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.