Effectiveness of Driver Isolation and Testing in User Space

Oliver Schwahn*
TU Darmstadt, Germany
os@cs.tu-darmstadt.de

1 The Device Drivers Problem

Device Drivers have often been identified as a major source of reliability issues in commodity operating systems (OSs). Static analyses of the Linux kernel show that the majority of detected bugs reside in driver code [2, 10]. These findings suggest that driver code is particularly prone to errors or software bugs. One explanation for this observation is that driver development in kernel space is generally a challenging task requiring experience and expert knowledge. Drivers are inherently complex due to their asynchronous nature, the close interaction with kernel and hardware interfaces, and the often implemented support for different hardware devices in one driver. Moreover, kernel space development suffers from a lack of tool support compared to user space development [7].

Bugs in driver code have severe consequences for the whole system. In most (monolithic) OSs, drivers are heavily coupled extensions of the kernel that are linked to the kernel binary at compile- or run-time. They share the same address space with the kernel and operate with highest privileges. Hence, buggy (or malicious) drivers can read/write arbitrary memory and program hardware devices and buses with unforeseeable effects on the system, ranging from crashes to permanent data corruption.

Existing approaches that strive to improve the situation are manifold: (1) Driver isolation approaches (e.g. [1, 15]) attempt to contain faults and there effects such that the kernel is not affected. (2) Driver recovery (e.g. [14]) attempts to detect driver failures and (transparently) recover to a clean state. (3) Driver synthesis, new architectures and languages (e.g. [12, 13]) attempt to make writing drivers easier. (4) Testing approaches (e.g. [11]) try to make testing drivers easier and more comprehensive. However, none of the existing approaches is widely applied in practice although isolation approaches, which attempt to break the heavy coupling between drivers and the kernel, promise ad-hoc reliability improvements.

2 Research Questions

We plan to investigate why driver isolation techniques are not used in contemporary kernels and whether we can apply advances in testing techniques to driver testing to improve driver code quality.

*PhD advisor: Prof. Neeraj Suri, TU Darmstadt
Starting year: 2014

2.1 Effectiveness of Isolation

Relying on user space processes for isolating OS components is a natural choice as user space processes have existed for a long time in OSs. Their isolation capabilities are based on hardware supported address space separation and an unprivileged execution mode. Microkernel OSs such as the L4 family [6] make extensive use of user space processes for isolation. Approaches for moving drivers of monolithic systems into user space processes [1, 7, 8] follow the same idea by re-using existing isolation capabilities of the kernel rather than introducing new and complex mechanisms. Hence, we will focus on user space isolation.

2.1.1 Which Faults Can Be Isolated?

A natural question is how well a user space driver is isolated from the rest of the system, i.e., if a driver fault occurs, is the effect limited to the isolated driver or can the error propagate to affect other system components? The existing works only evaluate isolation properties against a small set of "expected" faults, but do not thouroughly investigate system behavior in the presence of different fault types.

We propose to study how effective isolation approaches are by performing fault injection experiments [3] on isolated and unisolated drivers. From such experiments, we hope to learn which techniques are effective in containing which types faults and why. Earlier studies identified different bug classes including protocol violations and concurrency bugs [12] as well as different classes of typical human programmer errors [9]. We expect different techniques to have different effectiveness for these fault classes. A preliminary study we performed has shown that systems relying on shared kernel data have high potential to corrupt kernel data as precise automated data checks are difficult. Moreover, protocol violation bugs are often not taken into account.

2.1.2 Isolation-Performance Trade-off

User space isolation approaches such as SUD [1] strive to isolate whole drivers. However, it remains unclear whether whole drivers or driver code fractions should be isolated for a practical system. Isolating drivers usually requires manual effort, e.g., rewriting code for new interfaces, and imposes overheads during runtime which are often considered too high for wide adoption, e.g., due to additional indirection and data synchronization. If isolating only half the code of a driver imposes only half the overhead in the common

case but offers good enough isolation properties due to a smart selection of isolated code, such a partitioning could be a viable option worth of systematic exploration.

We propose to use dynamic analysis to capture the overheads that a partitioning into kernel space and isolated user space would impose for drivers. Combining these results with a graph-based driver code model obtained from static analysis, we can formulate the driver partitioning problem as a linear optimization problem. Using a linear solver, we can explore the space of possible driver partitionings trading-off performance with amount of isolated code. Such an approach allows to fine-tune the isolation-performance trade-off to the operational environment and application requirements. Preliminary results we obtained using this approach and the Microdrivers framework [7] for driver synthesis suggest that such an approach is practicable.

2.2 Driver Testing

We argue that due to the involved overheads, complexity, and often unclear isolation capabilities, more efforts should be invested in improving driver code quality. Testing is a valuable quality assurance measure in software development. Since drivers are a critical part of the OS, they should be thoroughly tested. Unfortunately, driver testing has been recognized as being inherently difficult [4, 11].

2.2.1 Testing in User Space

Existing approaches for driver testing often rely on complex tool chains and are resource and time intensive. For instance, SymDrive [11], relies on a combination of driver source code annotation and transformation, vitalization, and symbolic execution for checking drivers against known bug patterns. However, a lightweight approach with fast feedback for developers that allows for traditional unit testing techniques in user space appears to be desirable.

We propose to treat drivers similarly to libraries in user space, where interfaces can be systematically tested, by applying techniques similar to those used for user space driver isolation systems. For instance, SUD [1] relies on User-Mode Linux (UML) [5] for providing kernel services in user space. The device hardware can be abstracted with traditional mocking or stubbing techniques or with device models constructed using machine learning [4] or generated from specifications [13], depending on the tests to perform. Such user space testing would make driver code accessible to the plethora of debugging and testing tools and techniques available for user space development.

2.2.2 Test Parallelization

Thorough testing requires large numbers of test cases, especially if a coverage criterion like branch coverage should be achieved. Traditional sequential test execution takes time and leaves available computing resources unused. With multi-core CPUs being the default, testing should ideally be performed in parallel, i.e., multiple tests should run in parallel without affecting the result validity. Previous work on parallel robustness testing [16] demonstrates that test parallelization can greatly improve test throughput. However, it has also shown that care must be taken to preserve result validity with increasing parallelism.

With driver tests being performed in user space similar to library tests, established techniques can be applied to drivers. To reduce the burden of writing parallelizable tests on the developer, we propose to rely on static dependency analysis to identify independent tests that can be executed in parallel without invalidating test results. We furthermore propose to study if using multiple threads for parallel execution instead of processes is feasible to reduce resource consumption and obtain even higher efficiency.

References

[1] Silas Boyd-wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *USENIX Annu. Tech. Conf.*

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proc. eighteenth ACM Symp. Oper. Syst. Princ. (SOSP '01).*

[3] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. 2015. Automatic fault injection for driver robustness testing. *Proc. 2015 Int. Symp. Softw. Test. Anal. - ISSTA 2015* (2015).

[4] Domenico Cotroneo, Luigi De Simone, Francesco Fucci, and Roberto Natella. 2015. MoIO: Run-time monitoring for I/O protocol violations in storage device drivers. In *26th Int. Symp. Softw. Reliab. Eng.*

[5] Jeff Dike. 2005. The User-mode Linux Kernel Home Page. (2005). http://user-mode-linux.sourceforge.net

[6] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 – What Have We Learnt in 20 Years of L4 Microkernels?. In *ACM SIGOPS Symp. Oper. Syst. Princ.*

[7] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. 2008. The Design and Implementation of Microdrivers. In *Proc. 13th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS XIII).*

[8] Ben Leslie, Peter Chubb, Nicholas Fitzroy-dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. 2005. User-level Device Drivers: Achieved Performance. *J. Comput. Sci. Technol.* (2005).

[9] R Natella, D Cotroneo, J A Duraes, and H S Madeira. 2013. On Fault Representativeness of Software Fault Injection. *Softw. Eng. Trans.* (2013).

[10] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS XVI).*

[11] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. 2012. SymDrive: Testing Drivers Without Devices. In *Proc. 10th USENIX Conf. Oper. Syst. Des. Implement. (OSDI'12).*

[12] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009. Dingo: Taming Device Drivers. In *ACM Eur. Conf. Comput. Syst. (EuroSys '09).*

[13] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic device driver synthesis with termite. In *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ. - SOSP '09.*

[14] M Swift, M Annamalai, B Bershad, and H Levy. 2006. Recovering Device Drivers. *ACM Trans. Comput. Syst.* (2006).

[15] Michael M Swift, Brian N Bershad, and Henry M Levy. 2003. Improving the Reliability of Commodity Operating Systems. In *Proc. Ninet. ACM Symp. Oper. Syst. Princ. (SOSP '03).*

[16] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. 2015. No PAIN, No Gain?: The Utility of PArallel Fault INjections. In *Proc. 37th Int. Conf. Softw. Eng. (ICSE '15).*