection Procedure

An important goal of fault injection is to mimic real software bugs. Previous OS-level robustness tests have injected faults at different locations in the system, including (1) the application programming interface (API) [Koopman and DeVale, 1999; Kalakech et al., 2004], (2) selected kernel subsystems [Arlat et al., 2002; Gu et al., 2003], (3) the driver programming interface (DPI) [Albinet et al., 2004], and (4) the actual driver code [Durães and Madeira, 2002]. Although these studies evaluated different OS characteristics, a common pattern is the use of *external faults* injected at the interfaces and *internal faults* injected into the test target. Interestingly, a comparison of internal faults and external faults found that they induce different kinds of errors [Jarboui et al., 2002; Moraes et al., 2006]. In particular, external faults were not able to produce errors matching the error patterns provoked by real faults in drivers. In other words, external faults are not representative for residual software faults. Therefore, our approach is to inject mutations representative for common hardware faults and programming errors directly into the driver.

Each SWIFI test run is defined by the following parameters: fault type to be used, number of SWIFI trials, number of faults injected per trial, and driver targeted. 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. The workload used is designed to exercise the driver's functionality, that is, reading from and writing to the device. If the injected fault is on the driver's execution path it will be activated. A driver crash triggers the test suite to sleep for 10 seconds, allowing the driver manager to refresh the driver—transparently to application programs and without user intervention as described in Sec. 4.2. When the test suite awakens, it looks up the process ID (PID) of the (restarted) driver, and continues injecting faults until the experiment finishes.

The test suite injects faults without requiring changes to the targeted driver or the rest of the OS. In particular, we use a variant of UNIX process tracing (ptrace) to control execution of the driver and corrupt its code segment at run time. We do not alter the data segment to simulate wrong initialization of global or static variables, since we believe it to be more likely that programming errors occur in the actual program code. For each fault injection, the code to be mutated is found by calculating a random offset into the driver's 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 break down and inspect the instructions' properties. Finally, the test suite injects the selected fault by writing the corresponding code modification into the driver's text segment using the ptrace system call. Finally, the driver is allowed to run again and may activate the fault injected.

During the SWIFI tests we verified that the driver could successfully execute its workload and inspected the system logs for anomalies afterward. In order to collect data we instrumented the test environment to produce debug output for specific interesting actions. The test target itself, however, was run unmodified in order not

to influence the experiment. The results presented below are based on the following data. For each SWIFI trial the test suite outputs the fault type, number of faults injected, and whether the test target has been restarted since the previous trial. The kernel and trusted servers and drivers that interact with the driver log violations of the driver's isolation policy. Finally, if a process crashes, the driver manager logs the component name, crash reason, and whether it could be restarted.

Fault Types and Test Coverage

Our test suite injected a meaningful subset of all fault types supported by the fault injector [Ng and Chen, 1999; Swift et al., 2005]. For example, faults targeting dynamic memory allocation (malloc) were left out because this is not used by our drivers. This selection process led to 8 suitable fault types, which are summarized in Fig. 5.1. First, binary faults flip a bit in the program text to emulate hardware faults [Kanawati et al., 1995]. This can cause a wide variety of crashes, but is difficult to relate to software bugs. It does, however, give some indication of how resilient the system is to certain kinds of hardware errors, such as bit flips caused by cosmic rays or bad memory banks. The other fault types approximate a range of C-level programming errors commonly found in system code. For example, pointer faults corrupt address calculation and source and destination faults respectively corrupt the right-hand and left-hand assignment operand in order to emulate pointer management errors, which were found to be a major cause (27%) of system outages [Sullivan and Chillarege, 1991]. Similarly, control faults change loop or branch conditions to mimic off-by-one and other checking errors; parameter faults omit operands loaded from the stack to change function invocations; and omission faults can lead to a variety of errors due to missing statements [Chillarege et al., 1992]. Finally, random faults are randomly selected from the other fault types.

Our SWIFI methodology is aligned with the fault and failure model described in Sec. 2.5. We limited ourselves to simulating soft intermittent faults, which were found to be a common crash cause [Gray, 1986; Chou, 1997]. If a fault is triggered and causes a driver failure, we refresh the driver and start over with a clean state. We

Fault type	Text affected	Code mutation performed
Binary	Random address	Flip one random bit in the selected instruction
Pointer	In-memory operand	Corrupt address calculation (ModR/R byte or SIB byte)
Source	Assignment	Corrupt the right-hand operand's address calculation
Destination	Assignment	Corrupt the left-hand operand's address calculation
Control	Loop or branch	Swap 'rep' and 'repe' or invert the branch condition
Parameter	Function invocation	Delete operand loaded from stack, e.g. movl 4(ebp))
Omission	Random address	Replace the selected instruction with NOP instructions
Random	One of the above	Corresponding code mutation from above mutations

Figure 5.1: Fault types and code mutations used for SWIFI testing. Our test suite can either inject faults of a predefined fault type or randomly pick one from the seven unique fault types.

did not inject other faults types, such as hard permanent faults or Byzantine failures, since they are outside the scope of MINIX 3's protection mechanisms.

We have iteratively refined our design by injecting increasingly larger numbers of faults using several different system configurations. We have not attempted to emulate all possible (internal) error conditions [Christmansson and Chillarege, 1996; Durães and Madeira, 2006] because we believe that the real issue is exercising the (external) techniques used to confine the test target. To illustrate this point, for example, algorithmic errors would primarily test the driver's functionality rather than MINIX 3's fault-tolerance mechanisms. For this reason, we also did not attempt to analyze the fraction of driver code executed during testing. Instead, we focused on the behavior of the (rest of the) system in the presence of a faulty component, and injected increasingly larger number of faults to increase the likelihood of finding shortcomings in MINIX 3's defense mechanisms. While complete coverage cannot be guaranteed without more formal approaches, our extensive SWIFI tests proved to be very effective and pinpointed various design problems. Analysis of the system logs also showed that we obtained a good test coverage, since the SWIFI tests stressed each of the isolation techniques presented in Sec. 3.3.

A final point worth mentioning is that we performed far more rigorous SWIFI tests than related work that attempts to isolate drivers. Previous efforts often limited their tests to a few thousand faults, which may not be enough to trigger rare faults and find all the bugs in the rest of the system. For example, Nooks [Swift et al., 2005], Safedrive [Zhou et al., 2006], and BGI [Castro et al., 2009] reported results on the basis of only 2000, 44, and 3375 fault injections, respectively. However, in our experiments, we found that this is not nearly enough. Instead, millions of faults were injected before we found no more bugs in the fault-tolerance mechanisms, because some bugs have a low probability of being triggered. To make the software very reliable, even these bugs must be found and removed. MINIX 3 can now survive several millions of fault injections, which strengthens our trust in its design.

Driver Configurations Tested

In order to ensure that our tests are representative we have experimented with each of the device classes discussed in Sec. 4.3. We selected network-device drivers as our primary test target after we found that networking forms by far the largest and fastest-growing driver category in Linux 2.6. Nevertheless, we also experimented with block-device drivers and characters-device drivers. Fig. 5.2 summarizes the drivers and devices tested; the exact hardware configurations and workloads used are described along with the results of each experiment in Secs. 5.1.2–5.1.4. Because the first three configurations use the same MINIX 3 driver binary, we use the device identifiers to distinguish the experiments in the text below.

We believe that the selected test targets cover a representative set of complex interactions. Although each of the drivers represents at most thousands of lines of code, complexity should not be assessed on the basis of lines of code. Instead, com-

Class	Driver	Device	Bus	I/O method	Results
Network	DP8390	Emulated NE2000	ISA	Programmed	Sec. 5.1.2
	DP8390	NS DP8390 card	ISA	Programmed	Sec. 5.1.2
	DP8390	Realtek RTL8029 card	PCI	Programmed	Sec. 5.1.2
	RTL8139	Realtek RTL8139 card	PCI	DMA	Sec. 5.1.2
	FXP	Intel PRO/100 card	PCI	DMA	Sec. 5.1.2
Block	ATWINI	Sitecom CN-033 card	SATA	Mixed	Sec. 5.1.3
Character	ES1371	Ensoniq ES1371 card	PCI	DMA	Sec. 5.1.4

Figure 5.2: Overview of the MINIX 3 driver configurations that were subjected to fault injection.

plexity should be measured by a driver's interactions with the surrounding software and hardware, which determine the possible failure modes. We tested drivers for several different hardware configurations, including network, block, and character devices, because, as explained in Sec. 4.3, each driver stack has different recovery properties. In addition, we ensured that our tests covered the full spectrum of isolation mechanisms devised. For example, we have tested drivers using both programmed I/O and DMA. Moreover, all drivers heavily interact with the kernel, system servers and support drivers such as the PCI-bus driver and IOMMU driver. Our test setup also heavily relied on MINIX 3's ability to restart failed drivers on the fly, but we did not stress the state-management facilities offered by the data store, because all of the drivers tested are stateless.

5.1.2 Network-device Driver Results

First of foremost, we tested MINIX 3's ability to withstand failures in the network stack. Because network-device drivers can be recovered transparently, as described in Sec. 4.3.1, we were able to automate the test procedure and inject literally millions of faults in order to stress test the system's fault-tolerance techniques. For these experiments we used the following hardware configurations:

1. Emulated NE2000 ISA on Bochs v2.2.6.
2. NS DP8390 ISA card on Pentium III 700 MHz.
3. Realtek RTL8029 PCI card on Pentium III 700 MHz.
4. Realtek RTL8139 PCI card on AMD Athlon64 X2 3800+.
5. Intel PRO/100 PCI card on AMD Athlon64 X2 3800+.

The workload used in all SWIFI tests caused a continuous stream of network I/O requests in order to exercise the drivers' functionality and increase the probability that the injected faults are triggered. In particular, we sent TCP requests to a remote *daytime* server. The workload is transparent to the working of the drivers, however, since they simply put INET's message buffers on the wire (and vice versa) without inspecting the actual data transferred.

Robustness against Failures

The first experiment was designed to stress test our fault-tolerance techniques by inducing driver failures with a 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 4 driver configurations for each of the 8 fault types used. As expected, the drivers repeatedly crashed and had to be restarted by the driver manager.

The test results are shown in Fig 5.3, which gives the total number of failures, and Fig. 5.4, which gives a stacked histogram highlighting the failure reasons for each fault type and each driver. For example, for random faults injected into the NE2000, DP8390, RTL8139, and PRO/100 drivers we found a total number of 826, 552, 819, and 931 failures, respectively. These failures are subdivided based on the failure reasons logged by the driver manager. For example, for random faults injected into the NE2000 driver the driver manager reported 349 (42.3%) failures where the driver was signaled due to a CPU or MMU exception, 454 (54.9%) internal panics, and 23 (2.8%) missing heartbeats. 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. Moreover, in all these cases, the system was able to recover the driver transparently and without data loss.

The figures also show that different fault types affected the drivers in different ways. For example, source and destination faults more consistently caused failures than pointer and omission faults. In addition, we also observed some differences between the drivers themselves, which is clearly visible for pointer and control faults. For example, for pointer faults the NE2000, DP8390, RTL8139, and PRO/100 drivers failed 293, 108, 849, and 757 times, respectively. Since one driver may be programmed to panic and exit upon the first failure, whereas the other may repeatedly retry failed operations until the driver manager kills it due to a missing heartbeat, the differences seem logical for the configurations with different drivers. However, the effect is also present for the NE2000 and DP8390 configurations that

Driver	NE2000	DP8390	RTL8139	PRO/100
Source faults	947	907	877	960
Destination faults	954	915	883	970
Pointer faults	293	108	849	757
Parameter faults	555	921	890	986
Control faults	798	980	279	884
Binary faults	933	708	899	932
Omission faults	705	729	711	425
Random faults	826	552	819	931
Total failures	**6011**	**5820**	**6207**	**6845**

Figure 5.3: Total number of failures induced for 4 network-device drivers and 8 fault types. For each driver and fault type we conducted 1000 SWIFI trials injecting 100 faults each.

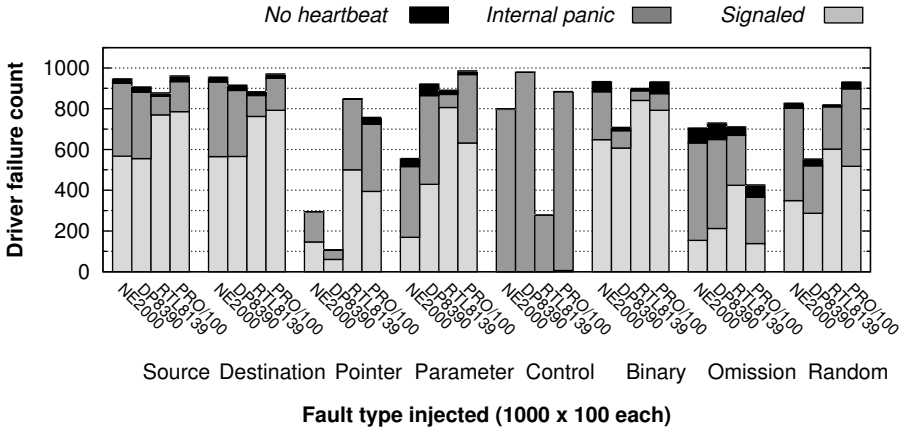


Figure 5.4: Number of driver failures and failure reasons for 4 network-device drivers and 8 fault types. For each driver and fault type we conducted 1000 SWIFI trials injecting 100 faults each.

use the same driver binary. 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 timings of the fault injections.

Unauthorized Access Attempts

Next, we analyzed the nature and frequency of unauthorized access attempts and correlated the results to the classification of privileged driver operations in Fig. 3.1. In order to do so we instrumented the system with additional debugging output and conducted 100,000 SWIFI trials that each injected 1 random fault into the RTL8139 driver. The results reported in Fig. 5.5 should be taken as approximate because the abundance of measurement data from various components cluttered the system logs and sometimes caused unintelligible entries when multiple messages that logically belonged together were written in an interleaved fashion. Furthermore, while MINIX 3 has many sanity checks in the system libraries linked into the driver, we have focused on only the logs from the kernel and the trusted system servers, since their checks cannot be circumvented.

The test results provide various insights into the working of our defenses. The driver manager detected 5887 failures that caused the RTL8139 driver to be replaced: 3738 (63.5%) exits due to internal panics, 1870 (31.8%) crashes due to CPU or MMU exceptions, and 279 (4.7%) kills due to missing heartbeats. However, as shown in Fig. 5.5, the number of unauthorized access attempts found in the system logs was up to three orders of magnitude higher, totaling 2,162,054. This could happen because not all error conditions are immediately fatal and certain failed operations were repeatedly retried. For example, the logs revealed 1,754,886 (81.1%) device I/O attempts that were denied because the registers requested did not belong to the RTL8139 card. Likewise, we found 390,741 (18.5%) IPC calls that were re-

Unauthorized access	# Violations	Percentage
Total violations detected	2,162,054	100.00%
Unauthorized CPU access	**1851**	**0.09%**
General protection fault (interrupt vector 13)	1593	0.07%
Stack exception (interrupt vector 12)	133	0.01%
Invalid opcode (interrupt vector 6)	103	0.00%
Divide error (interrupt vector 0)	11	0.00%
Breakpoint (interrupt vector 3)	7	0.00%
Debug exception (interrupt vector 1)	4	0.00%
Unauthorized memory access	**16,964**	**0.78%**
Grant ID is out of range of grantor's grant table	14,830	0.69%
Invalid grantor endpoint or invalid grant ID	1332	0.06%
Memory region requested exceeds memory granted	448	0.02%
Access type requested violates grant modifiers	326	0.02%
Grant ID refers to unused or invalid memory grant	28	0.00%
Unauthorized device I/O	**1,754,886**	**81.08%**
Device port or register not allowed by policy	1,754,886	81.08%
Unauthorized IPC	**390,741**	**18.05%**
Kernel call not allowed by driver policy	198,487	9.17%
Kernel call rejected due to invalid call number	123,518	5.71%
IPC trap with invalid destination IPC endpoint	51,041	2.36%
IPC attempted to unauthorized destination	15,214	0.70%
Unauthorized request rejected by IPC target	2361	0.11%
IPC trap with invalid IPC primitive number	20	0.00%

Figure 5.5: Unauthorized access attempts found in the system logs for an experiment with the RTL8139 driver that consisted of 100,000 SWIFI trials injecting 1 random fault each.

jected because the kernel call or system service requested was not allowed by the driver's isolation policy. Code inspection confirmed that the RTL8139 driver repeatedly retried failed operations before exiting due to an internal panic, being killed by the driver manager due to a missing heartbeat, or causing an exception due to subsequent fault injections. Despite all these failures, we again found that the base system was never (noticeably) affected by the driver's misconduct.

Availability under Faults

We also tested the driver's sensitivity to faults. In order to do so we have conducted 100,000 SWIFI trials that each injected 1 random fault into the DP8390 driver, and measured how many faults it takes to disrupt the driver and how many more are needed for a crash. Disruption means that the driver can no longer successfully service its workload, but has not yet failed in a way detectable by the driver manager. After injecting a fault several things can happen. If the fault injected is not on the

path executed, the driver continues normal execution as if nothing happened. If the fault injected is triggered, the fault activation can change the driver's execution in various ways, including no (immediately) noticeable effect, a nonfatal error where the driver deviates from its specified behavior but does not crash, or a fatal error that causes the driver to be replaced. We were able to distinguish these effects by maintaining a connection to a remote server and checking for availability after each SWIFI trial. If the connection works fine, the fault is either not triggered or has no noticeable effect. If the connection does not work, the driver is either disrupted or has crashed, which can be told apart based on the driver manager logs.

Figs. 5.6 and 5.7 show the distribution of the number of faults needed to disrupt and crash the DP8390 driver and RTL8139 driver, respectively. Although the RTL8139 driver seems slightly more sensitive to faults than the DP8390 driver, a similar pattern is visible for both drivers. On the one hand, we found that disruption usually happens after only a few faults. For example, we observed 664 disruptions and 136 crashes for the DP8390 driver and 1245 disruptions and 815 crashes for

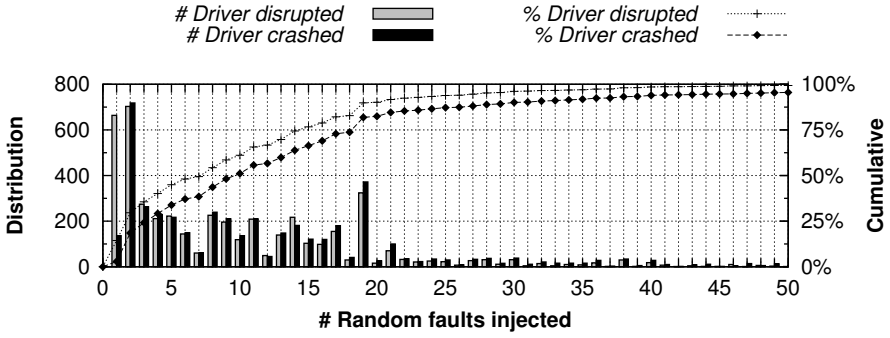


Figure 5.6: Number of faults needed to disrupt and crash the DP8390 driver during 100,000 random SWIFI trials. Crashes show a long tail to the right and surpass 99% only after 263 faults.

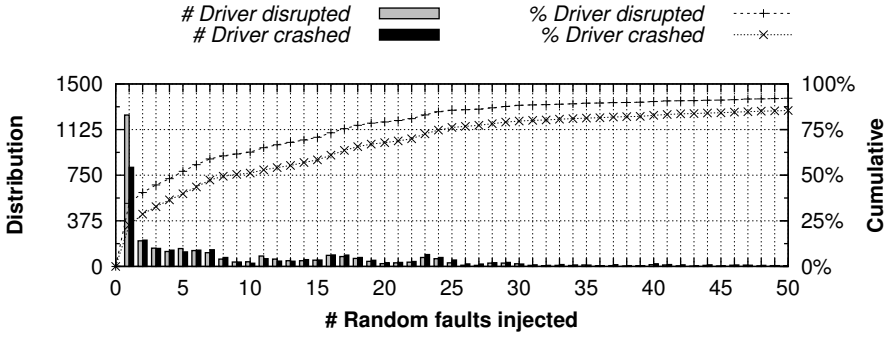


Figure 5.7: Number of faults needed to disrupt and crash the RTL8139 driver during 100,000 random SWIFI trials. Crashes show a long tail to the right and surpass 99% only after 282 faults.

the RTL8139 driver after just 1 fault injection. On the other hand, we found that the number of faults needed to induce a crash can be high and shows a long tail to the right. For example, the cumulative distribution of the number of faults needed to crash the DP8390 driver surpasses 99% only after 263 fault injections. One run even required 2484 fault injections before the driver crashed. The median numbers of fault injections needed to disrupt and crash the DP8390 driver were 8 faults and 10 faults, respectively. The medians for the RTL8139 driver were 5 faults and 9 faults, respectively. Experiments with other drivers gave similar results. On the basis of these findings we picked a fault load of 100 faults per SWIFI trial for the stress tests used to assess MINIX 3's robustness.

Software and Hardware 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 underscore this point, Fig. 5.8 briefly summarizes some of the problems that we encountered (and subsequently fixed) during the SWIFI tests. Interestingly, we found many obscure bugs even though MINIX 3 was already designed with dependability in mind, which illustrates the usefulness of fault injection.

A final but important result is that we experienced several insurmountable hardware problems that put an upper limit on the amount of fault isolation and failure resilience that can achieved. While the hardware is, in principle, not directly affected by the faults injected into the driver software, in some cases the fault injection caused the driver to program its device in such way that hardware problems became apparent. Such hardware problems are virtually impossible to deal with in software, and are not specific to our design. Unfortunately, these kind of problems can only be solved with help from PC hardware manufacturers.

- Kernel stuck in infinite loop during load update due to inconsistent scheduling queues.
- Process manager using synchronous IPC blocked by a driver not willing to receive it.
- Driver request to perform SENDREC with nonblocking flag goes undetected and fails.
- IPC call to SENDREC with invalid target ANY not detected and kept pending forever.
- IPC call to NOTIFY with invalid target ANY caused a panic rather than erroneous return.
- Kernel panic due to dereferencing an uninitialized privilege structure pointer.
- Network driver went into silent mode due to bad parameters upon driver manager restart.
- Driver manager's priority was too low to request heartbeats from a looping driver.
- System-wide starvation occurred due to excessive debug messages during kernel calls.
- Isolation policy allowed driver to make arbitrary memory copies, corrupting the INET server.
- Driver reprogrammed RTL8139 hardware's PCI device ID (unexpected code in the driver).
- Wrong IOMMU setting caused legitimate DMA operation to fail and corrupt the file system.
- Interrupt line table filled up because clean-up after driver exit was not correctly done.

Figure 5.8: Selected bugs that were encountered (and subsequently fixed) during SWIFI testing.

The hardware failures manifested in two different ways. First, for one network card, hardware shortcomings thwarted MINIX 3's fault isolation. SWIFI tests with the Realtek RTL8029 PCI card repeatedly caused the entire system to freeze within just a few SWIFI trials. We narrowed down the problem to writing a specific (unexpected) value to an (allowed) control register of the PCI device—presumably causing the PCI bus to hang. We believe this to be a peculiarity of the specific device and a weakness in the PCI-bus architecture rather than a shortcoming of MINIX 3. A possible workaround would be to inspect all of the driver's I/O requests at the kernel level—something we were not willing to do.

Second, for two other network cards, the effectiveness of recovery was limited by the hardware. Fortunately, such cases occurred very infrequently; less than 0.1% in these series of tests. In particular, the emulated NE2000 ISA card was put in an unrecoverable state in fewer than 10 cases, whereas the PRO/100 PCI card showed a similar problem in under 25 cases. The DP8390 ISA and RTL8139 PCI cards did not have this problem. Although the device driver could be successfully restarted, the cards could not be reinitialized by the restarted driver. The rest of the OS was not affected by this problem, but a low-level BIOS reset was needed to get the card to work again. If the card had had a 'master-reset' command, the driver could have solved the problem, but our card did not have this.

5.1.3 Block-device Driver Results

We also tested MINIX 3's ability to deal with failures in the storage stack. Because the MINIX file system does not provide end-to-end integrity, we augmented the storage stack with the filter driver presented in Sec. 4.4. We used an AMD Athlon64 X2 3200+ machine with a Sitecom CN-033 Serial ATA PCI RAID controller and two hard disk drives, each of which was controlled by an independent ATWINI driver. The faults were injected into one of the two ATWINI drivers. The workload consisted of writing and reading back 5-MB randomly generated data using *dd*. We checked the I/O stream's data integrity by comparing the SHA-1 hashes afterward.

Manual and Automated Fault Injection

We started out by testing the principle working of the filter driver's protection techniques using a small number of carefully selected, manually injected faults. First, we manipulated the code of one of the ATWINI drivers in order to mimic data-integrity violations. For example, we let the driver respond OK while not doing any work, changed the disk address to be read, and so on. Second, we provoked driver crashes and other erroneous behavior in order to emulate driver-protocol failures. Third, we caused (permanent) failures on one partition in order to test recovery with help of the backup partition. These tests confirmed the filter driver's correct working with respect to detection of data corruption and protocol violations, retrying of failed operations, recovery of corrupted data, and graceful degradation.

Next, we conducted a series of automated fault-injection experiments. For each test run we attempted 40 SWIFI trials that each injected 25 faults into the running ATWINI driver. This fault load ensures that each SWIFI trial has a high probability to induce a driver failure. The filter driver was configured to use checksumming but no mirroring, so that the targeted partition would not be shut down due to repeated driver failures. The filter driver's recovery strategy was set to a maximum of $M = 3$ driver restarts and $N = 3$ retries per request. Experiments with different parameters showed that further recovery attempts are usually pointless. The results of 7 representative test runs are shown in Fig. 5.9.

The results show a mixed picture. The successful test runs, T.1–T.3, indicate that the filter driver is indeed effective in dealing with misbehaving and crashing block-device drivers. Here, the system logs revealed a total of 94 driver restarts due to 17 (18.1%) internal panics, 22 (23.4%) CPU or MMU exceptions, 6 (6.4%) missing heartbeats, and 49 (52.1%) filter-driver complaints. The breakdown of problems shows that the filter driver can detect both data-integrity problems and driver-protocol violations, such as time-outs and unexpected replies. If retrying did not help, the filter driver asked the driver manager to replace the ATWINI driver—except when a request was undeliverable due to a missing driver, in which case the driver

SWIFI test run	**T.1**	**T.2**	**T.3**	**T.4**	**T.5**	**T.6**	**T.7**
SWIFI trials × 25 faults	40	40	40	12	9	18	13
Total driver requests	1648	1724	1796	566	249	745	504
Driver requests failed	0	0	0	0	2	11	12
SWIFI test result	OK	OK	OK	*Hang*	*Hang*	*Hang*	*Hang*
Driver-manager restarts	**33**	**31**	**30**	**11**	**18**	**63**	**59**
Driver exit due to panic	5	7	5	0	0	3	3
Crashed due to exception	9	5	8	4	3	3	2
Missing driver heartbeat	1	4	1	2	10	38	38
Filter-driver complaint	18	15	16	5	5	19	16
Filter-driver output	**92**	**88**	**95**	**26**	**18**	**64**	**49**
Driver dead when sending	0	1	1	1	5	24	25
Driver receive time-out	18	14	17	5	4	7	10
Unexpected IPC reply	24	33	33	9	3	15	6
Legitimate request failed	35	40	38	11	3	18	8
Bad checksum detected	15	0	6	0	3	0	0
Read-after-write failed	0	0	0	0	0	0	0
ATWINI-driver output	**1**	**4**	**2**	**5**	**17**	**84**	**95**
Controller not ready	1	4	2	5	15	73	77
Reset failed, drive busy	0	0	0	0	2	11	13
Timeout on command	0	0	0	0	0	0	5

Figure 5.9: Results of seven SWIFI tests with 40 SWIFI trials that each injected 25 faults of a random type into the ATWINI driver. Results are ordered by the number of requests that failed.

manager automatically restarted the crashed driver. In all these cases, the driver manager replaced the crashed or misbehaving driver with a fresh copy. Even though a 100% success rate cannot be guaranteed without a backup to recover from, we found that the filter driver's best-effort recovery was generally effective, especially after a requesting a dynamic update of a bad driver.

More Hardware Limitations

The remaining test runs, T.4–T.7, did not run to completion because the system hung before completing the 40 SWIFI trials. While the filter driver behaved as intended, the Sitecom CN-033 PCI card did not, and limited the number of faults we could inject. Compared to the network-device driver tests we experienced a relative large number of cases where (1) the CN-033 controller was confused and required a BIOS reset and (2) the test PC completely froze, presumably due to a PCI bus hang. We also encountered a small number of filter-to-driver requests with unrecoverable failures, but the mere fact that we can detect these failures and warn the user is an improvement over silent data corruption. Interestingly, test run T.4 hung the system without failed driver requests, although there might have been a race condition in logging the filter driver's messages. For test runs T.5–T.7 ATWINI's diagnostic output clearly showed that the controller had difficulties with the driver's deviation from normal behavior: we observed frequent warnings that the controller was not ready, controller resets failed, or commands timed out. These are hardware problems and there is nothing the OS can do when a buggy driver issues an I/O command that causes the device to fail in a way that cannot be recovered in software.

5.1.4 Character-device Driver Results

Finally, we have experimented with character-device driver recovery where the I/O stream is interrupted and data may be lost. In particular, we injected faults into a driver for the Ensoniq ES1371 PCI audio card. The experiments were run on an AMD Athlon64 X2 3200+. The workload consisted of playing a song with the *mplayer* media player. The O_REOPEN flag was added to mplayer's open call in order to tell the virtual file system (VFS) to recover automatically and reassociate the file descriptor with the restarted driver; EREOPENED errors for failures occurring during an I/O operation were ignored by the mplayer application.

Effectiveness of Isolation and Recovery

To start with, we conducted 1000 SWIFI trials of 100 random faults each in order to stress test the character-device driver defenses. This showed that the ES1371 driver could be successfully recovered, although hiccups in the audio playback generally occurred, as discussed below. In total, this experiment injected 100,000 faults and induced 484 detectable failures: 347 (71.7%) CPU or MMU exceptions, 8 (1.7%) internal panics, and 129 (26.6%) missing heartbeat messages. Interestingly, the audio

driver showed a much higher number of failures due to missing heartbeat messages than the network-device and block-device drivers. We were unable to find the underlying reasons from the logs, but suspect that this is due to differences in how the driver is programmed.

In order to analyze how the audio playback is affected by the ES1371 driver failures, we connected the audio card's *line out* to the *line in* on a second PC. Fig. 5.10 shows the uninterrupted playback of a regularly shaped audio sample and the interrupted playback during 10 SWIFI trials of 100 random faults each. Three different effects are visible. First, if the driver fails to program the device in time, sample repetition occurs because the card's output buffer is no longer refreshed; the card simply continues playing whichever sample it finds in the buffer. Next, the fault injection may corrupt the audio sample, causing the card to output noise. This can happen because the driver uses double buffering of audio data, that is, it prepares a copy of the audio data in its address space, which may be garbled by the fault injection before the data is actually read by the device. Alternatively, the driver may program the audio card with a wrong I/O address pointing to arbitrary text or data in the driver's address space. Finally, DMA read operations that are rejected by the IOMMU cause the card's output buffer to be filled with ones, which translates to silence in the playback. This can happen if the driver crashes and its DMA protection domain is invalidated by the IOMMU driver or if the driver provides an unauthorized I/O address due to the fault injection.

The results show that, in general, normal operation continued after recovering the driver. In contrast, drivers failures in OSes with a monolithic kernel may bring down the entire system. If we translate this outcome to a normal usage scenario with infrequent intermittent driver failures, recovery at the price of a small hiccup in the audio playback brings a huge dependability improvement for the end user.

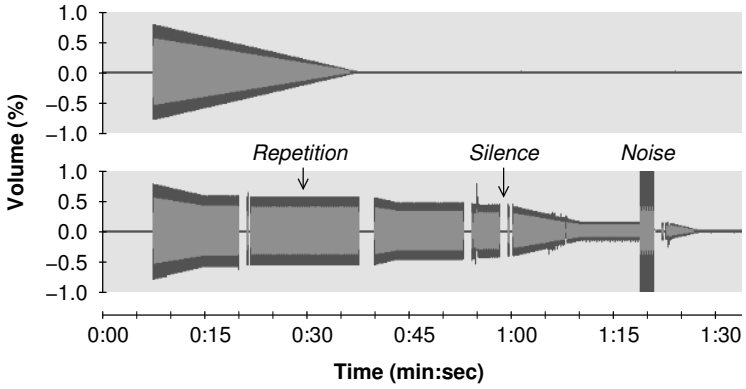


Figure 5.10: Normal playback of a regular audio sample (top) and playback with 10 SWIFI trials injecting 100 random faults each into the Ensoniq ES1371 PCI audio driver (bottom). Recovery is transparent to the application, but the I/O is repeatedly interrupted.

Application-level Transparency

In a few cases, the fault injection not only affected the ES1371 driver, but also induced a failure of the mplayer application due to unexpected driver replies. Since the code to handle I/O control operations (IOCTLs) is driver-specific and may, in principle, return any status code, the VFS cannot detect invalid return values and simply forwards the result to the application. A filter driver wrapping the ES1371 driver might be effective, but we did not investigate this option. Instead, we feel that this problem should have been handled by a more defensive programming style in the mplayer application. Applications making system calls should be prepared to handle unexpected return values in a sensible way and should not just crash.

A related problem is that, in some cases, the fault injection caused the driver to execute an unwanted but otherwise legitimate IOCTL without crashing the driver. For example, in one case, the mixer settings were changed to a different playback frequency. In such an event, recovery of state with help of the data store is not possible, since the data store cannot distinguish good requests from bad ones, and also would be updated with the wrong value. Although these kind of failures cannot be prevented, they rarely occurred and were easily dealt with by restarting the mplayer application, which causes the mixer settings to be reset.

5.2 Performance Measurements

Although the focus of this work is dependability rather than performance, we realize that performance is important for the system's usability. This section presents selected experiments assessing MINIX 3's performance. The results show that (1) the user-perceived overhead depends on the workload and (2) the inherent overhead due to a modular design seems to be less of an issue than careful optimization.

5.2.1 Costs of Fault Isolation

One objection that is often raised about modular designs is that they require additional context switches and data copies when user-level modules interact with one another. To find out how much overhead our design incurred, we conducted a number of benchmarks that trigger IPC between the MINIX 3 applications, servers, drivers, and kernel. Below, we present the results of a series of MINIX-specific tests as well as a cross-platform comparison with Linux and FreeBSD.

MINIX-specific Tests

As a first data point, MINIX 3 feels fast and responsive for research and development usage, such editing files and (re)compiling system components. To illustrate this point, we conducted the following measurements on an AMD AthlonXP X2 4400+ with 1-GB RAM. The time interval between leaving the multiboot monitor

and getting the login prompt is about 6 sec. At that time a POSIX-conformant OS is ready to use. Furthermore, the OS can do a full build of itself in about 19 sec. This includes 256 calls to the MINIX 3 ANSI C compiler and 33 calls to the linker in order to build the kernel and all the standard servers and drivers. As an aside, other measurements showed that the MINIX 3 C compiler (*cc*) is faster than the GNU C compiler (*gcc*), while the performance of the produced executables is only slightly worse at the default optimization level [Ahmad, 2008]. Finally, when presented with a new machine, MINIX 3 will install itself from the live CD in about 10 minutes.

Benchmarks comparing MINIX 2.0.4 to MINIX 3.0.0 showed that the transition from in-kernel to user-level drivers incurred a performance penalty of 5%–10% [Herder et al., 2006]. These experiments were conducted on an AMD Athlon64 3200+ with 1-GB RAM. The results are shown in Fig. 5.11. First, system call times for in-kernel versus user-level drivers showed an average overhead of 12%. For example, creating and removing a directory had 7% overhead, opening and closing a file had 9% overhead, and renaming a file had 16% overhead. With such simple calls, the extra context switching required by user-level drivers slows the call down measureably. Nevertheless, even though the percent difference for renaming a file is 16%, the delta in time is only 960 nsec per call, so even with 10,000 calls/sec the loss is only 9.6 msec/sec, under 1%. This makes clear that the workload determines the user-perceived overhead. Therefore, we also measured the performance of actual applications rather than pure system call times. The run times for various applications showed an average overhead of 6%. For example, building a boot image had 7% overhead, whereas grepping a 64-MB file was only 1% slower. These differences show that I/O-bound programs have more overhead than CPU-bound programs that do not depend on user-level drivers.

Next, we measured the performance of the MINIX 3 storage stack. We first compared the disk read throughput of MINIX 2.0.4 and MINIX 3.0.0 for both the raw-device and file-system interface using various I/O unit sizes. This experiment is

System call	MINIX 2.0.4	MINIX 3.0.0	Delta	Ratio
getpid	0.831 μs	1.011 μs	0.180 μs	1.22
lseek	0.721 μs	0.797 μs	0.076 μs	1.11
open+close	3.048 μs	3.315 μs	0.267 μs	1.09
read 64k+lseek	81.207 μs	87.999 μs	6.792 μs	1.08
write 64k+lseek	80.165 μs	86.832 μs	6.667 μs	1.08
creat+wr+del	12.465 μs	13.465 μs	1.000 μs	1.08
fork	10.499 μs	12.399 μs	1.900 μs	1.18
fork+exec	38.832 μs	43.365 μs	4.533 μs	1.12
mkdir+rmdir	13.357 μs	14.265 μs	0.908 μs	1.07
rename	5.852 μs	6.812 μs	0.960 μs	1.16

Figure 5.11: System call times for in-kernel drivers (MINIX 2.0.4) versus user-level drivers (MINIX 3.0.0). All times are wall-clock times in microseconds.

Raw reads	MINIX 2.0.4	MINIX 3.0.0	Delta	Ratio
1 KB	2.602 μs	2.965 μs	0.363 μs	1.14
16 KB	17.907 μs	19.968 μs	2.061 μs	1.12
256 KB	303.749 μs	332.246 μs	28.497 μs	1.09
4 MB	6184.568 μs	6625.107 μs	440.539 μs	1.07
64 MB	16.729 s	17.599 s	0.870 s	1.05

Figure 5.12: Raw reads for in-kernel drivers (MINIX 2.0.4) versus user-level drivers (MINIX 3.0.0). All times are in microseconds, except for the 64-MB operations, where they are in seconds.

File reads	MINIX 2.0.4	MINIX 3.0.0	Delta	Ratio
1 KB	2.619 μs	2.904 μs	0.285 μs	1.11
16 KB	18.891 μs	20.728 μs	1.837 μs	1.10
256 KB	325.507 μs	351.636 μs	26.129 μs	1.08
4 MB	6962.240 μs	7363.498 μs	401.258 μs	1.06
64 MB	16.907 s	17.749 s	0.841 s	1.05

Figure 5.13: File reads for in-kernel drivers (MINIX 2.0.4) versus user-level drivers (MINIX 3.0.0). All times are in microseconds, except for the 64-MB operations, where they are in seconds.

representative for MINIX 3's modularity, since the I/O triggered IPC between the test program, file server, ATWINI driver, and kernel. The results are shown in Figs. 5.12 and 5.13. The overhead ranged from 14% and 11% for 1-KB units to 7% and 6% for 4-MB units, respectively. The (relative) overhead thus decreases for larger I/O units. Therefore, we changed the file-system block size from 1 KB to 8 KB, and again measured the file-system throughput. Interestingly, we found that MINIX 3 now outperformed MINIX 2 despite the use of user-level drivers. The point we want to make is that an 8% performance hit due to user-level drivers is on the same order of magnitude as the gains or losses from configuring system parameters.

We also ran several application-level benchmarks with the filter driver on an AMD Athlon64 X2 Dual Core 4400+ with 1-GB RAM and two identical 500-GB Western Digital Caviar SE16 SATA hard-disk drives (WD5000AAKS). We used a standard MINIX 3 file system with a 4-KB block size and a 32-MB buffer cache. The test script created a new file system on the test partition, mounted it on */mnt*, copied the MINIX 3 installation, and executed the actual benchmark in a chroot jail. After each benchmark we synchronized the cache to disk, which is included in the reported run times. The average results out of three test runs are shown in Fig. 5.14. These results again show that the workload dominates the user-perceived overhead. First, workloads where writes dominate reads show higher overheads. Second, while the filter driver's overhead is visible for I/O-bound jobs, it is negligible for CPU-intensive jobs, even with the best protection strategy. For example, with both checksumming and mirroring, the overhead compared to running without filter is 28% for copying the source tree, 13% for doing a file system check, only 4% for building the MINIX 3 OS, and 0% for building the system libraries.

Benchmark	No Filter	Mirror	Checksum	Both
Copy root FS	14.89 (1.00)	15.44 (1.04)	17.11 (1.15)	18.34 (1.23)
Find and touch	2.75 (1.00)	2.83 (1.03)	2.94 (1.07)	2.91 (1.06)
Build libraries	28.84 (1.00)	29.10 (1.01)	28.82 (1.00)	28.72 (1.00)
Build MINIX 3	14.26 (1.00)	14.69 (1.03)	14.79 (1.04)	14.86 (1.04)
Copy source tree	2.54 (1.00)	2.73 (1.07)	3.06 (1.20)	3.26 (1.28)
Find and grep	5.16 (1.00)	5.23 (1.01)	5.65 (1.10)	5.67 (1.10)
File system check	3.46 (1.00)	3.55 (1.03)	3.91 (1.13)	3.91 (1.13)
Delete root FS	10.72 (1.00)	11.20 (1.05)	12.30 (1.15)	13.07 (1.22)

Figure 5.14: Application-level benchmarks for various filter driver configurations. Shown are the average run times in seconds and performance relative to 'No Filter' (in parentheses).

Finally, we tested the performance of the network stack. Because we initially did not have drivers for gigabit Ethernet cards, we measured the performance of a Fast Ethernet card. In particular, we used an Intel PRO/100 card, which is capable of carrying network traffic at a rate of 100 Mbit/s. This experiment triggered IPC between the test program, virtual file system (VFS), network server (INET), PRO/100 driver, and kernel. We first transferred a 512-MB file from the local network, and were able to drive the Ethernet at full speed. We also ran a loopback test and observed a throughput of 1.7 Gbit/s, which is roughly equivalent to both sending at 1.7 Gbit/s and receiving at 1.7 Gbit/s at the same time. A later port of the Intel PRO/1000 driver confirmed that MINIX 3 can indeed saturate gigabit Ethernet. However, the high number of messages associated with gigabit Ethernet incurred a high CPU load, and showed a need for further optimization [Linnenbank, 2009].

Cross-platform Comparison

In order to see where MINIX 3 stands compared to other OSes we have briefly contrasted MINIX 3 to Linux and FreeBSD. In particular, we compared the throughput and CPU utilization of sequential access to the raw-device interface of MINIX 3.1.2 to that of Linux 2.6.18 and FreeBSD 6.1. For each OS we used the out-of-the-box configuration with the default parameters. This represents a worst case for MINIX 3, since the overhead incurred by the user-level driver cannot be amortized over the costs associated with disk seeks and file-system logic. The experiment was done on an AMD AthlonXP 2200+ configured with 512-MB RAM and a 40-GB Maxtor 6E040L0 hard disk drive. Because the on-disk location influences the performance, we ensured that all disk I/O was done from the same test partition. We focused on the read performance in order to prevent possible caching effects from the disk controller. The comparison exemplifies the overhead of MINIX 3's modular design, since it involves IPC between the application, VFS server, file server, ATWINI driver, and kernel. In contrast, both Linux 2.6 and FreeBSD 6.1 have a monolithic kernel and require only a single system call to do I/O.

The test script measured the performance of sequential access to the raw-device interface by reading 2 GB worth of data using I/O unit sizes ranging from 1 KB to 256 KB. The throughput was calculated by putting the I/O operations between two gettimeofday calls and dividing the amount of data read by the test's duration. In order eliminate semantic differences between the CPU loads reported by each OS, the CPU utilization was measured by running the *dhrystone* CPU benchmark [Weicker, 1984] in parallel with the test. First, a base run without I/O was done in order to determine how many dhrystone iterations/sec could be done on an idle system. For each test run we measured the number of dhrystone iterations performed in parallel (D_{real}) and extrapolated the base rate to the expected number of dhrystone iterations for an idle system (D_{idle}). Since the difference between the two is attributable to the test run, the CPU utilization then was calculated as $(D_{idle} - D_{real})\ /\ D_{idle}$.

The results are plotted in Fig. 5.15. The results show that the disk throughput increases and CPU utilization decreases for larger I/O unit sizes. Compared to Linux 2.6, MINIX 3 shows a throughput degradation of 45.6% for 1-KB units, 31.2% for 4-KB units, 7.0% for 16-KB units, but no overhead for larger unit sizes. However, MINIX 3 also has a lower CPU utilization—possibly because the MINIX 3 code base is simpler and contains fewer optimization strategies—which might mean that there is room for improvement. The maximum throughput of 57.8 MB/s is reached at 64-KB units, but the CPU utilization still slowly decreases for larger unit sizes. The differences in performance are caused, in part, by additional context switches and data copies. However, FreeBSD 6.1 also has a lower throughput than Linux 2.6, with a degradation of 16.9% for 1-KB units and 12.9% for 4-KB units. In other words, the gap between MINIX 3 and FreeBSD is roughly equivalent to the gap between FreeBSD 6.1 and Linux 2.6. This shows that the impact of user-level drivers is comparable to other trade-offs in system design.

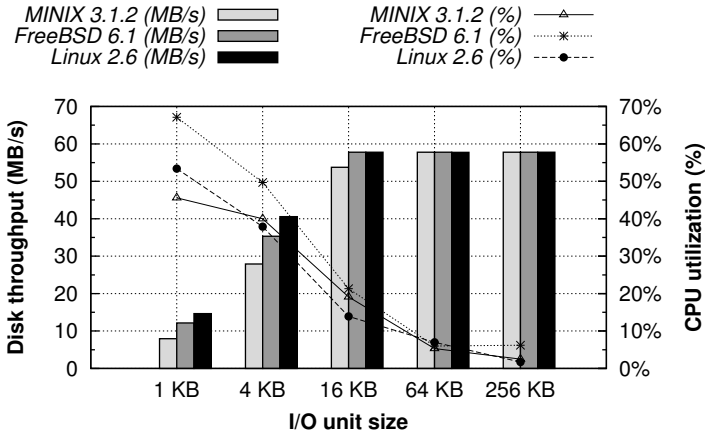


Figure 5.15: Cross-platform comparison of disk throughput and CPU utilization for a 2-GB read with varying I/O units from the raw block device with a sequential access pattern.

We want to emphasize that this test setup represents a worst-case scenario for MINIX 3. As we have seen above, the actual user-perceived overhead depends on the workload and can be significantly lower. Moreover, it has to be noted that the comparison not only highlights the costs of a modular design, but also is influenced by many other factors, including differences in storage-stack strategies, amount of optimization, compiler quality, memory management algorithms, and so on. Linux 2.6 and FreeBSD 6.1 are far more mature than MINIX 3, and the test results have to be interpreted with this difference in mind. We have not attempted to analyze and remove bottlenecks, since our research focuses on dependability. Nevertheless, several possible performance optimizations are mentioned in Sec. 6.4.

5.2.2 Costs of Failure Resilience

In order to determine the overhead introduced by our failure-resilience mechanisms we simulated driver crashes while I/O was in progress, and compared the performance to an uninterrupted I/O transfer. The test script first initiates the I/O transfer, and then repeatedly looks up the driver's process ID and crashes the driver using a SIGKILL signal. The test was run with varying intervals between the simulated crashes. The recovery policy directly restarted the driver without introducing delays. After the I/O completed we verified that no data corruption took place by comparing the checksums of the data transferred. In all cases, we observed full transparent recovery. The results are shown in Figs. 5.16 and 5.17 and discussed below,

Network-stack Performance

We first measured the overhead for the recovery of network-device drivers using the Realtek RTL8139 PCI Ethernet driver. Each test initiated a TCP transfer using the *wget* utility to retrieve a 512-MB file from the local network. We ran multiple tests with the period between the simulated crashes ranging from 1 to 15 seconds. In

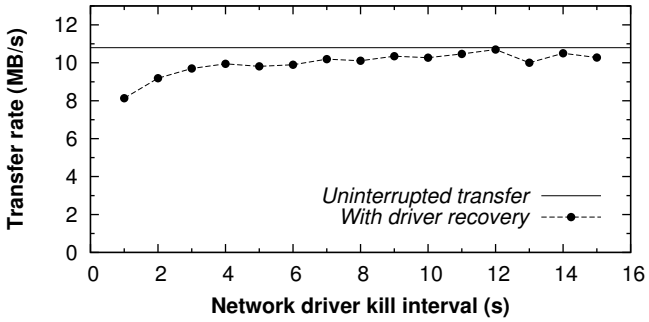


Figure 5.16: Throughput when using *wget* to retrieve a 512-MB file from the local network with and without repeatedly killing the Realtek RTL8139 driver with various time intervals.

all cases, *wget* successfully completed, with the only noticeable difference being a small performance degradation as shown in Fig. 5.16. In order to verify that data integrity was preserved we compared the MD5 checksums of the received data and the original file. No data corruption was found. The uninterrupted transfer time was 47.4 sec with a throughput of 10.8 MB/s. The interrupted transfer times ranged from 47.9 sec to 63.0 sec, with a throughput of 10.7 MB/s and 8.1 MB/s, for simulated crashes every 1 sec and 12 sec respectively. The mean recovery time for the network driver failures was 0.5 sec. The loss in throughput due to network driver failures and the subsequent recovery ranged from 25.0% to just 0.9% in the best case.

Storage-stack Performance

We also measured the overhead of block-device driver recovery by repeatedly sending a SIGKILL signal to the ATWINI hard disk driver while reading a 1-GB file filled with random data using *dd*. The input was immediately redirected to *sha1sum* in order to calculate the SHA-1 checksum. Again, we killed the driver with varying intervals between the simulated crashes. Since the MINIX 3 file system does not guarantee end-to-end integrity and we did not yet have the filter driver discussed in Sec. 4.4, this experiment potentially could have caused data corruption. Nevertheless, we found that the data transfer successfully completed with the same SHA-1 checksum in all cases. The transfer rates are shown in Fig. 5.17. The uninterrupted disk transfer completed in 31.3 sec with a throughput of 32.7 MB/s. The interrupted transfer times ranged from 83.1 sec to 34.7 sec, with a throughput of 12.3 MB/s and 30.5 MB/s, for simulated crashes every 1 sec and 15 sec, respectively. The performance overhead of disk driver recovery ranged from 62.4% to about 6.7% in this test. Because the amount of work to be done to clean up the killed driver and start a new one, and thus the recovery time needed, is roughly the same for each driver, the higher recovery overhead compared to the previous experiment is due to the higher steady-state I/O transfer rate.

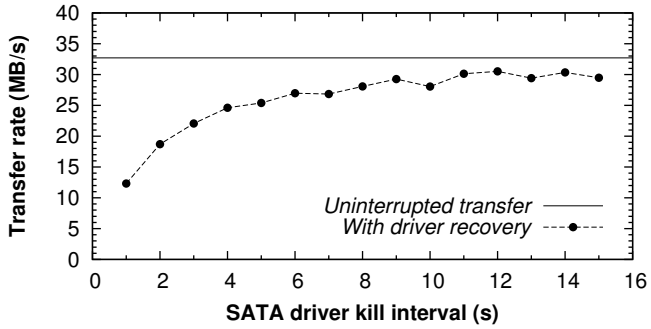


Figure 5.17: Throughput when using *dd* to read a 1-GB file from the hard disk with and without killing the ATWINI driver with various time intervals.

5.3 Source-code Analysis

Finally, we performed a source-code analysis in order to measure the system's evolution over time and estimate the amount of engineering effort that went into it. As a metric we counted the number of lines of code (LoC) using the source-code line counter *sclc.pl* [Appleton, 2003]. The size of the code base was measured in lines of assembly instructions and C code, excluding comments, and blank lines. System library code was excluded from the line counts because it is not specific to the driver and (hopefully) better tested. Headers files were also excluded based on the premise that constant and function definitions do not add to the code complexity. For comparison purposes we performed the analysis for both MINIX 3 and Linux 2.6.

5.3.1 Evolution of MINIX 3

Our analysis of the MINIX 3 code base spans a 4-year period since the official release with 6-month deltas, and includes 9 SVN revisions ranging from r1171 to r5545. We have focused on the core OS consisting of the kernel and the user-level drivers and servers. The sources were obtained from the MINIX 3 source-code repository [Vrije Universiteit Amsterdam, 2009]. Although a small community of MINIX enthusiasts has made valuable contributions, the majority of changes to the core system is attributable to the in-house research and development team. An interesting data point in this respect is that during the time frame analyzed the full-time MINIX 3 team has grown from 1 graduate student and 1 scientific programmer to 5 graduate students and 3 scientific programmers. In addition, there have been various contributions from students doing term projects. The results of our MINIX 3 analysis are shown in Figs. 5.18 and 5.19 and discussed below.

Since the official release in October 2005 the kernel has grown by 101.6% from 3413 LoC to 6881 LoC. We observed a steady increase in the kernel-call handler code, which indicates continuous addition of new features. Two notable events

MINIX 3		Kernel		Drivers		Servers	
Version	*Release date*	*LoC*	*Growth*	*LoC*	*Growth*	*LoC*	*Growth*
r1171	18 Oct 2005	3413	0.0%	22,777	0.0%	27,134	0.0%
r2145	18 Apr 2006	4014	17.6%	26,441	16.1%	28,921	6.6%
r2623	16 Oct 2006	4457	30.6%	28,148	23.6%	31,724	16.9%
r2864	19 Apr 2007	4759	39.4%	28,436	24.8%	36,055	32.9%
r3044	17 Oct 2007	5231	53.3%	30,574	34.2%	37,642	38.7%
r3152	14 Apr 2008	5439	59.4%	34,988	53.6%	38,698	42.6%
r3187	3 Oct 2008	5439	59.4%	34,988	53.6%	38,886	43.3%
r4226	17 Apr 2009	6284	84.1%	34,888	53.2%	41,708	53.7%
r5545	19 Oct 2009	6881	101.6%	35,092	54.1%	45,817	68.9%

Figure 5.18: Source-code analysis of the MINIX 3 kernel and the user-level drivers and servers for a 4-year period since its official release with 6-month deltas.

are a restructuring of the architecture-dependent and the architecture-independent code (r2864) and the introduction of a new virtual memory (VM) subsystem (r4226 and r5545). The latter caused numerous kernel changes and explains the 1442-LoC increase in just 12 months. (As an aside, the memory-management subsystem in the Linux kernel is 34,702 LoC. Although it is obviously more feature-rich than MINIX 3's memory-management subsystem, it also measures more than 5 times the size of the entire MINIX 3 kernel.) A similar picture exists for the user-level drivers and servers. The drivers have grown by 54.1% from 22,777 LoC to 35,092 LoC. While the amount of code per driver sometimes increased, most of the growth is because the number of drivers has grown from 15 to 21 drivers. The average size of a MINIX 3 driver is still only 1671 LoC. The servers have equally grown by 68.9% from 27,134 LoC to 45,817 LoC. Notable changes include the addition of the virtual file system (VFS) (r2864), virtual memory (VM) (r4226), and System V IPC (r5545). These results show that MINIX 3 has grown significantly due to the addition of new features and functionality. Still, as discussed below, the 6881-LoC MINIX 3 kernel seems hardly bloated compared to the 5,319,731-LoC Linux 2.6 kernel. Of course, the MINIX 3 trusted computing base (TCB) also includes several components running at the user level, but even if these are counted, the difference in size is still two orders of magnitude.

Directory	LoC	TCB	Explanation
src/kernel	2095	yes	MINIX 3 kernel
src/kernel/arch	2730	yes	Architecture-dependent code
src/kernel/system	2056	yes	Kernel-call handlers
src/drivers/pci	2905	yes	Generic PCI-bus driver
src/drivers/iommu/amddev	431	yes	AMD-DEV IOMMU driver
src/drivers/libdriver	466	no	Device-independent driver interface
src/drivers/libdriver-asyn	594	no	Asynchronous driver interface
src/drivers/net/dp8390	3176	no	NE2000/ DP8390/ RTL8029 driver
src/drivers/net/fxp	2530	no	Intel PRO/100 driver
src/drivers/net/rtl8139	2469	no	Realtek RTL8139 driver
src/drivers/block/atwini	2100	no	Generic SATA driver
src/drivers/audio/es1371	1295	no	Ensoniq ES1731 audio driver
src/drivers/audio/framework	728	no	Audio-driver framework
src/servers/pm	2605	yes	Process manager
src/servers/rs	2477	yes	Driver manager
src/servers/ds	390	yes	Data store
src/servers/inet	20,133	no	Network server
src/servers/vfs	7109	yes	Virtual file system
src/servers/mfs	4878	no	MINIX file server
src/servers/vm	4635	yes	Virtual memory server

Figure 5.19: Lines of executable code (LoC) for the most important MINIX 3 components. The figure also shows whether the component is part of the trusted computing base (TCB).

Zooming in on the changes required to isolate and recover drivers, we find that most engineering effort went into preparing the core OS to deal with untrusted drivers. Although the driver manager measures only 2477 LoC, it heavily relies on the rest of the TCB to isolate drivers. For example, the PCI-bus driver and IOMMU driver measure another 2905 LoC and 431 LoC, respectively. In addition, the run-time checks performed by the TCB usually required new code to look up the isolation policy and check the driver's authorization. Furthermore, several new kernel calls had to be added to control access to privileged resources, such as peripheral devices and memory. Another important change to the TCB concerns the handling of IPC. In order to protect against blockage due to untrusted drivers, synchronous IPC has been replaced by asynchronous and nonblocking IPC. This is evidenced, for example, by the new 594-LoC *libdriver-asyn* library. The recovery support also affected various parts of the TCB. Most of the driver manager's logic to start dynamically servers and drivers could be reused, but new code was needed for defect detection and execution of recovery scripts. Likewise, the system servers needed little change to support restarting drivers, but gained new code to retry failed I/O operations. Most of this error-handling logic could be centralized in the device I/O routines. Finally, the kernel is not aware of defect detection and recovery.

Most importantly, the fault-tolerance mechanisms required only limited modifications to the drivers themselves. The majority of the changes concerned removing drivers from the kernel. This involved mostly replacing direct function calls with kernel calls, but did not demand structural changes to the way drivers work. The fault isolation is mostly transparent to the working of the driver: unauthorized access attempts will be denied, but the driver is unaware of the underlying mechanisms that constrain it. Several small modifications were required for the run-time memory-protection mechanisms, however. For example, the use of safe copies and IOMMU protection generally affected a few lines of code at all sites relating to memory access. In addition, the changes needed to implement failure resilience were also limited. In general, drivers are only required to reply to heartbeat and shutdown requests from the driver manager. For most drivers this change comprised only 5 LoC in the shared driver library to handle the new request types. Device-specific driver code almost never had to be changed. For a few drivers, however, the code to initialize the hardware had to be modified in order to support reinitialization after a restart. Overall, the changes required are negligible compared to the amount of driver code that potentially can be guarded by our fault-tolerant design.

A final point worth mentioning is that porting device drivers from other OSes to MINIX 3 can be done with relatively little effort. Because MINIX 3 mostly looks and feels like a normal UNIX OS, the hardest part is understanding how the hardware works and separating the driver code from the foreign OS. Once that has been done, it is relatively straightforward to get it to work under MINIX 3, where it can immediately benefit from MINIX 3's protection mechanisms. As a case in point, the Intel PRO/1000 gigabit Ethernet driver was ported from DragonFly BSD to MINIX 3 by a single student in the course of two weeks [Linnenbank, 2009].

5.3.2 Evolution of Linux 2.6

We have also analyzed the Linux 2.6 code base. This analysis spans a 5-year period since its official release with 6-month deltas, and includes 11 versions ranging from 2.6.0 to 2.6.27.11. The kernel images were obtained from the Linux kernel archives [Linux Kernel Organization, Inc., 2009]. Linux 2.6 kernel development is done by a large, distributed community of about 5000 developers representing over 500 corporations [Kroah-Hartman et al., 2009]. Still, a small number of developers and corporations is responsible for the majority of the kernel changes. Looking at individual contributors, the top 10 developers contributed 11.9% of the changes. Looking at corporate support, the developers employed by the top 10 companies, including Red Hat, IBM, and Novell, contributed 43.5% of the changes. The results of our Linux 2.6 analysis are shown in Figs. 5.20 and 5.21 and discussed below.

The Linux kernel shows a sustained linear growth in LoC of about 5.5% every 6 months. In 5 years, the kernel has grown by 65.4% from 3.2 MLoC to over 5.3 MLoC. The */drivers* subsystem is by far the largest subsystem and comprises about half the kernel code base. In 5 years, the */drivers* subsystem has grown by 76.4% and now surpasses 2.7 MLoC. The second largest subsystem is */arch* and comprises 1.1 MLoC. Next, the */fs*, */net*, and */sound* subsystems—which can be regarded as special kinds of OS extensions—together comprise another 1.2 MLoC. The core */kernel* subsystem is relatively small and comprises 70,756 LoC or 1.3% of the entire kernel. Within the */drivers* subsystem, network drivers are by far the largest and fastest-growing driver category. In the past 5 years, network drivers have grown by 77% and now comprise 683,375 LoC. This means that network drivers alone are responsible for 12.9% of the entire kernel code base. Interestingly, these findings match the trends found during an earlier study of Linux 1.0 to Linux 2.3 over 6-year lifespan (96 versions) [Godfrey and Tu, 2000].

Linux 2.6		Entire kernel		Drivers subsystem		
Version	*Release date*	*LoC*	*Growth*	*LoC*	*Growth*	*Ratio*
2.6.0	18 Dec 2003	3,216,751	0.0%	1,564,699	0.0%	0.486
2.6.7	16 Jun 2004	3,427,140	6.5%	1,713,226	9.5%	0.500
2.6.10	24 Dec 2004	3,594,857	11.8%	1,806,297	15.4%	0.502
2.6.12.3	15 Jul 2005	3,757,899	16.8%	1,903,616	21.7%	0.507
2.6.14.6	08 Jan 2006	3,947,373	22.7%	2,001,213	27.9%	0.507
2.6.17.5	15 Jul 2006	4,203,430	30.7%	2,093,988	33.8%	0.498
2.6.19.2	10 Jan 2007	4,403,895	36.9%	2,197,216	40.4%	0.499
2.6.22.1	10 Jul 2007	4,680,941	45.5%	2,341,407	49.6%	0.500
2.6.23.13	09 Jan 2008	4,729,971	47.0%	2,388,677	52.7%	0.505
2.6.26.0	13 Jul 2008	5,177,093	60.9%	2,639,686	68.7%	0.510
2.6.27.11	14 Jan 2009	5,319,731	65.4%	2,760,476	76.4%	0.519

Figure 5.20: Source code analysis of the Linux 2.6 kernel and the device-driver subsystem for a 5-year period since its official release in October 2005 with 6-month deltas.

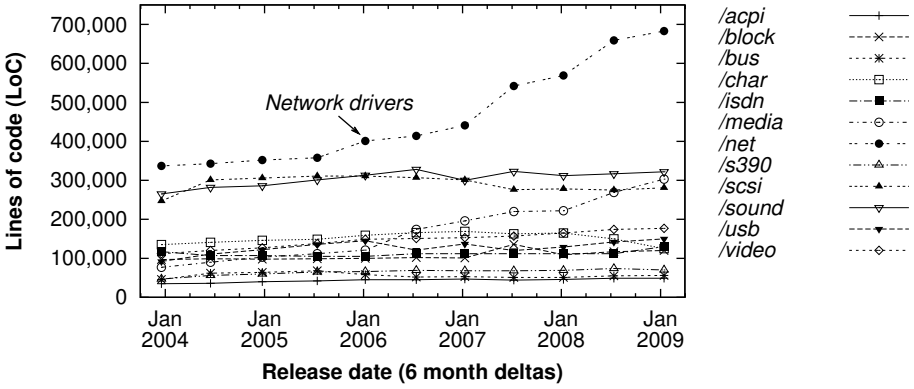


Figure 5.21: Linux 2.6 driver growth in lines of executable code (excluding comments and white space). Network-device drivers are both the largest and fastest-growing driver category.

These results show that both MINIX 3 and Linux 2.6 have experienced a substantial growth of the code base due to the addition of new features and functionality. However, there are several important differences as well. To start with, Linux 2.6 has a larger and more complex code-base in all respects. Looking at just the kernel, we found that the Linux 2.6 kernel is three orders of magnitudes larger than the MINIX 3 kernel. While this is partly because Linux 2.6 is more mature than MINIX 3, the real reason is that Linux 2.6 implements the entire OS in the kernel. This means that all the code runs in a single protection domain with no fault isolation between the components. In contrast, MINIX 3 completely compartmentalizes the OS, implementing only the most crucial mechanisms at the kernel level and all policies in independent user processes. While it is hard to quantify this difference in structure, we strongly believe that it makes the MINIX 3 code base much more manageable.

Chapter 6

Related Work

This chapter surveys related work and compares it to our research. As discussed in Sec. 1.5, we primarily focus on run-time systems for improving OS dependability and distinguish four different approaches. First, in-kernel sandboxing isolates drivers inside the kernel. Second, virtualization techniques can safely run multiple services on a single computer platform. Third, formal methods build on advances in safe languages and verification tools. Fourth, user-level frameworks remove drivers from the kernel and run them in independent user processes. Approaches aimed at prevention of bugs, such as static driver analysis, are outside the scope of this work, since they cannot protect the OS against bugs that are not found.

Below, we assess the working of each class and discuss several representative concrete systems. In particular, Sec. 6.1 introduces in-kernel sandboxing, Sec. 6.2 covers virtualization techniques, Sec. 6.3 presents formal methods, and Sec. 6.4 describes user-level frameworks. For each class we first provide a high-level discussion of the techniques used and then study a concrete system in more detail. Finally, Sec. 6.5 briefly compares MINIX 3 to the other approaches.

6.1 In-kernel Sandboxing

In-kernel sandboxing restricts the driver's execution environment without removing the driver from the kernel by setting up separate protection domains and intercepting unsafe calls from the driver to the core kernel. As discussed below, this can be done using either *hardware-enforced protection* or *software-based isolation*. One particular benefit of in-kernel sandboxing is that it requires only minor modifications to existing drivers and commodity OSes. Furthermore, the use of wrapping and interposition allows catching different kinds of faults than just hardware protection can. For example, BGI provides dynamic type safety for kernel objects [Castro et al., 2009]. On the downside, the approach typically requires substantial run-time support, which adds kernel complexity and increases the burden of maintenance. For

example, Nooks added 22,266 lines of code (LoC) to the Linux kernel [Swift et al., 2005]. In addition, the level of indirection introduced by wrapping and interposition can result in significant performance overheads. Nevertheless, in-kernel sandboxing is an important technique for retrofitting dependability in legacy OSes.

6.1.1 Hardware-enforced Protection

A first approach to in-kernel sandboxing is to run drivers in separate hardware-enforced protection domains. The basic idea is to use the MMU hardware to set up intra-address-space protection for untrusted extensions [Chase et al., 1994]. Such protection can be realized by loading each extension in a separate, less-privileged memory segment that falls in the kernel address space. In this way, the MMU ensures that extensions cannot directly access unauthorized memory regions or corrupt kernel memory other than their own. Since extensions and the core kernel run in different protection domains, direct cross-domain communication is no longer possible. Instead, control and data transfer between protection domains is done using wrappers or capabilities that interpose all communication using a variant of a lightweight remote procedure call (LRPC) [Bershad et al., 1990]. If the access is authorized, the stub routine changes the protection domain, and executes the requested function using the caller's thread.

Intra-address-space protection has been used by a range of *single-address-space operating system* (SASOS) implementations, including Opal [Chase et al., 1994], Nemesis [Leslie et al., 1996], and Mungi [Heiser et al., 1998]. A SASOS runs all OS services in a globally shared virtual address space, which reduces the complexity of pointer management and facilitates data sharing. Although virtual addresses are context independent, the access rights depend on the protection domain in which a thread executes, limiting its access to a specific set of pages at a specific instant. Capabilities are commonly used to enforce the use of well-defined interfaces and protect system objects [Vochteloo et al., 1993]. For example, Mungi isolates drivers using capabilities that enforce fine-grained protection for devices and OS services [Leslie et al., 2004].

In-kernel sandboxing has also been used to retrofit dependability in legacy OSes. For example, Palladium [Chiueh et al., 1999], Kernel Plugins [Ganev et al., 2004], Nooks [Swift et al., 2005], and CuriOS [David et al., 2008] use this technique to isolate untrusted extensions inside the kernel. By changing the module loader, extensions can be loaded in their own protection domain. In addition, all kernel API calls are dynamically linked with wrappers that transparently interpose all outside communication. In this way, access to privileged kernel functionality can be mediated and integrity constraints enforced. For example, Nooks created extensive wrappers for each class of device drivers in order to track the use of kernel resources and perform consistency checks [Swift et al., 2005].

The trust in this approach lies with the correctness of the wrapper-code or capabilities that are responsible for cross-domain transfers. It is crucial that mem-

ory protection is set up before control is transferred to the extension. In addition, the consistency checks should be implemented correctly. If in-kernel extensions can use privileged CPU instructions, complete isolation is sacrificed. For example, Mungi runs extensions with user-mode CPU privileges [Leslie et al., 2004], whereas Nooks' extensions still have kernel-mode CPU privileges [Swift et al., 2005]. Therefore, a buggy or malicious driver can still change the page tables, perform unauthorized I/O, disable interrupts, or halt the CPU. This strategy represents a trade-off between hard isolation guarantees and practicality of the approach.

Case Study: Nooks

Nooks is a reliability subsystem for the Linux kernel that aims to address a large fraction of driver problems with only minor changes to legacy code [Swift et al., 2005]. The design of Nooks is shown in Fig. 6.1. Kernel extensions are isolated using both hardware-enforced protection domains and software-based interposition. Upon loading the Linux module loader gives the extension read-only access to kernel memory other than its own and binds kernel calls to wrappers that interpose on all communication. Cross-domain communication is done using an extension procedure call (XPC) that changes the page table, copies data structures to and from the extension, if need be, and calls the requested function. In this way, the Nooks isolation manager can track the use of kernel resources and perform consistency checks. If the extension causes an MMU hardware exception or invokes a kernel service improperly, Nooks releases all resources used by the extension, unloads the extension, and signals a user-level recovery agent. The recovery agent can run a script to restart and reconfigure the driver after a failure. However, because extensions still run with kernel-mode CPU privileges, nothing can prevent them from reloading the page table and corrupting the rest of the kernel. This is unlikely to happen accidentally though. In addition, Nooks does not provide IOMMU support to protect against invalid direct memory access (DMA).

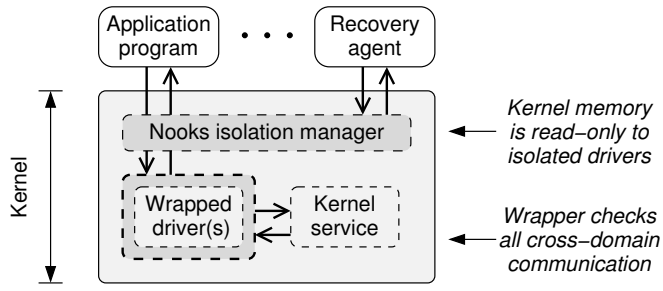


Figure 6.1: Hardware-enforced protection in Nooks. Nooks sets up protection domains for kernel memory and wraps drivers in a layer of protective software. If a failure is detected, the isolation manager calls a user-level recovery agent that can restart the driver.

In a successor project, Nooks' failure detection mechanisms were augmented with *shadow drivers* to make recovery more transparent to applications [Swift et al., 2006]. A shadow driver is not a replica of the real driver, but implements only the services needed for recovery. In passive mode, a shadow driver monitors all communication between the driver and the kernel, and logs configuration messages that change the driver's state. When a driver failure is reported by Nooks, the shadow driver impersonates the failed driver and governs the recovery procedure. It accepts requests until a new driver has been started and, depending on the state collected, immediately replies, drops the request, or queues it for later processing. In addition, the shadow driver restarts the failed driver, restores its state by replaying logged configuration messages, and resubmits pendings requests from the queue. This approach thus supports recovery of stateful drivers. However, shadow drivers cannot guarantee exactly-once behavior for driver requests and must rely on higher-level protocols to maintain data integrity, just like is done in MINIX 3.

The Nooks code base consist of 22,266 LoC, including 14,396 LoC of wrapper code, all of which runs in the kernel domain and thus must be trusted. About two thirds of the 248 wrappers were used to isolate drivers; the rest were meant for a file system (VFAT) and an in-kernel web server (kHTTPd). Out of eight drivers tested, seven drivers required no code changes and only 13 lines had to be changed in the eighth driver. Support for shadow drivers added about 3300 LoC to Nooks and about 2150 LoC for other support infrastructure. Individual shadow drivers were much smaller, for example, 198 LoC for the class of network-device drivers. A software-implemented fault-injection (SWIFI) test injecting 2000 faults showed that Nooks could prevent 99% of 365 Linux crashes, but only half of the nonfatal extension failures, rendering the service unavailable to applications in 210 cases. Another SWIFI test with 2400 fault injections showed that shadow drivers could automatically recover 65% of 390 applications failures. The effectiveness was limited by Nooks' failure detection mechanism, which did not detect, for example, I/O requests that were never completed and errors in the driver's device interaction. Finally, performance measurements showed that the overhead incurred by Nooks ranged from no overhead to 56% in the worst case. Shadow drivers imposed a negligible additional overhead of 1% on average for nine applications tested.

6.1.2 Software-based Isolation

A completely different technique for in-kernel sandboxing is *software-based fault isolation* (SFI) [Wahbe et al., 1993]. SFI modifies the object code of drivers in such a way that it cannot execute unsafe instructions. In particular, the compiler or binary rewriter inserts run-time *software guards* before every instruction that jumps or writes to an address that cannot be statically verified to fall within the driver's (logical) protection domain. The software guard verifies that the driver has authorization for the computed address before writing to it. Optionally, load instructions can also be guarded in order to prevent malicious code from reading unauthorized memory.

In addition, cross-domain calls are mediated by arbitration code in order to verify that the call performed by the other domain is safe. This provides protection against memory corruption by buggy code.

Traditionally, SFI was used to enforce memory protection, but several advances made it possible to specify richer protection policies. For example, *control flow integrity* (CFI) instruments binaries in order to ensure that the code path executed adheres to a static policy that comprises a control-flow graph [Abadi et al., 2005]. CFI is enforced by inserting known labels at each branch destination and preceding branches with run-time guards that verify that the destination contains the expected label. These checks ensure that calls enter functions only at the beginning and returns transfer control to a point after a valid call site, and thereby prevent code-injection and return-to-libc attacks. Next, XFI builds on CFI to provide a generalized form of SFI that can enforce memory access constraints, restrict the use of interfaces, prevent execution of privileged instructions, and provide system state integrity guarantees [Erlingsson et al., 2006]. Finally, BGI extends these checks with dynamic type safety for kernel objects [Castro et al., 2009].

The use of software rather than hardware protection represents a trade-off with respect to execution time overhead. Although software protection provides faster cross-domain communication than hardware protection, the binary instrumentation incurs an overhead proportional to the code size. Because SFI requires every unsafe instruction to be preceded by a run-time guard, the execution-time overhead of memory-intensive applications can be nearly 200% [Seltzer et al., 1996]. XFI reduces this performance penalty by checking memory ranges with a fast path for common accesses and hoisting software guards out of frequently executed code paths such as loops. Benchmarks show that XFI incurs an overhead ranging from 1% to 94% [Erlingsson et al., 2006]. Recently, Native Client (NaCl) implemented SFI-like protection for browser plug-ins using hardware segments, such that load and store instructions do not have to be preceded by a software guard [Yee et al., 2009]. Normal hardware page protection is still required between processes. This model allows for less fine-grained policies, but the performance overhead was limited to 12% in the worst case and less than 5% on average.

The protection provided by SFI depends either on the correctness of the compiler or binary patching tool or on an independent verifier. The absence of hardware protection domains, means that all trust lies with the software guards. The simplest approach is to assume that the tools used generate sufficient software guards for all unsafe instructions, just like ordinary compilers are trusted to work correctly [Thompson, 1984]. Alternatively, an independent verification tool can be used to check all code paths for unprotected unsafe accesses. Because of all the low-level complexity involved certain corner cases may be missed though. For example, Native Client's validator logic was shown to contain flaws that can lead to memory corruption in the run-time system [Hawkes, 2009]. In addition, the support infrastructure, such as the interposition libraries that enforce integrity constraints, introduces additional kernel-level complexity that must be trusted.

Case Study: BGI

BGI provides fast byte-granular protection for existing Windows drivers with low overhead and no modifications to the source code [Castro et al., 2009]. The working of BGI is depicted in Fig. 6.2. A BGI compiler takes an unmodified Windows driver and replaces all direct kernel API calls with calls to a trusted interposition library. In addition, unsafe accesses, such as direct memory writes and indirect jumps, are instrumented with software guards. In-line assembly is disallowed by the compiler to prevent the extension from executing privileged CPU instructions. The software guards and wrapper functions enforce a fine-grained access control model through run-time checking, granting, and revoking of memory write and ownership rights, kernel call rights, and type rights. In particular, for each byte of virtual memory BGI maintains a list of access rights per untrusted domain. Memory write and ownership rights prevent corruption of kernel memory, kernel call rights are used to enforce control flow integrity, and type rights are used to enforce *dynamic type safety* for kernel objects. The latter dynamically changes the set of operations allowed on kernel objects in order to catch (temporal) errors when using complex kernel interfaces. Memory reads by the driver and direct memory access (DMA) from the device layer are unprotected in order to limit BGI's overhead.

BGI also provides limited recovery support. If the interposition library detects an extension failure, BGI unloads all extensions in the domain, releases all the resources held, and restarts the extensions. Because BGI relies on the Windows plug-and-play (PnP) manager, only PnP drivers can currently be recovered. Moreover, recovery transparent to applications is not supported. When an extension call is attempted during the recovery procedure, the interposition library returns an error code. Re-

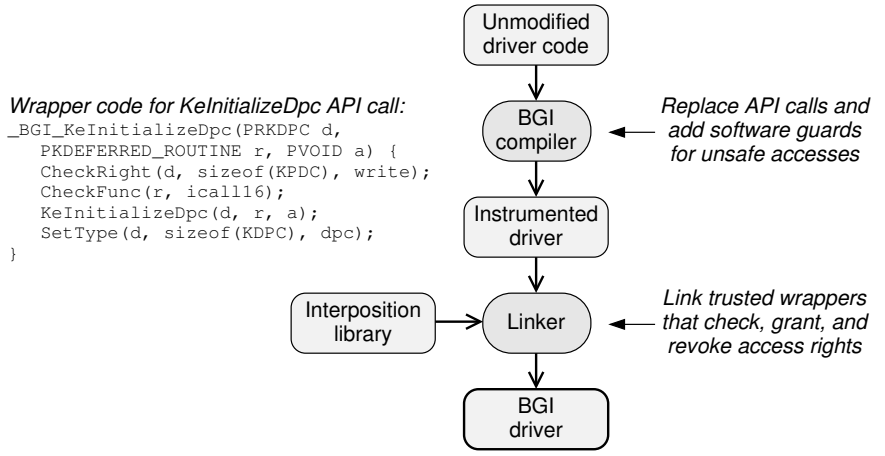


Figure 6.2: Software-based fault isolation in BGI. A BGI compiler replaces direct kernel API calls with wrapper calls and adds software guards before direct memory accesses and indirect calls.

quests from the PnP manager to remove the device are automatically acknowledged, however. The last unload call causes all resources associated with the domain to be released. There is no need for a separate object tracker as in Nooks because BGI already keeps track of all ownership rights for each domain.

BGI was applied to 16 Windows drivers that comprise over 400,000 LoC and use 350 different API functions from WDM, IFS, NDIS, and KMDF [Microsoft Corp., 2009]. BGI required wrappers for 262 kernel API calls and 88 extension callbacks, some of which could be automatically generated from source annotations. The size of the resulting interposition library is 16,700 LoC. In order to test BGI's effectiveness two drivers were subjected to fault-injection experiments. Buggy BGI drivers were produced by injecting 5 random bugs into the driver source before calling the BGI compiler. In total, 675 buggy driver versions were tested by running them in an isolated BGI domain. BGI was able to prevent 47%–60% of all Windows hangs and 98%–100% of all Windows crashes. Driver hangs were caused by infinite loops and resource leaks, which are not checked for by the BGI wrappers. Because BGI can only recover PnP drivers, the recovery could only be tested with one driver. During 50 test runs the recovery was successful in 19 out of 21 failures detected by BGI. Finally, performance measurements showed that the overhead incurred by BGI is limited: for TCP and UDP network benchmarks the average CPU overhead was 8% with an average throughput degradation of 2%.

6.2 Virtualization Techniques

Virtualization is used to run multiple services on a single system [Smith and Nair, 2005; Rosenblum and Garfinkel, 2005]. The basic idea is to create a virtual execution environment, known as a *virtual machine* (VM), by replicating the computer hardware, or a variant thereof, in software. Below we discuss both flavors: *full virtualization* and *paravirtualization*. The VM runs under the control of a small privileged kernel, commonly referred to as a *hypervisor* or *virtual machine monitor* (VMM). Hypervisors share architectural commonalities with microkernels, but export richer primitives and more closely resemble the hardware [Hand et al., 2005; Heiser et al., 2006]. Virtualization provides strong guarantees by running untrusted code in user-mode in a private address space monitored by the VMM. Unmodified driver reuse is possible by running the driver in its original OS in a VM. However, running multiple OSes in different VMs complicates resource and configuration management [Ganev et al., 2004, LeVasseur, pers. comm., 2006].

6.2.1 Full Virtualization

Full virtualization provides a faithful software replica of the underlying hardware and allows the guest OS to execute unmodified. Although the guest OS runs in a VM, it appears to the OS as though it runs on its own dedicated hardware. This technique

was originally developed by IBM in the 1960s to provide concurrent access to mainframe computers, with VM/370 as a notable example [Seawright and MacKinnon, 1979; Creasy, 1981]. Today, virtualization is widely used for consolidating servers, improving manageability, sandboxing untrusted code, and running different OSes on a single computer. Two common VM designs include running the virtual machine monitor (VMM) directly on the hardware, as in VMware ESX [Waldspurger, 2002] and Hyper-V [Kelbley et al., 2009], or using a *hosted architecture* where the VMM co-exists with a preinstalled OS, as in VMware Workstation [Sugerman et al., 2001], QEMU [Bellard, 2005], KVM [Kivity et al., 2007], and VirtualBox [Möller, 2008].

Unfortunately, full virtualization can incur a significant performance and resource penalty. On the one hand, the VMM must provide each VM with virtual privileged CPU instructions, memory spaces, and device I/O. This introduces overhead since the associated data structures must be updated for each access and more state has to be stored and loaded on each context switch. On the other hand, not all instructions of the x86 (IA-32) architecture are *classically virtualizable*, meaning that it is not possible to run the VM on the real hardware and apply a *trap-and-emulate* approach for privileged instructions [Popek and Goldberg, 1974]. This problem can be addressed either by running the guest OS in an emulator that dynamically translates nonvirtualizable instructions or by relying on x86-architecture virtualization extensions provided by AMD-V and Intel VT-x, but both strategies come at a price. For example, a web server benchmark showed a slowdown of 33%–62% compared to native execution [Adams and Agesen, 2006]. If backward compatibility is not an issue, modifying the guest OS to let it work together with the VMM is an efficient alternative, as discussed below.

Although virtualization provides strong inter-VM isolation guarantees, the protection is too coarse-grained to deal with intra-VM failures due to buggy drivers. If the guest OS has a monolithic structure, faults can still propagate and crash the

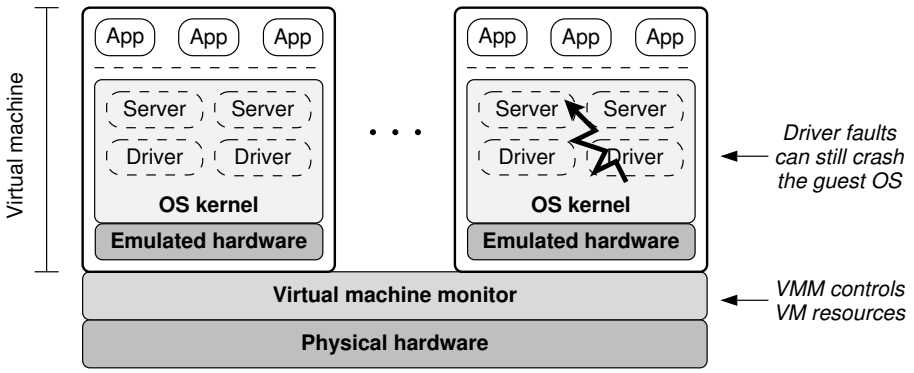


Figure 6.3: Full virtualization cannot isolate individual drivers. Although each virtual machine runs in isolation, a single driver fault in a guest OS can still take down an entire virtual machine.

entire VM, as illustrated in Fig. 6.3. Because a VM failure does not require a full machine reboot, a faster OS restart may result, but all running application programs and unsaved user data on that VM are still lost. If the virtualization platform supports snapshots, the state and data of the VM can be saved periodically and restored at a later point [Ta-Shma et al., 2008]. Such checkpointing speeds up recovery after a VM failure, since it is often possible to revert to a point in time just before the failure occurred. However, the effectiveness of recovery is inherently limited because all information between the last checkpoint and the failure is lost. Moreover, checkpoints may still contain the corruption that eventually leads to failure.

What is needed instead is running the core OS and untrusted extensions in different protection domains. Because full virtualization does not support such fine-grained isolation, we do not further discuss this approach.

6.2.2 Paravirtualization

Paravirtualization exposes an interface that is similar to the underlying hardware, but includes strategic modifications in order to increase performance or provide a richer programming interface. For example, Denali uses paravirtualization to prevent wasting CPU resources and provide a simplified view of I/O devices [Whitaker et al., 2002]. The guest OS can request VMM services via a *hypercall* that loads the parameters in registers and traps to the VMM. By adapting the guest OS to the underlying VMM all the code can be executed without run-time translation and near-native performance can be achieved. The only overhead is the use of hypercalls instead of direct hardware access. For example, a web server benchmark on a paravirtualized Linux OS running on the Xen VMM performed within 1% of native Linux [Barham et al., 2003].

An important extension that can be provided by paravirtualization is controlled VM-to-VM communication [Hohmuth et al., 2004]. This overcomes limitations of full virtualization because driver faults can be isolated from the core OS by running each untrusted driver in its original OS in a dedicated VM. External clients can interface with the driver via a translation module that runs in the driver OS and maps requests onto normal driver calls. Although the lowest parts of the driver OS need to be paravirtualized, device drivers can still use the normal kernel APIs and often do not have to be modified. In addition to driver isolation, simple recovery support may be provided by restarting the VM of a failed driver. This approach was recently demonstrated by various systems. For example, L^4Linux provides driver reuse via a paravirtualized Linux OS running on top of the L4 microkernel [LeVasseur et al., 2004]. Next, reuse of Linux drivers via a unified device API is done with the Xen VMM [Fraser et al., 2004]. Finally, VEXE'DD sets up virtual execution environments providing binary compatibility for Windows drivers using a modified version of Microsoft Virtual PC [Erlingsson et al., 2005].

There are several downsides to paravirtualization as well. To start with, the source code has to be available in order to paravirtualize the guest OS, which makes

the approach impractical for proprietary, closed-source OSes. Furthermore, the approach is relatively complex and requires an intimate understanding of the guest OS, which can make all kinds of assumptions about its execution environment. A related issue is the programming effort needed, although techniques to automate part of the work exist [LeVasseur et al., 2008]. Next, the approach does not work for drivers that cannot be replaced dynamically, including the interrupt controller, real-time clock, keyboard, and mouse in Linux. In these cases, the VMM must provide full hardware emulation. Finally, in order to achieve the same fine-grained compartmentalization as in multiserver OSes like MINIX 3, each server and driver should run in a dedicated VM. Such a design would have to face the same challenges as a multiserver OS, but the problems of resource management and configuration management become more pronounced when using VMs and legacy OSes rather than processes [LeVasseur, pers. comm., 2006].

The trust in this approach lies with the implementation of the hypervisor or VMM that sets up the hardware-enforced protection domains, translates or emulates privileged instructions, and manages the resources of each VM. Virtualization platforms have traditionally demonstrated to be capable of providing proper inter-VM protection. However, paravirtualization-based driver isolation gives up strict VM separation by allowing VM-to-VM communication and is complicated because VMMs typically do not support IPC, data copying, and the like [Hohmuth et al., 2004]. However, the case study below demonstrates that safe inter-VM communication can realized by building on microkernel technologies.

Case Study: L^{4}Linux

L^{4}Linux is a paravirtualized version of Linux running on top of the L4 microkernel [Härtig et al., 1997]. In order to keep the porting effort low only minimal changes were made to the architecture-dependent parts. Physical memory is mapped one-to-one into the L^{4}Linux server, which acts as a pager for applications running on Linux. All communication to L^{4}Linux induced by system calls, page faults, and interrupts is done using the native L4 IPC primitives. Application-to-L^{4}Linux system calls are mapped onto IPC using a modified version of the standard C library or a *trampoline* if binary compatibility is needed. This setup allows (re)using Linux functionality next to real-time components, as in DROPS [Härtig et al., 1998], or security extensions, as in PERSEUS [Pfitzmann and Stüble, 2001].

L^{4}Linux supports unmodified reuse of device drivers [LeVasseur et al., 2004]. Drivers are isolated by running them in a VM with their original OS to preserve semantics and prevent incompatibilities. A client OS communicates with reused drivers via a kernel module that implements a *virtual device* abstraction for each device class. Client-server communication with the device-driver OS (DD/OS) is implemented using L4 IPC and memory sharing. A translation module in the DD/OS catches the IPC and forwards the request to the actual driver. For example, client-side disk operations are converted into server-side block request to the Linux block

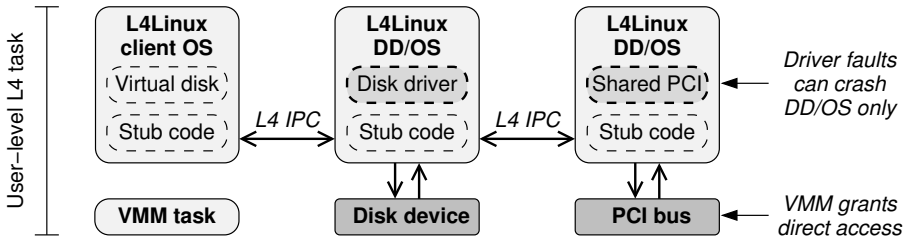


Figure 6.4: Paravirtualization supports safe and unmodified driver reuse by running them in their original, paravirtualized OS inside a VM. The client OS and device-driver OS (DD/OS) communicate via VM-to-VM communication primitives provided by the L4 microkernel.

layer. A pass-through mechanism controlled by the VMM—which is implemented as a separate user-level L4 task—grants drivers direct access to only the device(s) they control. Driver reuse can be recursively applied to hardware that is inherently shared and cannot be partitioned. For example, stub code in each DD/OS forwards PCI-related requests to a PCI DD/OS that centrally controls PCI-bus access. This setup is illustrated in Fig. 6.4. The granularity of isolation depends on the number of drivers colocated per DD/OS. Failed drivers potentially can be recovered via a VM reboot, but this was not implemented.

The original paravirtualization effort required about 6500 LoC to adapt Linux' architecture-dependent code. The additional effort required for driver reuse was about the same, with 1024 LoC for common functionality used by both the client and server, 2184 LoC for virtual device drivers added to the client, and 3304 LoC for translation modules added to the server. Application-level benchmarks showed that L^{4}Linux performs within 5%–10% of vanilla Linux—up to 7 times faster than MkLinux running on top of Mach [Barbou des Places et al., 1996]. Performance measurements showed that the cost of driver reuse is also reasonable. For example, running 3 DD/OS instances has a 6-MB memory footprint and consumes just 0.36% of the CPU in a steady state. Furthermore, an encapsulated network driver displayed a throughput degradation of just 3%–8%, although the CPU utilization was relatively large at 1.6×–2.2× higher than normal. We are not aware of fault-injection testing or another form of empirical dependability assessment in the context of L^{4}Linux.

6.3 Formal Methods

Formal methods isolate extensions by exploiting advances in programming languages and verification tools. Many of the ideas have been around for decades, but were recently revisited to address dependability problems. Below we focus on *language-based protection* and *driver synthesis*. The use of safe, high-level languages prevents many driver problems caused by the inherent complexity of low-level languages, such as C and C++. A problem with this approach is that it often

breaks with current programming models and throws away all legacy. Virtualization presents a possible path to adoption, however, by running a trusted OS for important applications next to an untrusted OS for legacy support. An alternative strategy is to support the programmer by automatically generating safe driver code. Arriving at a correct specification that can be used to synthesize the driver is a hard problem though, but automatic reverse engineering may be of help [Chipounov and Candea, 2010]. In some cases, program analysis and verification tools can be used to find driver bugs and prove code correct [e.g. Ball et al., 2006; Zamfir and Candea, 2010]. Such tools are orthogonal to the work presented here, however, since they cannot protect the OS against driver bugs that are not found. This thesis primarily focuses on systems that ensure run-time safety. Nevertheless, even though it may not yet be possible to deploy formal methods on a large scale, it is important to assess novel dependability techniques for next-generation OSes.

6.3.1 Language-based Protection

Currently, OSes are usually written in low-level languages like C or C++, which make heavy use of memory pointers and low-level bit manipulations that are hard to verify and a rich source of bugs. Therefore, various projects investigated the use of safe languages that eliminate memory corruption due to invalid pointers and buffer overruns through a combination of static, compile-time and dynamic, run-time checks. In particular, *type safety* makes it impossible to construct a pointer to an arbitrary memory location or to perform illegal operations. In addition, *memory safety* ensures the validity of references by preventing null-pointer references and references to deallocated memory. However, the run-time checks and garbage collection incur an overhead that is approximately proportional to the amount of code executed. On the other hand, protection domain crossings are cheaper than with hardware-based protection. We have seen a similar trade-off for software-based fault isolation above. Various systems have experimented with safe languages, including Modula-3 in SPIN [Bershad et al., 1995], Java in JavaOS [Mitchell, 1996], Cyclone in OKE [Bos and Samwel, 2002], Deputy in SafeDrive [Zhou et al., 2006], and Sing# in Singularity [Hunt and Larus, 2007].

In order to guarantee safety all code that cannot be statically verified must contain sufficient checks inserted by the compiler and run under the control of the run-time system. Unsafe code that does not meet this requirement is considered trusted. This includes, for example, the run-time system itself as well as low-level C and assembly code for bootstrapping the system and performing I/O. This safety requirement precludes the use of existing device drivers and application programs writting in an unsafe language. In addition, porting of legacy code is complicated because all unsafe code must be rewritten in a safe language. This is an important drawback, but a possible solution is to combine language-based protection with hardware protection. For example, Singularity explored augmenting pure software isolation with hardware protection, although this model is primarily used as a defense against

potential failures in software isolation mechanisms [Aiken et al., 2006]. This combination of language-based protection and hardware protection resembles the structure of a microkernel-based multiserver system, such as MINIX 3.

A related approach that often relies on the use of (functional) languages is formal verification. While formal verification potentially gives high assurance, the high computational complexity associated with formal verification limits its application to individual subsystems. Although, some projects explicitly aim to construct a formally verified general-purpose OS [Shapiro et al., 2004], to date, checking an entire OS for correctness has been infeasible. The current state-of-the-art proved the functional correctness of the seL4 microkernel by deriving its implementation from a formally checked prototype written in Haskell [Klein et al., 2009]. With a formal model of the kernel in place, it becomes possible to reason about the trustworthiness of (modular) systems built on top of it [Heiser, 2005]. We are not aware of device drivers for which a formal correctness proof has been established, however.

Language-based protection relies on the correctness of the compiler and run-time system. For example, the compiler must be trusted to generate safe code and structures for run-time checking and garbage collection. A buggy compiler that generates unsafe low-level code may corrupt the OS even if the high-level implementation is correct. For example, dependability testing revealed a bug in the Modula-3 compiler used by SPIN [Small and Seltzer, 1996] and the Bartrok compiler used by Singularity is 'likely to contain bugs' [Hunt and Larus, 2007]. Therefore, it is important to perform verification at a level close to the native code, for example, using *typed assembly language* [Glew and Morrisett, 1999]. Bugs in the run-time system, including the memory allocator and garbage collector, also may break the isolation, but practical safety and correctness proofs were recently demonstrated [e.g. Hawblitzel and Petrank, 2009]. Finally, hardware faults corrupting a pointer value or computation may also break software isolation [Govindavajhala and Appel, 2003]. Although this potentially poses a threat with hardware-based isolation as well, the problem is less likely to occur since only a few components run with a privileged CPU mode and less code needs to be trusted at run time.

Case Study: Singularity

The Singularity project builds on advances in programming languages, run-time systems, and program analysis tools to explore new strategies toward dependable systems [Hunt and Larus, 2007]. The design of Singularity is centered around three architectural principles. First, all code executes in the context of a *software-isolated process* (SIP), which is a closed object space consisting of a set of memory pages and one or more threads of execution [Hunt et al., 2007]. Each SIP has its own run-time system and garbage collector. Since SIPs rely on the use of static verification and run-time checks, multiple SIPs can safely run inside a single address space. Second, although SIPs cannot directly share data, they can communicate by passing messages over *channels* [Fähndrich et al., 2006]. The Sing# language in-

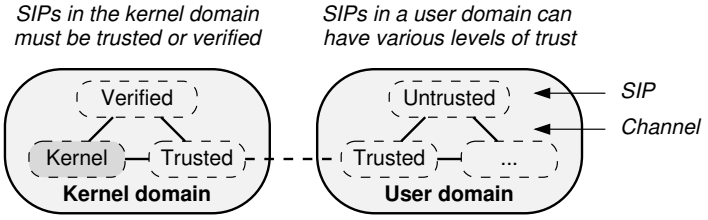


Figure 6.5: Combined hardware and software isolation in Singularity. Software isolated processes (SIPs) running in the kernel domain are either trusted or verified. User domains can contain code with various levels of trust. Channels copy data between protection domains.

corporates support for channel contracts that describe valid messages and message sequences leading to other states in the state machine. Contracts are amenable to static verification, which prevents execution of channel operations in a wrong protocol state. Exchange of large data structures is done by passing a reference and transferring exclusive ownership via an *exchange heap*. Third, all programs are associated with a manifest that is consulted at install time to verify that all resources needed by a driver are present and that no configuration conflicts occur [Spear et al., 2006]. Driver manifests are generated by the compiler on the basis of programmer declarations. Like the MINIX 3 isolation policies, these declarations include the IPC and device-I/O resources needed by the driver.

Although Singularity primarily relies on software isolation, the use of hardware-protected *domains* has also been explored [Aiken et al., 2006]. Each domain has a private MMU-protected address space and, optionally, a lower CPU privilege level. A domain can contain one or more SIPs and has its own exchange heap. Internal communication is still done using reference passing, but data copying is used for external communication between domains. Singularity can selectively combine software and hardware isolation depending on the level of trust needed. Code running in the kernel domain must be either trusted or verified. This is illustrated in Fig. 6.5. If the kernel, system services, and applications are all run in separate hardware protection domains, the configuration mimics a microkernel-based multiserver system like MINIX 3. One of the motivating factors is to defend against failures in software isolation mechanisms. As an aside, regardless of the use of software isolation or hardware isolation, IOMMU hardware support is needed in order to protect against memory corruption by incorrectly programmed devices.

The Singularity kernel is approximately 165,000 LoC. While 90% of the kernel consists of safe Sing# code, a significant portion of about 16,500 LoC is unsafe code, such as C++ and assembly language. This code comprises performance-critical functionality and low-level code, including the garbage collector, memory manager, I/O subsystem, kernel debugger, and initialization code. A comparison of software isolation and hardware isolation showed that the latter incurs a performance cost of up to 25%–33%. However, the comparison seems unfavorable because context

switches forced unnecessary translation-lookaside-buffer (TLB) flushes, each IPC call required two message copies, and software-isolation overhead was still present. A meaningful comparison is only possible if known optimizations to reduce the costs of hardware protection are applied. Finally, we are not aware of an empirical dependability assessment of Singularity.

6.3.2 Driver Synthesis

Another method to improve dependability is synthesizing drivers based on formal specifications of the device and OS interface. One approach is to support programmers with *domain-specific languages* that provide high-level abstractions for expressing error-prone low-level driver code, such as register access and bit manipulation. For example, several projects have attempted to formalize device interactions using an *interface definition language* (IDL), including Devil [Mérillon et al., 2000], NDL [Conway and Edwards, 2004], and HAIL [Sun et al., 2005]. Because the IDL captures the layout of device registers and allows specifying access constraints, consistent use of interfaces can be enforced and safe code for device access can be generated automatically. Likewise, OS protocols have been captured using a state-machine-based language in Dingo [Ryzhyk et al., 2009a]. The protocol specification can be mapped onto a driver implementation and used at run time to check for protocol violations. However, while domain-specific languages help improving driver quality by supporting programmers, they address only part of the problem and cannot provide isolation guarantees for untrusted drivers.

Complete *driver synthesis* further reduces the impact of human error on driver reliability. If drivers are automatically generated from formal device specifications provided by the hardware manufacturer, programming mistakes can no longer occur and driver correctness can be guaranteed by construction. In addition, because driver development normally is complex and time-consuming, driver synthesis can potentially cut down on development costs. Termite demonstrated a concrete tool chain to perform driver synthesis based on formal specifications of the OS and device [Ryzhyk et al., 2009b]. However, a key challenge for driver synthesis is the correct definition of the formal specifications used the generate the code. Although OS vendors and hardware manufacturers are in a good position to provide an accurate and complete model of operation, current practices still require substantial manual effort from the driver developer, which makes wide-scale adoption of driver synthesis unlikely in the short term. However, since synthesized drivers can coexist with normal drivers, they can be adopted incrementally. Finally, since synthesized drivers can safely run inside the kernel without run-time checks, their performance is on par with manually developed drivers.

The working and correctness driver synthesis depends on two aspects. First, the formal specifications used to generate the code must be correct. Because OS specifications are shared by many drivers, they will be extensively tested over time. Interface changes or bug fixes need only be reflected in the OS specification in order

to regenerate all drivers for that OS. The correctness of device specifications can potentially be established by checking correspondence with the actual device behavior, as defined in its register-transfer-level description [Ryzhyk et al., 2009b]. Second, the tool chain used to synthesize drivers must be trusted. For example, the code generator must correctly transform the driver's state machine into runnable C code.

Case Study: Termite Termite provides automatic synthesis of drivers based on formal specifications of device, device-class, and OS interfaces [Ryzhyk et al., 2009b]. Each of the specifications deals with a separate concern. First, the device specification contains hardware registers and interrupt lines and describes how the device reacts to software commands. Second, the device-class specification encompasses events that characterize a class of similar devices, such as Ethernet controllers. Events for the class of Ethernet controllers include, for example, packet transmission and link status change. In practice, many I/O devices are not 100% compliant, which means that extended versions of the device-class specification must be created to support features that are unique to a device. Third, the OS specification defines a state machine with the requests that must be handled by the driver, the ordering in which these requests can occur, and how the driver should respond. By separating the OS specification, a single device specification can be used to synthesize drivers for any supported OS. The device specification and OS specification sometimes refer to functionality in the device-class specification. This is illustrated in Fig. 6.6.

The goal of Termite driver synthesis is to generate driver code that complies with all interface specifications. The actual driver-synthesis procedure consists of three phases. Phase one aggregates all specifications into a single, aggregate state machine that contains all possible states and transitions. The state machine adheres to the specified ordering of operations, but certain execution traces may still contain deadlocks or infinite loops. Therefore, phase two computes a driver strategy that reaches a target state in a finite number of steps. This results in a driver state machine that guarantees forward progress. Finally, in phase three, the driver state machine is used to generate the actual C code. The resulting driver implementa-

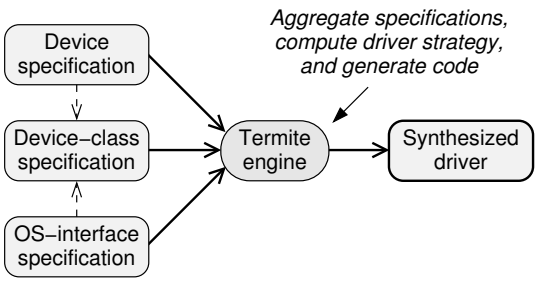


Figure 6.6: Formal method to synthesize drivers in Termite. Device, device-class, and OS specifications are aggregated, a driver strategy machine is computed, and driver code is generated.

tion consists of a C structure that describes the driver's state and entry points that handle incoming requests. The generated driver assumes a single thread of execution, whereas many OSes provide a multithreaded execution environment. This problem is solved by running the driver in a thin wrapper that serializes concurrent requests [Ryzhyk et al., 2009a].

Termite was used to synthesize two drivers for Linux, namely a low-bandwidth SD-host-controller driver and a USB-to-Ethernet driver. The native drivers consisted of 1174 LoC and 1200 LoC, respectively. Because the corresponding devices were based on proprietary designs, the device specifications had to be manually created with help of the data sheets and the original drivers. The resulting device specifications were 653 LoC and 463 LoC and the OS specifications were 641 LoC and 309 LoC, respectively. Because it is computationally infeasible to include values of a variable during the driver synthesis, the Termite engine manipulates variables symbolically. However, one of the two drivers was not compatible with this approach, and required manually written support functionality comprising 110 LoC. The synthesized drivers measured 4667 LoC and 2620 LoC, respectively. Performance measurements showed that both drivers performed virtually identical to native drivers, even under heavy workloads.

6.4 User-level Frameworks

Finally, user-level frameworks that encapsulate drivers in UNIX processes can be used to isolate extensions. Two key properties leading to strong isolation are address-space separation enforced by the (IO)MMU and user-mode privileges enforced by the CPU. User-level framework are typically deployed in microkernel and multiserver environments, but they are also slowly being adopted by commodity OSes. Below we focus on two variants, namely full *process encapsulation* and *split-driver architectures*. MINIX 3 fits in the former variant. Several projects have shown that the performance of user-level frameworks can be competitive with only 5%–10% overhead compared to native execution [Härtig et al., 1997; Gefflaut et al., 2000; Leslie et al., 2005a]. Another advantage of user-level frameworks is that normal programming practices can be used. Although user-level frameworks are not fully compatible with existing drivers, drivers may be ported relatively easily and future hardware requires new drivers in any event. Developers do not need to learn new languages or adopt new programming styles. In fact, as discussed in Sec. 1.4, the development cycle is shortened because drivers can be written and tested like ordinary applications. Therefore, user-level frameworks represent a pragmatic approach to dependability for both current and future OSes.

6.4.1 Process Encapsulation

User-level frameworks that apply full process encapsulation have a long and rich history in microkernel and multiserver systems, as shown in Fig. 6.7. At an early

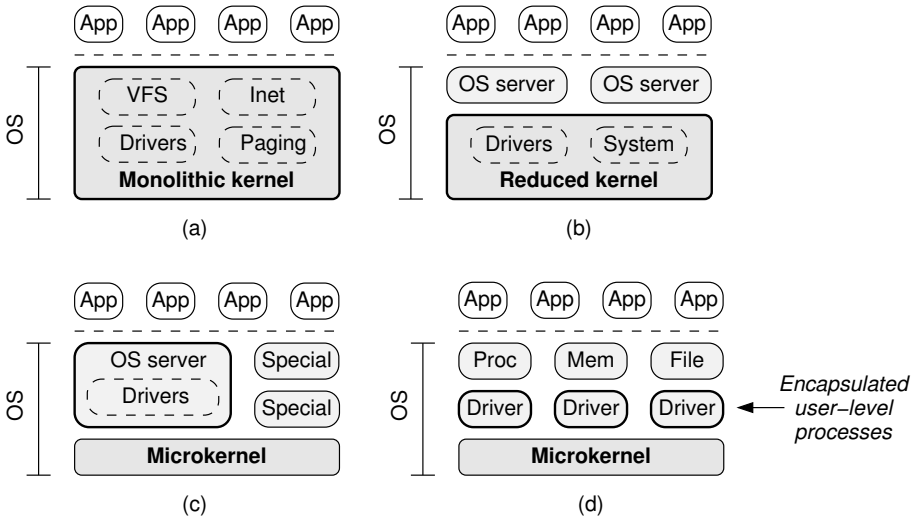


Figure 6.7: Granularity of process encapsulation over the years: (a) monolithic design, 1970s, e.g. BSD UNIX; (b) reduced-kernel design, 1980s, e.g. Mach; (c) single-server design, 1990s, e.g. L^{4}Linux; and (d) multiserver design, 2000s, e.g. SawMill Linux and MINIX 3.

stage, the OS personality was run as a single process on top of a reduced kernel that still contained drivers, for example, UNIX on the Mach microkernel [Accetta et al., 1986; Golub et al., 1990]. More recently, single-server OSes that encapsulate drivers have been built on top of the L4 microkernel, including L^{4}Linux [Härtig et al., 1997] and Wombat [Leslie et al., 2005b]. However, since drivers still run closely integrated with the OS, these designs cannot improve OS dependability in the face of driver bugs. A driver crash would still take down the entire OS server and all applications depending on it. Other designs provide more fine-grained isolation by encapsulating individual device drivers in user processes or even compartmentalizing the entire OS. Examples of such modular systems include, Chorus/MiX [Armand, 1991], QNX [Hildebrand, 1992], Mach-US [Stevenson and Julin, 1995], SawMill Linux [Gefflaut et al., 2000] and GNU Hurd [Le Mignot, 2005]. Despite all these effort to encapsulate drivers, these system do not specifically aim to improve OS dependability in the face of buggy drivers and have not been subjected to fault-injection tests like MINIX 3.

Recently, process encapsulation of drivers has also been proposed for commodity OSes like Windows and Linux, but, to date, has not yet been fully adopted. An early project showed how to construct user-level drivers on Windows using an in-kernel proxy that routes requests between the application and the driver [Hunt, 1997]. This design suffers from significant performance degradation due to the level of indirection introduced by the proxy: each driver request must cross the kernel/user-level boundary at least four times. Nevertheless, Windows Driver Foundation (WDF) uses

a similar approach for the User-Mode Driver Framework (UMDF) [Microsoft Corp., 2007]. UMDF currently provides support for user-level software drivers, but cannot manage drivers that control hardware devices or that rely on kernel resources. The Jungo WinDriver framework also supports user-level driver development by mediating device access with a proprietary kernel module [Shmerling, 2001]. However, the framework seems to promote running drivers as a kernel plug-in in order to achieve better performance. Experiments with user-level drivers on Linux are more promising in this respect. For example, one project provided support infrastructure for user-level PCI drivers, and showed that the performance of user-level IDE disk and gigabit Ethernet drivers is comparable to native execution with CPU overheads limited to just a few percent [Chubb, 2004; Leslie et al., 2005a]. However, even though buggy drivers were listed as one of the motivating factors, we are not aware of an empirical dependability evaluation.

Trust lies with the process manager, which is responsible for providing the process execution model, as well as with the driver manager, which is responsible for installing driver policies. In addition, the hardware must be trusted to function correctly, but this is not different from any of the other techniques discussed in this chapter. Fortunately, hardware provides a much higher level of correctness due to the substantially more formal approach in the development process. Nevertheless, sometimes even hardware suffers from bugs that threaten dependability, as evidenced by CPU specification updates and errata [Kaspersky and Chang, 2008].

Case Study: the L4 family

Although MINIX 3 has different goals, our work bears many relations to L4. L4 is a family of microkernel APIs with implementations available for many architectures, including x86 (IA-32), MIPS, and ARM. The original L4/x86 kernel was written in assembly language, but later versions are written in C and C++ to improve portability. The L4 microkernel is built around three basic concepts: mechanisms to construct address spaces recursively, threads that execute inside an address space, and interprocess communication (IPC). The widely used L4Ka::Pistachio kernel is implemented on the basis of these ideas [L4Ka Project, 2003]. However, certain features needed for fine-grained isolation are lacking in L4Ka::Pistachio. For example, since all IPC is synchronous and the use of IPC is not protected, a buggy driver can potentially block another thread or send arbitrary messages throughout the system, respectively. The newer seL4 kernel overcomes these limitations by providing asynchronous IPC facilities and capabilities for fine-grained authorization. Its implementation comprises about 8700 lines of C code and 600 lines of assembly language. A major milestone was the formal verification of the seL4 kernel's functional correctness [Klein et al., 2009].

While various L4-based user-level frameworks exist, their potential for building a highly robust version of UNIX that can automatically recover from driver failures was not yet fully explored. Previous designs have often focused on performance and

security rather than dependability. For example, DROPS provides real-time support for multimedia applications [Härtig et al., 1998] and Nizza securely wraps untrusted extensions to reduce the size of the trusted computing base (TCB) [Singaravelu et al., 2006]. Although security and dependability are overlapping domains, there are important differences as well. A secure design may incorporate untrusted drivers by encrypting all data before passing it to the driver, but is not necessarily concerned with driver availability. This idea has been put as follows: "I don't care if it works, as long as it is secure [Gasser, 1988]." In contrast, a dependable design must be able to tolerate driver faults and repair failures on the fly. In this respect, the work closest to ours are two L4-based driver frameworks that show how untrusted drivers can be encapsulated [Härtig et al., 2003; Elphinstone and Götz, 2004]. However, the fault isolation was not augmented with failure resilience, and the dependability evaluation was limited to a qualitative analysis.

To conclude, an important area where MINIX 3 can build on L4 is performance. While modular designs sometimes have been criticized for being slow, work on L4 clearly showed that high-performance microkernel-based systems can be built. For example, fast IPC is realized using direct message transfers, putting arguments in CPU registers, and minimizing cache-miss rates [Liedtke, 1993]. Likewise, address space switching without the need for a costly translation-lookaside-buffer (TLB) flush is done by sharing page tables between processes and using segments for protection on the x86 (IA-32) architecture [Liedtke, 1995]. Furthermore, SawMill Linux showed how multiserver IPC protocols can help reduce context-switching and data-copying overheads [Gefflaut et al., 2000]. Finally, even IPC stub-code generation was optimized for performance [Haeberlen et al., 2000]. All these efforts have proven extremely helpful to increase the performance and usability of multiserver systems. It was shown, for example, that user-level frameworks can perform within 5%–10% of a monolithic design [Härtig et al., 1997; Gefflaut et al., 2000]. It may be possible to build on these results in order to improve MINIX 3's performance. Although performance improvements do not directly increase dependability, they may make it affordable to employ certain dependability techniques.

6.4.2 Split-driver Architectures

A middle ground is provided by split-driver architectures that leave the lowest-level driver functionality in the kernel, but puts other parts in isolated user-level processes. For example, performance-critical operations such as device I/O and interrupt handling may be left in the kernel, whereas code for driver initialization, device configuration, error handling, and reporting statistics may be run at the user level. Analysis of 297 drivers showed that, on average, 67% of the code could be moved to user level [Ganapathy et al., 2008]. Applications are not aware of the driver split and still interface with the kernel-level part, which signals the user-level part if there is work to do. Various split-driver variants have been proposed. For example, the Microdrivers project marshals and copies shared data on function calls [Ganapathy et al.,

2008], whereas Windows UMDF sets up a shared memory region accessible to both driver parts [Lee et al., 2009].

Although the literature gives two motivating factors for split drivers, namely performance and compatibility, the added value over user-level frameworks that provide full process encapsulation seems limited. Because experiments with process encapsulation already demonstrated near-native performance [Chubb, 2004; Leslie et al., 2005a], the main benefit of split drivers is the ability to retain the same kernel interface and work with unmodified OS kernels. However, the downside of this design is that it still leaves untrusted code in the kernel, separates functionality that logically belongs together, and complicates the driver development cycle. Furthermore, even though a large fraction of the work required can be automated using a driver rewriting tool [Ganapathy et al., 2008], developers still are in the loop and must annotate the original source code and test the split driver. While tool support for split drivers has great potential for improving the dependability of the legacy driver code base, a user-level driver framework based on process encapsulation seems a better choice for newly developed drivers.

The trust model of split-driver architectures is similar to that of full process encapsulation, but with the important distinction that the kernel-level driver parts also must be trusted. In other words, this design weakens the protection in favor of performance and compatibility. Furthermore, although the majority of driver code typically is not performance critical and can be moved to the user level, bugs may not be distributed uniformly due to differences in code complexity. The remaining kernel-level parts should therefore be protected using in-kernel sandboxing techniques, such as the ones described in Sec. 6.1. For example, XFI protection has been suggested to isolate the in-kernel parts in UMDF [Lee et al., 2009].

Case Study: Microdrivers

Microdrivers provides a split-driver architecture for the Linux OS [Ganapathy et al., 2008]. As shown in Fig. 6.8, the design consists of user-level and kernel-level driver parts, stub code, and run-time libraries. The U-driver, K-driver, and all stub code are automatically generated with help of a DriverSlicer tool. In order to do so, the user must specify the driver's critical root functions and add marshaling annotations to the driver's source code. The DriverSlicer operates in two stages. First, a splitter component builds a call graph starting at the root functions and determines which code should stay in the kernel. Second, a code generator emits the U-driver and K-driver as well as the RPC code for kernel-user communication. The code generator limits the amount of data transferred for complex data structures by copying only the fields that are actually accessed on the other side. The Microdriver run-time libraries are responsible for control and data transfer across protection domains and synchronization of driver data structures. The current implementation does not include defect detection mechanisms, however. Therefore, bugs in the user-level part can still propagate to the kernel.

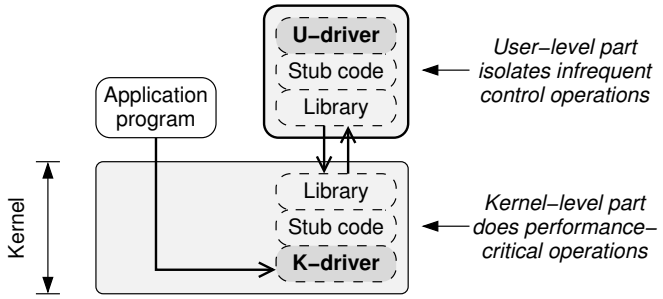


Figure 6.8: Split-driver architecture used by Microdrivers: applications interface with the normal kernel APIs. The kernel-level part (K-driver) calls the user-level part (U-driver) for control operations that are not performance critical.

With the Microdriver framework in place the use of additional protection techniques has been studied. To start with, the Decaf project aims to (re)write user-level drivers in a safe language, such as Java [Renzelmann and Swift, 2009]. The DriverSlicer supports the programmer in converting a driver written in C to a Decaf driver written in Java by generating stub code and marshaling code for both kernel-user communication and C-Java communication. A user-level support library provides access to low-level C functionality that is not available in Java. Another project attempts to protect the kernel against device-driver vulnerabilities [Butt et al., 2009]. In particular, an RPC monitor mediates all communication with the kernel in order to prevent the user-level driver from making unauthorized kernel calls and preserve the integrity of kernel data structures. Permissible control transfers are extracted by statically analyzing the driver's possible responses to an upcall from the kernel part. Permissible data transfers are automatically inferred during a training phase with benign workloads.

The Microdrivers code base consists of about 10,000 LoC for the DriverSlicer and 6750 LoC for the kernel-level and user-level run-time libraries. The DriverSlicer was run on 4 drivers, which required annotations for 1%–6% of the driver code. The amount of code that could be moved to the U-driver ranged from 19% to 65%. The generated kernel-level stub code comprised 6100 LoC up to 37,900 LoC. Testing showed that the stub code was not always correctly generated, however. While debugging the code generator represents a one-time effort, this problem makes clear that the DriverSlicer is part of the trusted computing base (TCB). Experiments with 5 Decaf drivers showed that the amount of user-level code that could be converted to Java ranged from 13% to 100%. Since the majority of code converted is on the control path rather than the data path, Microdrivers performed nearly identical to native drivers. The Decaf drivers also displayed a near-native performance, but the latency to initialize the driver was 5.2× higher on average due to the safe-code tax. Microdrivers were not subjected to automated fault-injection testing, because the current implementation does not provide a recovery subsystem.

6.5 Comparison

Having surveyed four different approaches to deal with untrusted drivers, we now briefly compare our work to related efforts. First, we contrast the use of user-level frameworks to the use of in-kernel sandboxing, virtualization techniques, and formal methods. While all approaches can effectively isolate untrusted drivers, each approach seems to target a particular application. For example, in-kernel sandboxing is particularly suited for retrofitting dependability into legacy OSes, whereas formal methods explore novel techniques for next-generation OSes. User-level frameworks like MINIX 3 are somewhere in the middle of this spectrum, and present a pragmatic approach to improve the dependability of both current and future systems. Interestingly, user-level frameworks can often be combined with other fault-tolerance techniques—while the converse is not always true. In this way, multiple levels of protection can be selectively provided for important components. For example, Microdrivers [Ganapathy et al., 2008] experimented with user-level drivers in a safe language and monitored communication for protocol violations. Another benefit discussed in Sec. 1.4.3 is that user-level frameworks keep the UNIX look and feel, and even seem to make driver development and maintenance easier—something which is not clear for the other approaches.

Furthermore, we want to point out that the use of hardware protection versus software isolation is a recurring theme. We have seen hardware protection with, for instance, virtualization techniques and user-level frameworks, whereas software isolation is used with in-kernel sandboxing and formal methods. Several interesting differences exist. First, software and hardware pose different performance tradeoffs. Although software isolation often incurs a run-time overhead due to the in-line checks needed to guarantee safety, cross-domain communication is less expensive because hardware tables do not have to be reloaded. However, both techniques can achieve good performance, as demonstrated by L4 [Liedtke, 1995] and Singularity [Hunt and Larus, 2007]. Second, software isolation based on formal methods breaks with legacy code because code must be either verified or trusted. In contrast, hardware protection can be used to run legacy code in untrusted domains. Third, the correctness of software isolation is a concern, because verification is not done at execution time, which leaves a window of vulnerability. For example, bugs may be introduced by the compiler or caused by hardware faults corrupting a pointer value or a computation. Therefore, hardware protection remains valuable as a defense-in-depth [Aiken et al., 2006].

Next, we compare how MINIX 3 ranks against other systems that performed an empirical dependability assessment. In particular, both Nooks [Swift et al., 2005] and BGI [Castro et al., 2009] used software-implemented fault-injection (SWIFI) tests similar to the tests we did on MINIX 3. In the comparison below, we ignore hardware limitations, such as PCI bus hangs, since these represent an issue orthogonal to system design. Nooks was subjected to thousands of fault injections, which showed that Linux crashes could be prevented in up to 99% of the cases, but trans-

parent recovery could be provided in only 65% of the cases. BGI was also subjected to thousands of fault injections, which showed that Windows crashes could be prevented in 98%–100% of the cases, but only 47%–60% of the hangs could be prevented. Transparent recovery is not supported by BGI. In contrast, we injected literally many millions of faults, inducing far more driver failures than any of the other experiments, and showed that—hardware limitations aside—MINIX 3 was able to survive 100% of the faults and could recover in over 99.9% of the failures. In other words, MINIX 3 not only was tested with more rigor than its peers, but also demonstrated the highest degree of fault tolerance.

Finally, we highlight some differences between MINIX 3 and other user-level frameworks. Although many modular systems have been built, the primary focus seems to have been performance and security. For example, L4 [Liedtke, 1995] clearly showed that microkernel-based systems can achieve high performance. As suggested in Sec. 6.4.1, these results provide various ideas for improving the performance of MINIX 3. However, the focus of our research has been dependability. To the best of our knowledge, we are the first to explore this area in depth by actually building a complete OS and subjecting it to rigorous testing. For example, while process encapsulation provides coarse-grained isolation, the literature often does not discuss how fine-grained per-driver policies and dynamic access control mechanisms are realized—although L4-based user-level driver frameworks form a notable exception [Härtig et al., 2003; Elphinstone and Götz, 2004]. Furthermore, we are not aware of user-level frameworks that combine fault isolation with failure resilience beyond a simple unload-and-restart approach. Finally, none of the user-level frameworks was subjected to extensive SWIFI tests to assess empirically the system's dependability like we did for MINIX 3.

Chapter 7

Summary and Conclusion

This chapter brings this thesis to an end by summarizing the research, highlighting the lessons learned, and recapitulating the contributions. The central question in this thesis was whether it is possible to build a highly dependable OS. We have met this challenge by building a fault-tolerant OS, MINIX 3, that isolates untrusted drivers and recovers failed ones. Although many of the concepts presented are not completely new, no one has put all the pieces together and subjected the resulting system to rigorous dependability testing. This effort has led to various new insights and advanced the state of the art in dependable OS design.

The remainder of this chapter is organized as follows. To start with, Sec. 7.1 summarizes the research problems and our solutions. Next, Sec. 7.2 presents the lessons that we learned. Then, Sec. 7.3 recapitulates the thesis' contributions, discusses practical applications, and outlines directions for future research. Finally, Sec. 7.4 tells how the resulting OS, MINIX 3, can be obtained.

7.1 Summary of this Thesis

In this thesis, we have researched how we can build a highly dependable OS that is tolerant to driver failures. We first introduced the problem statement and described our approach. Then, we presented in detail the fault-tolerance techniques employed by MINIX 3. Below, we briefly reiterate these issues.

7.1.1 Problem Statement and Approach

We started out with the observation that one of the biggest problems with using computers is that they do not meet user expectations regarding reliability, availability, safety, security, maintainability, etc. Dependability of the operating system (OS) is of particular importance, because of the central role of the OS in virtually any computer system. Although dependability problems with commodity PC operating systems (OSes), such as Windows, Linux, FreeBSD, and MacOS, are well-known,

mobile and embedded OSes are not much different and face similar challenges. For example, a survey of Windows users showed that 77% of the customers experienced 1–5 crashes per month, whereas 23% of the customers experienced more than 5 monthly crashes [Orgovan and Dykstra, 2004]. In order to change this situation and improve the end-user experience, this thesis has addressed the problem of buggy device drivers, which were found to be responsible for the majority of all OS crashes. To begin with, Chaps. 1 and 2 introduced the problem domain and gave an architectural overview of our solution, respectively.

Why Systems Crash

A study of the literature suggested that unplanned downtime is mainly due to faulty system software [Gray, 1990; Thakur et al., 1995; Xu et al., 1999]. Within this class OS failures are especially problematic because they take down all running applications and destroy unsaved user data [Ganapathi and Patterson, 2005]. We identified that OS extensions and device drivers in particular are responsible for the majority of OS crashes. Drivers comprise up to 70% of the code base and have an error rate 3–7 times higher than other code [Chou et al., 2001]. Indeed, Windows XP crash dumps showed that 70% of all crashes are caused by drivers, whereas 15% is unknown due to severe memory corruption [Orgovan and Dykstra, 2004]. Fixing buggy drivers is infeasible due to the large and complex code base as well as continuously changing software and hardware configurations. For example, 25 new and 100 revised Windows drivers are released per day [Glerum et al., 2009].

We then focused on the more fundamental principles that cause OS crashes. We asserted that software is buggy by nature on the basis of various studies that reported fault densities ranging from 0.5 to 75 faults per thousand lines of code (LoC) [Hatton, 1997]. For example, the open-source FreeBSD OS was found to have 1.89 faults/KLoC [Dinh-Trong and Bieman, 2005]. Bugs could be correlated to (a) limited exposure of newly developed driver code [Ostrand et al., 2005] and (b) maintainability problems with existing drivers due to changing kernel interfaces and unwieldy growth of the code base [Padioleau et al., 2006]. Our analysis of Linux 2.6 underscored this problem: the kernel has grown by 65.4% in just 5 years, and now surpasses 5.3 MLoC. Over 50% of the code is taken up by the */drivers* subsystem, which consists of 2.7 MLoC. Not surprisingly, Linux' creator, Linus Torvalds, recently called the kernel 'bloated and huge' [Modine, 2009].

In addition, we pointed out several design flaws that make current OSes so susceptible to bugs. The main problem is the use of a monolithic kernel structure that runs the entire OS as a single binary program running with all powers of the machine and no protection barriers between the modules. This design violates the principle of least authority (POLA) [Saltzer and Schroeder, 1975] by granting excessive privileges to untrusted driver code. This code is often provided by third parties, who may be ignorant of system rules. Because all the code runs in a single protection domain, a single driver bug can easily spread and crash the entire OS.

High-level Design

Having stated the problem we outlined our approach to improve OS dependability. We advocated the use of a modular, multiserver OS design that runs untrusted driver code in independent user-level processes, just like is done for ordinary application programs. In this model, CPU and (IO)MMU hardware protection is complemented with OS-level software protection in order to enforce strictly separate protection domains. As a consequence, the effects of a user-level bug that is triggered will be less devastating. The compartmentalization made it possible to explore the idea of fault tolerance, that is, the ability to continue to operate normally in the presence of faults and failures [Nelson, 1990]. In particular, we outlined two fault-tolerance strategies: (1) fault isolation to increase the mean time to failure (MTTF) and (2) failure resilience to reduce the mean time to recover (MTTR) [Gray and Siewiorek, 1991]. MTTF and MTTR are equally important to increase OS availability. Finally, we described several other benefits of a modular OS design, including a short development cycle, normal programming model, easy debugging, and simple maintenance.

Next, we introduced the open-source MINIX OS [Tanenbaum, 1987], which we used to prototype and test our ideas. We first described how MINIX 3 runs all drivers, servers, and applications in independent user-level processes on top of a small microkernel of about 7500 LoC [Herder, 2005]. Interprocess communication (IPC) is done by passing small, fixed-length messages between process address spaces. For example, the application that wants to do I/O sends a message to the virtual file system (VFS), which forwards the request to the corresponding device driver, which, in turn, may request the kernel to do the actual I/O. We also presented the support infrastructure that we added to MINIX 3 to manage all servers and drivers: the driver manager. The driver manager can be instructed by the administrator via the service utility in order start and stop system services on the fly and set custom policies for the system's fault-tolerance mechanisms.

Context and Scope

We also examined how technological advances and changing user expectations provided a suitable context for this work. Hardware improvements made it possible to revisit the design choices of the past. On the one hand, modern hardware provides better support for isolation, for example, in the form of IOMMUs that can protect against memory corruption due to unauthorized direct memory access (DMA) [Intel Corp., 2008; Advanced Micro Devices, Inc., 2009]. Although certain hardware limitations still exist, for example, shared IRQ lines that may cause interdriver dependencies, we have been able to work around most of them. On the other hand, performance has increased to the point where software techniques that previously were infeasible or too costly have become practical. For example, modular designs have been criticized for performance problems due to context switching and data copying, but we showed how hardware advances have dramatically reduced the absolute costs and thereby mitigated the problem [Moore, 1965]. For example,

an IPC roundtrip on MINIX 3 takes only 1 μs on modern hardware. Moreover, the performance-versus-dependability trade-off has changed and most users would now be more than willing to sacrifice some performance for improved dependability.

Because several other research groups have also acknowledged the problem of buggy drivers, we briefly surveyed related attempts to make drivers more manageable. In particular, we classified the approaches as in-kernel sandboxing, virtualization techniques, formal methods, and user-level frameworks. MINIX 3 fits in the latter class. In-kernel sandboxing uses wrapping and interposition to improve the dependability of legacy OSes, but cannot always provide hard safety guarantees. Virtualization and paravirtualization run legacy OSes in a sandbox, but cannot elegantly isolate individual drivers in a scalable manner. Formal methods use languages-based protection and driver synthesis to improve dependability, but often are not backward compatible. In contrast, user-level frameworks balance these factors by restructuring only the OS internals to provide hard safety guarantees, but keeping the UNIX look and feel for higher layers. While various user-level frameworks have been built, many projects focused on performance and security. To date, no one had explored the limits of isolating and recovering faulty drivers.

Finally, we discussed the assumptions underlying our design and pointed out known limitations. The fault and failure model our system is designed for encompasses soft intermittent faults in drivers, including transient physical faults, interaction faults, and elusive development faults or residual faults that remain after testing [Avižienis et al., 2004]. Such bugs are a common crash cause [Chou, 1997]. They are referred to as Heisenbugs, because their activation is hard to reproduce. By isolating faults and making failures fail-stop [Schlichting and Schneider, 1983], the Heisenbug hypothesis can be exploited by simply retrying failed operations [Gray, 1986]. As an example, consider resources leaks that underlie 25.7% of the defects found in a study of 250 open-source projects [Coverity, Inc., 2008]. While our design does not attempt to prevent such bugs, the damage can be contained in the buggy module and the problem tends to go away after a restart.

7.1.2 Fault-tolerance Techniques

Because software seems to be buggy by nature, we have attempted to make our OS fault tolerant. As a first defense, we isolated drivers from each other and the rest of the OS in order to limit the damage that can result from faults. In addition, because isolated faults can still cause local failures, we made the OS resilient so that failures can be masked to higher levels. To make things concrete, a bug in, say, an audio driver may cause the sound to stop, but may not crash the entire OS. In this way, the OS gets a chance to restart the failed driver and resume normal operation. In the best case, the user does not even notice that there has been a problem. The lion's share of this thesis focused on techniques for achieving such a level of OS fault tolerance. In particular, Chaps. 3 and 4 presented our fault-isolation and failure-resilience techniques, respectively.

Achieving Fault Isolation

The goal of fault isolation is to prevent problems from spreading beyond the module in which a fault is contained. A buggy device driver may cause a local failure, but a global failure of the entire OS should never occur. Our method to achieve this was to restrict untrusted code according to the principle of least authority, granting each component access to only those resources needed to do its job.

We started out by classifying the privileged operations that drivers need to perform and that, unless properly restricted, are root causes of fault propagation. We distinguished four orthogonal classes that map onto the core components found in any computer system (CPU, memory, peripheral devices, and system software), and subdivided each class into subclasses. We also discussed the threats posed by bugs in each class. This classification allowed us to reason systematically about the protection mechanisms needed to restrict drivers. This led to a set of general rules demanding no-privilege defaults in order to isolate faults in each class.

First, we removed all untrusted drivers from the kernel and encapsulated each in an independent UNIX process, so that they became more manageable and could be controlled like ordinary applications. To do so we had to disentangle the drivers' dependencies on the trusted computing base (TCB). For example, drivers need to do device I/O and interrupt handling, copy data between processes, access kernel information, and so on, which they can no longer directly do at the user level. The solution was to add new system calls and kernel calls, such as SAFECOPY and VDEVIO, that allow authorized drivers to access privileged resources in a controlled manner. We also added a new driver manager in order to administer all servers and drivers. This support infrastructure constitutes the MINIX 3 user-level driver framework.

With the user-level driver framework in place, we developed further techniques for curtailing the driver's privileges in each of the operational classes that we distinguished. We gained proper fault isolation through a combination of structural constraints imposed by a multiserver design, fine-grained per-driver isolation policies, and run-time memory-protection mechanisms. For example, CPU usage and memory access were constrained by running each driver in a user-mode process with a private address space. Furthermore, since drivers typically have different I/O and IPC requirements, we associated each with a dedicated isolation policy that grants access to only the resources needed. Finally, because memory allocation typically involves dynamically allocated buffers, we also developed a new delegatable grant mechanism for safe byte-granular memory sharing at run time.

Achieving Failure Resilience

The goal of failure resilience is to recover quickly from failures such that normal operation can continue with minimum disturbance of application programs and end users. In other words, driver problems should be detected automatically and repaired transparently. Building on the fault-isolation properties discussed above, our method

to achieve this was to monitor drivers at run time and design the OS such that it can handle on-the-fly replacement of failing or failed drivers.

We started out by discussing how the driver manager can detect failures. We did not use introspection to detect erroneous system states, but instead focused on detecting component failures, that is, deviations from the specified service. This led to three defect detection techniques. First, all drivers are monitored for unexpected process exits due to CPU or MMU exceptions and internal panics. Second, the driver manager proactively checks for liveness by periodically requesting a driver heartbeat message. Third, dynamic updates can be requested by the administrator or trusted OS components. In this way, informed decisions can be made for problems that cannot be detected by the driver manager, such as server-to-driver protocol violations.

Next, we showed how the system can recover once a failure has been detected. By default the driver manager directly restarts failed drivers, but if more flexibility is needed, individual drivers can be associated with a recovery script. This script is a normal shell script that can be used, for example, to log the failed component and its execution environment, to implement a binary exponential backoff for repeated failures, to send a failure alert to a remote administrator, and so on. Once a driver has been restarted, the rest of the system is informed so that further recovery, such as reissuing pending I/O requests, can be done. In addition, the driver may need to reset its device and recover lost state. Although we provided a data store to backup state, we also pointed out various gaps in state management. Therefore, we assume fail-stop behavior where erroneous state transformations do not occur. This did not pose a problem since the MINIX 3 drivers are mostly stateless.

Having outlined the basic ideas we analyzed the effectiveness of MINIX 3's failure-resilience mechanisms. Two properties are important in this respect. First, recovery is transparent if it can be done internal to the OS without returning an error to application programs and without user intervention. Second, recovery is full or lossless if no user data is lost or corrupted. We studied recovery schemes for different OS components. First, the effectiveness of driver recovery depends on (a) whether I/O is idempotent, that is, can be repeated without affecting the end result, and (b) whether the protocol stack provides end-to-end integrity. We found that full transparent recovery is possible for network-device drivers and block-device drivers, but not always for character-device drivers. Second, server recovery is typically limited by the amount of internal state that is lost during a restart.

7.2 Lessons Learned

All concepts described in this thesis have been prototyped and tested in MINIX 3. Our experimental evaluation was based on software-implemented fault injection (SWIFI), performance measurements, and a source code analysis. Chap. 5 gave a detailed overview of the raw results of these experiments. Below, we present the most important lessons that we have learned.

7.2.1 Dependability Challenges

We have taken an empirical approach toward dependability and have iteratively refined our fault-tolerance techniques using extensive SWIFI tests. We injected 8 fault types at run time into the text segment of unmodified drivers. The faults types used were shown to be representative for both transient hardware errors and common programming errors. We targeted various driver configurations covering a representative set of interactions with the surrounding software and hardware. The SWIFI tests have led to the following insights.

- To start with, we learned that extensive SWIFI campaigns are needed to find the majority of the bugs. While the SWIFI tests immediately proved helpful to debug the system, some hard-to-trigger bugs showed up only after several design iterations and injecting many millions of faults. However, in the past, for example, Nooks [Swift et al., 2005] and BGI [Castro et al., 2009] were subjected to only 2000 and 3375 fault injections, respectively, which is almost certainly not enough to find all the bugs. Even though robustness to injected faults gives no quantitative prediction of robustness to real faults, we believe that our extensive SWIFI campaign sets a new standard for hardening systems against possible bugs.

- Stress tests demonstrated that our design can indeed tolerate faults occurring in untrusted drivers. In a test targeting 4 different network-device drivers, MINIX 3 was able withstand 100% of 3,200,000 common, randomly injected faults. Although the targeted drivers failed 24,883 times, the OS was never affected and user programs correctly executed despite the driver failures. In fact, the OS could transparently recover failed network-device drivers in 99.9% of the cases. This result represents a much higher degree of fault tolerance than other systems that empirically assessed their dependability. While SWIFI testing alone cannot prove our design correct, we are assured that our fault-tolerance techniques help to improve OS dependability.

- Even if full recovery was not possible, our fault-tolerance techniques still improved dependability, either by limiting the consequences of failures or by speeding up recovery. For example, in a few cases the restarted network-device driver could not reinitialize the hardware, which caused the networking to stop working, but did not affect the rest of the OS. Furthermore, while audio-driver recovery was inherently limited because the I/O operations were not idempotent, recovery at the cost of hiccups in the audio playback was still possible. Finally, although recovery of the network server (INET) was not transparent due to the amount of internal state lost on a restart, recovery scripts helped automating the recovery procedure for a remote web server where minor downtime was tolerable. These examples show that full transparent recovery is not always needed to improve dependability.

- Experiments with a filter driver in the storage stack showed that monitoring of untrusted drivers can improve availability. While the driver manager could not detect driver-specific protocol violations, we found recovery triggered by complaints from trusted components to be effective. We did not use this strategy in the network stack, but the logs revealed that the number of unauthorized access attempts can be several orders of magnitude higher than the number of failures seen by the driver manager. The use of more introspection and proactive recovery thus seems worth investigating.

- Although this research focused on mechanisms rather than policies, it must go hand in hand with careful policy definition. At some point, a driver's isolation policy accidentally granted access to a kernel call for copying arbitrary memory without the grant-based protection, causing memory corruption in the network server. Even though policy definition is an orthogonal issue, it is key to the effectiveness of the mechanisms provided. We 'manually' reduced the privileges granted by the driver's policy, but techniques such as formalized interfaces [Ryzhyk et al., 2009a] and compiler-generated manifests [Spear et al., 2006] may be helpful to define correct policies.

- Finally, the SWIFI tests also revealed several hardware limitations that suggest that safe systems cannot be realized by software alone. First, while the OS could successfully restart failed drivers, the hardware devices were sometimes put in an unrecoverable state and required a BIOS reset. This shows that all devices should have a master reset command to allow recovery from unexpected (software) bugs. Second, tests with two PCI cards even caused the entire system to freeze, presumably due to a PCI bus hang. We believe this to be a weakness of the PCI bus, which should have confined the problem and shut down the malfunctioning device. These problems must be addressed by hardware manufacturers through more fault-tolerant hardware designs.

7.2.2 Performance Perspective

Although our primary focus was dependability, we also conducted several performance measurements on MINIX 3, FreeBSD, and Linux in order to determine the costs of our fault-tolerance techniques. The benchmarks exemplified MINIX 3's modular design by triggering IPC between applications, servers, drivers, and the kernel. We did not optimize the system, however, and the results should be taken as a rough estimate. Nevertheless, we gained various insights.

- We found that MINIX 3 is fast and responsive for typical research and development usage. For example, on modern hardware, installing MINIX 3 on a fresh PC takes about 10 minutes, a full build of the OS can be done under 20 sec, and a reboot of the machine takes just over 5 sec. In addition, the infrastructure to start and stop system services on the fly helps to speed up testing of

new components. Measurements also showed that the overhead introduced by MINIX 3's fault-tolerance mechanisms is limited. These characteristics make MINIX 3 very suitable as a prototype platform.

- Performance measurements showed that the user-perceived overhead is mostly determined by the usage scenario rather than MINIX 3's raw performance. System-call-level microbenchmarks showed an average overhead of 12% for user-level versus in-kernel drivers, whereas the average overhead decreased to only 6% for application-level macrobenchmarks. Most overhead was found for I/O-bound workloads, whereas CPU-bound workloads displayed a negligible overhead or no overhead at all. While these findings are not surprising, they show how important it is to take the workload into account when assessing the system's usability.

- The comparison of MINIX 3 with other OSes showed how trade-offs in system design affect the overall performance. While a small performance gap was visible between MINIX 3 and FreeBSD 6.1, a roughly equivalent gap exists between FreeBSD 6.1 and Linux 2.6. This shows that the use of a modular design has a similar effect on performance as other system parameters, including storage-stack strategies, compiler quality, memory management algorithms, and so on. None of these aspects have been optimized in MINIX 3, which indicates that there is room for improvement. While we cannot remove the inherent costs incurred by a modular design, various independent studies have already shown that the overhead can be limited to 5%–10% through careful analysis and removal of bottlenecks [Liedtke, 1993, 1995; Härtig et al., 1997; Gefflaut et al., 2000; Haeberlen et al., 2000; Leslie et al., 2005a].

7.2.3 Engineering Effort

The last part of our experimental evaluation consisted of a source-code analysis of MINIX 3 and Linux 2.6. For both systems we started with the initial release and picked subsequent versions with 6-month deltas. The analysis spanned 4 years of MINIX 3 development (9 versions) and 5 years of Linux 2.6 development (11 versions). Measurements were done by counting lines of executable code (LoC). The source-code analysis has led to the following insights.

- The source code analysis showed that the reengineering effort needed to make MINIX 3 fault tolerant is both limited and local. In general, all interactions with drivers, which are now considered untrusted, required a more defensive programming style, such as performing additional access control and sanity checks. These OS changes represent a one-time effort. We also found that some of the existing drivers had to be modified, for example, to use safe system call variants. Fortunately, such changes typically required little effort. Porting or writing new drivers is not complicated because we were able to maintain a UNIX-like development environment.

- The previous finding makes us believe that our ideas might be applicable to other OSes with only modest changes. Although the process encapsulation provided by our prototype platform, MINIX 3, enabled us to implement and test our ideas with relatively little engineering effort, a similar trend toward isolation of untrusted extensions on other OSes is ongoing. For example, there already have been experiments with user-level drivers on both Linux [Chubb, 2004] and Windows [Microsoft Corp., 2007]. Once the drivers run at the user-level, these systems may be able to build on the fault-isolation and failure-resilience techniques presented here.

- Although the MINIX 3 code base grew by 64.4% to 87.8 KLoC in 4 years, MINIX 3's source-code evolution seems sustainable. Because MINIX 3 runs most of the OS in isolated user-level compartments, the addition of new functionality generally does not pose a direct risk to the trusted computing base (TCB). However, due to new, low-level infrastructure the kernel also doubled in size and now measures 6881 LoC—and further growth is expected with multicore support and kernel threads still in the pipeline. Nevertheless, we expect the kernel's size to stabilize once it becomes more mature. A case in point is that more mature microkernels such as L4Ka::Pistachio and seL4 measure on the order of 10 KLoC [Heiser, 2005]. The bottom line is that compartmentalization makes the code base more manageable.

- We also found that the Linux 2.6 kernel grew by 65.4% to 5.3 MLoC in 5 years, which may pose a threat to its dependability. The problem is not only that Linux 2.6 is several orders of magnitude larger than MINIX 3, but also that the entire OS is part of the kernel. All source-code evolution thus directly affects the size and complexity of the TCB. We realize that not all drivers can be active at the same time, but Linux' monolithic design means that any fault can potentially be fatal. Despite its huge code base the Linux 2.6 kernel has been very stable, but structural changes still seem advisable. Even though the problem is no longer ignored [Modine, 2009], the concerns raised thus far seem to focus on performance rather than dependability. A different mindset is needed in this respect.

- A final point worth mentioning is that, in our experience, driver developement and maintenance indeed has become easier due to MINIX 3's modular design. Despite all the fault-tolerance techniques that we introduced, we were able to maintain the normal UNIX programming model. Interestingly, the changes even helped to shorten the driver development cycle, since drivers now can be programmed, tested, and debugged without tedious reboots after each build. Furthermore, we ran into various driver problems that could be fixed either automatically or by dynamically updating the faulty driver with a new or patched version. While we did not quantitatively analyze these benefits, we do believe that the use of a modular OS design indeed leads to higher productivity.

7.3 Epilogue

Arriving at the end of this thesis it is time to reflect. Below, we briefly summarize the main contributions, discuss possible applications, and outline future research areas.

7.3.1 Contribution of this Thesis

The goal of this work was to build a dependable OS that can survive and recover from common bugs in device drivers. We realized this goal by making the OS fault tolerant such that it can continue to operate normally in the presence of faults and failures. Although many of the ideas and techniques presented are not completely new, their combined potential to improve OS dependability had not yet been fully explored. By doing so we have made the following major contributions:

- We demonstrated how OS dependability can be improved without sacrificing the widely used UNIX programming model. In contrast to related work, we have redesigned only the OS internals, maintaining backward compatibility with existing software and presenting an incremental path to adoption.
- We presented a classification of privileged device-driver operations that are root causes of fault propagation and developed a set of fault-isolation techniques for each class in order to limit the damage bugs can do. We believe this to be an important result for any effort to isolate drivers in any system.
- We presented a failure-resilient OS design that can detect and recover from a wide range of failures in drivers and other critical components, transparently to applications and without user intervention. We believe that many of these ideas are also applicable in a broader context.
- We have evaluated our design using extensive software-implemented fault-injection (SWIFI) testing. In contrast to earlier efforts, we literally injected millions of faults, which allowed us to find also many rare faults and demonstrate improved dependability with high assurance.
- We showed how recent hardware virtualization techniques can be used to overcome shortcomings of previous isolation techniques. At the same time we discussed a number of PC-hardware shortcomings that still allow even a properly isolated device driver to hang the entire system.
- Finally, we have not only designed, but also implemented the complete system, resulting in a freely available, open-source OS, MINIX 3, which can be obtained from the official website at http://www.minix3.org/. The MINIX 3 OS effectively demonstrates the practicality of our approach.

All in all, we believe that our effort to build a fault-tolerant OS that can withstand the threats posed by buggy device drivers represents a small step toward more dependable OSes and helps to improve the end-user experience.

7.3.2 Application of this Research

We hope that our research will find its way into practical applications so that actual end users can benefit from a more dependable computing environment. The direct real-world impact of this research is the MINIX 3 OS, which has already been downloaded over 250,000 times and sparked many discussions about OS design [e.g. Slashdot, 2005, 2006, 2008]. However, we believe that an even wider audience may be reached by improving MINIX 3's usability and applying the ideas put forward in this thesis to other OSes.

Adoption of MINIX 3

Wide-spread adoption of MINIX 3 can only become a reality if the system becomes more usable for ordinary end users. While MINIX 3 is not yet competitive with much more mature systems, there is clearly enough already to eliminate any doubt that it could be done given 'some' programming effort. At the same time, we realize that the system still has plenty of shortcomings.

The completeness of MINIX 3 needs to be assessed at various levels. For example, at the application level, about 500 UNIX programs have already been written or ported, but many of the larger, widely used applications are still missing, including the GNOME and KDE desktop environments, the Firefox web browser, and a Java virtual machine. Looking at library and framework support, MINIX 3 provides basic POSIX compliance, but better POSIX compatibility and more system libraries would be needed to support developers. At the driver level, MINIX 3 provides the most crucial drivers to run on standard hardware and popular emulators, but still needs drivers for a range of peripheral devices, I/O buses, and hardware standards, such as the Universal Serial Bus (USB) and Advanced Configuration and Power Interface (ACPI). At the architecture level, MINIX 3 currently runs on the x86 (IA-32) architecture, but does not yet support the MIPS and ARM architectures that are widely used in embedded systems. In addition, MINIX 3 lacks support for multiprocessor architectures. Finally, looking at user support, there is a small community to support MINIX 3 users, but better developer documentation and end-user manuals would be needed in order to reach a wider audience.

Due to the small size of the core MINIX team, it will be hard to address all of these issues, and the gap with more mature general-purpose OSes is likely to grow. Therefore, a more promising approach seems to target a narrow application domain. One important area where MINIX 3 is already widely used is education and research. The development of additional course and training materials could potentially help to spread the ideas underlying MINIX 3's design even further. Another, more practical possibility would be to showcase MINIX 3's features by building a highly dependable dedicated system, for example, a set-top box, cell phone, voting machine, router, firewall, and so on. Besides less demanding functional requirements, the primary nonfunctional requirement for such applications is aligned with MINIX 3's goals: showing how dependable systems can be built.

Improving Commodity OSes

Finally, even if MINIX 3 itself would not become a mainstream OS, commodity OSes such as Windows, Linux, FreeBSD, and MacOS can potentially benefit from the ideas presented in this thesis. Although we have implemented our design in MINIX 3, many of the techniques are generally applicable and can be ported to improve the dependability of other systems. For example, IOMMU protection against invalid DMA requests can be adopted even with in-kernel drivers. Likewise, drivers could be associated with fine-grained isolation policies to restrict access to privileged OS functionality. For example, the Windows *hardware abstraction layer* (HAL) might be a suitable starting point to enforce such protection. However, even though the engineering effort was modest in the case of MINIX 3, it is hard to estimate the amount of work required without intimate knowledge of the target OS.

In order to benefit most, however, we believe it is crucial to remove drivers from the kernel. Fortunately, as discussed in Sec. 6.4, work in this area is already in progress and user-level driver frameworks are slowly making their way into commodity OSes [Microsoft Corp., 2007; Leslie et al., 2005a]. While the sheer size of Windows and Linux makes it infeasible to transform all existing in-kernel drivers, user-level drivers can be adopted incrementally. As a starting point, the most widely used drivers and the most error-prone drivers (as pinpointed by crash-dump analysis) should be moved out of the kernel. In addition, drivers for new hardware should be developed at the user level by default. Once drivers run at the user level it becomes easier to apply the full set of fault-tolerance techniques described in this thesis.

7.3.3 Directions for Future Research

The research on MINIX 3 has opened several opportunities for future work. Below we suggest two potentially fruitful areas for follow-up research projects.

Dependability and Security

Follow-up studies can build on our work to further enhance dependability and security. In particular, as suggested in Sec. 1.5, it would be interesting to investigate whether MINIX 3's fault-isolation techniques can be combined with other protection techniques. For example, wrapping and interposition could be used to monitor drivers for protocol violations, which proved useful in our filter-driver case study. Likewise, it may be worthwhile to focus on driver intrusion detection [Butt et al., 2009], since code that is relatively error-prone might also be relatively vulnerable. Furthermore, many of the other techniques discussed in Chap. 6, such as software-based fault isolation and safe languages, could be applied to improve the protection of user-level drivers. With drivers running in independent processes, each individual driver can potentially get a different protection strategy.

Another interesting area would be to improve the system's failure-resilience techniques. To start with, dynamic updates may be facilitated by using a cooperative

model whereby the system is requested to converge to a stable state before replacing a component [Giuffrida and Tanenbaum, 2009]. Next, improved state management, including transaction support, is needed to deal with crashes of stateful components. Possible approaches based on checkpointing state using the data store were already outlined in Sec. 4.2.3. Furthermore, the restart process might be made more transparent to the rest of the OS by associating system services with stable, virtual IPC endpoints. Finally, rather than cleaning up and restarting components after a problem has been detected, OS services could be replicated a priori using a shadow driver or 'hot standby', so that a backup process can take over and impersonate the primary process in case of a failure [Swift et al., 2006]. State synchronization between the primary and backup would still be needed though.

In addition to fault tolerance for drivers, it also may be interesting to investigate whether similar ideas can be applied to system servers and application programs. Making system servers, such as file servers, untrusted helps reducing the size of the TCB and thereby makes it easier to reason about the dependability or security of a system. The strategies could be similar to those for drivers, but it would be important to overcome the current problems relating to state management. It also may be possible to take some of our techniques to the application layer. Modern applications, such as web browsers, office suites, and photo editors, allow their base functionality to be extended via (third-party) plug-ins, but put the user at risk because buggy and potentially hostile plug-ins are run with the same user ID and in the same address space as the host application. What would be needed instead is to isolate plug-ins from the application and each using separate protection domains. This is analogous to isolating drivers from the core OS.

Performance and Multicore

Although this research has not focused on performance, we realize that performance is important for the system's usability. In addition, because there sometimes is a trade-off between performance and dependability, improving the system's performance may also improve the practicality of certain dependability techniques. Even though various research projects have achieved good performance, within 5%–10% of a monolithic design [Härtig et al., 1997; Gefflaut et al., 2000], modular systems are still being criticized. Therefore, we believe that it is important to make MINIX 3's performance more competitive to commodity OSes. While this is, in part, a matter of careful engineering, such as profiling and removal of bottlenecks, there is ample opportunity to build on and possibly extend previous research projects. Sec. 6.4.1 already listed several techniques that were successfully used in L4. For example, one interesting area may be to look into multiserver IPC protocols that aim to minimize context switching and data copying [Gefflaut et al., 2000]. In particular, with the recent addition of a new virtual-memory (VM) subsystem to MINIX 3, it may be worthwhile to investigate VM-based optimizations such as setting up shared memory regions to transfer efficiently IPC messages and data structures.

Another interesting research question is whether modular, multiserver OSes can take advantage of multicore platforms. After approximately three decades of yearly performance gains of 40%–50%, CPU speeds no longer seem to increase in line with Moore's law, and most performance benefits are expected from multicore CPU designs [Larus, 2009]. However, monolithic OSes might be difficult to scale because of inter-CPU locking complexities and data-locality issues on massive multicore architectures. Multiserver OS designs, in contrast, seem to map more naturally onto multicore systems. For example, it might be possible to compartmentalize the OS such that the file server and the network server along with their clients run on different sets of CPUs, so that they can be active at the same time. Furthermore, the original criticism against modular OS designs—expensive IPC—might turn out to be a benefit in the long run. In particular, message-passing-based IPC might be cheaper than shared memory due to the latency of fetching remote memory [Baumann et al., 2009]. Recent chip designs even feature hardware support for message passing [Feldman, 2009]. Future research should further investigate how modular designs compare to monolithic designs with respect to complexity and performance when they are scaled to massive multicore architectures.

7.4 Availability of MINIX 3

To conclude this thesis, we would like to emphasize that the MINIX 3 OS is free and open-source software. All the fault-tolerance mechanisms as well as the test infrastructure described in this thesis are publicly available from the MINIX 3 source-code repository [Vrije Universiteit Amsterdam, 2009]. The official MINIX 3 website can be found at http://www.minix3.org/ and provides a live CD, beta versions, source code, user documentation, news and updates, a community wiki, contributed software, and more. Since the official release of MINIX 3 in October 2005, over 250,000 people have already downloaded the OS, resulting in a large and growing user community that communicates using the Google Group *minix3*. MINIX 3 is being actively developed, and your help and feedback are much appreciated.

References

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proc. 12th ACM Conf. on Computer and Communications Security*, pp. 340–353, Nov. 2005.

Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 USENIX Summer Conf.*, pp. 93–113, June 1986.

Adams, K. and Agesen, O. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proc 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, Oct. 2006.

Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Specification. White Paper, PID 34434, Rev. 1.26, Feb. 2009.

Ahmad, F. S. MINIX 3 C Compiler Performance: Comparing the Amsterdam Compiler Kit to the GNU Compiler Collection on x86 Systems. Bachelor's Thesis, Dept. of Computer Science, Vrije Universiteit Amsterdam, June 2008.

Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., and Larus, J. Deconstructing Process Isolation. In *Proc. 2006 ACM SIGPLAN Ws. on Memory System Performance and Correctness*, pp. 1–10, Oct. 2006.

Albinet, A., Arlat, J., and Fabre, J.-C. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proc. 34th Int'l Conf. on Dependable Systems and Networks*, pp. 867–876, June/July 2004.

Appleton, B. Source Code Line Counter (sclc.pl), Apr. 2003. Available online: http://www.cmcrossroads.com/bradapp/clearperl/sclc-cdiff.html.

Arlat, J., Fabre, J.-C., Rodríguez, M., and Salles, F. Dependability of COTS Microkernel-Based Systems. *IEEE Trans. on Computers*, 51(2):138–163, Feb. 2002.

Armand, F. Give a Process to your Drivers! Tech. Report CS/TR-91-97, Chorus Systèmes, Sept. 1991.

Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable*

and Secure Computing, 1(1):11–33, Jan.–Mar. 2004.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough Static Analysis of Device Drivers. In *Proc. 1st EuroSys Conf.*, pp. 73–85, Apr. 2006.

Barbou des Places, F., Stephen, N., and Reynolds, F. D. Linux on the OSF Mach3 Microkernel. In *Proc. 1st Conf. on Freely Redistributable Software*, paper no. 5, Feb. 1996.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proc. 19th ACM Symp. on Operating Systems Principles*, pp. 164–177, Oct. 2003.

Bartlett, J. F. A NonStop Kernel. In *Proc. 8th ACM Symp. on Operating Systems Principles*, pp. 22–29, Dec. 1981.

Basili, V. R. and Perricone, B. T. Software Errors and Complexity: An Empirical Investigation. *Comm. of the ACM*, 27(1):42–52, Jan. 1984.

Baumann, A., Appavoo, J., Wisniewski, R. W., Silva, D. D., Krieger, O., and Heiser, G. Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proc. 2007 USENIX Annual Technical Conf.*, pp. 337–350, June 2007.

Baumann, A., Peter, S., Schüpbach, A., Singhania, A., Roscoe, T., Barham, P., and Isaacs, R. Your Computer is Already a Distributed System. Why isn't Your OS? In *Proc. 12th Ws. on Hot Topics in Operating Systems*, paper no. 12, May 2009.

Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *Proc. 2005 USENIX Annual Technical Conf., FREENIX Track*, pp. 41–46, Apr. 2005.

Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8(1):37–55, Feb. 1990.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility Safety and Performance in the SPIN Operating System. In *Proc. 15th ACM Symp. on Operating Systems Principles*, pp. 267–283, Dec. 1995.

Blair, M., Obenski, S., and Bridickas, P. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Tech. Report GAO/IMTEC-92-26, U.S. General Accounting Office, Feb. 1992.

Bos, H. and Samwel, B. Safe Kernel Programming in the OKE. In *Proc. 5th Conf. on Open Architectures and Network Programming*, pp. 141–152, June 2002.

Boutin, P. Blue Screen of Death: Why your Computer Still Crashes. *Slate Magazine*, Sept. 2004. Available online: http://www.slate.com/id/2107471/.

Butt, S., Ganapathy, V., Swift, M. M., and Chang, C.-C. Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In *Proc. 25th Annual Computer Security Applications Conf.*, pp. 301–310, Dec. 2009.

Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot—A Technique for Cheap Recovery. In *Proc. 6th Symp. on Operating System Design and Implementation*, pp. 31–44, Dec. 2004.

Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast Byte-Granularity Software Fault Isolation. In *Proc. 22nd ACM Symp. on Operating Systems Principles*, pp. 45–58, Oct. 2009.

Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. Sharing and Protection in a Single-Address-Space Operating System. *ACM Trans. on Computer Systems*, 12(4):271–307, Nov. 1994.

Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M.-Y. Orthogonal Defect Classification—A Concept for In-Process Measurements. *IEEE Trans. on Software Engineering*, 18(11):943–956, Nov. 1992.

Chipounov, V. and Candea, G. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proc. 5th EuroSys Conf.*, pp. 167–180, Apr. 2010.

Chiueh, T.-c., Venkitachalam, G., and Pradhan, P. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proc. 17th ACM Symp. on Operating Systems Principles*, pp. 140–153, Dec. 1999.

Chou, A., Yang, J.-F., Chelf, B., Hallem, S., and Engler, D. An Empirical Study of Operating System Errors. In *Proc. 18th ACM Symp. on Operating Systems Principles*, pp. 73–88, Oct. 2001.

Chou, T. C. K. Beyond Fault Tolerance. *IEEE Computer*, 30(4):47–49, Apr. 1997.

Christmansson, J. and Chillarege, R. Generation of an Error Set that Emulates Software Faults—Based on Field Data. In *Proc. 26th Int'l Symp. on Fault-Tolerant Computing*, pp. 304–313, June 1996.

Chubb, P. Get More Device Drivers out of The Kernel! In *Proc. 2004 Ottawa Linux Symp.*, pp. 149–161, July 2004.

Conway, C. L. and Edwards, S. A. NDL: A Domain-Specific Language for Device Drivers. In *Proc. 2004 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, pp. 30–36, June 2004.

Coverity, Inc. Open Source Report 2008. White Paper on Software Quality and Security, 2008. Available online: http://scan.coverity.com/report/.

Creasy, R. J. The Origin of the VM/370 Time-Sharing System. *IBM Systems Journal*, 25(5):483–490, Sept. 1981.

David, F. M. and Campbell, R. H. Building a Self-Healing Operating System. In *Proc. 3rd IEEE Int'l Symp. on Dependable, Autonomic and Secure Computing*, pp. 3–10, Sept. 2007.

David, F. M., Chan, E., Carlyle, J. C., and Campbell, R. H. CuriOS: Improving Reliability through Operating System Structure. In *Proc. 8th USENIX Symp. on Operating System Design and Implementation*, pp. 59–72, Dec. 2008.

Department of Defense. Trusted Computer System Evaluation Criteria. Department of Defense Standard DOD 5200.28-STD, Library No. S225,7ll, Dec. 1985. Rainbow Series Library, Orange Book.

DiBona, C. and Ockman, S. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, 1st ed., Jan. 1999. Appendix A: The Tanenbaum-Torvalds Debate.

Dinh-Trong, T. T. and Bieman, J. M. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Trans. on Software Engineering*, 31 (6):481–494, June 2005.

Dowson, M. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, Mar. 1997.

Durães, J. A. and Madeira, H. S. Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. In *Proc. 9th Pacific Rim Int'l Symp. on Dependable Computing*, pp. 201–209, Dec. 2002.

Durães, J. A. and Madeira, H. S. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Trans. on Software Engineering*, 32(11):849–867, Nov. 2006.

Elphinstone, K. and Götz, S. Initial Evaluation of a User-Level Device Driver Framework. In *Proc. 9th Asia-Pacific Computer Systems Architecture Conf.*, pp. 256–269, Sept. 2004.

Endres, A. An Analysis of Errors and Their Causes in System Programs. *ACM SIGPLAN Notices*, 10(6):327–336, June 1975.

Erlingsson, U., Roeder, T., and Wobber, T. Virtual Environments for Unreliable Extensions: A VEXE'DD Retrospective. Tech. Report MSR-TR-2005-82, Microsoft Research, June 2005.

Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software Guards for System Address Spaces. In *Proc. 7th USENIX Symp. on Operating System Design and Implementation*, pp. 75–88, Nov. 2006.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R., and Levi, S. Language support for fast and reliable message-based communication in singularity OS. In *Proc. 1st EuroSys Conf.*, pp. 177–190, Apr. 2006.

Feldman, M. Intel Unveils 48-Core Research Chip. HPCwire >> Features, Dec.

2009. Available online: http://www.hpcwire.com/features/Intel-Unveils-48-Core-Research-Chip-78378487.html.

Fenton, N. E. and Ohlsson, N. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, 26(8):797–814, Aug. 2000.

Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williams, M. Safe Hardware Access with the Xen Virtual Machine Monitor. In *1st Ws. on Operating System and Architectural Support for the on demand IT InfraStructure*, paper no. 5, Oct. 2004.

Ganapathi, A. and Patterson, D. Crash Data Collection: A Windows Case Study. In *Proc. 35th Int'l Conf. on Dependable Systems and Networks*, pp. 280–285, June/July 2005.

Ganapathi, A., Ganapathi, V., and Patterson, D. Windows XP Kernel Crash Analysis. In *Proc. 20th Large Installation and System Administration Conf.*, pp. 149–159, Dec. 2006.

Ganapathy, V., J.Renzelmann, M., Balakrishnan, A., Swift, M. M., and Jha, S. The Design and Implementation of Microdrivers. In *Proc 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 168–178, Mar. 2008.

Ganev, I., Eisenhauer, G., and Schwan, K. Kernel Plugins: When a VM is Too Much. In *Proc. 3rd Virtual Machine Research and Technology Symp.*, pp. 83–96, May 2004.

Garfinkel, S. History's Worst Software Bugs. Wired.com, Nov. 2005. Available online: http://www.wired.com/news/technology/bugs/0,2924,69355,00.html.

Gasser, M. *Building a Secure Computer System*. Van Nostrand Reinhold, May 1988.

Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K. J., Uhlig, V., Tidswell, J. E., Deller, L., and Reuther, L. The SawMill Multiserver Approach. In *Proc. 9th ACM SIGOPS European Ws.*, pp. 109–114, Sept. 2000.

Giuffrida, C. and Tanenbaum, A. S. Cooperative Update: A New Model for Dependable Live Update. In *Proc. 2nd Int'l Ws. Hot Topics in Software Upgrades*, paper no. 1, Oct. 2009.

Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., and Hunt, G. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proc. 22nd ACM Symp. on Operating Systems Principles*, pp. 103–116, Oct. 2009.

Glew, N. and Morrisett, G. Type-Safe Linking and Modular Assembly Language. In *Proc. 26th ACM SIGPLAN-SIGACT Symp. on the Principles of Programming*

Languages, pp. 250–261, Jan. 1999.

Godfrey, M. W. and Tu, Q. Evolution in Open Source Software: A Case Study. In *Proc. 16th Int'l Conf. on Software Maintenance*, pp. 131–142, Oct. 2000.

Golub, D. B., Dean, R. W., Forin, A., and Rashid, R. F. UNIX as an Application Program. In *Proc. 1990 USENIX Summer Conf.*, pp. 87–95, June 1990.

Govindavajhala, S. and Appel, A. W. Using Memory Errors to Attack a Virtual Machine. In *Proc. 2003 IEEE Symp. on Security and Privacy*, pp. 154–165, May 2003.

Gray, J. Why Do Computers Stop and What Can Be Done About It? In *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, pp. 3–12, Jan. 1986.

Gray, J. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–481, Oct. 1990.

Gray, J. and Siewiorek, D. P. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, Sept. 1991.

Gu, W., Kalbarczyk, Z., Iyer, R. K., and Yang, Z. Characterization of Linux Kernel Behavior under Errors. In *Proc. 33rd Int'l Conf. on Dependable Systems and Networks*, pp. 459–468, June 2003.

Gunawi, H. S., Prabhakaran, V., Krishnan, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Improving File System Reliability with I/O Shepherding. In *Proc. 21st ACM Symp. on Operating Systems Principles*, pp. 293–306, Oct. 2007.

Guo, P. J. and Engler, D. Linux Kernel Developer Responses to Static Analysis Bug Reports (short paper). In *Proc. 2009 USENIX Annual Technical Conf.*, pp. 285–292, June 2009.

Haeberlen, A., Liedtke, J., Park, Y., Reuther, L., and Uhlig, V. Stub-Code Performance is Becoming Important. In *Proc. 1st Ws. on Industrial Experiences with Systems Software*, Oct. 2000.

Hand, S., Warfield, A., Fraser, K., Kotsovinos, E., and Magenheimer, D. Are Virtual Machine Monitors Microkernels Done Right? In *Proc. 10th Ws. on Hot Topics in Operating Systems*, p. 1ff, June 2005.

Härtig, H., Hohmuth, M., Liedtke, J., Wolter, J., and Schönberg, S. The Performance of μ-Kernel-Based Systems. In *Proc. 17th ACM Symp. on Operating Systems Principles*, pp. 66–77, Oct. 1997.

Härtig, H., Baumgartl, R., Borriss, M., Hamann, C.-J., Hohmuth, M., Mehnert, F., Reuther, L., Schönberg, S., and Wolter, J. DROPS: OS Support for Distributed Multimedia Applications. In *Proc. 8th ACM SIGOPS European Ws.*, pp. 203–209, Sept. 1998.

Härtig, H., Löser, J., Mehnert, F., Reuther, L., Pohlack, M., and Warg, A. An I/O Architecture for Microkernel-Based Operating Systems. Tech. Report TUD-FI03-08, Dresden University of Technology, July 2003.

Hatton, L. Reexamining the Fault Density-Component Size Connection. *IEEE Software*, 14(2):89–97, Mar./Apr. 1997.

Hawblitzel, C. and Petrank, E. Automated Verification of Practical Garbage Collectors. *ACM SIGPLAN Notices*, 44(1):441–453, Jan. 2009.

Hawkes, B. Exploiting Native Client. Presented at Hacking at Random, Aug. 2009.

Heijmen, T. Radiation-induced Soft Errors in Digital Circuits. A Literature Survey. Tech. Report Nat.Lab. Unclassified Report 2002/828, Philips Electronics Nederland BV, Feb. 2002.

Heiser, G. Secure Embedded Systems need Microkernels. *USENIX ;login:*, 30(6): 9–13, Dec. 2005.

Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J. The Mungi Single-Address-Space Operating System. *Software—Practice & Privacy*, 28(9): 901–928, July 1998.

Heiser, G., Uhlig, V., and LeVasseur, J. Are Virtual-Machine Monitors Microkernels Done Right? *ACM SIGOPS Operating System Review*, 40(1):95–99, Jan. 2006.

Herder, J. N. Towards a True Microkernel Operating System. Master's Thesis, Dept. of Computer Science, Vrije Universiteit Amsterdam, Feb. 2005.

Herder, J. N., Bos, H., and Tanenbaum, A. S. A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. Tech. Report IR-CS-018, Vrije Universiteit Amsterdam, Jan. 2006.

Hildebrand, D. An Architectural Overview of QNX. In *Proc. of the USENIX Ws. on Micro-kernels and Other Kernel Architectures*, pp. 113–126, Apr. 1992.

Hohmuth, M., Peter, M., Härtig, H., and Shapiro, J. S. Reducing TCB Size by using Untrusted Components: Small Kernels versus Virtual-Machine Monitors. In *Proc. 11th ACM SIGOPS European Ws.*, paper no. 22, Sept. 2004.

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. Software Rejuvenation: Analysis, Module and Applications. In *Proc. 25th Int'l Symp. on Fault-Tolerant Computing*, pp. 381–390, June 1995.

Hunt, G. Creating User-Mode Device Drivers with a Proxy. In *Proc. 1st USENIX Windows NT Ws.*, paper no. 8, Aug. 1997.

Hunt, G., Aiken, M., Fähndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., and Wobber, T. Sealing OS processes to Improve Dependability and Safety. In *Proc. 2nd EuroSys Conf.*, pp. 341–354, Mar. 2007.

Hunt, G. C. and Larus, J. R. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating System Review*, 41(2):37–49, Apr. 2007.

Institute of Electrical and Electronics Engineers. *Information Technology-Portable Operating System Interface (POSIX ®)—Part 1: System Application: Program Interface (API) [C Language]*. IEEE, 1st ed., Oct. 1990. Also known as IEEE Std. 1003.1-1990. Approved as Int'l Std. ISO/IEC 9945-1:1990.

Intel Corp. Intel Virtualization Technology for Directed I/O. Architecture Specification, Rev. 1.2, Order Number D51397-004, Sept. 2008.

Ishikawa, H., Nakajima, T., Oikawa, S., and Hirotsu, T. Proactive Operating System Recovery. Presented at 20th ACM Symp. on Operating Systems Principles, Poster Session, Oct. 2005.

Jarboui, T., Arlat, J., Crouzet, Y., Kanoun, K., and Marteau, T. Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study. In *Proc. 9th Pacific Rim Int'l Symp. on Dependable Computing*, pp. 51–58, Dec. 2002.

Johnson, R. and Wagner, D. Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symp.*, pp. 119–134, Aug. 2004.

Kadav, A., Renzelmann, M. J., and Swift, M. M. Tolerating Hardware Device Failures in Software. In *Proc. 22nd ACM Symp. on Operating Systems Principles*, pp. 59–72, Oct. 2009.

Kalakech, A., Kanoun, K., Crouzet, Y., and Arlat, J. Benchmarking The Dependability of Windows NT4, 2000 and XP. In *Proc. 34th Int'l Conf. on Dependable Systems and Networks*, pp. 681–686, June/July 2004.

Kanawati, G. A., Kanawati, N. A., and Abraham, J. A. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Trans. on Computers*, 44(2):248–260, Feb. 1995.

Karp, A. H. Enforce POLA on Processes to Control Viruses. *Comm. of the ACM*, 46(12):27–29, Dec. 2003.

Kaspersky, K. and Chang, A. Remote Code Execution Through Intel CPU Bugs. Presented at Hack in the Box Security Conf., Oct. 2008.

Kelbley, J., Sterling, M., and Stewart, A. *Windows Server 2008 Hyper-V: Insiders Guide to Microsoft's Hypervisor*. Sybex, 1st ed., Apr. 2009.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the Linux Virtual Machine Monitor. In *Proc. 2007 Ottawa Linux Symp.*, pp. 225–230, June 2007.

Klein, G. Operating System Verification—An Overview. *Sādhanā*, 34(1):27–69, Feb. 2009.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe,

D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proc. 22nd ACM Symp. on Operating Systems Principles*, pp. 207–220, Oct. 2009.

Koopman, P. and DeVale, J. Comparing the Robustness of POSIX Operating Systems. In *Proc. 29th Int'l Symp. on Fault-Tolerant Computing*, pp. 30–37, June 1999.

Krioukov, A., Bairavasundaram, L. N., Goodson, G. R., Srinivasan, K., Thelen, R., Arpaci-Dusseau, A. C., and Arpaci-Dussea, R. H. Parity lost and Parity Regained. In *Proc. 6th USENIX Conf. on File and Storage Technologies*, pp. 1–15, Feb. 2008.

Kroah-Hartman, G., Corbet, J., and McPherson, A. Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. The Linux Foundation, White Paper, Aug. 2009. Available online: http://www.linuxfoundation.org/publications/whowriteslinux.pdf.

Kuznetsov, V., Chipounov, V., and Candea, G. Testing Closed-Source Binary Device Drivers with DDT. In *Proc. 2010 USENIX Annual Technical Conf.*, pp. 159–172, June 2010.

L4Ka Project. The L4Ka::Pistachio Microkernel. White Paper, May 2003. Available online: http://l4ka.org/projects/pistachio/pistachio-whitepaper.pdf.

Larus, J. Spending Moore's Dividend. *Comm. of the ACM*, 52(5):62–69, May 2009.

Le Mignot, G. The GNU Hurd. Extended Abstract of Talk at Libre Software Meeting, Dijon, France, July 2005.

Lee, I. and Iyer, R. K. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *Proc. 23rd Int'l Symp. on Fault-Tolerant Computing*, pp. 20–29, June 1993.

Lee, I. and Iyer, R. K. Software Dependability in the Tandem GUARDIAN System. *IEEE Trans. on Software Engineering*, 21(5):455–467, May 1995.

Lee, M., Wieland, P., Ganapathy, N., Erlingsson, U., Abadi, M., and Richardson, J. Split User-Mode/Kernel-Mode Device Driver Architecture. Microsoft Corp., United States Patent Application, Pub. No. US 2009/0138625 A1, May 2009.

Lemos, R. Device Drivers Filled with Flaws, Threaten Security. SecurityFocus, May 2005. Available online: http://www.securityfocus.com/news/11189.

Leslie, B., FitzRoy-Dale, N., and Heiser, G. Encapsulated User-level Device Drivers in the Mungi Operating System. In *Proc. 1st Int'l Ws. on Object Systems and Software Architectures*, paper no. 20, Jan. 2004.

Leslie, B., Chubb, P., Fitzroy-Dale, N., Gotz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y.-T., Elphinstone, K., and Heiser, G. User-Level Device Drivers: Achieved Performance. *J. Computer Science and Technology*, 20(5):654–664, Sept. 2005a.

Leslie, B., van Schaik, C., and Heiser, G. Wombat: A Portable User-Mode Linux for Embedded Systems. Presented at 6th Linux.Conf.Au, Apr. 2005b.

Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J. Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th Symp. on Operating System Design and Implementation*, pp. 17–30, Dec. 2004.

LeVasseur, J., Uhlig, V., Yang, Y., Chapman, M., Chubb, P., Leslie, B., and Heiser, G. Pre-Virtualization: Soft layering for Virtual Machines. In *Proc. 13th Asia-Pacific Computer Systems Architecture Conf.*, pp. 1–9, Aug. 2008.

Leveson, N. G. and Turner, C. S. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

Liedtke, J. Improving IPC by Kernel Design. In *Proc. 14th ACM Symp. on Operating Systems Principles*, pp. 175–188, Dec. 1993.

Liedtke, J. On μ-Kernel Construction. In *Proc. 15th ACM Symp. on Operating Systems Principles*, pp. 237–250, Dec. 1995.

Linnenbank, N. Implementing the Intel PRO/1000 on MINIX 3. Tech. Report IR-CS-052, Vrije Universiteit Amsterdam, Dec. 2009.

Linux Kernel Organization, Inc. The Linux 2.6 Kernel Archives, Dec. 2009. Available online: http://www.kernel.org/pub/linux/kernel/v2.6/.

Lions, J. *A Commentary on the Sixth Edition UNIX Operating System*. Dept. of Computer Science, The University of New South Wales, 1977.

Mancina, A., Faggioli, D., Lipari, G., Herder, J. N., Gras, B., and Tanenbaum, A. S. Enhancing a Dependable Multiserver Operating System with Temporal Protection via Resource Reservations. *Real-Time Systems*, 43(2):177–210, Oct. 2009.

Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. Devil: an IDL for Hardware Programming. In *Proc. 4th Symp. on Operating System Design and Implementation*, pp. 17–30, Oct. 2000.

Merlo, E., Dagenais, M., Bachand, P., Sormani, J. S., Gradara, S., and Antoniol, G. Investigating Large Software System Evolution: The Linux Kernel. In *Proc. 26th Int'l Computer Software and Applications Conf.*, pp. 421–426, Aug. 2002.

Microsoft Corp. Architecture of the User-Mode Driver Framework. Windows Hardware Developer Central (WHDC), White Paper, Feb. 2007. Available online: http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx.

Microsoft Corp. Windows Driver Kit. Windows Hardware Developer Central (WHDC), Dec. 2009. Available online: http://www.microsoft.com/wdk.

Mitchell, J. G. JavaOS: Back to the Future (Abstract). In *Proc. 2nd Symp. on Operating System Design and Implementation*, p. 1, Oct. 1996.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. on Software Engineering and Methodology*, 11(3):309–346, July 2002.

Modine, A. Linus Calls Linux 'Bloated and Huge': No Diet Plan in Sight. The Register, Sept. 2009. Available online: http://www.theregister.co.uk/2009/09/22/linus_torvalds_linux_bloated_huge/.

Möller, U. Optimizing the Desktop using Sun XVM VirtualBox. Sun Microsystem, Inc., Sun BluePrints Online, Part No. 820-7121-10, Rev. 1.0, Nov. 2008.

Moore, G. E. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):114ff, Apr. 1965.

Moraes, R., Barbosa, R., Durães, J., Mendes, N., Martins, E., and Madeira, H. Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent? In *Proc. 6th European Dependable Computing Conf.*, pp. 53–64, Oct. 2006.

Mourad, S. and Andrews, D. On the Reliability of the IBM MVS/XA Operating System. *IEEE Trans. on Software Engineering*, 13(10):1135–1139, Oct. 1987.

Murphy, B. Automating Software Failure Reporting. *ACM Queue*, 2(8):42–48, Nov. 2004.

Myhrvold, N. P. The Next Fifty Years of Software (Software: The Crisis Continues!). Presented at ACM97 Conf., Mar. 1997.

Nelson, V. P. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Computer*, 23(7):19–25, July 1990.

Ng, W. T. and Chen, P. M. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proc. 29th Int'l Symp. on Fault-Tolerant Computing*, pp. 76–83, June 1999.

Orgovan, V. and Dykstra, W. Online Crash Analysis—Higher Quality At Lower Cost. Presented at 13th Microsoft Windows Hardware Engineering Conf., May 2004.

Ostrand, T. J. and Weyuker, E. J. The Distribution of Faults in a Large Industrial Software System. In *Proc. 12th Int'l Symp. on Software Testing and Analysis*, pp. 55–64, July 2002.

Ostrand, T. J., Weyuker, E. J., and Bell, R. M. Predicting the Location and Number

of Faults in Large Software Systems. *IEEE Trans. on Software Engineering*, 31 (4):340–355, Apr. 2005.

Ou, G. Surge of Killer Device Drivers Leave no OS Safe. ZDNet, Oct. 2006. Available online: http://blogs.zdnet.com/Ou/?p=347.

Ozment, A. and Schechter, S. E. Milk or Wine: Does Software Security Improve with Age? In *Proc. 15th USENIX Security Symp.*, pp. 93–104, July/Aug. 2006.

Padioleau, Y., Lawall, J. L., and Muller, G. Understanding Collateral Evolution in Linux Device Drivers. In *Proc. 1st EuroSys Conf.*, pp. 59–71, Apr. 2006.

Pfitzmann, B. and Stüble, C. PERSEUS: A Quick Open-Source Path to Secure Signatures. In *Proc. 2nd Ws. on Microkernel-Based Systems*, paper no. 14, Oct. 2001.

Popek, G. J. and Goldberg, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Comm. of the ACM*, 17(7):412–421, July 1974.

Poulsen, K. Tracking the Blackout Bug. SecurityFocus, Apr. 2004. Available online: http://www.securityfocus.com/news/8412.

Renesse, R. v., Staveren, H. v., and Tanenbaum, A. S. Performance of the World's Fastest Distributed Operating System. *ACM SIGOPS Operating System Review*, 22(4):25–34, Oct. 1988.

Renzelmann, M. J. and Swift, M. M. Decaf: Moving Device Drivers to a Modern Language. In *Proc. 2009 USENIX Annual Technical Conf.*, pp. 187–200, June 2009.

Rescorla, E. Security Holes... Who Cares? In *Proc. 12th USENIX Security Symp.*, pp. 75–90, Aug. 2003.

Ritchie, D. M. and Thompson, K. The UNIX Time-Sharing System. *Comm. of the ACM*, 17(7):365–375, July 1974.

Rosenblum, M. and Garfinkel, T. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, 38(5):39–47, May 2005.

Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. Dingo: Taming Device Drivers. In *Proc. 4th EuroSys Conf.*, pp. 275–288, Apr. 2009a.

Ryzhyk, L., Chubb, P., Kuz, I., Sueur, E. L., and Heiser, G. Automatic Device Driver Synthesis with Termite. In *Proc. 22nd ACM Symp. on Operating Systems Principles*, pp. 73–86, Oct. 2009b.

Saltzer, J. and Schroeder, M. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-end Arguments in System Design. *ACM Trans. on Computer Systems*, 2(4):277–288, Nov. 1984.

Schach, S., Jin, B., Wright, D., Heller, G. Z., and Offutt, A. J. Maintainability of the Linux Kernel. *IEE Proceedings – Software*, 149(1):18–23, Feb. 2002.

Schlichting, R. D. and Schneider, F. B. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. on Computer Systems*, 1 (3):222–238, Aug. 1983.

Seawright, L. H. and MacKinnon, R. A. VM/370—A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, Jan.–Mar. 1979.

Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. 2nd Symp. on Operating System Design and Implementation*, pp. 213–227, Oct. 1996.

Shapiro, J., Doerrie, M. S., Northup, E., Sridhar, S., and Miller, M. Towards a Verified, General-Purpose Operating System Kernel. In *Proc. 1st Ws. on Operating Systems Verification*, pp. 1–18, Oct. 2004. NICTA Formal Methods Program.

Shapiro, J. S. Vulnerabilities in Synchronous IPC Designs. In *Proc. 2003 IEEE Symp. on Security and Privacy*, pp. 251–262, May 2003.

Shmerling, L. Writing a Device Driver Made Easy: WinDriver—The "Java" of Device Drivers. *Dedicated Systems Magazine*, pp. 52–53, Q3 2001.

Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing TCB Complexity for Security-sensitive Applications: Three Case Studies. In *Proc. 1st EuroSys Conf.*, pp. 161–174, Apr. 2006.

Sivathanu, G., Wright, C. P., and Zadok, E. Ensuring Data Integrity in Storage: Techniques and Applications. In *Proc. 2005 ACM Ws. on Storage Security and Survivability*, pp. 26–36, Nov. 2005.

Slashdot. Andy Tanenbaum Releases Minix 3. Slashdot forum thread, Oct. 2005. Available online: http://slashdot.org/article.pl?sid=05/10/24/1049200.

Slashdot. Microkernel: The Comeback? Slashdot forum thread, May 2006. Available online: http://slashdot.org/article.pl?sid=06/05/08/1058248.

Slashdot. The Great Microkernel Debate Continues. Slashdot forum thread, Jan. 2008. Available online: http://slashdot.org/article.pl?sid=08/01/31/1617254.

Small, C. and Seltzer, M. A Comparison of OS Extension Technologies. In *Proc. 1996 USENIX Annual Technical Conf.*, pp. 41–54, Jan. 1996.

Smith, J. and Nair, R. The Architecture of Virtual Machines. *IEEE Computer*, 38 (5):32–38, May 2005.

Spear, M. F., Roeder, T., Hodson, O., Hunt, G. C., and Levi, S. Solving the Starting Problem: Device Drivers as Self-describing Artifacts. In *Proc. 1st EuroSys Conf.*, pp. 45–57, Apr. 2006.

Stevenson, J. M. and Julin, D. P. Mach-US: UNIX on Generic OS Object Servers. In *Proc. 1995 USENIX Technical Conf.*, pp. 119–130, Jan. 1995.

Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. 2001 USENIX Annual Technical Conf.*, pp. 1–14, June 2001.

Sullivan, M. and Chillarege, R. Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems. In *Proc. 21st Int'l Symp. on Fault-Tolerant Computing*, pp. 2–9, June 1991.

Sun, J., Yuan, W., Kallahalla, M., and Islam, N. HAIL: A Language for Easy and Correct Device Access. In *Proc. 5th ACM Int'l Conf. on Embedded Software*, pp. 1–9, Sept. 2005.

Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the Reliability of Commodity Operating Systems. *ACM Trans. on Computer Systems*, 23(1):77–110, Feb. 2005.

Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering Device Drivers. *ACM Trans. on Computer Systems*, 24(4):333–360, Nov. 2006.

Ta-Shma, P., Laden, G., Ben-Yehuda, M., and Factor, M. Virtual Machine Time Travel using Continuous Data Protection and Checkpointing. *ACM SIGOPS Operating System Review*, 42(1):127–134, Jan. 2008.

Tanenbaum, A. S. A UNIX Clone with Source Code for Operating Systems Courses. *ACM SIGOPS Operating System Review*, 21(1):20–29, Jan. 1987.

Tanenbaum, A. S. and Woodhull, A. S. *Operating Systems Design and Implementation*. Prentice Hall, 3rd ed., Jan. 2006.

Thakur, A., Iyer, R., Young, L., and Lee, I. Analysis of Failures in the Tandem NonStop-UX Operating System. In *Proc. 6th Int'l Symp. on Software Reliability Engineering*, pp. 40–50, Oct. 1995.

Thompson, K. Reflections on Trusting Trust. *Comm. of the ACM*, 27(8):761–763, Aug. 1984. ACM Turing Award Lecture.

Torvalds, L. What Would You Like to See Most in MINIX? Message to the Usenet Newsgroup comp.os.minix, Aug. 1991.

Torvalds, L. and Diamond, D. *Just for Fun: The Story of an Accidental Revolutionary*. HarperCollins, 1st ed., May 2001.

Vochteloo, J., Russell, S., and Heiser, G. Capability-Based Protection in the Mungi Operating System. In *Proc. 3rd Int'l Ws. on Object Orientation in Operating Systems*, pp. 108–115, Dec. 1993.

Vrije Universiteit Amsterdam. The MINIX 3 SVN Repository, Dec. 2009. Available

online: https://gforge.cs.vu.nl/svn/minix/trunk/src/.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proc. 14th ACM Symp. on Operating Systems Principles*, pp. 203–216, Dec. 1993.

Waldspurger, C. A. Memory Resource Management in VMware ESX Server. In *Proc. 5th Symp. on Operating System Design and Implementation*, pp. 181–194, Dec. 2002.

Weicker, R. P. Dhrystone: A Synthetic Systems Programming Benchmark. *Comm. of the ACM*, 27(10):1013–1030, Oct. 1984.

Whitaker, A., Shaw, M., and Gribble, S. D. Denali: A Scalable Isolation Kernel. In *Proc. 10th ACM SIGOPS European Ws.*, pp. 10–15, July 2002.

Willmann, P., Rixner, S., and Cox, A. L. Protection Strategies for Direct Access to Virtualized I/O Devices. In *Proc. 2008 USENIX Annual Technical Conf.*, pp. 15–28, June 2008.

Wong, K. Mac OS X on L4. Bachelor's Thesis, The University of New South Wales, Dec. 2003.

Xu, H., Du, W., and Chapin, S. J. Detecting Exploit Code Execution in Loadable Kernel Modules. In *Proc. 20th Annual Computer Security Applications Conf.*, pp. 101–110, Dec. 2004.

Xu, J., Kalbarczyk, Z., and Iyer, R. K. Networked Windows NT System Field Failure Data Analysis. In *Proc. 6th Pacific Rim Int'l Symp. on Dependable Computing*, pp. 178–185, Dec. 1999.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. 2009 IEEE Symp. on Security and Privacy*, pp. 79–93, May 2009.

Zamfir, C. and Candea, G. Execution Synthesis: A Technique for Automated Software Debugging. In *Proc. 5th EuroSys Conf.*, pp. 321–334, Apr. 2010.

Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. SafeDrive: Safe and Recoverable Extensions using Language-based Techniques. In *Proc. 7th USENIX Symp. on Operating System Design and Implementation*, pp. 45–60, Nov. 2006.

List of Citations

Abadi et al. [2005], 105
Accetta et al. [1986], 118
Adams and Agesen [2006], 108
Advanced Micro Devices, Inc. [2009], 30, 127
Ahmad [2008], 90
Aiken et al. [2006], 113, 114, 123
Albinet et al. [2004], 76
Appleton [2003], 96
Arlat et al. [2002], 76
Armand [1991], 118
Avižienis et al. [2004], 1, 16, 35, 128
Ball et al. [2006], 6, 112
Barbou des Places et al. [1996], 111
Barham et al. [2003], 109
Bartlett [1981], 36
Basili and Perricone [1984], 6
Baumann et al. [2007], 14, 59
Baumann et al. [2009], 139
Bellard [2005], 108
Bershad et al. [1990], 102
Bershad et al. [1995], 112
Blair et al. [1992], 2
Bos and Samwel [2002], 112
Boutin [2004], 1
Butt et al. [2009], 122, 137
Candea et al. [2004], 60
Castro et al. [2009], 78, 101, 105, 106, 123, 131
Chase et al. [1994], 102
Chillarege et al. [1992], 77
Chipounov and Candea [2010], 112
Chiueh et al. [1999], 102
Chou et al. [2001], 4–6, 126
Chou [1997], 33, 35, 77, 128
Christmansson and Chillarege [1996], 78
Chubb [2004], 119, 121, 134
Conway and Edwards [2004], 115
Coverity, Inc. [2008], 35, 128
Creasy [1981], 108
David and Campbell [2007], 36
David et al. [2008], 102
Department of Defense [1985], 26
DiBona and Ockman [1999], 20
Dinh-Trong and Bieman [2005], 6, 7, 126
Dowson [1997], 2
Durães and Madeira [2002], 76
Durães and Madeira [2006], 78
Elphinstone and Götz [2004], 120, 124
Endres [1975], 6
Erlingsson et al. [2005], 109
Erlingsson et al. [2006], 105
Fähndrich et al. [2006], 113
Feldman [2009], 139
Fenton and Ohlsson [2000], 6
Fraser et al. [2004], 109
Ganapathi and Patterson [2005], 3, 126
Ganapathi et al. [2006], 5, 10, 39, 40
Ganapathy et al. [2008], 120, 121, 123

Ganev et al. [2004], 102, 107
Garfinkel [2005], 2
Gasser [1988], 120
Gefflaut et al. [2000], 17, 117, 118, 120, 133, 138
Giuffrida and Tanenbaum [2009], 138
Glerum et al. [2009], 4, 6, 14, 126
Glew and Morrisett [1999], 113
Godfrey and Tu [2000], 99
Golub et al. [1990], 118
Govindavajhala and Appel [2003], 113
Gray and Siewiorek [1991], 7, 13, 127
Gray [1986], 33, 35, 36, 77, 128
Gray [1990], 3, 126
Gu et al. [2003], 76
Gunawi et al. [2007], 70
Guo and Engler [2009], 8
Härtig et al. [1997], 10, 16, 17, 110, 117, 118, 120, 133, 138
Härtig et al. [1998], 110, 120
Härtig et al. [2003], 120, 124
Haeberlen et al. [2000], 17, 120, 133
Hand et al. [2005], 107
Hatton [1997], 6, 126
Hawblitzel and Petrank [2009], 113
Hawkes [2009], 105
Heijmen [2002], 59
Heiser et al. [1998], 102
Heiser et al. [2006], 107
Heiser [2005], 113, 134
Herder et al. [2006], 90
Herder [2005], 11, 20, 42, 127
Hildebrand [1992], 118
Hohmuth et al. [2004], 109, 110
Huang et al. [1995], 59
Hunt (pers. comm.) [2010], 5, 59
Hunt and Larus [2007], 112, 113, 123
Hunt et al. [2007], 113
Hunt [1997], 118
Institute of Electrical and Electronics Engineers [1990], 20
Intel Corp. [2008], 30, 127
Ishikawa et al. [2005], 59
Jarboui et al. [2002], 76
Johnson and Wagner [2004], 39
Kadav et al. [2009], 5, 39
Kalakech et al. [2004], 76
Kanawati et al. [1995], 77
Karp [2003], 17
Kaspersky and Chang [2008], 119
Kelbley et al. [2009], 108
Kivity et al. [2007], 108
Klein et al. [2009], 113, 119
Klein [2009], 27
Koopman and DeVale [1999], 76
Krioukov et al. [2008], 71
Kroah-Hartman et al. [2009], 99
Kuznetsov et al. [2010], 6
L4Ka Project [2003], 119
Larus [2009], 17, 139
LeVasseur (pers. comm.) [2006], 107, 110
LeVasseur et al. [2004], 109, 110
LeVasseur et al. [2008], 110
Lee and Iyer [1993], 3
Lee and Iyer [1995], 36
Lee et al. [2009], 121
Lemos [2005], 17
Leslie et al. [1996], 102
Leslie et al. [2004], 102, 103
Leslie et al. [2005a], 17, 117, 119, 121, 133, 137
Leslie et al. [2005b], 118
Leveson and Turner [1993], 2
Le Mignot [2005], 118
Liedtke [1993], 16, 17, 31, 120, 133
Liedtke [1995], 17, 21, 32, 47, 120, 123, 124, 133
Linnenbank [2009], 92, 98
Linux Kernel Organization, Inc. [2009], 99
Lions [1977], 19

Möller [2008], 108
Mérillon et al. [2000], 115
Mancina et al. [2009], 21
Merlo et al. [2002], 5
Microsoft Corp. [2007], 119, 134, 137
Microsoft Corp. [2009], 107
Mitchell [1996], 112
Mockus et al. [2002], 6
Modine [2009], v, 7, 126, 134
Moore [1965], 32, 127
Moraes et al. [2006], 76
Mourad and Andrews [1987], 36
Murphy [2004], 4, 5
Myhrvold [1997], 8
Nelson [1990], 12, 127
Ng and Chen [1999], 77
Orgovan and Dykstra [2004], 3, 5, 39, 126
Ostrand and Weyuker [2002], 6
Ostrand et al. [2005], 6, 126
Ou [2006], 17
Ozment and Schechter [2006], 17
Padioleau et al. [2006], 7, 126
Pfitzmann and Stüble [2001], 110
Popek and Goldberg [1974], 108
Poulsen [2004], 2
Renesse et al. [1988], 31
Renzelmann and Swift [2009], 122
Rescorla [2003], 8
Ritchie and Thompson [1974], 1, 19
Rosenblum and Garfinkel [2005], 107
Ryzhyk et al. [2009a], 5, 40, 115, 117, 132
Ryzhyk et al. [2009b], 115, 116
Saltzer and Schroeder [1975], 8, 38, 126
Saltzer et al. [1984], 66
Schach et al. [2002], 7
Schlichting and Schneider [1983], 36, 65, 128
Seawright and MacKinnon [1979], 108
Seltzer et al. [1996], 105
Shapiro et al. [2004], 113
Shapiro [2003], 40
Shmerling [2001], 119
Singaravelu et al. [2006], 16, 120
Sivathanu et al. [2005], 71
Slashdot [2005], 136
Slashdot [2006], 136
Slashdot [2008], 136
Small and Seltzer [1996], 113
Smith and Nair [2005], 107
Spear et al. [2006], 114, 132
Stevenson and Julin [1995], 118
Sugerman et al. [2001], 108
Sullivan and Chillarege [1991], 39, 77
Sun et al. [2005], 115
Swift et al. [2005], 77, 78, 102, 103, 123, 131
Swift et al. [2006], 31, 36, 104, 138
Ta-Shma et al. [2008], 109
Tanenbaum and Woodhull [2006], 20
Tanenbaum [1987], 20, 127
Thakur et al. [1995], 3, 4, 126
Thompson [1984], 105
Torvalds and Diamond [2001], 20
Torvalds [1991], 20
Vochteloo et al. [1993], 102
Vrije Universiteit Amsterdam [2009], 96, 139
Wahbe et al. [1993], 104
Waldspurger [2002], 108
Weicker [1984], 93
Whitaker et al. [2002], 109
Willmann et al. [2008], 49
Wong [2003], 40
Xu et al. [1999], 3, 14, 59, 126
Xu et al. [2004], 59
Yee et al. [2009], 105
Zamfir and Candea [2010], 112
Zhou et al. [2006], 78, 112

Abbreviations

A	availability
ACL	access control list
ACPI	advanced configuration and power interface
API	application programming interface
ATA	advanced technology attachment
BIOS	basic input/output system
CFI	control flow integrity
CPU	central processing unit
DEV	device exclusion vector
DHCP	dynamic host configuration protocol
DMA	direct memory access
DNS	domain name system
DPI	driver programming interface
FS	file system
HAL	hardware abstraction layer
IA-32	Intel architecture, 32-bit
IDE	integrated drive electronics
IDL	interface definition language
IFS	installable file system
I/O	input/output
IOCTL	I/O control operation
IOMMU	I/O memory management unit
IPC	interprocess communication
IRQ	interrupt request
ISA	instruction set architecture
KMDF	kernel-mode driver framework
LoC	lines of code
MD5	message-digest algorithm 5
MLFQ	multilevel-feedback-queue scheduler
MMU	memory management unit
MSI	message-signaled interrupt
MTTF	mean time to failure

MTTR	mean time to recover
NDIS	network driver interface specification
NOP	no-operation
OS	operating system
PC	personal computer
PCI	peripheral component interconnect
PIC	programmable interrupt controller
PID	process identifier
PnP	plug and play
POLA	principle of least authority
POSIX	portable operating system interface
RAID	redundant array of inexpensive disks
RAM	random access memory
RPC	remote procedure call
SASOS	single-address-space operating system
SATA	serial advanced technology attachment
SCSI	small computer system interface
SFI	software-based fault isolation
SHA-1	secure-hash algorithm 1
SIP	software-isolated process
SVN	subversion version control system
SWIFI	software-implemented fault injection
TCB	trusted computing base
TCP	transmission control protocol
TLB	translation lookaside buffer
UDP	user datagram protocol
UMDF	user-mode driver framework
USB	universal serial bus
VFS	virtual file system
VM	virtual machine *or* virtual memory
VMM	virtual machine monitor
WDF	Windows driver foundation
WDM	Windows driver model
x86	Intel 8086 architecture

Publications

Parts of this thesis have been published before. The five key publications that form the basis of most of the material presented here are listed below. Pointers to additional publications are given on page *vi*.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. 6th European Dependable Computing Conf. (EDCC'06)*, pages 3–12, Coimbra, Portugal, Oct. 2006.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers In *Proc. 37th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN'07)*, pages 41–50, Edinburgh, United Kingdom, June 2007.[1]

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Countering IPC Threats in Multiserver Operating Systems. In *Proc. 14th IEEE Pacific Rim Int'l Symp. on Dependable Computing (PRDC'08)*, pages 112–121, Taipei, Taiwan, Dec. 2008.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault Isolation for Device Drivers. In *Proc. 39th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN'09)*, pages 33–42, Estoril-Lisbon, Portugal, July 2009.

Jorrit N. Herder, David C. van Moolenbroek, Raja Appuswamy, Bingzheng Wu, Ben Gras, and Andrew S. Tanenbaum. Dealing with Driver Failures in the Storage Stack. In *Proc. 4th Latin-American Symp. on Dependable Computing (LADC'09)*, pages 119–126, João Pessoa, Paraíba, Brazil, Sept. 2009.[2]

[1] This paper won the William C. Carter Award.
[2] This paper was awarded 'best paper.'

Biography

Jorrit Niek Herder (1978) was born in Alkmaar, Netherlands. After completing secondary education at Petrus Canisius College in Alkmaar, he obtained a Propaedeutic degree in Building Architecture from Delft University of Technology, Netherlands in 1997. Looking for more technical challenges he took up courses in Applied Physics before switching to Computer Science at Vrije Universiteit Amsterdam, Netherlands in 1999. Around the same time, he became intrigued by the Internet and started a one-man business in web development and consulting that he successfully ran next to his studies. In 2005, he received a M.Sc. degree in Computer Science (cum laude) from Vrije Universiteit Amsterdam with a dual specialization in Software Engineering and Internet & Web Technology.

Between 2005 and 2009 Jorrit conducted his Ph.D. research at Vrije Universiteit Amsterdam, aiming to build a highly dependable operating system. In this role, he was closely involved in the design and implementation of the MINIX 3 operating system (OS). He is the principle architect of the MINIX 3 user-level driver framework and the system's fault-tolerance mechanisms. This work has led to over 20 publications in international conferences, journals, magazines, and technical reports. The MINIX 3 OS was also released online and still attracts about 25,000 visitors and over 10,000 downloads each month. In 2007, he received the William C. Carter Award for his contributions to the field of dependable computing.

During his Ph.D. Jorrit has also been involved in various other activities. From 2006 until 2009, he managed the graduate course Practical Work Operating Systems and helped supervising B.Sc. and M.Sc. thesis projects. He set up and mentored projects for Google Summer of Code 2008 and 2009, one of which was published at a dependability conference and won the 'best paper' award. In addition, he did several internships and research visits. In 2007, he was a research intern at Microsoft Research in Mountain View, California and visited Scuola Superiore Sant'Anna in Pisa, Italy. In 2008, he interned as a software engineer at Google in New York, New York and visited National ICT Australia in Sydney, Australia.

In the second half of 2009, Jorrit held a postdoctoral research position at Vrije Universiteit Amsterdam. As of March 2010, he has been working as a software engineer for Google in Sydney, Australia.