IMPROVING THE DEVELOPMENT AND TESTING OF DEVICE DRIVERS

by

Matthew J. Renzelmann

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 6/12/2013

The dissertation is approved by the following members of the Final Oral Committee:

Michael Swift, Assistant Professor, Computer Sciences

Remzi Arpaci-Dusseau, Professor, Computer Sciences

Somesh Jha, Professor, Computer Sciences

Ben Liblit, Associate Professor, Computer Sciences

Parmesh Ramanathan, Professor, Electrical and Computer Engineering

© Copyright by Matthew J. Renzelmann 2013
All Rights Reserved

To Jana, my wife, and to Michael and Linda, my parents.

ACKNOWLEDGMENTS

As I understand it, this section is where I spill my guts and thank everyone who made this document possible. I will do my best not to forget anyone, and if I do, I apologize.

First, I would like to thank my adviser, Michael Swift. Someone I know once characterized Mike as a "feedback machine: you put problems in, and feedback comes out." Mike, you have provided me with invaluable feedback over the years, and I feel very lucky to have worked with you. It has been a lot of fun, and if any prospective student is reading this, Mike is a pleasure to work with.

I would also like to thank my committee members: Remzi Arpaci-Dusseau, Somesh Jha, Ben Liblit, and Parmesh Ramanathan, both for taking the time to read this, and also for your help. Remzi, I enjoyed meeting you and Andrea during the department's 2005 Visit Weekend. You helped convince me to go to school here. Somesh, I enjoyed working with you on Microdrivers, and taking your class on security. Ben, your class on software artifacts was a lot of fun and I have since recommended it to several others. Parmesh, we have not yet met formally, but I certainly appreciate your help with my dissertation.

My stay at UW would have been much less enjoyable without some of the students with whom I've had the pleasure of interacting over the years: Arini Balakrishnan, Siddharth Barman, Theophilus Benson, Vinod Ganapathy, Neelam Goyal, Asim Kadav, Thawan Kooburat, Sankaralingam Panneerselvam, Kevin Roundy, Mohit Saxena, Swaminathan Sundararaman, Andres Tack, Adwait Tumbde, and Haris Volos. Your friendship and camaraderie have meant a lot to me, and I'm fortunate to have known you all.

Several of the projects described in this dissertation would not be possible without the help of others. On Decaf Drivers, I'd like to thank Vinod Ganapathy, and Arini Balakrishnan for your work on Microdrivers. Decaf would not have been possible otherwise. I must also thank Robert Grimm for your help with Jeannie and XTC, which were an integral part of Decaf. I also want to thank Vitaly Chipounov, for answering a number of questions I had about S^2E. Finally, I'd especially like to thank Asim Kadav for your collaboration on several projects, including both Decaf and SymDrive. Asim, you made trips to the office much more fun than they might otherwise have been, and I appreciate your friendship over the years.

During my time at UW, I participated in two internships, at Intel and at Microsoft Research. Both of these internships provided me with valuable experience. At Intel, it was a pleasure working with Doug Nelson on some interesting performance problems. At Microsoft, I'd like to thank Ed Nightingale and Jeremy Condit, with whom I had fun developing a new, cloud-based file system.

I also feel compelled to express a heartfelt thanks to the doctors and staff at the University of Wisconsin Hospital, who helped ensure I did not perish from appendicitis one week before the 2012 OSDI paper deadline. That would have been rather inconvenient. Although our submission was accepted, I hope my heavily-medicated state of mind did not show through in the writing.

Muddling my way through a dissertation would have been much less fun if it were not for my closest friends, to whom I owe a considerable debt of gratitude: Ben and Brenna Yang, Kevin Springborn, Brooks and Jill Wheatley, Jeremiah and Kimberly VanDamme. Knowing you all has been a real pleasure, and I hope our friendships endure for many years to come.

I also must thank my parents, Michael and Linda, without whom this dissertation most certainly would not exist. I'm deeply indebted to you both and am fortunate to have you as parents, and I don't tell you how much I love you often enough. I'm sure you're relieved that your kid is finally, after all these years, done with school. I hope that you received your money's worth: all I've really learned is that my education is just beginning.

Finally, I would like to thank my wife, Jana, for her patience while I finished this project. I know I was a bit optimistic when I delivered my initial estimate for when I'd be done – please forgive me. I love you very much and am lucky to have you. Fortunately (?), I'm about to abandon my dissertation,[1] which means we can soon start a new life together.

[1] *Dissertations are not finished; they are abandoned.* —Fred Brooks

CONTENTS

Contents iv

List of Tables vi

List of Figures vii

Abstract viii

1 Introduction 1
- *1.1 Motivation* . . . 1
- *1.2 Thesis and Contributions* . . . 6
- *1.3 Dissertation Organization* . . . 9

2 Background and Related Work 10
- *2.1 Device Drivers* . . . 10
- *2.2 Reliability* . . . 14
- *2.3 Static Analysis* . . . 17
- *2.4 Symbolic Execution* . . . 19
- *2.5 Conclusions* . . . 21

3 Decaf Drivers 22
- *3.1 Design of Decaf Drivers* . . . 23
- *3.2 Implementation* . . . 29
- *3.3 Experimental Results* . . . 38
- *3.4 Case Study: E1000 Network Driver* . . . 43
- *3.5 Summary* . . . 49

4 SymDriveProto: Testing Drivers without Devices Prototype 50
- *4.1 Motivation and Background* . . . 50
- *4.2 SDP Introduction* . . . 54
- *4.3 Design* . . . 55
- *4.4 Implementation* . . . 62
- *4.5 Evaluation* . . . 64
- *4.6 Limitations* . . . 69
- *4.7 Conclusions* . . . 71

5 SymDrive: Testing Drivers without Devices 72

5.1 Design . . . 73
5.2 Checkers . . . 84
5.3 Evaluation . . . 87
5.4 Conclusions . . . 96

6 SymDriveCluster: Automating SymDrive 97

6.1 Motivation . . . 98
6.2 Automatically Testing one Driver . . . 101
6.3 Testing Many Drivers . . . 107
6.4 Evaluation . . . 111
6.5 Summary . . . 114

7 Lessons Learned and Future Work 115

7.1 Lessons Learned . . . 115
7.2 Future Work . . . 123

8 Conclusions 131

A Checkers 133

A.1 Writing a Checker . . . 133
A.2 Checker List . . . 134

B SymDrive Support Library 141

B.1 Structures . . . 141
B.2 Functions . . . 142

C Bugs Found 146

C.1 Linux v3.1.1 . . . 146
C.2 Linux v2.6.29 . . . 148
C.3 FreeBSD 9 . . . 155
C.4 Other Drivers . . . 155
C.5 Other Bugs . . . 157

Bibliography 159

LIST OF TABLES

3.1 Decaf code complexity 38
3.2 Drivers converted to the Decaf architecture 40
3.3 Performance of Decaf Drivers 42
3.4 Patch statistics for the E1000 case study 47

4.1 Drivers tested with SDP 65
4.2 Driver performance with SDP 69

5.1 Implementation size of SymDrive 74
5.2 Drivers tested with SymDrive 89
5.3 Bugs found 89
5.4 Code coverage 92
5.5 Patched code coverage 94

6.1 SDC results 112

LIST OF FIGURES

1.1 Drivers look the same as 40 years ago 3

2.1 Symbolic execution example 19

3.1 Decaf Drivers architecture 25
3.2 Sample Jeannie stub 31
3.3 XDR example 36
3.4 Exception handling example 45
3.5 Rewritten code example 46

4.1 SDP architecture 56
4.2 State space explosion example 60
4.3 Symbolic execution limitation example 61

5.1 SymDrive architecture 75
5.2 SymGen instrumentation example 84
5.3 Example checkers 86

6.1 Symbolic execution scalability limitation 105
6.2 SDC architecture 107

ABSTRACT

Writing software to interact with external devices is difficult for a variety of reasons. First, this software often executes in the operating system kernel, where a single error can cause the entire system to fail. Second, most major operating systems today require device drivers to be written in C, a language developed roughly 40 years ago. Third, the kernel execution environment is complex, and in some systems, constantly changing, which leads to increased development costs and unreliable code. Fourth, executing drivers requires specific device hardware, which the developer may not have available. Consequently, many Linux driver patches include the comment “compile tested only.” These problems make driver development a resource- and effort-intensive endeavor. This dissertation presents two new systems that improve the process of device-driver development and testing.

We first present Decaf Drivers, a system for incrementally converting existing Linux kernel drivers to Java programs in user mode. With support from program-analysis tools, Decaf separates out performance-sensitive code and generates a customized kernel interface that allows developers to move the remaining code into Java over time. On the five drivers we tested, the Decaf Drivers system achieves performance close to native kernel drivers and requires few changes to the Linux kernel.

We then present three versions of SymDrive, which is a system for testing Linux and FreeBSD drivers without their devices present. The system uses symbolic execution to remove the need for device hardware. The first prototype, called SymDriveProto (SDP), allows developers to test drivers without hardware by combining traces of hardware operation with symbolic execution. The second version, SymDrive, adds three new capabilities relative to SDP and existing work. First, SymDrive uses static analysis and source-to-source transformation to greatly reduce the effort of testing a new driver. Second, SymDrive incorporates a comprehensive test framework to allow developers to check many driver correctness properties. Finally, SymDrive provides an execution-tracing tool to identify how a patch changes I/O to the device and to compare device-driver implementations. We finally discuss SymDriveCluster (SDC), which allows developers to test drivers and find bugs automatically with a cluster of machines.

1 INTRODUCTION

One of the largest source code components in modern computer operating systems (OSs) is device drivers. Device drivers are OS extensions that function at the interface between the hardware and the rest of the OS. Unfortunately, they are difficult to create. Writing drivers places many more demands on developers than does writing other software. The result is often unreliable, buggy code that requires significant resources to maintain. Given the substantial investments made in existing drivers, this dissertation examines ways in which we can improve the reliability and programmability of these drivers, without wasting the resources and effort already spent on their development.

To accomplish these goals, we present a system to allow developers to migrate drivers into other languages called Decaf Drivers [113]. This system allows developers to continue using existing drivers written in C, while also acting as a mechanism that allows rewriting drivers gradually into user-mode Java, as time permits. Our results show that Decaf Drivers provides a practical approach to migrating substantial driver code out of C and out of the kernel.

We then present a tool called SymDrive that allows driver developers to use symbolic execution to test device drivers even when the associated hardware is not available [112]. We built three versions of SymDrive: an initial prototype called *SymDriveProto* (*SDP* for short), a robust implementation called *SymDrive*, and a more scalable version that runs drivers on cluster-based infrastructure called *SymDriveCluster* (*SDC*). Our results show that SymDrive is an effective tool for finding bugs of many types in existing device drivers, and serves as an effective starting point for fully-automated cluster-based driver testing.

1.1 Motivation

In this section, we first examine the size and nature of the developer community that writes device drivers. We then consider two major reasons that these developers need better ways to write and test drivers: (1) existing drivers are unreliable, and (2) developing and testing drivers is costly.

1.1.1 Development Community

Device drivers are a critical part of operating systems, yet are developed by a broad community. Modern versions of Linux include over 3,200 driver versions in the kernel source tree, developed by over 300 people and entailing millions of lines of code [126]. Moreover, the growth of drivers continues: according to the CLOC tool, Linux v3.8.4 contains approximately six million lines of driver code in the `drivers` directory, as compared to approximately four million lines in the rest of the kernel [1]. Similarly, Windows Vista released with over 30,000 available device drivers [3], and Windows Error Reporting statistics indicate that in 2005, 25 new drivers and 100 revised drivers were released per day [62]. Given the volume of code produced, it is clearly important to support developers involved with driver programming.

In addition to the volume of driver code in modern operating systems, many driver authors are not necessarily well-versed in kernel implementation details. For example, Linux driver authors do not routinely contribute code to the Linux kernel itself, and the Windows kernel source code is often unavailable to driver developers. Consequently, it is likely that many driver developers are not kernel experts, and given the rate at which drivers are being written, it is reasonable to expect that occasional development mistakes will take place.

1.1.2 Reliability

Device drivers are a frequent source of system unreliability. Two studies indicate that drivers are the source of at least 85% of all kernel crashes in Windows [124, 59]. Another Microsoft study suggests that device hardware failures frequently cause driver failure [8], because drivers are often not written to handle the necessary corner cases. Although faulty drivers are causing fewer errors than in the past on Windows-based systems, they still represent a major source of failure compared to faulty hardware [62]. The result is that a variety of system failures stem from poorly written device drivers.

We will next consider three specific reasons for unreliable drivers: (1) the programming language used, (2) the programming model and execution environment, and (3) the lack of tool support.

Kernel Programming Languages

> *C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.*
>
> — Dennis Ritchie [115]

In 1972 [121], Dennis Ritchie, Ken Thompson, and others wrote the Fourth Edition of UNIX (UNIXv4) using C instead of assembly language. The device drivers in this early operating system were also written in C. The same year, Intel introduced the 8008 processor, running at 500kHz and using 3,500 transistors [77]. Today, two of the most widely-used operating systems, Linux and Windows, are both written in C, but Intel now mass-produces CPUs with over three billion transistors [76]. Although technology has clearly moved forward, operating systems development processes and techniques continue to use a four-decade-old language that Dennis Ritchie himself acknowledges as quirky and flawed.

Despite attempts to change how drivers are developed, they continue to be written as they have been: in the kernel and in C. For example, Figure 1.1 compares code from an early Unix driver [114] with code from a corresponding driver in Linux. Even with decades of engineering, driver code in modern versions of Linux bears a striking resemblance to that found in the original versions of Unix.

Unfortunately, C does not support many of the features found in newer languages. For example, modern languages include many of the following capabilities: (a) exceptions, (b) safe pointers, (c) strong type checking, (d) garbage collection, (e) extensive data structure libraries, (f) strong compile-time error checking, (g) object-oriented programming, (h) templates and generics, and (i) reflection. These features often enable developers to write code more efficiently and with fewer bugs, but are generally unavailable in the Windows, Linux, and Mac OS kernels.

Kernel Programming Model

Early versions of Unix had only a handful of drivers, totaling a few kilobytes of code, that were written by a single developer: Dennis Ritchie [114]. In

serial_ks8695.c

```
...
/* stop bits */
if (termios->c_cflag & CSTOPB)
    lcr |= URLC_URSB;
/* parity */
if (termios->c_cflag & PARENB) {
    ...
    else if (termios->c_cflag & PARODD)
        lcr |= URPE_ODD;
    else
        lcr |= URPE_EVEN;
} ...
```

dc.c

```
rtp = tp = &dc11[dev.d_minor];
rtp->t_addr = addr = DCADDR + dev.d_minor*8;
rtp->t_state =| WOPEN;
addr->dcrcsr =| IENABLE|CDLEAD;
if ((rtp->t_state&ISOPEN) == 0) {
    rtp->t_quit = 034;      /* FS */
    rtp->t_intrup = 0177;      /* DEL */
    addr->dcrcsr = IENABLE|CDLEAD|SPEED1;
    addr->dctcsr = IENABLE|SPEED1|STOP1|RQSEND;
    rtp->t_state = ISOPEN | WOPEN;
    rtp->t_flags = ODDP|EVENP|ECHO;
} ...
```

Figure 1.1: Drivers today are often not written very differently from drivers written 40 years ago. The Linux driver on the left is from Linux v3.8.4, released in March 2013. The UNIX driver on the right is from the UNIX nsys kernel, from January 1973.

this model, the driver developer was also intimately familiar with the rest of the operating system, and could make well-informed decisions about how to construct drivers as a result.

In contrast, operating system kernels today are enormous and incorporate millions of lines of code, making it impractical for individual developers to learn all their nuances before developing extensions. Consequently, writing quality device driver code is difficult, as evidenced by the many books and conferences devoted to the subject. The net result of this difficulty is unreliability and unavailability [125].

Writing kernel code is also difficult for a variety of other reasons, unrelated to the complexity of the system as a whole. Drivers often execute in a heavily concurrent environment with extensive synchronization requirements. These requirements can lead to subtle race conditions and deadlocks that manifest only under specific circumstances.

Drivers also require complex memory management techniques, such as mapping regions of memory for specific uses during unusual execution contexts. An incorrect memory allocation while the driver holds a spin lock, for example, can lead to a deadlock. Similarly, an incorrect memory access, perhaps with an invalid pointer, can easily crash the operating system.

Moreover, the kernel often imposes implicit requirements on drivers, and

the programming language, compiler, and kernel often do not enforce these requirements. For example, the Linux kernel includes few correctness checks, and as a result, a driver bug can lead not only to crashes, but undefined behavior. By design, kernels often assume the device driver is correct and not malicious, but the result is that a bug or mistaken assumption can lead to a failure much more easily.

Debugging and Tool Support

Historically, user-mode application developers have enjoyed better tool support than kernel-mode developers. This disparity likely stems from two causes. First, since there are more user-mode developers, the demand for tool support is greater, which leads to better support. As an example, Valgrind [105] is a popular tool for diagnosing memory leaks and pointer errors in user-mode software, but is largely inapplicable to device drivers because it does not support analysis of kernel code.

The techniques Valgrind uses, such as JIT compilation, may also be inapplicable to kernel code, which leads to a second problem: the kernel execution environment makes tool development more difficult or impossible. Using JIT compilation with the kernel, for example, may not work when an interrupt handler is executing. Similarly, interactively debugging some drivers is problematic because single stepping the driver could trigger a timeout condition and alter the driver's behavior unexpectedly. These problems are inherent to the kernel execution environment, and preclude using some tools and techniques that would otherwise help developers.

1.1.3 Development Cost

Device drivers are critical to operating-system reliability, yet are expensive to test and debug. Aside from the time it takes to debug drivers, which can be substantial given the technique and tool limitations outlined previously, these expenses stem from the need to have all supported device hardware on hand in order to test a given driver thoroughly. These high costs cause developers to skip testing altogether, which can lead to more bugs.

A single driver may support dozens of devices, and testing all the driver's code may be impractical. For example, one of the 18 supported medium access controllers in the E1000 driver requires an additional EEPROM read operation while configuring flow-control and link settings, and this operation

may fail. Testing the error handling in this driver requires any test suite to consider specifics of each supported chip, which may become costly, and also requires access to all the supported devices. Error handling code is particularly time consuming and expensive to test because this code may only execute if the device reports an error or malfunctions, such as by returning an invalid value. Simulating this condition with standard hardware is often not feasible. Moreover, the error handling code may only apply when a specific support device is present. Testing drivers thoroughly requires a significant financial investment, and testing some parts of drivers, such as error handling code, remains impractical for most individual developers even when the hardware is available.

The need for device hardware can prevent testing altogether. Over two dozen driver Linux and FreeBSD patches include the comment “compile tested only,” indicating that the developer was unable or unwilling to run the driver. Core kernel developers often do not have access to the hardware needed by device drivers. Revisions to the driver/kernel interface occasionally necessitate large, cross-cutting changes affecting many drivers, and the hardware requirements for an individual to test all of these changes could be very high [108]. In these cases, driver code is left without sufficient testing.

1.2 Thesis and Contributions

Although device drivers and the process of their development suggest a variety of long-term improvements, we can also improve their quality immediately by selectively addressing some of the most pressing problems. In particular, our thesis is that we can improve driver quality without redesigning existing operating systems by using two new tools that allow developers to (a) gradually rewrite existing drivers using a modern programming language and (b) test existing device drivers without their associated hardware.

To accomplish these goals, we first created a system called Decaf Drivers to allow driver developers to rewrite drivers using Java instead of C [113]. Decaf Drivers provides a framework that allows developers to take an existing Linux driver, and move most of the code out of the kernel into user mode. Once in user mode, it allows developers to rewrite the driver, one function at a time, into Java. Using modern programming language techniques keeps this process seamless.

We also created a tool called SymDrive that provides an improved way to test device drivers [112]. SymDrive allows developers to execute Linux and FreeBSD drivers inside a virtual machine, even when the device hardware is not available, by using symbolic execution. This tool also provides a test framework to allow developers to check driver correctness properties along multiple execution paths. We will next look at each of these tools in additional detail.

1.2.1 Decaf Drivers

Decaf Drivers provides developers a way to migrate Linux kernel drivers, all of which are written in C, first into user-mode C code and then into Java. To accomplish this, Decaf Drivers builds on the Microdrivers system [60, 61] to automatically split drivers into kernel-mode C, user-mode C, and user-mode Java components. Decaf calls these pieces the driver nucleus, driver library, and decaf driver respectively.

Initially, after splitting a specific driver, the driver nucleus and driver library contain the original C driver code, and the decaf driver is empty. The driver nucleus executes as a kernel module, and the driver library executes as a separate user-mode process. The driver library acts as a temporary staging ground, because once the driver is split, the developer can easily migrate any driver functions in the driver library into the decaf driver by writing the corresponding Java code. Once the developer migrates all the driver library functionality into the decaf driver, the driver library is no longer necessary. The result is a driver split into a small, high-performance kernel-mode component written in C, and a larger, user-mode component written in Java.

To split the original device driver, Decaf Drivers uses DriverSlicer, which takes an existing driver as input, and produces the driver nucleus and driver library as output. The driver nucleus and driver library each contain a subset of the original driver's functions, and the split depends on whether a driver's entry point executes at high-priority in the kernel. High-priority code, such as interrupt handlers and timers, remains in the driver nucleus. All other functionality, such as initialization, cleanup, and other control-path code can execute safely in the driver library, and eventually, the decaf driver.

Once split, driver execution relies on three additional pieces of infrastructure: the *object tracker*, the *nuclear runtime*, and the *decaf runtime*. First, the object tracker stores a correspondence between any data structures used in each of the three driver components. When transferring control from the driver nucleus to

the decaf driver, for example, the object tracker ensures that pointers within the necessary objects are copied correctly. Second, the nuclear runtime and decaf runtime provide necessary runtime support for the decaf driver. These runtime components implement XPC, or eXtension Procedure Call [125], to allow the decaf driver to communicate with the driver nucleus.

Decaf Drivers provides a high-performance approach to moving driver code out of the kernel and into a modern language, and requires low developer effort. Experiments with Decaf show that it can execute existing drivers with less than 1% performance penalty, that it can execute most driver code in user-mode Java, and can employ Java language features to simplify development.

1.2.2 SymDrive

Re-writing drivers using Java is an effective way to prevent many bugs and simplify development. However, this approach requires re-writing thousands of drivers. Until these rewrites take place, we still need to test and improve the drivers we already have.

To improve the process of testing existing drivers, our second contribution is the development of SymDrive. This system uses symbolic execution [24, 31] to test device drivers even when their hardware is not available. By supplying symbolic data in place of device input, SymDrive allows developers to see how a driver would react in response to many possible forms of device input.

Our first prototype implementation, called *SymDriveProto* (or SDP for short), builds on Microdrivers [60, 61]. In contrast to Microdrivers, SDP moves the entire driver into user mode. SDP then executes the driver symbolically. This implementation uses the KLEE symbolic execution framework (SEF) [24] to provide the symbolic execution functionality. By using this prototype on seven different network and sound drivers, we were able to find seven bugs.

Unfortunately, SDP's implementation has three major limitations. First, Microdrivers itself does not currently work with all driver classes because of the design complexity of some device drivers. Second, because execution is split between kernel and user mode, SDP cannot pass symbolic data to the kernel. Instead, when returning or calling into the kernel with symbolic data, SDP chooses feasible example values for each symbol to use instead, in a process called *concretization*. When concretizing these values, SDP terminates any additional execution paths, which prevents SDP from exploring many feasible paths. Third, SDP requires a device trace, which is essentially a database of

previously recorded driver/device hardware interactions. This trace is necessary to initialize drivers with SDP quickly.

Given these limitations, we undertook the implementation of an improved tool, called SymDrive. Instead of relying on KLEE, SymDrive builds on S^2E [32, 33], which executes a complete virtual machine symbolically. By allowing the entire operating system to execute along multiple paths, SymDrive overcomes most of the limitations of SDP, and concretizes data much less frequently. SymDrive does not require a hardware trace to initialize the driver, and it supports many more driver types than SDP or Microdrivers. To initialize drivers without any hardware details, SymDrive prioritizes execution of paths that lead to successful driver initialization.

Experiments with SymDrive show that it works with 21 Linux drivers and 5 FreeBSD drivers, running on five hardware buses. Using SymDrive, we found 39 bugs in these drivers, and submitted patches for many of them. These results demonstrate that SymDrive is an effective tool for finding bugs in drivers.

Although SymDrive successfully finds bugs in unfamiliar device drivers, it is not scalable. In order to achieve the results it did, we still needed to test each driver individually. Testing a driver involves setting up the appropriate environment to test it, annotating the driver if needed, invoking its functionality on a case-by-case basis, and analyzing the results. This process is too slow given the number of drivers present in modern operating systems. Instead, we believe that using SymDrive as an automatic bug-finding tool would be more useful. For example, using SymDrive to test Linux driver patches automatically before they were committed to the kernel would be useful.

To address these limitations and allow SymDrive to scale more effectively, we extended SymDrive to automate the process of testing a device driver. The enhanced tool, called *SymDriveCluster* (or simply *SDC*), runs on unmodified Linux device drivers and requires no per-driver annotations, executes each driver on its own machine in a cluster-based environment automatically, and can find many more bugs than SymDrive could on its own.

1.3 Dissertation Organization

This dissertation consists of eight chapters. The first two chapters provide introductory and background material to provide additional context for the research. Chapter 3 discusses the Decaf Drivers system and presents results achieved. Chapters 4, 5, and 6 discuss the three versions of SymDrive: SymDriveProto,

SymDrive, and SymDriveCluster. Finally, chapters 7 and 8 present some of the lessons learned during this research, ideas for future work, and conclusions.

2 BACKGROUND AND RELATED WORK

This section presents background information and related work on device drivers and driver reliability. First, it discusses drivers in general, and how they interface with the rest of the operating system and hardware. Second, it discusses device driver reliability, and some previous work that addresses the problem. Finally, it presents information on static source code analysis and symbolic execution, because these forms of program analysis are related.

2.1 Device Drivers

Device drivers are operating-system extensions that provide an interface between a device and the rest of the system. Most pieces of hardware installed on PCs, servers, and phones today require a device driver. Linux uses the term *module* to refer to any operating system extension, whereas Windows uses the term *driver*. In this dissertation, we use the term *device driver* or simply *driver* to refer to an operating system extension that also interacts with device hardware.

2.1.1 Device Driver Requirements

Regardless of operating system, many device drivers must serve a wide-ranging set of functions. Some of the most common functions of device drivers include (a) initializing the device, (b) multiplexing access to the device, and (c) abstracting the interface to the device.

Most devices require initialization. This process generally consists of reading and writing control registers on the hardware to put the device into a state that enables its primary function. For example, when the driver for a network interface card loads, it needs to carry out a number of steps before the card is ready to send and receive packets, such as configuring the device's MAC address, and registering the device with the kernel. Once these steps are complete, the OS can use the device by invoking the driver's interface as needed.

The need to multiplex access to the device is common to all modern operating systems. For example, a disk-controller driver must allow multiple applications to read and write data at the same time, even if these applications are not otherwise related. This requirement can complicate driver design, because it necessitates synchronization among multiple independent threads.

Finally, device drivers must abstract the device functionality according to the host operating system's driver interface. Many devices use unique, vendor-specific communication protocols to interact with the host OS. Drivers for these devices serve as translators, by converting system requests into hardware-specific commands. Similarly, drivers translate device notifications, such as interrupts, into the appropriate kernel invocations. In all cases, drivers need to interact with the rest of the machine according to the operating system's interface. In Linux, the driver and kernel interfaces are broad, and often span dozens of functions and data structures.

Each interface function may have special requirements. For example, the kernel may invoke some driver functions with interrupts disabled or with a spin lock held. Similarly, the driver may invoke some kernel functions only under certain conditions, such as after invoking some other set of kernel functions or from within particular execution contexts. These implicit requirements extend beyond function parameters and return values, and contribute to the complexity of driver development.

Some drivers represent exceptions to each of these rules. An especially simple device running on a simple bus may require very little initialization, though the driver may still need to interact with the OS simply to notify it of the device's presence. Similarly, some drivers may lock out multiple users if the hardware lacks the necessary support. For example, a sound card may allow users to play only one sound at a time, in which case multiplexing access is not necessary. Finally, many USB devices from different vendors use the same drivers, which removes most of the need for vendor- and hardware-specific command translations. Nevertheless, all drivers must take care to interact with the kernel and device correctly, and most drivers have several or all of these requirements.

2.1.2 Driver/OS Interface

Operating system driver architectures define the interface between the driver and the rest of the system, as well as the environment. In Linux, Windows, and Mac OS, device drivers generally run in kernel mode and are written in C, or a subset of C++ in the case of Mac OS X.

Operating systems incorporate varying approaches to supporting device drivers. The first approach is to treat the driver as a kernel-mode OS extension. In this model, all interaction with the kernel takes place via standard function

calls. Both Windows and Linux use this model today [38, 97]. Executing drivers as low-privilege user-mode processes is the second approach. The benefit is greater OS reliability, because a driver crash will not necessarily cause the system to fail. The Minix operating system uses this approach [69], and the Windows User-Mode Driver Framework introduces a similar capability to Windows [99].

The operating system's driver framework typically specifies different device classes, which are simply categories of devices with similar functionality that drivers must support. For example, the Mac OS X kernel includes classes for network, sound, and graphics drivers [6]. All drivers in a specific class export a consistent interface to the kernel. For example, sound drivers must provide a "play" function, regardless of how the hardware implements this feature. This approach simplifies OS design, because it can communicate with any device of a given class using the same interface.

Drivers must provide this consistent device-class interface to the operating system, despite the diverse functionality provided by hardware within a given class, so operating systems typically include a mechanism to extend it. For example, the Linux E1000 driver supports a variety of command-line options for features specific to the E1000, such as an "interrupt throttling rate," which limits the maximum number of interrupts that the card will ever generate in a given amount of time. This feature is not part of any operating system's standard network device interface and is instead specific to this device. The Windows, Mac OS X, and Linux architectures also incorporate an I/O control command, which drivers can use to implement nearly any desired extension.

2.1.3 Driver/Device Interface

Each piece of hardware often has its own interface, with its own unique characteristics. Hardware vendors define the interfaces for their devices, and these interfaces can vary significantly, even with different revisions of the same device. Drivers communicate with devices using four primary mechanisms on most platforms today: (a) I/O memory, (b) port I/O, (c) direct memory access, and (d) interrupts. These channels are the same regardless of the bus the device uses.

I/O Memory and Port I/O

Many devices expose a set of registers to the driver. These registers allow the driver to read and write data, and issue commands. The two mechanisms

used to expose these registers are I/O memory and port I/O. I/O memory is the most common approach, and involves mapping the device registers as a range of memory addresses. The driver then interacts with the device by reading and writing those memory addresses. Port I/O acts similarly, but is only used in older PCI devices. Instead of reading and writing memory addresses, the driver reads and writes a range of numeric identifiers accessed with special CPU instructions called *ports*. Both mechanisms allow the driver to interact with the device directly.

Direct Memory Access

DMA, or *direct memory access*, is a mechanism by which the device and the driver communicate by reading and writing a particular piece of memory. This mechanism is unrelated to I/O memory, which involves using memory addresses to access device registers. Once the driver configures a piece of memory for use with DMA, it communicates the details to the device, which can then read and write that memory directly without using the CPU or interrupting the driver.

Interrupts

Interrupts are the final communication mechanism. In contrast to the three mechanisms listed above, which involve transferring data between the device and the driver, interrupts allow the device to notify the driver of an event. The steps that take place when the device generates an interrupt are as follows:

- The device hardware enters a state in which it needs to notify the driver of an event. The device notifies the CPU that an interrupt has occurred.
- The CPU receives notification of the interrupt, and immediately transfers control to a system-wide interrupt handler.
- The system then invokes the device driver's *interrupt handler*, which is a function for responding to the interrupt and notifying the device appropriately.

Device Buses

Nearly all devices that interact with a CPU in phones, tablets, and computers use I/O memory, port I/O, DMA, and interrupts. These same abstractions work regardless of the means by which the device attaches to the CPU, called the *bus*.

Each bus has unique characteristics, which heavily influence the design of drivers for devices running on that bus. For example, the PCI bus provides drivers with few abstractions to interact with the device. PCI drivers read and write device registers via memory-mapped or port I/O instructions, which provides high performance at the expense of additional driver complexity. Other buses provide drivers with higher-level communication protocols. USB device drivers do not interact with the bus directly, but instead use a special driver called the *host controller driver* or HCD. The HCD provides abstractions for interacting with the device that build on the mechanisms outlined previously. Buses such as I^2C and SPI are similar to USB in this respect.

2.2 Reliability

Previous work has considered a variety of approaches toward improving driver reliability. The primary approaches are to: (a) isolate drivers, (b) use type safety and other language features (c) simplify driver code, (d) provide better driver test frameworks, (e) formally specify drivers (f) improve driver interfaces and (g) re-write the kernel. This section discusses each of these areas of related work in more detail.

2.2.1 Driver Isolation

One common way to isolate drivers from the rest of the operating system is to execute them as standalone user-mode processes, or in their own protection domains. Microkernels often use this model [135, 70, 69, 81], but sometimes suffer from reduced performance. User-mode driver frameworks for Linux and Windows also exist [98, 127, 85], but do not support all drivers or driver classes.

User-mode device drivers offer developers a number of significant benefits. Moving drivers drivers either partially [127, 61] or completely [7, 34, 98, 129, 69, 49, 20] into user mode also provides development benefits, but may incur a performance penalty and not support all classes of devices. If the driver experiences an unrecoverable error, the driver can be terminated or restarted, and the rest of the operating system can continue execution without significant disruption. Developers can also apply advanced debugging and analysis tools to the driver as it executes, because he or she can treat it like any other process on the system [7, 34]. Tools such as Valgrind [105], for example, make diagnosing memory-usage problems much simpler. In this environment,

developers can leverage most of the same development tools used by thousands of other application developers.

Like Decaf Drivers, Linux UIO drivers leave part of the driver in the kernel, while the bulk executes at user level [127]. However, Decaf Drivers allows developers to transition existing user-mode C driver code into Java, whereas UIO requires C, and does not require rewriting drivers from scratch. Moreover, Decaf Drivers leaves performance-critical code such as interrupt handlers in the kernel, while UIO drivers defer most interrupt handler code to user-mode.

Other systems provide alternatives to user-mode isolation. Nooks runs each driver in its own protection domain [124], which isolates the driver from the rest of the kernel. Bugs in the driver will rarely crash the entire machine. Tolerating faults by leveraging memory protection is another approach [125, 130, 27, 54]. Similarly, executing a driver in its own virtual machine provides strong isolation guarantees at the cost of performance [55, 58, 86, 90, 13]. If the driver crashes, it will not necessarily impact the rest of the machine.

Driver isolation improves operating system reliability, but does not address the difficulty of writing drivers, or the difficulties of debugging drivers. However, it can significantly improve reliability compared to running un-isolated drivers in the kernel.

2.2.2 Driver Safety

The language drivers are written with can provide type safety and other guarantees. For example, CCured and Ivy [36, 22, 137] augment C with additional features that help programmers eliminate bugs. This approach is a form of isolation, but is distinct from the previously listed techniques in that the drivers execute in the kernel as they always have, but use language and compiler features to guarantee various driver properties, such as not accessing memory incorrectly. This approach has the benefit that it relies on existing driver code.

Similar to language isolation, enforcing safety checks at the language level [120, 74] also improves driver reliability, because certain classes of bugs are eliminated. SPIN [72] and the J Kernel [128] also have kernels written in type safe languages, though SPIN uses device drivers written in C and the J-Kernel uses the host operating system's drivers. More recently, a real-time JVM was ported to the Solaris kernel [106], and other systems have run drivers using Java on Solaris [132, 133]. The approaches used in these systems are often most appropriate when writing new drivers from scratch.

Redesigning the kernel completely to incorporate type safety can also simplify driver development and improve reliability. Several new kernels include type safety as an explicit design criterion [72, 47, 73]. The CuriOS kernel incorporates safety in a more general sense by considering security and fault tolerance as design criteria [44]. However, this approach is extremely costly. As discussed previously, existing operating systems include millions of lines of driver code, and thousands of drivers [126, 3, 62], and discarding this investment would be very costly.

2.2.3 Cross-Platform Frameworks

Considerable work has been conducted to simplify development of drivers across platforms. For example, Jungo WinDriver and the Uniform Driver Interface [78, 110] make it easy to run a single driver on multiple platforms. The result is that developers only need to implement the driver once, rather than repeatedly. However, WinDriver builds a custom interface on top of existing driver architectures, which necessarily impacts performance. Moreover, many drivers are written by copying and pasting existing code [51, 87]. Thus, it may still be easier for a driver developer to modify an existing C driver than to write a new driver from scratch, even if the environment is simpler to program.

2.2.4 Test Frameworks

Test frameworks provide automated testing environments for drivers. IBM maintains the Linux Test Project (LTP) with the aim of providing a set of tests that "validate the reliability, robustness, and stability of Linux," including the kernel and associated drivers [75]. The main drawback to test suites such as the LTP is the need for the device hardware in order to execute the test. In addition, LTP tests execute at the system-call level, and thus cannot verify properties of individual driver entry points.

Test frameworks such as the Linux Test Project (LTP) [75] and Microsoft's Driver Verifier (DV) [92, 96] can invoke drivers and verify their behavior, but require the device be present. LTP tests at the system-call level and thus cannot verify properties of individual driver entry points. Driver Verifier, in contrast, checks that drivers are executing correctly at runtime by tracking their interaction with the kernel. SymDrive can use these frameworks, either as checkers, in the case of DV, or as a test program, in the case of LTP. Similar to

Driver Verifier, Diagnosys also detects bugs by interposing on various kernel interfaces [19].

2.2.5 Driver-Specific Languages

Domain-specific languages for hardware access and for common driver logic are another approach to improving driver quality and reliability [91, 37, 116, 29]. The advantage of this approach is that the language designer can tailor the language to the unique requirements of drivers. Aside from the need to rewrite drivers using these languages from scratch, another disadvantage is the importance of including necessary functionality in the language. For example, Dingo assumes that drivers are written as state machines [116], but in practice this assumption may not hold because drivers often contain additional data-processing functionality and threading that does not fit well into the language model [80].

2.2.6 Formal Specifications

Another way to improve driver reliability is to require developers to write specifications of the device's behavior. Formal specifications express a device's or a driver's operational requirements. Once written, tools can check the driver's implementation against its specification [14, 118, 129]. It is also possible to synthesize driver code directly from the specification [117]. However, this approach requires creation of specifications for each driver or device. Amani et al. argue that the existing driver architecture is too complicated to be formally specified, and propose a new architecture to simplify verification [4].

2.3 Static Analysis

Static analysis is a commonly-used strategy for finding bugs or other problems by examining source code. It has the primary benefits of being fast, easy to use, and scalable. Using static analysis to find bugs in driver code is an important technique for improving their reliability.

Static analysis tools can find specific kinds of bugs common to large classes of drivers, such as misuses of the driver/kernel [9, 10, 11, 101, 107] or driver/device interface [79] and ignored error codes [68, 123]. Static bug-finding tools are often faster and more scalable than symbolic execution, and can often check the kernel and other software in addition to drivers [16, 41].

Better tool support can also simplify driver development. For example, Coccinelle is a tool that simplifies the task of making large, cross-cutting changes to many drivers, such as when the operating system interface is changed [107]. These tools make it easier to write and maintain device drivers.

Microsoft Static Driver Verifier [101] combined with the Specification Language for Interface Checking [12] are another approach to static analysis. SDV combines a kernel model with SLIC specifications to execute drivers along multiple executions paths, while checking adherence to the specifications along each path. The specifications are written in the SLIC language, which uses C-like syntax, and check conditions such as function call pre- and post conditions. Using an operating system model is effective for testing Windows drivers because the driver/kernel interface changes more slowly than that of Linux [67].

The Windows driver/kernel interface also often supports binary compatibility over comparatively long periods of time, so drivers designed on one version of Windows work automatically on newer versions. This property is important because it means that writing and maintaining an operating system model is comparatively inexpensive: developers would write the model once, and then continue to use it to check driver behavior for many years at a time. In contrast, Linux changes continuously, which would necessitate constant revision to the model.

Static Analysis Limitations

Unfortunately, the same approximations that makes static analysis tools fast and scalable result in several key limitations. First, most static analysis techniques do not address bugs that manifest across driver entry point invocations, such as when state is corrupted during one call and accessed during another. Second, static analysis has limited access to driver and kernel state, which precludes checking driver behavior. For example, tracking the contents of memory and other dynamic driver and kernel data is difficult to do statically. Static analysis also rarely supports the full functionality of C, such as pointer arithmetic, aliasing, inline assembly code, and casts. Instead, these tools usually operate on a restricted subset of the language, which can lead to false and missing bug reports. Finally, static tools either require a model of kernel behavior, which in Linux changes regularly [67], or they do not verify driver/kernel interaction and consequently miss bugs.

```
// Produce an unconstrained symbolic value:
int x = read_from_device();

// Compare the symbol stored in x against
// a constant, and constrain the symbol
// along each possible path appropriately.
// Symbolic execution forks the program state,
// and allows the developer to observe the
// behavior along both paths, with x
// constrained appropriately along each.
if (x > 5) {
  printk ("Branch A: Value > 5\n");
} else {
  // The engine notifies the developer
  // that a crash occurs along this branch
  panic ("Branch B: Value <= 5.  Crash!");
}
```

Figure 2.1: Initially, the variable x stores an unconstrained symbol. When execution reaches the conditional statement, the underlying runtime forks execution, with the symbol stored in x constrained appropriately along each path.

2.4 Symbolic Execution

Symbolic execution allows a program's input to be replaced with a *symbolic value*, which represents all possible values the data may have. In order to execute a program symbolically, a runtime framework called a *symbolic-execution engine* is necessary. This engine runs the code and tracks which values are symbolic and which have fully-defined (*i.e.*, *concrete*) values, such as initialized variables. When the program compares a symbolic value, the engine forks execution into multiple *paths*, one for each outcome of the comparison. It then executes each path with the symbolic value constrained by the chosen outcome of the comparison.

Figure 2.1 shows an example code snippet, which we might execute symbolically. In this example, the predicate $x > 5$ forks execution by copying the running program. In one copy, the code executes the path where $x \leqslant 5$ and the other executes the path where $x > 5$. Subsequent comparisons can further constrain a value. In places where specific values are needed, such as printing a value, the engine can concretize data by producing a single value that satisfies all constraints over the data.

Symbolic execution detects bugs either through illegal operations, such as dereferencing a null pointer, or through explicit assertions over behavior, and

can show the state of the executing path at the failure site.

There are numerous prior approaches to symbolic execution [5, 21, 26, 24, 31, 64, 82, 119, 131]. However, most of these approaches apply to standalone programs with limited environmental interaction. In particular, they focus primarily on single-threaded user-mode programs whose behavior depends only on parameters and file accesses. Drivers, in contrast, execute as a library and make frequent calls into the kernel. BitBlaze supports environment interaction but not I/O or drivers [111].

In addition, many of them search for specific kinds of bugs such as crashes and assertion violations, rather than allowing the developer to track the program's logical state as it executes to verify correctness. To limit symbolic execution to manageable state, previous work limited the set of symbolically executed paths by applying smarter search heuristics and/or by limiting program inputs [25, 65, 82, 83, 89, 134]. SymDrive uses a combination of approaches to handle this issue, such as path pruning and prioritization.

Other systems combine static analysis with symbolic execution [40, 42, 53, 109]. SymDrive uses static analysis to insert checkers and to dynamically guide the path selection policy from code features such as loops and return values. In contrast, these systems use the output of static analysis directly within the symbolic execution engine to select paths. Execution Synthesis [136] combines symbolic execution with static analysis, but is designed to reproduce existing bug reports with stack traces, and is thus complementary to SymDrive.

Symbolic execution is also a highly parallelizable program-analysis technique. One approach to improving performance is to distribute the analysis across many machines, as in Cloud9 [23]. This technique is more sophisticated than that used in SDC, which instead independently tests each driver on its own virtual machine. However, we have not found that the additional implementation complexity of Cloud9 would be of significant enough benefit in SDC.

DDT and S^{2}E

The DDT and S^{2}E systems have been used for finding bugs in binary drivers [32, 33, 82]. SymDrive is built upon S^{2}E but significantly extends its capabilities in three ways by leveraging driver source code. First and most important, SymDrive automatically detects the driver/kernel interface and generates code to interpose checkers at that interface. In contrast, S^{2}E requires programmers to identify the interface manually and write plugins that execute

outside the kernel, where kernel symbols are not available, though S^2E and SymDrive both support re-using existing testing tools. Second, SymDrive automatically detects and annotates loops, which in S^2E must be identified manually and specified as virtual addresses. As a result, the effort to test a driver is much reduced compared to S^2E. Third, checkers in SymDrive are implemented as standard C code executing in the kernel, making them easy to write, and are only necessary for kernel functions of interest. When the kernel interface changes, only the checkers affected by interface changes must be modified. In contrast, checkers in S^2E are written as plugins outside the kernel, and the consistency model plugins must be updated for all changed functions in the driver interface, not just those relevant to checks.

2.5 Conclusions

Developers have invested a significant amount in the development of thousands of Linux and Windows device drivers. Some approaches to improving driver quality and reliability would require ignoring this investment, which we believe is impractical. Instead, we should develop tools and techniques that enable developers to introduce modern programming languages into these existing drivers, and to test them more easily. In doing so, we can preserve the existing investment in device drivers while also improving their quality.

To accomplish these goals, this thesis takes two approaches to improving driver quality. First, Decaf Drivers combines splitting drivers with rewriting them, to provide high performance, better maintainability, and additional correctness guarantees. In contrast, SymDrive focuses primarily on testing drivers more easily and more widely, to improve the quality of existing implementations.

3 DECAF DRIVERS

Decaf Drivers takes a best-effort approach to simplifying driver development by allowing most driver code to be written at user level in languages other than C. Decaf sidesteps many potential problems by leaving code that is critical to performance or compatibility in the kernel in C. All other code can move to user level and to another language; we use Java for our implementation, as it has rich tool support for code generation, but the architecture does not depend on any Java features. The Decaf architecture provides common-case performance comparable to kernel-only drivers, but reliability and programmability improve as large amounts of driver code can be written in Java at user level.

The goal of Decaf Drivers is to provide a clear migration path for existing drivers to a modern programming language. User-level code can be written in C initially and converted entirely to Java over time. Developers can also implement new user-level functionality in Java. Previous efforts that have demonstrated how to execute kernel C code safely [137] are compatible with Decaf Drivers.

We implemented Decaf Drivers in the Linux 2.6.18.1 kernel by extending the Microdrivers infrastructure [61]. Microdrivers provided the mechanisms necessary to convert existing drivers into a user-mode and kernel-mode component. The resulting driver components were written in C, consisted entirely of preprocessed code, and offered no path to evolve the driver over time.

The contributions of our work are threefold. First, Decaf Drivers provides a mechanism for converting the user-mode component of microdrivers to Java through cross-language marshaling of data structures. Second, Decaf supports incremental conversion of driver code from C to Java on a function-by-function basis, which allows a gradual migration away from C. Finally, the resulting driver code can easily be modified as the operating system and supported devices change, through both editing of driver code and modification of the interface between user and kernel driver portions.

We demonstrate this functionality by converting five drivers to decaf drivers, and rewriting either some or all of the user-mode C code in each into Java. We find that converting legacy drivers to Java is straightforward and quick.

We analyze the E1000 gigabit network driver for concrete evidence that Decaf simplifies driver development. We find that using Java exceptions reduced the amount of code and fixed 28 cases of missing error handling. Furthermore,

updating the driver to a recent version predominantly required changes to the Java code, not kernel C code. Using standard workloads, we show while decaf drivers are slower to initialize, their steady-state performance is within 1% of native kernel drivers.

In the following section, we describe the Decaf architecture. Section 3.2 provides a discussion of our Decaf Drivers implementation. We evaluate performance in Section 3.3, follow with a case study of applying Decaf Drivers to the E1000 driver, and finally conclude.

3.1 Design of Decaf Drivers

The primary goal of Decaf Drivers is to simplify device driver programming. We contend that the key to dramatically improving driver reliability is to simplify their development, and that requires:

- User-level development in a modern language.
- Near-kernel performance.
- Incremental conversion of existing drivers.
- Support for evolution as driver and kernel data structures and interfaces change.

User-level code removes the restrictions of the kernel environment, and modern languages provide garbage collection, rich data structures, and exception handling. Many of the common bugs in drivers relate to improper memory access, which is solved by type-safe languages; improper synchronization, which can be improved with language support for mutual exclusion; improper memory management, addressed with garbage collection; and missing or incorrect error handling, which is aided by use of exceptions for reporting errors.

Moving driver code to advanced languages within the kernel achieves some of our goals, but raises other challenges. Support for other languages is not present in operating system kernels, in part because the kernel environment places restrictions on memory access [133]. Notably, most kernels impose strict rules on when memory can be allocated and which memory can be touched at high priority levels [39, 95]. Past efforts to support Java in the Solaris kernel bears this out [106]. In addition, kernel code must deal gracefully with low memory situations, which may not be possible in all languages [28].

The Decaf architecture balances these conflicting requirements by *partitioning* drivers into a small kernel portion that contains performance-critical code and a large, user-level portion that can be written in any language. In support of the latter two goals, Decaf provides tools to support migration of existing code out of the kernel and to generate and re-generate marshaling code to pass data between user mode and the kernel.

3.1.1 Microdrivers

We base Decaf Drivers on *Microdrivers*, a user-level driver architecture that provides both high performance and compatibility with existing driver and kernel code [61]. Microdrivers partition drivers into a kernel-level *k-driver*, containing only the minimum code required for high-performance and to satisfy OS requirements, and a user-level *u-driver* with everything else. The k-driver contains code with high bandwidth or low-latency requirements, such as the data-handling code and driver code that executes at high priority, such as interrupt handlers. The remaining code, which is often the majority of code in a driver, executes in a user-level process. While the kernel is isolated from faults in the user-level code, systems such as SafeDrive [137] or XFI [54] can be used to isolate and recover from faults in the kernel portion.

To maintain compatibility with existing code, the *DriverSlicer* tool can create microdrivers from existing driver code. This tool identifies high-priority and low-latency code in drivers that must remain in the kernel and creates two output files: one with functions left in the kernel (the k-driver), and one with everything else (the u-driver). With assistance from programmer annotations, DriverSlicer generates RPC-like stubs for communication between the k-driver and u-driver. The kernel invokes microdrivers normally by either calling into the k-driver or into a stub that passes control to the u-driver.

The Microdrivers architecture does not support several features necessary for widespread use. First, after splitting the driver, Microdrivers produces only preprocessed C output, which is unsuitable for evolving the driver once split. Second, Microdrivers only supports C in the u-driver, and provides no facility for moving to any other language.

3.1.2 Decaf Drivers Overview

Decaf Drivers extends Microdrivers by addressing the deficiencies outlined previously: Decaf Drivers supports (1) writing user-level code in a language

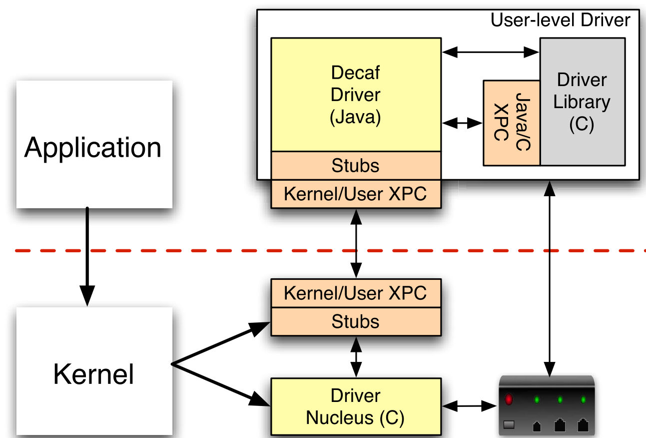

Figure 3.1: The Decaf Drivers architecture. The OS kernel invokes driver nucleus code or stubs that communicate with the decaf driver via an extension procedure call (XPC).

other than C, (2) incremental conversion of a legacy driver to a new driver architecture, and (3) evolving the driver as interfaces and data structures change.

Decaf Drivers partitions drivers into two major components: the *driver nucleus*[1] that executes in the kernel for performance and compatibility; and the user-level *decaf driver* written in any language that supports marshaling/unmarshaling of data structures. However, user-level driver code may need to perform actions that are not expressible in all languages, such as directly controlling the hardware with instructions such as *outb*. This code resides in the user-level *driver library*, which executes normal C code. The driver library also provides a staging ground when migrating C code out of the kernel, where it can execute before being converted to another language.

While the architecture supports any language, our implementation is written for Java and we refer to code in the decaf driver as being written in Java. Using Java raises the issue of communicating data structures between languages, in contrast to C++. We believe that other languages that provide mechanisms for invoking native C code, such as Python, would also work well with the Decaf

[1] We re-christened the k-driver and u-driver from Microdrivers to more descriptive names reflecting their purpose and implementation, not just their execution mode.

Drivers architecture.

At runtime, all requests to the driver enter through the kernel. The kernel directly invokes functionality implemented by the driver nucleus. Functionality implemented at user level enters through a stub that transfers control to user level and dispatches it to the driver library or the decaf driver. The user-level components may invoke each other or call back into the kernel while processing a request.

The Decaf architecture consists of two major components:

1. *Extension Procedure Call (XPC)* for communication between kernel/user level and between C and Java.

2. *DriverSlicer* to generate marshaling code for XPC.

We next discuss these two components in detail.

3.1.3 Extension Procedure Call

Extension procedure call, created as part of the Nooks driver isolation subsystem [125], provides procedure calls between protection domains. XPC in Decaf Drivers provides five services to enable this cooperation: *control transfer* to provide procedure call semantics (*i.e.*, block and wait); *object transfer* to pass language-level objects, such as structures, between domains; *object sharing* to allow an object to exist in multiple domains; and *synchronization* to ensure consistency when multiple domains access a shared object. *Stubs* invoke XPC services for communication between domains.

The two primary domains participating in driver execution are the driver nucleus and the decaf driver. However, driver functionality may also exist in the driver library, both when migrating code to another language or for functionality reasons. For example, code shared across operating systems may be left in C. Thus, the Decaf architecture also provides XPC between the decaf driver and the driver library to provide access to complex data structures requiring conversion between languages. The decaf driver may directly invoke code in the driver library for simple library calls.

Cross-Domain Control Transfer

The control-transfer mechanism performs the actions of the runtime in an RPC system [17] to pass control from the calling thread to a thread in the

target domain. If the decaf driver and the driver library execute in a single process, the control transfer mechanism can be optimized to re-use the calling thread rather than scheduling a new thread to handle the request.

Cross-Domain Object Transfer

XPC provides customized marshaling of data structures to copy only those fields actually accessed at the target. Thus, structures defined for the kernel's internal use but shared with drivers are passed with only the driver-accessed fields. In addition, XPC provides cross-language conversion, converting structures making heavy use of C language features for performance (*e.g.*, bit fields) to languages without such control over memory layout.

Object Sharing

Driver components may simultaneously process multiple requests that reference the same object. If two threads are accessing the same object, they should work on a single copy of this object, as they would in the kernel, rather than on two separate copies. Similar to Nooks [125], Decaf Drivers XPC uses an object tracker that records each shared object, extended to support two user-level domains. When transferring objects into a domain, XPC consults the object tracker to find whether the object already exists. If so, the existing object can be updated, and if not, a new object must be allocated.

Synchronization

Synchronized access to data is a challenging problem for regular device drivers because they are reentrant. For example, a device may generate an interrupt while a driver is processing an application request, and the interrupt handler and the request handler may access the same data. To prevent corruption, driver writers must choose from a variety of locking mechanisms based on the priority of the executing code and of potential sharers [88].

The Decaf Drivers synchronization mechanism provides regular locking semantics. If code in one domain locks an object, code in other domains must be prevented from accessing that object while the lock is held. Furthermore, Decaf ensures that the holder of a lock has the most recent version of the objects it protects.

Stubs

Similar to RPC stubs, XPC stubs contain calls specific to a single remote procedure: calls into marshaling code, object tracking code, and control transfer code. These can be written by hand or generated by the DriverSlicer tool. Calls to native functions must be replaced with calls to stubs when the function is implemented in another domain.

3.1.4 DriverSlicer

The XPC mechanism supports the execution of Decaf Drivers, but does little on its own to simplify the writing of drivers. This task is achieved by the DriverSlicer tool, which enables creation of decaf drivers from existing kernel code written in C. DriverSlicer provides three key functions: (1) partitioning, to identify code that may run outside the kernel, (2) stub generation to enable communication across language and process boundaries, and (3) generation of the driver nucleus and user-level C code to start the porting process. Furthermore, DriverSlicer can regenerate stubs as the set of supported devices, driver data structures, and kernel interfaces change.

Partitioning

Given an existing driver, DriverSlicer automatically partitions the driver into the code that must remain in the kernel for performance or functionality reasons and the code that can move to user level. This feature is unchanged from the Microdrivers implementation of DriverSlicer. As input, it takes an existing driver and type signatures for *critical root functions*, *i.e.*, functions in the kernel-driver interface that must execute in the kernel for performance or functionality reasons. DriverSlicer outputs the set of functions reachable from critical root functions, all of which must remain in the kernel. The remaining functions can be moved to user level. In addition, DriverSlicer outputs the set of entry-point functions, where control transfers between kernel mode and user mode. The user-mode entry points are the driver interface functions moved to user mode. The kernel entry points are OS kernel functions and critical driver functions called from user mode.

Stub Generation

DriverSlicer creates stubs automatically based on the set of kernel and user entry points output from the partitioning stage. With the guidance of programmer annotations [61], DriverSlicer automatically generates marshaling code for each entry-point function. In addition, DriverSlicer emits code to marshal and unmarshal data structures in both C and Java, allowing complex data structures to be accessed natively in both languages.

Driver Generation

DriverSlicer emits C source code for the driver nucleus and the driver library. The driver library code can be ignored when functions are rewritten in another language. The source code produced is a partitioning of the original driver source code into two source trees. Files in each tree contain the same include files and definitions, but each function is in only one of the versions, according to where it executes.

To support driver evolution, DriverSlicer can be invoked repeatedly to generate new marshaling code as data structures change. The generated driver files need only be produced once since the marshaling code is segregated from the rest of the driver code.

3.1.5 Summary

The Decaf architecture achieves our four requirements. The decaf driver itself may be implemented in any language and runs at user level. The driver nucleus provides performance near that of native kernel drivers. DriverSlicer provides incremental conversion to C through automatic generation of stubs and marshaling code both for kernel-user communication and C-Java communication. Finally, DriverSlicer supports driver evolution through regeneration of stubs and marshaling code as the driver changes.

3.2 Implementation

We implemented the Decaf Drivers architecture for the Linux 2.6.18.1 kernel and re-wrote five device drivers into decaf drivers. We use a modified version of DriverSlicer from Microdrivers [61] to generate code for XPC stubs and marshaling, and implemented extensions to generate similar code between the driver library and decaf driver.

The driver nucleus is a standard Linux kernel module and the decaf driver and driver library execute together as a multithreaded Java application. Our implementation relies on Jeannie [71] to simplify calling from C into Java and back. Jeannie is a compiler that allows mixing Java and C code at the expression level, which simplifies communication between the two languages. Languages with native support for cross-language calls, such as C♯ [100], provide the ability to call functions in different languages, but do not allow mixing expressions in different languages.

Decaf Drivers provides runtime support common to all decaf drivers. The runtime for user-level code, the *decaf runtime*, contains code supporting all decaf drivers. The kernel runtime is a separate kernel module, called the *nuclear runtime*, that is linked to every driver nucleus. These runtime components support synchronization, object sharing, and control transfer.

3.2.1 Extension Procedure Call

Decaf Drivers uses two versions of XPC: one between the driver nucleus and the driver library, for crossing the kernel boundary; and another between the driver library and the decaf driver, for crossing the C-Java language boundary. XPC between kernel and user mode is substantially similar to that in Microdrivers, so we focus our discussion on communication between C and Java.

The Decaf implementation always performs XPCs to and from the kernel in C, which allows us to leverage existing stub and marshaling support from Microdrivers. An upcall from the kernel always invokes C code first, which may then invoke Java code. Similarly, downcalls always invoke C code first before invoking the kernel. While this adds extra steps when invoking code in the decaf driver, it adds little runtime overhead as shown by the experiments in Section 3.3.

3.2.1.1 Java–C Control and Data Transfer

Decaf Drivers provides two mechanisms for the decaf driver to invoke code in the driver library: direct cross-language function calls and calls via XPC. Direct calls may be used when arguments are scalar values that can be trivially converted between languages, such as arguments to low-level I/O routines. XPC must be used when arguments contain pointers or complex data structures to provide cross-language translation of data types. In addition, downcalls from the decaf driver to the driver nucleus require XPC.

```
Class Ens1371 {
  ...
  public static int snd_card_register(snd_card java_card) {
    CPointer c_card = JavaOT.xlate_j_to_c (java_card);      <- Consult object tracker
    int java_ret;
    begin_marshaling ();
    copy_XDR_j2c (java_card);                                <- Marshal arguments
    end_marshaling ();
    java_ret = `snd_card_register ((void *) `c_card.get_c_ptr());   <- Call C function
    begin_marshaling ();
    java_card = (snd_card) copy_XDR_c2j (java_card, c_card);  <- Marshal out parameters
    end_marshaling ();
    return java_ret;
  }
```

Figure 3.2: Sample Jeannie stub code for calling from Java to C. The backtick operator ‘ switches the language for the following expression, and is needed only to invoke the C function.

In both cases, Decaf Drivers relies on the Jeannie language [71] to perform the low-level transfer between C and Java. Jeannie enables C and Java code to be mixed in a source file at the granularity of a single expression. The backtick operator (‘) switches between languages. From a combined source file, the Jeannie compiler produces a C file, a Java file, and Java Native Interface code allowing one to call the other. Jeannie provides a clean syntax for invoking a Java function from C and vice versa. When invoking simple functionality in C, the decaf driver can inline the C code right into a Java function.

When invoking a function through XPC, Decaf Drivers uses RPC-style marshaling to transfer complex objects between Java and C. While Jeannie allows code in one language to read variables declared in the other, it does not allow modifications of those variables. Instead, Decaf uses the XDR marshaling standard [48] to marshal data between the driver library and the decaf driver, which we discuss in Section 3.2.2.3.

We write Decaf stubs in Jeannie to allow pure Java code to invoke native C code. The stubs invoke XDR marshaling code and the object tracker. Figure 3.2 shows an example of a stub in Jeannie. As shown in this figure, the following steps take place when calling from the decaf driver to the driver nucleus:

1. The decaf driver calls the Jeannie stub.

2. The stub invokes the object tracker to translate any parameters to their equivalent C pointers.

3. The stub, acting as an XPC client, invokes an XDR routine to marshal the Java parameters.

4. While marshaling these parameters, the XDR code uses inheritance to execute the appropriate marshaling routine for the object.

5. The same stub then acts as an XPC server, and unmarshals the Java objects into C.

6. While unmarshaling return values, the C stubs call specialized functions for each type.

We write stubs by hand because our code generation tools can only produce pure Java or C code, but the process could be fully automated.

3.2.1.2 Java-C Object Sharing

Object sharing maintains the relationship between data structures in different domains with an *object tracker*. This service logically stores mappings between C pointers in the driver library, and Java objects in the decaf driver. Marshaling code records the caller's local addresses for objects when marshaling data. Unmarshaling code checks the object tracker before unmarshaling each object. If found, the code updates the existing object with its new contents. If not found, the unmarshaling code allocates a new object and adds an association to the object tracker. For kernel/user XPC, the unmarshaling code in the kernel consults the object tracker with a simple procedure call, while unmarshaling code in the driver library must call into the kernel.

The different data representations in C and Java raise two difficulties. First, Java objects do not have unique identifiers, such as the address of a structure in C. Thus, the decaf runtime uses a separate user-level object tracker written in Java, which uses object references to identify Java objects. C objects are identified by their address, cast to an integer.

Second, a single C pointer may be associated with multiple Java objects. When a C structure contains another structure as its first member, both inner and outer structures have the same address. In Java, however, these objects are distinct. This difference becomes a problem when a decaf driver passes the inner structure as an argument to a function in the driver library or driver nucleus. The user-level object tracker disambiguates the uses of a C pointer by additionally storing a *type identifier* with each C pointer.

When an object is initially copied from C to Java, marshaling code adds entries to the object tracker for its embedded structures. When an embedded structure is passed back from Java to C, the marshaling code will search for a C pointer with the correct type identifier. The object tracker uses the address of the C XDR marshaling function for a structure as its identifier.

Once an object's reference is removed from the object tracker, Java's garbage collection can free it normally. We have not yet implemented automatic collection of shared objects, so decaf drivers must currently free shared objects explicitly. Implementing the object tracker with weak references [66] and finalizers would allow unreferenced objects to be removed from the object tracker automatically.

3.2.1.3 Synchronization

Decaf Drivers relies on kernel-mode *combolocks* from Microdrivers to synchronize access to shared data across domains [61]. When acquired only in the kernel, a combolock is a spinlock. When acquired from user mode, a combolock is a semaphore, and subsequent kernel threads must wait for the semaphore. Combolocks also provide support for multiple threads in the decaf driver and allow these threads to share data with the driver nucleus and driver library.

However, combolocks alone do not completely address the problem. The driver nucleus must not invoke the decaf driver while executing high priority code or holding a spinlock. We use three techniques to prevent high-priority code from invoking user-level code. First, we direct the driver to avoid interrupting itself: the nuclear runtime disables interrupts from the driver's device with `disable_irq` while the decaf driver runs. Since user-mode code runs infrequently, we have not observed any performance or functional impact from deferring code.

Second, we modify the kernel in some places to not invoke the driver with spinlocks held. For example, we modified the kernel sound libraries to use mutexes, which allowed more code to execute in user mode. In its original implementation, the sound library would often acquire a spinlock before calling the driver. Driver functions called with a spinlock held would have to remain in the kernel because invoking the decaf driver would require invoking the scheduler. In contrast, mutexes allow blocking operations while they are held, so we were able to move additional driver functions into the decaf driver.

Third, we deferred some driver functionality to a worker thread. For example, the E1000 driver uses a watchdog timer that executes every two seconds. Since

the kernel runs timers at high priority, it cannot call up to the decaf driver when the timer fires. Instead, we convert timers to enqueue a work item, which executes on a separate thread and allows blocking operations. Thus, the watchdog timer can execute in the decaf driver.

3.2.2 DriverSlicer Implementation

DriverSlicer automates much of the work of creating decaf drivers. The tool is a combination of OCaml code written for CIL [103] to perform static analysis and generate C code, Python scripts to post-process the generated C code, and XDR compilers to produce cross-language marshaling code. DriverSlicer takes as input a legacy driver with annotations to specify how C pointers and arrays should be marshaled and emits stubs, marshaling routines, and separate user and kernel driver source code files.

3.2.2.1 Generating Readable Code

A key goal of Decaf Drivers is support for continued modification to drivers. A major problem with DriverSlicer from microdrivers is that it only generated preprocessed driver code, which is difficult to modify. The Decaf DriverSlicer instead patches the original source, preserving comments and code structure. It produces two sets of files; one set for the driver nucleus and one set for the driver library, to be ported to the decaf driver. This patching process consists of three steps.

First, scripts parse the preprocessed CIL output to extract the generated code (as compared to the original driver source). This code includes marshaling stubs and calls to initialize the object tracker. Other preprocessed output, such as driver function implementations, are ignored, as this code will be taken from the original driver source files instead.

Second, DriverSlicer creates two copies (user and kernel) of the original driver source. From these copies, the tool removes function implemented by the other copy. Any functions in the driver nucleus source tree that are now implemented in the driver library and any functions in the driver library source tree that are implemented in the driver nucleus or the kernel are either replaced with stubs or removed entirely. The stubs containing marshaling code are placed in a separate file to preserve the readability of the patched driver.

Finally, DriverSlicer makes several other minor modifications to the output. It adds `#include` directives to provide definitions for the functions used in the

marshaling code, and adds a function call in the driver nucleus `init_module` function to provide additional initialization.

3.2.2.2 Generating XDR Interface Specifications

The decaf driver relies on XDR marshaling to access kernel data structures. DriverSlicer generates an XDR specification for the data types used in user-level code from the original driver and kernel header files. The existing annotations needed for generating kernel marshaling code are sufficient to emit XDR specifications.

Unfortunately, XDR is not C and does not support all C data structures, specifically strings and arrays. DriverSlicer takes additional steps to convert C data types to compatible XDR types with the same memory layout. First, DriverSlicer discards most of the original code except for `typedefs` and structure definitions. DriverSlicer then rewrites these definitions to avoid functionality that XDR does not support. For example, a driver data structure may include a pointer to a fixed length array. DriverSlicer cannot output the original C definition because XDR would interpret it as a pointer to a single element. Instead, DriverSlicer generates a new structure definition containing a fixed length array of the appropriate type, and then substitutes pointers to the old type with a pointer to the new structure type.

As shown in Figure 3.3, DriverSlicer converts pointers to an array into a pointer to a structure, allowing XDR to produce marshaling code. This transformation does not affect the in-memory layout. In this way, the generated marshaling code will properly marshal the entire contents of the array. After generating the C output, a script runs which makes a few syntactic transformations, such as converting C's `long long` type to XDR's `hyper` type. The result is a valid XDR specification.

3.2.2.3 Generating XDR Marshaling Routines

DriverSlicer incorporates modified versions of the `rpcgen` [122] and `jrpcgen` [2] XDR interface compilers to generate C and Java marshaling code respectively. These modifications to the original tools support object tracking and recursive data structures.

As previously mentioned in Section 3.2.1.2, the tools emit calls into the object tracker to locate existing versions of objects passed as parameters. The

```
Original Structure:
struct e1000_adapter {
  ...
  struct e1000_tx_ring test_tx_ring;
  struct e1000_rx_ring test_rx_ring;
  uint32_t * __attribute__((exp(PCI_LEN)))
                config_space;
  int msg_enable;
  ...
};
```

```
XDR input:
struct array256_uint32_t {
  uint32_t array[256] ;
};

typedef struct array256_uint32_t
*array256_uint32_ptr;

struct e1000_adapter_autoxdr_c {
  ...
  struct e1000_tx_ring test_tx_ring ;
  struct e1000_rx_ring test_rx_ring ;
  array256_unit32_ptr config_space ;
  int msg_enable ;
  ...
};
```

Figure 3.3: Portions of a driver data structure above, and the generated XDR input below. The names have been shortened for readability. The annotation in the original version is required for DriverSlicer to generate marshaling code between kernel and user levels.

generated unmarshaling code consults the object tracker before allocating memory for a structure. If one is found, the existing structure is used.

The DriverSlicer XDR compilers support recursive data structures, such as circular linked lists. The marshaling code checks each object against a list of the objects that have already been marshaled. When the tool encounters an object again, it inserts a reference to the existing copy instead of marshaling the structure again. This feature extends also across all parameters to a function, so that passing two structures that both reference a third results in marshaling

the third structure just once.

The output of DriverSlicer is a set of functions that marshal or unmarshal each data structure used by the functions in the interface. It also emits a Java class for each C data type used by the driver. These classes are containers of public fields for every element of the original C structures. The generated classes provide a useful starting point for writing driver code in Java, but do not take advantage of Java language features. For example, all member variables are public. We expect developers to rewrite these classes when doing more development in Java.

3.2.2.4 Regenerating Stubs and Marshaling Code

As drivers evolve, the functions implemented in the driver nucleus or the data types passed between the driver nucleus and the decaf driver may change. Consequently, the stubs and marshaling code may need to be updated to reflect new data structures or changed use of existing data structures. While this could be performed manually, DriverSlicer provides automated support for regenerating stubs and marshaling code. Simply re-running DriverSlicer may not produce correct marshaling code for added fields unless it observes code in the user-level partition accessing that field. If this is Java code, it is not visible to CIL, which only processes C code.

When the decaf driver requires access to fields not previously referenced, whether they are new or not, a programmer must inform DriverSlicer to produce marshaling code. DriverSlicer supports an annotation to the original driver code to indicate that a field may be referenced by the decaf driver. A programmer adds the annotation `DECAF_XVAR (y);` where X is an `R`, `W`, or `RW` depending on whether the Java code will read, write, or read and write the variable, and `y` is the variable name. These annotations must be placed in entry-point functions through which new fields are referenced.

Thus, the new annotations ensure that DriverSlicer generates marshaling code to allow reading and/or writing the new variables in the decaf driver. A programmer can add new functions to the user/kernel interface with similar annotations. In the future, we plan to automatically analyze the decaf driver source code to detect and marshal these fields. In addition, we plan to produce a concise specification of the entry points for regenerating marshaling code, rather than relying on the original driver source.

Source Components	# Lines
Runtime support	
Jeannie helpers	1,976
XPC in Decaf runtime	2,673
XPC in Nuclear runtime	4,661
DriverSlicer	
CIL OCaml	12,465
Python scripts	1,276
XDR compilers	372
Total number of lines of code	23,423

Table 3.1: The number of non-comment lines of code in the Decaf runtime and DriverSlicer tools. For the XDR compilers, the number of additional lines of code is shown.

3.2.3 Code Size

Table 3.1 shows the size of the Decaf Drivers implementation. The runtime code, consisting of user-level helper functions written in Jeannie and XPC code in user and kernel mode, totals 9,310 lines. This code, shared by all decaf drivers, is comparable to a moderately sized driver.

DriverSlicer consists of OCaml code for CIL, Python scripts for processing the output, and XDR compilers. As the XDR compilers are existing tools, we report the amount of code we added. In total, DriverSlicer comprises 14,113 lines.

3.3 Experimental Results

The value of Decaf Drivers lies in simplified programming. The cost of using Decaf Drivers comes from the additional complexity of partitioning driver code and the performance cost of communicating across domain boundaries. We have converted four types of drivers using Decaf Drivers, and report on the experience and the performance of the resulting drivers here. We give statistics for the code we have produced, and answer three questions about Decaf Drivers: how hard is it to move driver code to Java, how much driver code can be moved to Java, and what is the performance cost of Decaf Drivers?

We experimented with the five drivers listed in Table 3.2. Starting with existing drivers from the CentOS 4.2 Linux distribution (compatible with RedHat Enterprise Linux 4.2) with the 2.6.18.1 kernel, and we converted them to Decaf Drivers using DriverSlicer. All our experiments except those for the

E1000 driver are run on a 3.0GHz Pentium D with 1GB of RAM. The E1000 experiments are run on a 2.5GHz Core 2 Quad with 4GB of RAM. We used separate machines because the test devices were not all available on either machine individually.

3.3.1 Conversion to Java

Table 3.2 shows for each driver, how many lines of code required annotations and how many functions were in the driver nucleus, driver library, and decaf driver. After splitting code with DriverSlicer, we converted to Java all the functions in user level that we observed being called. Many of the remaining functions are specific to other devices served by the same driver. The column "Lines of Code" reports the quantity of code in the original driver. The final column, "Orig. LoC" gives the amount of C code converted to Java.

The annotations affect less than 2% of the driver source on average. These results are lower than for Microdrivers because of improvements we made to DriverSlicer to more thoroughly analyze driver code. In addition to the annotations in individual drivers, we annotated 25 lines in common kernel headers that were shared by multiple drivers. These results indicate that annotating driver code is not a major burden when converting drivers to Java. We also changed six lines of code in the `8139too` and `uhci-hcd` driver nuclei to defer functions executed at high priority to a worker thread while the decaf driver or driver library execute; the code is otherwise the same as that produced by DriverSlicer.

While we converted the `8139too` and `ens1371` to Java during the process of developing Decaf Drivers, we converted the other two drivers after its design was complete. For `uhci-hcd`, a driver previously converted to a microdriver, the additional conversion of the user-mode code to a decaf driver took approximately three days. The entire conversion of the `psmouse` driver, including both annotation and conversion of its major routines to Java, took roughly two days. This experience confirms our goal that porting legacy driver code to Java, when provided with appropriate infrastructure support, is straightforward.

In four of the five drivers, we were able to move more than 75% of the functions into user mode. However, we were only able to convert 4% of the functions in `uhci-hcd` to Java because the driver contained several functions on the data path that could potentially call nearly any code in the driver. We expect that redesigning the driver would allow additional code to move to user

Driver		Lines of	DriverSlicer	Driver nucleus		Driver library		Decaf driver		
Name	Type	code	Annotations	Funcs	LoC	Funcs	LoC	Funcs	LoC	Orig. LoC
8139too	Network	1,916	17	12	389	16	292	25	541	570
E1000	Network	14,204	64	46	1715	0	0	236	7804	8693
ens1371	Sound	2,165	18	6	140	0	0	59	1049	1068
uhci-hcd	USB 1.0	2,339	94	68	1537	12	287	3	188	168
psmouse	Mouse	2,448	17	15	501	74	1310	14	192	250

Table 3.2: The drivers converted to the Decaf architecture, and the size of the resulting driver components.

level. In the psmouse driver, we found that most of the user-level code was device-specific. Consequently, we implemented in Java only those functions that were actually called for our mouse device.

The majority of the code that we converted from C to Java is initialization, shutdown, and power management code. This is ideal code to move, as it executes rarely yet contains complicated logic that is error prone [116].

3.3.2 Performance of Decaf Drivers

The Decaf architecture seeks to minimize the performance impact of user-level code by leaving critical path code in the kernel. In steady-state behavior, the decaf driver should never or only rarely be accessed. However, during initialization and shutdown, the decaf driver executes frequently. We therefore measure both the latency to load and initialize the driver and its performance on a common workload.

We measure driver performance with workloads appropriate to each type of driver. We use netperf [46] sending and receiving TCP/IP data to evaluate the 8139too and E1000 network drivers with the default send and receive buffer sizes of 85 KB and 16 KB. We measure the ens1371 sound driver by playing a 256Kbps MP3 file. The uhci-hcd driver controls low-bandwidth USB 1.0 devices and requires few CPU resources. We measure its performance by untarring a large archive onto a portable USB flash drive and record the CPU utilization. We do not measure the performance of the mouse driver, as its bandwidth is too low to be measurable, and we measure its CPU utilization while continuously moving the mouse for 30 seconds. For all workloads except netperf, we repeated the experiment three times. We executed the netperf workload for a single 600-second iteration because it performs multiple tests internally.

We measure the initialization time for drivers by measuring the latency to run the insmod module loader. While some drivers perform additional startup activities after module initialization completes, we found that this measurement provides an accurate representation of the difference between native and Decaf Drivers implementations.

Table 3.3 shows the results of our experiments. As expected, performance and CPU utilization across the benchmarks was unchanged. With E1000, we also tested UDP send/receive performance with 1 byte messages. The throughput is the same as the native driver and CPU utilization is slightly higher.

To understand this performance, we recorded how often the decaf driver

Driver Name	Workload	Relative Performance	CPU Utilization		Init. Latency		User/Kernel Crossings
			native	Decaf	native	SymDrive	
8139too	netperf-send	1.00	14 %	13 %	0.02 sec.	1.02 sec.	40
	netperf-recv	1.00	17 %	15 %	–	–	–
E1000	netperf-send	0.99	2.8 %	3.7 %	0.42 sec.	4.87 sec.	91
	netperf-recv	1.00	20 %	21 %	–	–	–
ens1371	mpg123	–	0.0 %	0.1 %	1.12 sec.	6.34 sec.	237
uhci-hcd	tar	1.03	0.1 %	0.1 %	1.32 sec.	2.67 sec	49
psmouse	move-and-click	–	0.1 %	0.1 %	0.04 sec.	0.40 sec.	24

Table 3.3: The performance of Decaf Drivers on common workloads and driver initialization.

is invoked during these workloads. In the `ens1371` driver, the decaf driver was called 15 times, all during playback start and end. A watchdog timer in the decaf driver executes every two seconds in the `E1000` driver. The other workloads did not invoke the decaf driver at all during testing. Thus, the added cost of communication with Java has no impact on application performance.

However, the latency to initialize the driver was substantially higher, averaging 3 seconds. The increase stems from cross-domain communication and marshaling driver data structures. Table 3.3 includes the number of call/return trips between the driver nucleus and the decaf driver during initialization. We expect that optimizing our marshaling interface to transfer data directly between the driver nucleus and the decaf driver, rather than unmarshaling at user-level in C and re-marshaling in Java, would significantly reduce the cost of invoking decaf driver code.

3.4 Case Study: E1000 Network Driver

To evaluate the software engineering benefits of developing drivers in Java, we analyze the Intel E1000 gigabit Ethernet decaf driver. We selected this driver because:

- it is one of the largest network drivers with over 14,200 lines of code.
- it supports 50 different chipsets.
- it is actively developed, with 340 revisions between the 2.6.18.1 and 2.6.27 kernels.
- it has high performance requirements that emphasize the overhead of Decaf Drivers.

With this case study, we address three questions:

1. What are the benefits of writing driver code in Java?
2. How hard is it to update driver code split between the driver nucleus and the decaf driver?
3. How difficult is it to mix C and Java in a driver?

We address these questions by converting the E1000 driver from the Linux 2.6.18.1 kernel to a decaf driver. Overall, we converted 236 functions to Java

in the decaf driver and left 46 functions in the driver nucleus. There are no E1000-specific functions in the driver library. Of the 46 kernel functions, 42 are there for performance or functionality reasons. For example, many are called from interrupt handlers or with a spinlock held.

The four remaining functions are left in the driver nucleus because of an explicit data race that our implementation does not handle. These functions, in the `ethtool` interface, wait for an interrupt to fire and change a variable. However, the interrupt handler changes the variable in the driver nucleus; the copy of the variable in the decaf driver remains unchanged, and hence the function waits forever. This could be addressed with an explicit call into the driver nucleus to wait for the interrupt.

3.4.1 Benefits from Java

We identified three concrete benefits of moving E1000 code to Java, and one potential benefit we plan to explore.

Error Handling

We found the biggest benefit of moving driver code to Java was improved error handling. The standard practice to handle errors in Linux device drivers is through `goto` statements to a set of labels based on when the failure occurred. In this idiom, an `if` statement checks a return value, and jumps to a label near the end of the function on error. The labels are placed such that only the necessary subset of cleanup operations are performed. This system is brittle because the developer can easily jump to an incorrect label or forget to test an error condition [123].

Using exceptions to signal errors and nested handlers to catch errors, however, ensures that no error conditions are ignored, and that cleanup operations take place in the proper order. We rewrote 92 functions to use *checked exceptions* instead of integer return codes. The compiler requires the program to handle these exceptions. In this process, we found 28 cases in which error codes were ignored or handled incorrectly. Some, but not all, of these have been fixed in recent Linux kernels.

Figure 3.4 shows an example from the `e1000_open` function. This code catches and re-throws the exceptions; using a `finally` block would either incorrectly free the resources under all circumstances, or require additional code to ensure the resources are freed only in the face of an error.

```
Decaf driver code:
public static void e1000_open(net_device netdev)
      throws E1000HWException {
  e1000_adapteradapter = netdev.priv;
  int err;
    try {
        /* allocate transmit descriptors */
        e1000_setup_all_tx_resources(adapter);

        try {
            /* allocate receive descriptors */
            e1000_setup_all_rx_resources(adapter);

            try {
                e1000_request_irq(adapter);
                e1000_power_up_phy(adapter);
                e1000_up(adapter);
                ...
            } catch (E1000HWException e) {
                e1000_free_all_rx_resources(adapter);
                throw e;
            }
        } catch (E1000HWException e) {
            e1000_free_all_tx_resources(adapter);
            throw e;
        }
    } catch (E1000HWException e) {
        e1000_reset(adapter);
        throw e;
    }
  }
```

Figure 3.4: Code converted to nested exception handling.

Checked exceptions also reduce the amount of code in the driver. Figure 3.5 shows an example. By switching to exceptions instead of integer return values, we cut 675 lines of code, or approximately 8%, from `e1000_hw.c` by removing code to check for an error and return. We anticipate that converting the entire driver to use exceptions would eliminate more of these checks.

Object Orientation

We found benefits from object orientation in two portions of the E1000 driver. In `e1000_param.c`, functions verify module parameters using range and set-membership tests. We use three classes to process parameters during module initialization. A base class provides basic parameter checking, and the two derived classes provide additional functionality. The appropriate class checks each module parameter automatically. The resulting code is shorter than the original C code and more maintainable, because the programmer is forced by the type system to provide ranges and sets when necessary.

In addition, we restructured the hardware accessor functions as a class. In

Original Code:
```
if(hw->ffe_config_state == e1000_ffe_config_active) {
   ret_val = e1000_read_phy_reg(hw, 0x2F5B,
                                         &phy_saved_data);
   if(ret_val) return ret_val;

   ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
   if(ret_val) return ret_val;

   msec_delay_irq(20);
   ret_val = e1000_write_phy_reg(hw, 0x0000,
                    IGP01E1000_IEEE_FORCE_GIGA);
   if(ret_val) return ret_val;
```

Decaf driver Code:
```
if(hw.ffe_config_state.value == e1000_ffe_config_active) {
    e1000_read_phy_reg(0x2F5B, phy_saved_data);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
     DriverWrappers.Java_msleep (20);
     e1000_write_phy_reg((short) 0x0000,
            (short) IGP01E1000_IEEE_FORCE_GIGA);
```

Figure 3.5: Code from `e1000_config_dsp_after_link_change` in `e1000_hw.c`. The upper box shows the original code with error handling code. The lower box shows the same code rewritten to use exceptions.

the original E1000 driver, these functions all required a parameter pointing to an `e1000_hw` structure. Just rewriting this code as a class removed 6.5KB of code that passes this structure as a parameter to other internal functions. This syntactic change does not affect code quality, but makes the resulting code more readable.

Standard Libraries

In comparison to the Java collections library, the Linux kernel and associated C libraries provide limited generic data-structure support. We found that the Java collections library provides a useful set of tools for simplifying driver code. In addition to rewriting the parameter code to use inheritance, we also used Java hash tables in the set-membership tests.

Potential Benefit: Garbage Collection

While the E1000 decaf driver currently manages shared objects manually, garbage collection provides a mechanism to simplify this code and prevent

Category	Lines of Code Changed
Driver nucleus	381
Decaf driver	4690
User/kernel interface	23

Table 3.4: Statistics for patches applied to E1000: the lines changed in the driver nucleus, in the decaf driver, and to shared data structures requiring new marshaling code.

resource leaks. When allocating data structures shared between the driver nucleus and decaf driver, the decaf drivers use a custom constructor that also allocates kernel memory at the same time and creates an association in the object tracker.

Rather than relying on the decaf driver to explicitly release this memory, we can write a custom finalizer to free the associated kernel memory when the Java garbage collector frees the object. This approach can simplify exception-handling code and prevent resource leaks on error paths, a common driver problem [94].

3.4.2 Driver Evolution

We evaluate the ability of Decaf Drivers to support driver evolution by applying all changes made to the E1000 driver between kernel versions 2.6.18.1 and 2.6.27. Because we continue to use the 2.6.18.1 kernel, we omitted the small number of driver changes related to kernel interface updates. We applied all 320 patches in two batches: those before the 2.6.22 kernel and those after. Overall, we found that modifying the driver was simple, and that the driver nucleus and decaf driver could be modified and compiled separately.

The changes are summarized in Table 3.4. The vast majority of code changes were at user level. Thus, the bulk of the development on E1000 since the 2.6.18.1 kernel would have been performed in Java at user level, rather than in the kernel in C. Furthermore, only 23 changes affected the kernel-user interface, for example by adding or removing fields from shared structures.

In these cases, we modified the kernel implementation of the data structure, and re-split the driver to produce updated versions of the Java data structures. To ensure that new structure fields are marshaled between the driver nucleus and decaf driver, we added one additional annotation for each new field to the original driver. These annotations ensure that DriverSlicer generates marshaling

code to allow reading and/or writing the new variables in the decaf driver.

3.4.3 Mixing C and Java

A substantial portion of the Decaf architecture is devoted to enabling a mix of Java and C to execute at user level. We have found two reasons to support both languages. First, when migrating code to Java, it is convenient to move one function at a time and then test the system, rather than having to convert all functions at once (as required by most user-level driver frameworks). This code is temporary and exists only during the porting process. We initially ran all user-mode E1000 functions in this mode and then incrementally converted them to Java, starting with leaf functions and then advancing up the call graph. Our current implementation has no driver functionality implemented in the driver library.

Jeannie makes this transition phase simple because of its ability to mix Java and C code without explicitly using the Java Native Interface. The ability to execute either Java or C versions of a function during development greatly simplified conversion, as it allowed us to eliminate any new bugs in our Java implementation by comparing its behavior to that of the original C code.

Second, and more important, there may be functionality necessary for communicating with the kernel or the device that is not possible to express in Java. These functions are *helper routines* that do not contain driver logic but provide an escape from the limits of a managed language. Some examples we have observed include accessing the `sizeof()` operator in C, which is necessary for allocating some kernel data structures, and for performing programmed I/O with I/O ports or memory-mapped I/O regions. While some languages, including C♯, support unsafe memory access, Java does not. However, we found that none of these helper routines are specific to the E1000 driver, and as a result placed them in the decaf runtime to be shared with other decaf drivers. As before, Jeannie makes using these helper routines in Java a straightforward matter.

Jeannie also simplifies the user-level stub functions significantly. These stubs include a simple mixture of C and Java code, whereas using JNI directly would significantly complicate the stubs.

3.5 Summary

Device drivers are a major expense and cause of failure for modern operating systems. With Decaf Drivers, we address the root of both problems: writing kernel code in C is hard. The Decaf architecture allows large parts of existing drivers to be rewritten in a better language, and supports incrementally converting existing driver code. Drivers written for Decaf retain the same kernel interface, enabling them to work with unmodified kernels, and can achieve the same performance as kernel drivers. Furthermore, tool support automates much of the task of converting drivers, leaving programmers to address the driver logic but not the logistics of conversion.

Writing drivers in a type-safe language such as Java provides many concrete benefits to driver programmers: improved reliability due to better compiler analysis, simplified programming due to richer runtime libraries, and better error handling with exceptions. In addition, many tools for user-level Java programming may be used for debugging.

4 SYMDRIVEPROTO: TESTING DRIVERS WITHOUT DEVICES PROTOTYPE

4.1 Motivation and Background

The next three chapters present three versions of *SymDrive*, termed SymDrive-Proto, SymDrive, and SymDriveCluster. These systems test Linux and FreeBSD drivers without devices. For brevity, we use the terms SDP and SDC to refer to SymDriveProto and SymDriveCluster, respectively. This section provides some motivation for using symbolic execution to test drivers, and ignores most implementation details. In this section alone, we use the term SymDrive to encompass any and all three systems.

The goal of our work in all cases is to improve driver quality through thorough testing and validation. To be successful, SymDrive must demonstrate (i) usefulness, (ii) simplicity, and (iii) efficiency. First, SymDrive must be able to find bugs that are hard to find using other mechanisms, such as normal testing or static analysis tools. Second, SymDrive must require low developer effort to test a new driver and therefore support many device classes, buses, and operating systems. Finally, SymDrive must be fast enough to apply to every patch.

The rest of this chapter is organized as follows. This section provides some additional background and motivation for using symbolic execution with device drivers, and discusses the approach taken in all three systems in broad terms. The remainder of this chapter focuses specifically on SymDriveProto, our initial prototype implementation. In section 4.2, we introduce SDP as one possible solution. Section 4.3 describes SDP's overall design, while section 4.4 provides implementation details of SDP. Next, section 4.5 evaluates its effectiveness, and section 4.6 discusses some limitations of this implementation. Finally, section 4.7 concludes.

4.1.1 Symbolic Execution

All three versions of SymDrive build on modified versions of the KLEE and S^2E symbolic execution frameworks, which allow executing device-driver code without the device being present.

By itself, KLEE executes user-mode processes symbolically [24]. Using KLEE for symbolic execution requires the developer to first recompile the

program into LLVM bitcode using the LLVM compiler [84]. Once in bitcode form, the developer can run the program using KLEE and introduce symbolic data into any byte of the process address space. This approach is useful for running command-line programs symbolically, but is not as amenable to testing device drivers because they run in the kernel and receive input from both the kernel and device.

In contrast to KLEE, S^2E executes a complete virtual machine as the program under test [32]. Thus, S^2E allows symbolic data to be used anywhere in the operating system, including drivers and applications. S^2E acts as a virtual machine monitor (VMM) that schedules CPU time between different paths. Each path is treated like a thread, and the S^2E scheduler selects which path to execute and when to switch execution to a different path. S^2E supports *plug-ins*, which are modules loaded into the VMM that can be invoked to record information or to modify execution. SymDrive uses plugins to implement symbolic hardware, path scheduling, and code-coverage monitoring.

Symbolic execution is often used to achieve high coverage of code by testing all possible inputs. For device drivers, symbolic execution provides an additional benefit: executing without the device. Unlike most code, driver code can not be loaded and executed without its device present. Furthermore, it is difficult to force the device to generate specific inputs, which makes it difficult to thoroughly test error handling.

4.1.2 Why Symbolic Execution?

Symbolic execution eliminates the hardware requirement, because it can use symbolic data for all device input. Symbolic execution uses the driver itself as a model of device behavior: any device behavior used by the driver will be exposed as symbolic data. Moreover, it may provide inputs that correctly functioning devices do not. However, because hardware can provide similarly unexpected driver input [79], this unconstrained device behavior is reasonable. Drivers should not crash simply because the device provided an errant value.

Unexpected device input leads not only to crashes, but can also lead to security vulnerabilities. For example, in March 2013, Microsoft patched a USB kernel driver vulnerability that would allow a malicious user to take over any running Windows machine by using a carefully-programmed USB key [93], even if no user was logged in. Symbolic execution may have been able to uncover this vulnerability because it may have been able to force the driver into the

unsafe execution state.

One alternative to symbolic execution is to code a software model of the device [104]. In examining the complexity of the device models used in QEMU to emulate hardware [15], we estimate the size of the model to be comparable to that of the associated driver in many cases. This approach allows more accurate testing but greatly increases the effort required.

In comparison to static analysis tools, symbolic execution provides several benefits. First, it uses existing kernel code as a model of kernel behavior rather than requiring a programmer-written model. Second, because driver and kernel code actually execute, it can reuse kernel debugging facilities, such as deadlock detection, and existing test suites. Thus, many bugs can be found without any explicit description of correct driver behavior. Third, symbolic execution can invoke a sequence of driver entry points, which allows it to find bugs that span invocations, such as resource leaks. By running the driver in real kernel, symbolic execution allows the developer to make any desired sequence of entry point invocations. In contrast, most static analysis tools concentrate on bugs occurring within a single entry point.

4.1.3 Why not Symbolic Execution?

While symbolic execution has previously been applied to drivers with DDT and S^{2}E, there remain open problems that preclude its widespread use:

Efficiency

The engine creates a new path for every comparison, and branchy code may create hundreds or thousands of paths, called *path explosion*. This explosion can be reduced by distinguishing and prioritizing paths that complete successfully. This approach enables executing deeper into the driver: if driver initialization fails, the operating system could not otherwise invoke most driver entry points. S^{2}E and DDT require complex, manually written annotations to provide this information. These annotations depend on kernel function names and behavioral details, which are difficult for programmers to provide. For example, the annotations often examine kernel function parameters, and modify the memory of the current path on the basis of the parameters. The path-scheduling strategies in DDT and S^{2}E favor exploring new code, but may not execute far enough down a path to test all functionality.

Simplicity

Existing symbolic testing tools require extensive developer effort to test a single class of drivers, plus additional effort to test each individual driver. For example, supporting Windows NDIS drivers in S^2E requires over 1,000 lines of code specific to this driver class. For example, the S^2E wrapper for the `NdisReadConfiguration` Windows function consists of code to (a) read all of the call's parameters, which is not trivial because the code is running outside the kernel, (b) fork additional paths for different possible symbolic return codes, (c) bypass the call to `NdisReadConfiguration` along these additional error paths, and (d) register a separate wrapper function, of comparable complexity, to execute when this call returns. Since developers need to implement similarly complex code for many other functions in the driver/kernel interface, testing many drivers becomes impractical in these systems. Thus, these tools have only been applied to a few driver classes and drivers. Expanding testing to many more drivers requires new techniques to automate the testing effort.

Specification

Finally, symbolic execution by itself does not provide any specification of correct behavior: a "hello world" driver does nothing wrong, nor does it do anything right, such as registering a device with the kernel. In existing tools, tests must be coded like debugger extensions, with calls to read and write remote addresses, rather than as normal test code. Allowing developers to write tests in the familiar kernel environment simplifies the specification of correct behavior.

Thus, our work focuses on improving the state of the art to greatly simplify the use of symbolic execution for testing, and to broaden its applicability to almost any driver in any class on any bus.

4.1.4 Symbolic Execution with SymDrive

Symbolic execution is clearly an effective tool for analyzing device drivers because of its ability to execute drivers along multiple execution paths.

The next three chapters consider three different implementations of SymDrive. The first, SDP, builds on KLEE directly. We used this design because by building on our existing Microdrivers and Decaf Drivers framework, we knew we could move driver code out of the kernel. Once in user-mode, we could use KLEE to execute it. When we began work on SDP, S^2E was unavailable.

After implementing SDP, we discovered several design limitations that reduced its scalability. To address these limitations, we implemented SymDrive and SDC, which build on a combination of S^2E and KLEE, and allow developers to test more drivers in greater depth.

4.2 SDP Introduction

The remainder of this chapter focuses on the first version of SymDrive, called SymDriveProto (SDP), which was an early prototype. SDP meets our initial goal of being able to test drivers without devices.

We developed SDP to improve the quality of driver code. SDP uses *symbolic execution* to simulate all possible hardware inputs to a device driver while also eliminating the need for the device to be present. Should a failure along a specific branch of execution take place, SDP reports the precise set of data produced by the device that leads to the failure. However, symbolic execution alone does not specify what constitutes a failure.

SDP provides a test framework for conducting a partial verification and validation of the driver's implementation. This framework provides assertions over driver behavior, state variables for tracking the driver's logical state, and an object tracker to record the status of memory objects. At every control transfer between the driver and the kernel, the test framework interposes pre- and post-condition evaluation. Furthermore, the test framework infers the driver's class from its behavior, allowing class-specific checking of device drivers.

We target SDP at the Linux driver development process. Using a large database of bugs and kernel programming requirements culled from code, documentation, and mailing lists, we constructed 47 checkers to validate and verify driver code. These checkers enforce rules that maintainers commonly check during code reviews: matched allocation/free calls, matched lock/unlock calls, memory leaks, and proper use of kernel APIs. We also provide checkers to detect problems that commonly require subsequent patches, such as vulnerability to denial-of-service resource leaks. Finally, we provide a mechanism to compare the behavior before and after a patch, to verify that the interaction of the driver and device does not change. This is particularly helpful for collateral evolutions [107], when interface changes require driver modifications that commonly are not tested.

SDP is closely related to the recent DDT project [82]: both systems provide symbolic execution of drivers. The primary difference is that DDT targets

verification by end users, and therefore supports binary code and a small set of generic checks, such as proper memory and lock use. In contrast, SDP targets the development process, and focuses on testing specific patches. It includes a comprehensive test framework that allows sophisticated tests, such as those that ensure the driver uses APIs properly, and a differencing mechanism, to determine how a patch changes driver behavior.

We implement SDP using KLEE [24], User-Mode Linux [45], and Microdrivers [61]. We applied SDP to seven drivers (sound and network, USB and PCI buses), and found seven bugs. Overall, we found that SDP can:

- find bugs in drivers, such as memory leaks, incorrect uses of the kernel API, hardware dependence bugs, and mismatches between memory allocator and free routines.
- provide assurance that a refactored driver interacts with the hardware the same way as an unmodified driver along multiple execution paths.
- achieve 100% code coverage in complex driver functions.
- find several forms of generic bugs, such as illegal pointer dereferences, similar to other systems [11, 50, 52, 79, 82, 102].

4.3 Design

The SDP architecture focuses on thorough testing of driver *patches* to ensure that any new code meets its specifications. This approach reflects the evolutionary nature of development, in which developers check in a new module and then gradually evolve it over time through patches. Even small drivers, such as the NE2000 network driver have had dozens of patches in the last several years.

4.3.1 Design Overview

SDP executes drivers down a single execution path until the driver is unloaded or a developer-specified point. Then, SDP systematically explores other execution paths symbolically. Along each path through the driver, SDP checks all driver entry point and kernel function pre- and post-conditions. Any failure terminates the current execution path with a detailed error report, and execution continues along the next path.

The implementation of SDP is shown in Figure 4.1. The OS kernel executes in a virtual machine and communicates with the driver under test, executing in

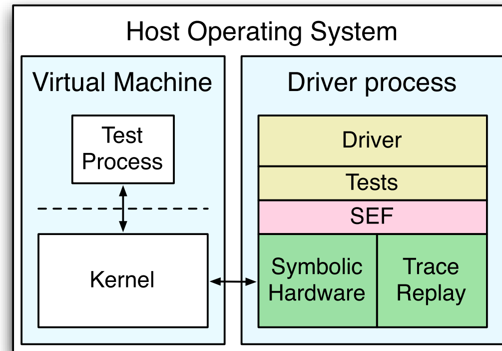

Figure 4.1: The SDP architecture.

a separate process, over IPC. The *symbolic execution framework* (*SEF*) governs the driver's execution as it runs symbolically. SDP provides *stubs* in the kernel and in the driver to copy data between the virtual machine and the driver.

4.3.2 Driver Interactions

A device driver communicates with two distinct entities: the device, through I/O requests, and the kernel, through function calls. We next describe how SDP handles both kinds of interaction.

4.3.2.1 Driver/Device Interaction

Drivers interact with devices according to well-defined, narrow interfaces. For PCI device drivers, this interface is comprised of I/O memory, port I/O, DMA memory, and interrupts. For USB drivers, the interface uses USB request blocks (URBs) that are sent to and from the device. While most previous driver research focuses on PCI devices, we include USB devices as they are prevalent, and USB versions of most standard devices (*e.g.*, network, storage, sound, and video controllers) are available.

For PCI devices, SDP provides symbolic data of the appropriate size each time the driver performs a read operation using memory or port I/O. Similarly, SDP treats the contents of DMA memory as symbolic. To accomplish this, we provide wrappers for the DMA allocation functions, such as

`pci_alloc_consistent`. When the driver allocates DMA memory, SDP allocates the region and makes its contents symbolic. SymDrive provides symbolic interrupts by directly invoking the driver's interrupt handler. Thus, SDP effectively simulates the presence of a piece of hardware, albeit one that might also return unexpected values.

USB drivers behave differently because they use an asynchronous packet-based protocol to communicate with the device. These packets, URBs, are used both to send and receive data. A driver submits an URB to the kernel USB framework, which then sends it through a USB host controller to the device. Packet submission is quick, and control returns to the driver immediately, before the device has a chance to respond. The URB includes a completion routine that the kernel invokes when the device sends its response. SDP provides symbolic USB hardware with symbolic URBs: a symbolic USB device sends a packet of symbolic data, including a constrained symbolic size to allow responses of different lengths. Similar to interrupts, SDP invokes completion routines spontaneously.

An alternative approach to supporting USB is to run the USB host controller driver on top of a symbolic PCI device. In doing so, we would supply the symbolic data at the lowest possible level in the stack. The USB device driver then would run on top of the host controller driver, and execute accordingly. Unfortunately, this approach is infeasible because USB URBs contain structured data, and the SDP would waste most of its time testing the underlying host controller's response to these invalid URBs.

4.3.2.2 Driver/Kernel Interaction

The driver also interacts with the kernel through calls to kernel functions for service. To ensure that symbolic execution is faithful to normal execution, SDP must ensure that the kernel responds appropriately when the driver invokes these functions. However, a difficulty arises when the driver forks symbolic execution: it can now invoke the kernel twice, on different execution paths. As described above, with SDP the kernel executes normally, preventing it from forking execution. Thus, the kernel can only be invoked along one of the many paths.

One solution to this problem is to fork the kernel's execution state when the driver itself forks, as in DDT [82]. However, this approach is not compatible with our symbolic execution framework, because it would require forking the

virtual machine running the OS, which is both time and memory intensive.

Instead, SDP supports two methods of interaction with the kernel, each with different benefits and drawbacks. Developers can choose the execution mode on a per-kernel function basis, and can vary it from one test run to another.

Concrete + Symbolic

In this approach, the driver executes kernel functions normally along one execution path through the driver, using a depth-first-search approach to choosing an execution path. Once this path terminates, all subsequent calls to the specified kernel function do not actually invoke the kernel; instead, return values and structure fields modified by the kernel function are marked symbolic. We determine the set of fields through static analysis of the kernel. Thus, the driver will execute with all possible (and a few more) return values. This execution strategy ensures that kernel functions that cause callbacks to the driver, such as registration functions (*e.g.*, `pci_register_driver`), correctly invoke the callback function. However, false positives are possible, as the driver can execute with return values the kernel would never generate.

Symbolic only

In this approach, the kernel is never invoked, even on the first path. Again, structure fields of any function parameters that the kernel might modify are marked as symbolic. This approach provides more thorough testing of driver code, because it is not constrained by a specific set of concrete values returned by the kernel.

As an example, suppose that the developer wishes to test the network driver `probe` function thoroughly. In this case, the developer could set the `pci_register_driver` to call into the kernel using the *concrete + symbolic* option, thus causing the kernel to invoke the driver's probe function. For all other kernel functions, the developer could use the *symbolic only* technique, to ensure that maximum code coverage is possible. By default, all kernel functions execute using the *concrete + symbolic* approach, which we have found represents an effective solution.

4.3.3 Limiting Path Explosion

Symbolic execution, by executing all possible paths through the kernel, can lead to very long testing times. Two important reasons for this are reentrancy, which

causes multiple threads to execute in the driver simultaneously, and lengthy, control-heavy driver initialization routines.

4.3.3.1 Driver reentrancy

Reentrancy in drivers arises for a variety of reasons, including interrupts, timer and work-queue callbacks, URB completions, and simultaneous calls from different threads. To explore all possible paths fully in the presence of reentrance, the symbolic execution framework would have to invoke all possible functions at every instruction boundary in a driver, which could cause a huge explosion of paths to explore.

To make testing more tractable, SDP restricts reentrant calls to occur only when control is transferred between the driver and the kernel. For example, SDP invokes the driver's interrupt handler both before calling a driver function and when the driver function returns. Similarly, USB drivers communicate with hardware asynchronously using the event-driven URB packet interface. Because SDP does not allow the driver to communicate with hardware, it instead calls completion routines at the same time it calls interrupt handlers: whenever control transfers between the driver and kernel.

For simultaneous calls into the driver, SDP restricts concurrency by only allowing one thread into the driver at a time; other threads block in the kernel while the thread executes. The same approach is used for timer and work-queue callbacks to prevent them from being delivered while a thread executes in the driver. This approach effectively serializes access to the driver, though, and prevents SDP from effectively testing for concurrency bugs.

4.3.3.2 Driver initialization

State explosion arises when symbolic values used in conditional statements cause execution to fork repeatedly, once for each path through the code. Each path results in a new symbolic execution state. Figure 4.2 shows a piece of driver code in which each conditional forks execution. Too many states cause symbolic execution to either take a long time to complete, or to stop testing some states. The problem is particularly acute for drivers that contain many conditional statements for checking device response status bits and chipset identifiers, especially in initialization code. Drivers often support many specific devices, and routinely contain extensive branching logic to implement this support.

```
e1000_config_dsp_after_link_change(…) { …
  if (hw->ffe_cfg_state == e1000_ffe_cfg_active) {
    ret_val = e1000_read_phy_reg(hw, 0x2F5B, &saved);
    if (ret_val) return ret_val;
    ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
    if (ret_val) return ret_val;
    mdelay(20);
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    mdelay(20);
    ret_val = e1000_write_phy_reg(hw, …);
    if (ret_val) return ret_val;
    hw->ffe_config_state = e1000_ffe_config_enabled;
  } …
}
```

Figure 4.2: State space explosion with symbolic execution. Each if-statement forks a new path of execution because `ret_val` is symbolic.

SDP provides a *replay* mechanism to fast-forward driver execution to the point of interest, such as the code affected by a patch. First, a developer with access to hardware creates a trace of all driver/device interactions with a trace tool. The trace includes all data provided by the hardware, and enough context information to allow it to be used for replay. Later, during testing, SDP can either provide symbolic values or provide values from the trace tool, which allows the driver to execute concretely on a single path.

Furthermore, device-driver initialization often relies on a large number of highly specific inputs from the device in order to proceed. For example, the E1000 network driver code shown in Figure 4.3 calculates a checksum over the device EEPROM data, and only continues if the checksum is valid. Forcing symbolic execution to derive the necessary constraints to satisfy the checksum calculation is infeasible, but replay provides a simple way to fast-forward over this code by providing the real EEPROM data from the device. Replay also simplifies testing of local changes: rather than testing all driver execution, including initialization, a developer may replay device interactions up to the changed code, and then switch to symbolic mode. Note that while the device is needed to generate the trace, this can be done once, and the trace can then be

```
unsigned short eeprom_data, i;
for (i = 0; i < (EEPROM_CHECKSUM_REG + 1); i++) {
    e1000_read_eeprom(..., &eeprom_data);
    checksum += eeprom_data;
}
if (checksum == (u16)EEPROM_SUM)
    return E1000_SUCCESS;
else return -E1000_ERR_EEPROM;
```

Figure 4.3: Symbolic execution is not appropriate for checksum calculations, because performance would be poor and the developer would gain little insight into the driver's behavior.

used to execute tests on a machine where the device is not present.

4.3.4 Testing Drivers

Symbolic execution can detect whether a driver will perform an illegal operation, such as reference an invalid address, on any execution path through a function. However, this proof does not provide any general correctness guarantees: a "hello world" program will not crash but will also not bring up an Ethernet interface properly.

Thus, achieving high-quality driver code requires both *validating* and *verifying* its operation to the extent practical. *Validation* establishes that the driver is solving the right problem and demonstrates that the driver does what it is supposed to do, such as initializing a data structure or registering with the kernel. In contrast, *verification* establishes that the implementation follows the rules to solve the problem. It ensures the absence of known bugs such as memory or lock leaks and null pointer dereferences. With SDP, we seek to ensure both driver verification and validation takes place.

To provide verification and validation functionality, the second major component of SDP is a *test framework*, which is used to determine whether a symbolically executing driver is behaving correctly. However, because SDP shares many design details with SymDrive, we will defer discussion of the test framework until the following chapter, and instead focus on SDP's overall design.

4.4 Implementation

This section describes the implementation of SDP, which includes the components outlined in Figure 4.1. SDP consists of five major components:

- A *virtual machine* for executing the kernel under test and a test program.
- *Stubs* for remote communication between the kernel and the driver.
- A *symbolic execution framework* for tracking symbolic data values and constraints, with symbolic hardware.
- A *trace generator* for creating traces of driver/device interaction and a replayer for use during testing.
- The *test framework* and associated API for invoking test code before and after invoking the driver.

We next describe the implementation of each component in turn.

4.4.1 Virtual Machine

SDP uses User Mode Linux (UML) as a virtual machine to provide a model of the kernel's execution. Because UML does not provide any hardware support, SDP implements virtual PCI and USB buses that cause the kernel to invoke the driver's functionality. These virtual buses invoke driver entry points, and provide easily configurable virtual devices to support operation of the desired drivers.

Creating a virtual PCI device requires a few basic identifiers, such as the device manufacturer and class. This information is available in existing device drivers.

In contrast, creating a simulated USB device is more involved. When a USB device is inserted, the USB core code in Linux exchanges several packets with the device to obtain the device's supported configurations, interfaces, and endpoints, in addition to the basic information of manufacturer and device class. SDP includes a tool to reproduce the details of the actual device in a virtual equivalent, allowing this information to be replayed to the virtual USB bus during testing.

4.4.2 Remote Driver Execution

SDP reuses DriverSlicer from Microdrivers [61], to enable the device driver to execute outside of the kernel. DriverSlicer generates stubs and marshaling code to transfer data and control between the driver and kernel appropriately. This control and data transfer takes place via an RPC mechanism over named pipes. This approach appears to the kernel as an architecture-specific feature that pauses kernel execution, which enables communication with the driver process even for high-priority code such as the packet transmit routine in network drivers.

In order to marshal data, a developer must annotate kernel data structures indicating how pointers are used. For example, character pointers must be annotated to distinguish between a null-terminated string, an array of bytes, and a single character. These annotations allow DriverSlicer to generate code to marshal and unmarshal data structures properly. Fortunately, the annotations are almost exclusively in kernel code, which need only be annotated once. DriverSlicer limitations also necessitate additional annotations in individual drivers in some cases, but DriverSlicer tells the developer where the annotation is necessary and how to write the annotation, making them easy to add.

4.4.3 Symbolic Execution with KLEE

SDP uses a modified version of KLEE [24] for symbolic execution. KLEE provides the execution environment and constraint solving capability necessary for symbolic execution. It also manages the forking that takes place as symbolic values are compared against each other. All driver code, including the test framework, executes using KLEE. The kernel executes natively, without KLEE.

By default, KLEE employs a depth-first search strategy (DFS) to explore the state space from the symbolic execution. To resolve issues related to driver loops performing repeated symbolic read operations, SDP switches to a breadth-first search strategy (BFS) once the developer unloads the driver. Without using BFS, SDP would waste too much time exploring uninteresting execution paths. DDT employs a similar heuristic that favors unexecuted code, though the purpose is the same [82].

4.4.4 Replay

The replay mechanism provides device inputs for the driver from a *hardware trace* to fast forward it to the point of interest. SDP includes a tracer that modifies a driver to log all interactions with the device. Instrumenting a driver requires the developer to add a single C preprocessor directive to the top of each driver file. The resulting instrumented driver uses customized device I/O routines for logging hardware interaction.

Running the instrumented driver on a system with compatible hardware produces a log of all device I/O operations. In the case of PCI drivers, the trace indicates the function name and line number of the original read operation, as well as the I/O operation type, such as `inb`, and the port number and data read. For USB drivers, the trace includes all the URBs received by the driver instead, as well as results from synchronous USB device interface functions such as `usb_control_msg`. The trace is a simple text file that developers can store in a source repository or share over the Internet.

Executing the driver with SDP will use the trace when it provides relevant data for the driver (*e.g.*, the next trace entry is for the correct hardware read operation, port, and driver function). Otherwise, SDP provides symbolic values. When a developer wants to test specific driver functionality, he or she comments out parts of the trace, thus forcing SDP to supply symbolic data at the desired point.

4.5 Evaluation

The purpose of the evaluation is to verify that SDP achieves its goal: can it verify and validate driver code, and can it do so without the device present? We test whether SDP can thoroughly test a variety of drivers, and whether it can find driver bugs unknown to us.

We have tested SDP on the drivers shown in Figure 4.1. These are the devices for which we have hardware, to generate traces, and for which we have annotated the kernel/driver interfaces. These drivers include sound and network drivers on both the PCI and USB buses. We report line counts using the CLOC utility [1]. In addition to executing these drivers symbolically, SDP executes a large fraction of the Linux kernel sound driver API symbolically. This sound library contains 30,180 lines of code. All tests took place on a machine running Red Hat Enterprise Linux 5.5 equipped with a quad-core Intel 2.66GHz Q9400

Driver	Class	Bus	LoC
8139too	Network	PCI	1904
ca0106	Sound	PCI	3052
cmipci	Sound	PCI	2717
e1000	Network	PCI	13973
ens1371	Sound	PCI	2110
pegasus	Network	USB	1541
usb-audio	Sound	USB	8094

Table 4.1: We have tested SDP with the seven PCI and USB drivers shown here.

CPU and 8GB of memory.

4.5.1 Invoking the Driver with SDP

Using SDP by itself does not invoke driver entry points. To test a driver, we carry out the following operations for each driver:

1. Create a virtual hardware device in UML by loading a module with appropriate parameters.
2. Load the driver with `insmod` and wait for initialization to complete.
3. Execute a workload.
4. Unload the driver
5. Allow SDP to continue symbolically executing the driver's code for another 10 minutes or less. If SDP is unable to cover all possible paths in that time, we abort further execution.

To test specific driver functionality, we execute workloads that trigger the appropriate driver entry point. For example, when the kernel attempts to send packets, it invokes the network driver's `start_xmit` routine. SDP itself calls the driver's interrupt handlers. Using `ifconfig` or related `ethtool` commands also invoke driver entry points. Similarly, the `mpg123` music player exercises a large piece of driver code by attempting to play an MP3 file.

In addition to these tests, we verified that SDP achieves 100% code coverage in several complex driver functions.

4.5.2 Bugs Found

We applied SDP with the checkers described in the previous section to the seven drivers listed in Table 4.1. Across these drivers, we found six distinct bugs, one of which appeared in two different drivers.

1. An allocator/deallocator mismatch in the e1000 driver. SDP reported that the attempt to free the pointer was invalid because the driver allocated it with another function.

2. A memory leak in the ca0106 driver. SDP reported leaked objects after unloading the driver.

3. A missed call to `put_device` on a `device_register` failure path in the sound library. SDP reported that the driver never called `put_device` after unloading the driver.

4. A hardware dependence bug in 8139too. SDP reported an invalid pointer dereference along an unlikely execution path.

5. A race condition in ca0106, in which the driver cleared a function pointer used in the interrupt handler before stopping interrupts.

6. Use of the `GFP_ATOMIC` memory flag in contexts that may block in both the pegasus and usb-audio drivers, which is unnecessary.

In addition, SDP reported several unusual code fragments, such as calls to `spin_lock_irqsave` followed by `spin_unlock`, `spin_lock`, and finally `spin_unlock_irqrestore`. Although correct, this code deserves additional review. SDP also reported a redundant permission check in the `ioctl` function of the E1000 driver.

We verified bugs #1, #2, and #4, and found them fixed in more recent kernel versions. Bugs #3, #5, and #6 we checked manually. The kernel's `device_register` function has a comment specifically reminding developers to call `put_device` if the call fails. We verified the `GFP_ATOMIC` issue in pegasus and usb-audio both by examining the functions in the stack traces and ensuring that the driver could safely block.

To find these bugs, we executed each driver using the workloads described earlier. We used the *concrete+symbolic* execution mode in all cases, except for the `device_register` bug. To find that bug, we used the *symbolic only*

execution mode for the `device_register` function. The ca0106 driver did not require a hardware trace to discover the memory leak or race condition, thus demonstrating the value of SDP even when a trace is not available.

Both the pegasus and usb-audio drivers (bugs listed under #6) are designed such that the driver always submits URBs using the `GFP_ATOMIC` flag even though the `GFP_KERNEL` flag would have been a better choice in some cases. This design stems from reusing the same code on both high and low priority paths, and no mechanism is currently present to distinguish the priority levels, so the driver conservatively chooses `GFP_ATOMIC` under all circumstances.

These bugs clearly demonstrate the diversity of potential issues facing driver developers. Although these bugs do not necessarily result in a driver crash, with the exception of bug #4, they all represent issues that need addressing. Furthermore, they demonstrate the importance of using automated tests, because using the drivers normally would not uncover the bugs.

4.5.3 Refactoring

To verify that our refactoring checks work correctly, we verify that SDP can distinguish between patches that change the driver/device interactions and safe refactorings that do not. We consider two existing drivers, 8139too and pegasus, apply five patches to each driver from the mainline kernel that refactor the code to improve readability or to upgrade the driver to current kernel programming practices. We use SDP to execute the original and the patched drivers and record the hardware interactions.

In each case, comparing the prefix tree representations of each driver's interaction with the hardware revealed no changes, because the refactoring is designed not to impact the driver's behavior. Thus, SDP confirmed that the refactorings do not affect the driver/device interaction.

To verify that patches to the driver/device interface are detected, we applied a recent patch to 8139too that changes the order of two low-level device operations and to pegasus that also changed its interaction with the hardware. The result in both cases clearly shows that the hardware operations were no longer the same. In the case of 8139too, the result was very clear because the trie showed the order of the two operations had reversed. The result from pegasus, a USB driver, was more difficult to interpret, because the change occurred in a function several calls removed from the actual device access. Nevertheless, it was clear that the patch impacted the driver's interaction with the hardware.

The information provided by the prefix tree in both cases provides enough detail that a developer could find the affected lines of code quickly.

4.5.4 Driver Setup Time

Testing a driver with SDP requires some additional effort beyond the normal kernel build process. After developing the bulk of SDP, we added support for the cmipci sound driver. It took less than four hours to run cmipci with SDP for the first time, of which three hours were spent adding support to our infrastructure for functionality not used by any of our previous drivers. We expect this effort to decrease as we test more drivers. The remaining time was spent setting up the driver with our build environment (5 minutes), installing the device in a test machine and recording a trace (25 minutes), annotating the driver for DriverSlicer (10 minutes), and finally 15 minutes to actually test the driver. A second developer with access to the trace would need only 15 minutes to build and test the driver symbolically.

4.5.5 Driver Execution

A final consideration in testing is execution time; fast tests can be run more frequently and find bugs earlier. In order to measure execution time, we timed how long it takes to load and unload each driver. We perform this test both using KLEE and symbolic execution and running the driver as a normal binary to measure the additional overhead of executing a driver with the SEF. We loaded and unloaded each driver three times, and used a trace to minimize the amount of symbolic data used. Refactoring trace support was enabled in all tests. Table 4.2 shows the results. Native execution is fairly fast, taking under 7 seconds, while symbolic execution can take over seven minutes. These tests times are reasonable when compared to the time it takes to install a driver and reboot a test machine.

We also measured the time it takes to execute a function symbolically, which varies based on how frequently it interacts with the device. As an example of how long it takes to thoroughly evaluate a function heavy on hardware operations, we consider the 8139too function `rtl8139_init_board`. SDP took 92 seconds to execute 113 paths through this function, on top of the 10 seconds shown in Figure 4.2 for the remaining driver load/unload code. Thus, testing even a complex function requires little additional time.

Driver Load/Unload Times (seconds)

	Native		With KLEE	
Driver	`insmod`	`rmmod`	`insmod`	`rmmod`
8139too	0.3	0.4	4.0	4.5
ca0106	2.5	0.5	56.5	6.9
cmipci	2.2	0.3	30.3	4.4
e1000	6.1	2.1	59.7	368.0
ens1371	3.6	0.5	89.5	14.2
pegasus	0.9	0.3	35.4	6.6
usb-audio	3.8	0.3	188.4	14.9

Table 4.2: This table shows how long it takes to load and unload each driver using the trace, in seconds. The second and third columns show the times when SDP is compiled without KLEE, and driver code executes natively, while the last two columns show execution time with KLEE.

We also examined memory usage. With symbolic execution, memory usage is highly dependent on the number of symbolic variables used and the number of paths executed. After loading the large usb-audio driver, with the trace, Linux reported that the driver process had a resident set size of approximately 194MB. We then modified the trace to exclude the `snd_usb_ctl_msg` function, which the driver uses extensively for hardware communication. After loading and unloading the driver a second time, we let it execute 378 additional execution paths. During this time, the largest memory usage we observed was 243 MB. In our tests, we have never seen the driver process use more than approximately 1GB of memory, because SDP focuses execution on specific driver function. However, if the developer employs a large amount of symbolic data in a branch heavy piece of code, the driver process may consume significant system resources.

4.6 Limitations

SDP suffers from three significant limitations, and a number of minor ones.

The first significant limitations relates to SDP's bug finding capability. Because SDP aggressively concretizes symbolic data in order to keep the kernel in a consistent state, SDP may miss bugs along other paths. This concretization happens whenever the driver invokes the kernel, so SDP is often unable to invoke many feasible code paths even within a single function. The solution here is to fork the kernel and operating system state in order to avoid these additional concretization operations. However, because SDP runs the User-

Mode Linux kernel natively, this approach is not possible. This limitation is the most significant, and precludes SDP from finding bugs that manifest on many code paths.

The second significant limitation relates to SDP's reliance on device trace data to fast-forward execution, which imposes some reliance on device hardware. Although developers can transmit this information easily via the Internet, it may still be difficult to test drivers that require unusual devices. In addition, SDP's use of a trace appears to conflict with SDP's goal of identifying chipset-specific bugs. Many device drivers support multiple chipsets, and may invoke some code paths only when a specific chip is present. SDP could mitigate this problem by executing drivers repeatedly with traces from each supported device provides complete coverage, but this approach is cumbersome.

We will next consider some minor limitations of SDP. SDP does report false positives under two circumstances. First, because SDP terminates driver execution when executing a symbolic path that returns to the kernel, the object tracker incorrectly reports some driver objects as having not been freed, when in reality, the driver would free them if SDP allowed it to continue executing.

SDP also reports false positives by default with the various `spin_lock` functions, if the driver does not pair them in a standard way. The example from the sound library, in which `spin_lock_irqsave` was matched with `spin_unlock`, is an example. In this case, SDP must make a trade-off between finding actual bugs, in which the developer accidentally mismatched the locking functions, and false positives, as in the example.

We are not currently aware of any specific false negatives among the checks that SDP does support. However, reduced reentrancy prevents SDP from detecting race conditions and deadlocks. Fortunately, various other static and dynamic approaches have demonstrated success at finding these synchronization errors, thus reducing the need for SDP to address the same problem [50, 57]. As processing power and symbolic execution techniques improve, it may become reasonable to reduce or eliminate this restriction, by instead simulating how driver functions would behave if their executions were interleaved.

The symbolic bus implementation in SDP also has limitations. Although we successfully used it to test PCI and USB drivers, the symbolic PCI bus would take considerable effort to set up correctly for all drivers. SDP's symbolic bus design is not scalable because it relies on re-implementing the existing Linux interface for each bus. Because Linux developers frequently change this interface, SDP would also require constant revision. Implementing support for

more buses in SDP is extremely challenging.

After we completed the development of SDP, the S^2E tool became available. As we will see in the next chapter, S^2E allows us to overcome many of these limitations. In particular, by leveraging what we learned here and by using S^2E, we can eliminate the need for the hardware trace while also finding more bugs.

4.7 Conclusions

SDP uses symbolic execution combined with a driver test infrastructure to test driver code without physical access to the corresponding device. Our results show that SDP can find subtle bugs in mature driver code, and can distinguish between patches that affect the driver/device interface and those that do not. Thus, SDP is a valuable tool that eases the burden of testing drivers.

Nevertheless, SDP has several key limitations. First, it requires creating a hardware trace before it can execute drivers meaningfully. Second, it never forks the kernel state, which causes it to miss some execution paths and, consequently, bugs. Finally, it is not scalable: each driver requires a number of manual code annotations. The next chapter discusses a reimplementation of this system simply called SymDrive.

5 SYMDRIVE: TESTING DRIVERS WITHOUT DEVICES

Although the SDP prototype is an effective tool for testing some drivers without their devices, it has several key limitations. After gaining experience with SDP, we decided to apply what we learned to create a new tool, called SymDrive, which addresses many of these limitations.

SymDrive uses static analysis to identify key features of the driver code, such as entry-point functions and loops. With this analysis, SymDrive produces an instrumented driver with callouts to test code that allows many drivers to be tested with no modifications. The remaining drivers require a few annotations to assist symbolic execution at locations that SymDrive identifies.

We designed SymDrive for three purposes. First, a driver developer can use SymDrive to test driver patches by thoroughly executing all branches affected by the code changes. Second, a developer can use SymDrive as a debugging tool to compare the behavior of a functioning driver against a non-functioning driver. Third, SymDrive can serve as a general-purpose bug-finding tool and perform broad testing of many drivers with little developer input.

SymDrive is built with the S^{2}E system by Chipounov et al. [32, 33] , which can make any data within a virtual machine symbolic and explore its effect. SymDrive makes device inputs to the driver symbolic, thereby eliminating the need for the device and allowing execution on the complete range of device inputs. In addition, S^{2}E enables SymDrive to further enhance code coverage by making other inputs to the driver symbolic, such as data from the applications and the kernel. When it detects a failure, either through an invalid operation or an explicit check, SymDrive reports the failure location and inputs that trigger the failure.

SymDrive extends S^{2}E with three major components. First, SymDrive uses *SymGen*, a static-analysis and code transformation tool, to analyze and instrument driver code before testing. SymGen automatically performs nearly all the tasks previous systems left for developers, such as identifying the driver/kernel interface, and also provides hints to S^{2}E to speed testing. Consequently, little effort is needed to apply SymDrive to additional drivers, driver classes, or buses. As evidence, we have applied SymDrive to eleven classes of drivers on five buses in two operating systems.

Second, SymDrive provides a *test framework* that allows *checkers* that validate driver behavior to be written as ordinary C code and execute in the

kernel. These checkers have access to kernel state and the parameters and results of calls between the driver and the kernel. A checker can make pre- and post-condition assertions over driver behavior, and raise an error if the driver misbehaves. Using bugs and kernel programming requirements culled from code, documentation, and mailing lists, we wrote 49 checkers comprising 564 lines of code to enforce rules that maintainers commonly check during code reviews, such as matched allocation/free calls across entry points, no memory leaks, and proper use of kernel APIs.

Finally, SymDrive provides an *execution-tracing* mechanism for logging the path of driver execution, including the instruction pointer and stack trace of every I/O operation. These traces can be used to compare execution across different driver revisions and implementations. For example, a developer can debug where a buggy driver diverges in behavior from a previous working one. We have also used this facility to compare driver implementations across operating systems.

We demonstrate SymDrive's value by applying it to 26 drivers, and find 39 bugs, including two security vulnerabilities. We also find two driver/device interface violations when comparing Linux and FreeBSD drivers. To the best of our knowledge, no symbolic execution tool has examined as many drivers. In addition, SymDrive achieved over 80% code coverage in most drivers, and is largely limited by the ability of user-mode tests to invoke driver entry points. When we use SymDrive to execute code changed by driver patches, SymDrive achieves over 95% coverage on 12 patches in 3 drivers.

5.1 Design

The SymDrive architecture focuses on thorough testing of drivers to ensure the code does not incorrectly use the kernel/driver interface, crash, or hang. We target test situations where the driver code is available, and use that code to simplify testing with a combination of symbolic execution, static code analysis and transformation, and an extensible test frameworkexecuting in the kernel.

The design of SymDrive is shown in Figure 5.1. The OS kernel and driver under test, as well as user-mode test programs, execute in a virtual machine. The symbolic execution engine provides symbolic devices for the driver. SymDrive provides stubs that invoke checkers on every call into or out of the driver. A test framework tracks execution state and passes information to plugins running in the engine to speed testing and improve test coverage.

Component	LoC
Changes to S^{2}E	1,954
SymGen	2,681
Test framework	3,002
Checkers	564
Support Library	1,579
Linux kernel changes	153
FreeBSD kernel changes	81

Table 5.1: Implementation size of SymDrive.

During the development of SymDrive, we considered a more limited design in which symbolic execution was limited to driver code. In this model, exploring multiple paths through the kernel was not possible; callbacks to the kernel instead required a model of kernel behavior to allow them to execute on multiple branches. After implementing a prototype of this design, we concluded that full-system symbolic execution is preferable because it greatly reduces the effort to test drivers by using real kernel code rather than a kernel model.

We implemented SymDrive for Linux and FreeBSD, as these kernels provide a large number of drivers to test. Only the test framework code running in the kernel is specialized to the OS. We made small, conditionally compiled changes to both kernels to print failures and stack traces to the S^{2}E log and to register the module under test with *see* .he breakdown of SymDrive code is shown in Table 5.1.

SymDrive consists of five components: (i) a modified version of the S^{2}E symbolic-execution engine, which consists of a SymDrive-specific plugin plus changes to S^{2}E; (ii) symbolic devices to provide symbolic hardware input to the driver; (iii) a *test framework* executing within the kernel that guides symbolic execution; (iv) the *SymGen* static-analysis and code transformation tool to analyze and prepare drivers for testing; and (v) a set of OS-specific *checkers* that interpose on the driver/kernel interface for verifying and validating driver behavior.

5.1.1 Virtual Machine

SymDrive uses S^{2}E [33] version 1.1-10.09.2011, itself based on QEMU [15] and KLEE [24], for symbolic execution. S^{2}E provides the execution environment, path forking, and constraint solving capability necessary for symbolic execution. All driver and kernel code, including the test framework, executes within an

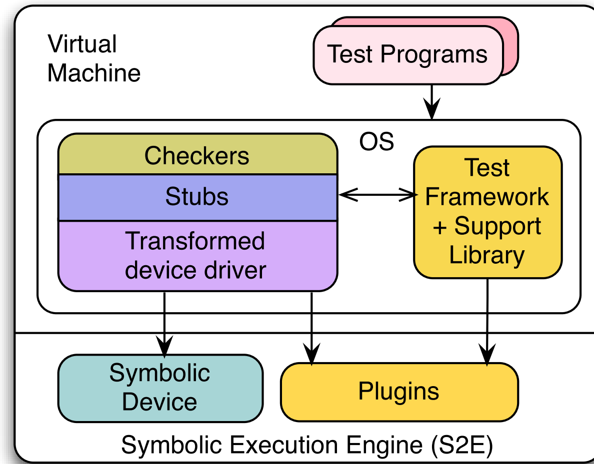

Figure 5.1: The SymDrive architecture. A developer produces the transformed driver with SymGen and can write checkers and test programs to verify correctness.

S^2E VM. Changes to *see* all into two categories: (i) improved support for symbolic hardware, and (ii) the SymDrive path-selection mechanism, which is an S^2E plugin.

SymDrive uses invalid x86 opcodes for communication with the VMM and S^2E plugins to provide additional control over the executing code. We augment S^2E with new opcodes for the test framework that pass information into our extensions. These opcodes are inserted into driver code by SymGen and also invoked directly by the test framework.

The purpose of the opcodes is to communicate source-level information to the SymDrive plugins, which uses the information to guide the driver's execution. The opcodes (i) control whether memory regions are symbolic, as when mapping data for DMA; (ii) influence path scheduling by adjusting priority, search strategy, or killing other paths; and (iii) support tracing by turning it on/off and providing stack information.

5.1.2 Symbolic Devices

Drivers interact with devices according to well-defined, narrow interfaces. For PCI device drivers, this interface is comprised of I/O memory accessed via normal loads and stores, port I/O instructions, bus operations, DMA memory, and interrupts. Drivers using other buses, such as SPI and I^2C, use functions provided by the bus for similar operations.

SymDrive provides a symbolic device for the driver under test, while at the same time emulating the other devices in the system. A symbolic device provides three key behaviors. First, it can be *discovered*, so the kernel loads the appropriate driver. Second, it provides methods to read from and write to the device and return symbolic data from reads. Third, it supports interrupts and DMA, if needed. SymDrive currently supports 5 buses on Linux: PCI (and its variants), I^2C (including SMBus), Serial Peripheral Interface (SPI), General Purpose I/O (GPIO), and Platform;[1] and the PCI bus on FreeBSD.

Device Discovery

When possible, SymDrive creates symbolic devices in the S^2E virtual machine and lets existing bus code discover the new device and load the appropriate driver. For some buses, such as I^2C, the kernel or another driver normally creates a statically configured device object during initialization. For such devices, we created a small kernel module, consisting of 715 lines of code, that creates the desired symbolic device.

Although SymDrive does currently not support USB, we believe that it could, but would require a more effort to implement a symbolic bus. USB device drivers interface with other drivers lower in a stack. The host controller interface driver communicates with the hardware, and modern operating systems typically only need one such driver for each version of USB. Supplying symbolic data at this level of abstraction is not practical, because it would need to pass through the host controller driver, and likely generate many uninteresting paths.

To address this issue, we would need to write a symbolic bus similar to the one used by SymDrive's I^2C support. I^2C provides a similar, high-level communication interface with the device, implemented in Linux's i2c-core and

---

[1] The "platform bus" is a Linux-specific term that encompasses many embedded devices. Linux's ARM implementation, for example, supports a variety of SoCs, each with its own set of integrated devices. The drivers for these devices are often of the "platform" type.

one of the various chip-specific drivers, e.g. PIIX4. We have not done a similar implementation for USB because of time constraints.

SymDrive can make the device configuration space symbolic after loading the driver by returning symbolic data from PCI bus functions with the test framework(similar to S^2E's NDIS driver support). PCI devices use this region of I/O memory for plug-and-play information, such as the vendor and device identifiers. If this data is symbolic, the device ID will be symbolic and cause the driver to execute different paths for each of its supported devices. Other buses have similar configuration data, such as “platform data” on the SPI bus. A developer can copy this data from the kernel source and provide it when creating the device object, or make it symbolic for additional code coverage.

Symbolic I/O

Most Linux and FreeBSD drivers do a mix of programmed I/O and DMA. SymDrive supports two forms of programmed I/O. For drivers that perform I/O through hardware instructions, such as `inb`, or through memory-mapped I/O, SymDrive directs S^2E to ignore write operations, because they do not return values that influence driver execution, and to return symbolic data from reads. The test framework overrides bus I/O functions, such as those used in I^2C drivers, to function analogously.

Symbolic Interrupts

After a driver registers an interrupt handler, the test framework invokes the interrupt handler on every transition from the driver into the kernel. This model represents a trade-off between realism and simplicity: it ensures the interrupt handler is called often enough to keep the driver executing successfully, but may generate spurious interrupts when the driver does not expect them.

Symbolic DMA

When a driver invokes a DMA mapping function, such as `dma_alloc_coherent`, the test framework uses a new S^2E opcode to make the memory act like a memory-mapped I/O region: each read returns a new symbolic value and writes have no effect. Discarding writes to DMA memory reflects the ability of the device to write the data via DMA at any time. The driver should not assume that data written here will be available subsequently. When the driver unmaps

the memory, the test framework directs S^{2}E to revert the region to normal symbolic data, so writes are seen by subsequent reads.

5.1.3 Test Framework

The test framework is a kernel module executing with the virtual machine that assists symbolic execution and executes checkers. SymDrive relies on the test framework to guide and monitor symbolic execution in three ways. First, the test framework implements policy regarding which paths to prioritize or deprioritize. Second, the test framework may inject additional symbolic data to increase code coverage. As mentioned above, it implements symbolic I/O interfaces for some device classes. Finally, it provides the VMM with a stack trace for *execution tracing*, which produces a trace of the driver's I/O operations.

The test framework supports several load-time parameters for controlling its behavior. When loading the test framework with `insmod` or FreeBSD's `kldload`, developers can direct the test framework to enable high-coverage mode (described in Section 5.1.3.1), tracing, or a specific symbolic device. To configure the device, developers pass the device's I/O capabilities and name as parameters. Thus, developers can script the creation of symbolic devices to automate testing.

SymDrive has to address two conflicting goals in testing drivers: (i) executing as far as possible along a path to complete initialization and expose the rest of the driver's functionality; and (ii) executing as much code as possible within each function for thoroughness.

5.1.3.1 Reaching Deeply

A key challenge in fully testing drivers is symbolically executing branch-heavy code, such as loops and initialization code that probes hardware. SymDrive relies on two techniques to limit path explosion in these cases: *favor-success scheduling* and *loop elision.* These techniques allow developers to execute further into a driver, and test functionality that is only available after initialization.

Favor-Success Scheduling

Executing past driver initialization is difficult because the code often has many conditionals to support multiple chips and configurations. Initializing a sound driver, for example, may execute more than 1,000 branches on hardware-specific details. Each branch creates additional paths to explore.

SymDrive mitigates this problem with a *favor-success* path-selection algorithm that prioritizes successfully executing paths, making it a form of best-first search. Notifications from the test framework increase the priority of the current path at every successful function return, both within the driver and at the driver/kernel interface. Higher priority causes the current path to be explored further before switching to another path. This strategy works best for small functions, where a successful path through the function is short.

At every function exit, the test framework notifies S^2E of whether the function completed successfully, which enables the VMM to prioritize successful paths to facilitate deeper exploration of code. The test framework determines success based on the function's return value. For functions returning integers, the test framework detects success when the function returns a value other than an `errno`, which are standard Linux and FreeBSD error values. On success, the test framework will notify the VMM to prioritize the current path. If the developer wishes to prioritize paths using another heuristic, he/she can add an annotation prioritizing any piece of code. We use this approach in some network drivers to select paths where the carrier is on, which enables execution of the driver's packet transmission code.

In order to focus symbolic execution on the driver, the test framework prunes paths when control returns to the kernel successfully. It kills all other paths still executing in the driver and uses an opcode to concretize all data in the virtual machine, so the kernel executes on real values and will not fork new paths. This ensures that a single path runs in the kernel and allows developers to interact with the system and run user-mode tests.

Loop Elision

Loops are challenging for symbolic execution because each iteration may fork new paths. S^2E provides an "EdgeKiller" plugin that a developer may use to terminate complex loops early, but requires developers to identify each loop's offset in the driver binary [33] and hence incur substantial developer effort.

SymDrive addresses loops explicitly by prioritizing paths that exit the loop quickly. Suppose an execution path A enters a loop, executes it once, and during this iteration more paths are created. If path A does not exit the loop after one iteration, SymDrive executes it for a second iteration and, unless it breaks out early, deprioritizes the second iteration because it appears stuck in the loop. SymDrive then selects some other path B that path A forked, and

executes it. SymDrive repeats this process until it finds a path that exits the loop. If no paths exit the loop promptly, SymDrive selects some path arbitrarily and prioritizes it on each subsequent iteration, in the hope that it will exit the loop eventually. If this path still does not exit the loop within 20 iterations, SymDrive prints a warning about excessive path forking as there is no evident way to execute the loop efficiently without manual annotation.

This approach executes hardware polling loops efficiently and automatically, and warns developers when loops cause performance problems. However, this approach may fail if a loop is present in uninstrumented kernel code. It can also result in worse coverage of code that executes only if a polling loop times out. Moreover, loops that produce a value, such as a checksum calculation, cannot exit early without stopping the driver's progress. However, we have not found these problems to be significant.

SymDrive's approach extends the EdgeKiller plugin in two directions. First, it allows developers to annotate driver source rather than having to parse compiled code. Second, source annotations persist across driver revisions, whereas the binary offsets used in the EdgeKiller plugin need updating each time the driver changes.

Annotating code manually to improve its execution performance does reduce SymDrive's ability to find bugs in that code. Wherever annotations were needed in the drivers we examined, we strove to write them in such a way as to execute the problematic loop at least once before terminating early. For example, after a checksum loop, we would add a line to return a symbolic checksum value, which could then be compared against a correct one.

5.1.3.2 Increasing Coverage

SymDrive provides a *high-coverage* mode for testing specific functions, for example those modified by a patch. This mode changes the path-prioritization policy and the behavior of kernel functions. When the developer loads the test framework module, he/she can specify any driver function to execute in this mode.

When execution enters the specified function, the test framework notifies S^2E to favor unexecuted code (the default *see* olicy) rather than favoring successful paths. The test framework terminates all paths that return to the kernel in order to focus execution within the driver. In addition, when the driver invokes a kernel function, the test framework makes the return value

symbolic. This mode is similar to the local consistency mode in S^{2}E [33], but requires no developer-provided annotations or plugins, and supports all kernel functions that return standard error values. For example, `kmalloc` returns a symbolic value constrained to be either NULL or a valid address, which tests error handling in the driver.

For the small number of kernel functions that return non-standard values, SymGen has a list of exceptions and how to treat their return values. The full list of exceptions for Linux currently contains 100 functions across all supported drivers. Of these, 64 are hardware-access functions, such as `inb` and `readl`, that always return symbolic data. A further 14 are arithmetic operations, such as `div32`. The remaining 22 functions return negative numbers in successful cases, or are used by the compiler to trigger a compilation failure when used incorrectly, such as `__bad_percpu_size`.

SymDrive also improves code coverage by introducing additional symbolic data in order to execute code that requires specific inputs from the kernel or applications. SymDrive can automatically make a Linux driver's module parameters symbolic, executes the driver with all possible parameters. Checkers can also make parameters to the driver symbolic, such as `ioctl` command values. This allows all `ioctl` code to be tested with a single invocation of the driver, because each comparison of the command will fork execution. In addition, S^{2}E allows using symbolic data anywhere in the virtual machine, so a user-mode test can pass symbolic data to the driver.

5.1.3.3 Execution Tracing

The test framework can generate execution traces, which are helpful to compare the execution of two versions of a driver. For example, when a driver patch introduces new bugs, the traces can be used to compare its behavior against previous versions. In addition, developers can use other implementations of the driver, even from another operating system, to find discrepancies that may signify incorrect interaction with the hardware.

A developer can enable tracing via a command-line tool that uses a custom opcode to notify SymDrive to begin recording. In this mode, an S^{2}E plugin records every driver I/O operation, including reads and writes to port, MMIO, and DMA memory, and the driver stack at the operation. The test framework passes the current stack to S^{2}E on every function call.

The traces are stored as a trie (prefix tree) to represent multiple paths

through the code compactly, and can be compared using the `diff` utility. SymDrive annotates each trace entry with the driver call stack at the I/O operation. This facilitates analysis of specific functions and comparing drivers function-by-function, which is useful since traces are subject to timing variations and different thread interleavings.

5.1.4 SymGen

All features of the test framework that interact with code, such as favor-success scheduling, loop prioritization, and making kernel return values symbolic are handled automatically via static analysis and code generation. The SymGen tool analyzes driver code to identify code relevant to testing, such as function boundaries and loops, and instruments code with calls to the test framework and checkers. SymGen is built using CIL [103].

Stubs

SymDrive interposes on all calls into and out of the driver with stubs that call the test framework and checkers. For each function in the driver, SymGen generates two stubs: a preamble, invoked at the top of the function, and a postscript, invoked at the end. The generated code passes the function's parameters and return value to these stubs to be used by checkers. For each kernel function the driver imports, SymGen generates a stub function with the same signature that wraps the function.

To support pre- and post-condition assertions, stubs invoke checkers when the kernel calls into the driver or the driver calls into the kernel. Checkers associated with a specific function `function_x` are named `function_x_check`. On the first execution of a stub, the test framework looks for a corresponding checker in the kernel symbol table. If such a function exists, the stub records its address for future invocations. While targeted at functions in the kernel interface, this mechanism can invoke checkers for any driver function.

Stubs employ a second lookup to find checkers associated with a function pointer passed from the driver to the kernel, such as a PCI probe function. Kernel stubs, when passed a function pointer, record the function pointer and its purpose in a table. For example, the Linux `pci_register_driver` function associates the address of each function in the `pci_driver` parameter with the name of the structure and the field containing the function. The stub for the

probe method of a pci_driver structure is thus named pci_driver_probe_check. FreeBSD drivers use a similar technique.

Stubs detect that execution enters the driver by tracking the depth of the call stack. The first function in the driver notifies the test framework at its entry that driver execution is starting, and at its exit notifies the test framework that control is returning to the kernel. Stubs also communicate this information to the VMM so that it can make path-scheduling decisions based on function return values.

Instrumentation

The underlying principle behind SymGen's instrumentation is to inform the VMM of source level information as it executes the driver so it can make better decisions about which paths to execute. SymGen instruments the start and end of each driver function with a call into the stubs. As part of the rewriting, it converts functions to have a single exit point. It generates the same instrumentation for inline functions, which are commonly used in the Linux and FreeBSD kernel/driver interfaces.

SymGen also instruments the start, end, and body of each loop with calls to short functions that execute SymDrive-specific opcodes. These opcodes direct the VMM to prioritize and deprioritize paths depending on whether they exit the loop quickly. This instrumentation replaces most of the per-driver effort required by S^2E to identify loops, as well as the per-class effort of writing a consistency model for every function in the driver/kernel interface. SymGen also inserts loop opcodes into the driver, as shown in Figure 5.2, to tell S^2E which paths exit the loop, and should receive a priority boost.[2]

For complex code that slows testing, SymGen supports programmer-supplied annotations to simplify or disable the code temporarily. Short loops and those that do not generate states require no manual developer effort. Only loops that must execute for many iterations and generate new paths on each iteration need manual annotation, which we implement through C #ifdef statements. For example, the E1000 network driver verifies a checksum over EEPROM, and we modified it to accept any checksum value. We have found these cases to be rare.

[2]One interesting alternative is to prioritize paths that execute loops in their entirety. The problem with this approach is that it may generate many states in the process, and slow testing.

```
s2e_loop_before(__LINE__, loop_id);
while(work--) {
  tmp___17 = readb(cp->regs + 55);
  if(!(tmp___17 & 16)) goto return_label;
  stub_schedule_timeout_uninterruptible(10L);
  s2e_loop_body(__LINE__, loop_id);
}
s2e_loop_after(__LINE__, loop_id);
```

Figure 5.2: SymGen instruments the start, end, and body of loops automatically. This code, from the `8139cp` driver, was modified slightly since SymGen produces preprocessed output.

5.1.5 Limitations

SymDrive is neither sound nor complete. The false positives we have experienced fall into two major categories. First, symbolic execution is slow, which may cause the kernel to print timing warnings and cause driver timers to fire at the wrong time. Second, our initial checkers were imprecise and disallowed (bizarre) behavior the kernel considers legal. We have since fixed the checkers, and have not seen them generate false positives.

Although we have observed no false negatives among the checkers we wrote, SymDrive cannot check for all kinds of bugs. Of 11 common security vulnerabilities [30], SymDrive cannot detect integer overflows and data races between threads, though support for overflow detection is possible in principle because the underlying VMM interprets code rather than executing it directly. In addition, SymDrive cannot achieve full path coverage for all drivers because SymDrive's aggressive path pruning may terminate paths that lead to bugs. SymDrive may also miss race conditions, such as those requiring the interrupt handler to interleave with another thread in a specific way.

5.2 Checkers

SymDrive detects driver/kernel interface violations with checkers, which are functions interposing on control transfer between the driver and kernel that verify and validate driver behavior. Each function in the driver/kernel interface can, but need not, have its own checker. Drivers invoke the checkers from stubs, described above, which call separate checkers at every function in the

driver/kernel interface. Since checkers run in the VM alongside the symbolically executing driver, they can verify runtime properties along each tested path.

The checkers use a *support library* that simplifies their development by providing much of their functionality. The library provides state variables to track the state of the driver and current thread, such as whether it has registered itself successfully and whether it can be rescheduled. The library also provides an object tracker to record kernel objects currently in use in the driver. This object tracker provides an easy mechanism to track whether locks have been initialized and to discover memory leaks. Finally, the library provides generic checkers for common classes of kernel objects, such as locks and allocators. The generic checkers encode the semantics of these objects, and thus do much of the work. For example, checkers for a mutex lock and a spin lock use the same generic checker, as they share semantics.

Writing a checker requires implementing checks within a call-out function. We have implemented 49 checkers comprising 564 lines of code for a variety of common device-driver bugs using the library API. Test #1 in Figure 5.3 shows an example call-out for `pci_register_driver`. The driver-function stub invokes the checker function with the parameters and return value of the kernel function and sets a `precondition` flag to indicate whether the checker was called before or after the function. In addition, the library provides the global `state` variable that a checker can use to record information about the driver's activity. As shown in this example, a checker can verify that the state is correct as a precondition, and update the state based on the result of the call. Checkers have access to the runtime state of the driver and can store arbitrary data, so they can find interprocedural, pointer-specific bugs that span multiple driver invocations.

Not every behavior requirement needs a checker. Symbolic execution leverages the extensive checks already included as kernel debug options, such as for memory corruption and locking. Most of these checks execute within functions called *from* the driver, and thus will be invoked on multiple paths. In addition, any bug that causes a kernel crash or panic will be detected by the operating system and therefore requires no checker.

We next describe a few of the 49 checkers we have implemented with SymDrive.

```
/* Test #1 */ void __pci_register_driver_check(...) {
  if (precondition) {
    assert (state.registered == NOT_CALLED);
    set_state (&state.registered, IN_PROGRESS);
    set_driver_bus (DRIVER_PCI);
  } else /* postcondition */ {
    if (retval == 0) set_state (&state.registered, OK);
    else set_state (&state.registered, FAILED);
  }
}

/* Test #2 */ void __kmalloc_check
  (..., void *retval, size_t size, gfp_t flags) {
  if (precondition)
    mem_flags_test(GFP_ATOMIC, GFP_KERNEL, flags);
  else /* postcondition */
    generic_allocator(retval, size, ORIGIN_KMALLOC);
}

/* Test #3 */ void _spin_lock_irqsave_check
  (..., void *lock) {
  // generic_lock_state supports pre/post-conditions
  generic_lock_state(lock,
    ORIGIN_SPIN_LOCK, SPIN_LOCK_IRQSAVE, 1);
}
```

Figure 5.3: Example checkers. The first checker ensures that PCI drivers are registered exactly once. The second verifies that a driver allocates memory with the appropriate `mem_flags` parameter. The third ensures lock/unlock functions are properly matched. Unlike Static Driver Verifier checkers [101], these checkers can track any path-specific run-time state expressible in C.

Execution Context

Linux prohibits the calling of functions that block when executing in an interrupt handler or while holding a spinlock. The execution-context checker verifies that flags passed to memory-allocation functions such as `kmalloc` are valid in the context of the currently executing code. The support library provides a state machine to track the driver's current context using a stack. When entering the driver, the library updates the context based on the entry point. The library also supports locks and interrupt management. When the driver acquires or releases a spinlock, for example, the library pushes or pops the necessary context.

Kernel API Misuse

The kernel requires that drivers follow the proper protocol for kernel APIs, and errors can lead to a non-functioning driver or a resource leak. The support library state variables provide context for these tests. For example, a checker can track the success and failure of significant driver entry points, such as the `init_module` and PCI `probe` functions, and ensure that if the driver is registered on initialization, it is properly unregistered on shutdown. Test #1 in Figure 5.3 shows a use of these states to ensure that a driver only invokes `pci_register_driver` once.

Collateral Evolutions

Collateral evolutions occur when a small change to a kernel interface necessitates changes in many drivers simultaneously. A developer can use SymDrive to verify that collateral evolutions [107] are correctly applied by ensuring that patched drivers do not regress on any tests.

SymDrive can also ensure that the desired *effect* of a patch is reflected in the driver's execution. For example, recent kernels no longer require that network drivers update the `net_device->trans_start` variable in their `start_xmit` functions. We wrote a checker to verify that `trans_start` is constant across `start_xmit` calls.

Memory Leaks

The leak checker uses the support library's object tracker to store an allocation's address and length. We implemented checkers to verify allocation and free requests from 19 pairs of functions, and ensure that an object's allocation and freeing use paired routines.

The API library simplifies writing checkers for additional allocators down to a few lines of code. Test #2 in Figure 5.3 shows the `generic_allocator` call to the library used when checking `kmalloc`, which records that `kmalloc` allocated the returned memory. A corresponding checker for `kfree` verifies that `kmalloc` allocated the supplied address.

5.3 Evaluation

The purpose of the evaluation is to verify that SymDrive achieves its goals: (i) usefulness, (ii) simplicity, and (iii) efficiency.

5.3.1 Methodology

As shown in Table 5.2, we tested SymDrive on 26 drivers in 11 classes from several Linux kernel revisions (13 drivers from 2.6.29, 4 from 3.1.1, and 4 that normally run only on Android-based phones) and from FreeBSD 9 (5 drivers). Of the 26 drivers, we chose 19 as examples of a specific bus or class and the remaining 7 because we found frequent patches to them and thus expected to find bugs.

All tests took place on a machine running Ubuntu 10.10 x64 equipped with a quad-core Intel 2.50GHz Intel Q9300 CPU and 8GB of memory. All results are obtained while running SymDrive in a single-threaded mode, as SymDrive does not presently work with S^{2}E's multicore support.[3]

To test each driver, we carry out the following operations:

1. Run SymGen over the driver and compile the output.
2. Define a virtual hardware device with the desired parameters and boot the SymDrive virtual machine.
3. Load the driver with `insmod` and wait for initialization to complete successfully. Completing this step entails executing at least one successful path and returning success, though it is likely that other failed paths also run and are subsequently discarded.
4. Execute a workload (optional). We ensure all network drivers attempt to transmit and that sound drivers attempt to play a sound.
5. Unload the driver.

If SymDrive reports warnings about too many paths from complex loops, we annotate the driver code and repeat the operations. For most drivers, we run SymGen over only the driver code. For drivers that have fine-grained interactions with a library, such as sound drivers and the `pluto2` media driver, we run SymGen over both the library and the driver code. We annotated each driver at locations SymDrive specified, and tested each Linux driver with 49 checkers for a variety of common bugs. For FreeBSD drivers, we only used the operating system's built-in test functionality.

[3]This limitation is an engineering issue and prevents SymDrive from exploring multiple paths simultaneously. However, because SymDrive's favor-success scheduling often explores a single path deeply rather than many paths at once, S^{2}E's multi-threaded mode would have little performance benefit.

Driver	Class	Bugs	LoC	Ann	Load	Unld.
*akm8975**	Compass	4	629	0	0:22	0:08
*mmc31xx**	Compass	3	398	0	0:10	0:04
tle62x0*	Control	2	260	0	0:06	0:05
me4000	Data Ac.	1	5,394	2	1:17	1:04
phantom	Haptic	0	436	0	0:16	0:13
lp5523*	LED Ctl.	2	828	0	2:26	0:19
apds9802*	Light	0	256	1	0:31	0:21
8139cp	Net	0	1,610	1	1:51	0:37
8139too	Net	2	1,904	3	3:28	0:35
be2net	Net	7	3,352	2	4:49	1:39
dl2k	Net	1	1,985	5	2:52	0:35
e1000	Net	3	13,971	2	4:29	2:01
et131x	Net	2	8,122	7	6:14	0:47
forcedeth	Net	1	5,064	2	4:28	0:51
ks8851*	Net	3	1,229	0	2:05	0:13
pcnet32	Net	1	2,342	1	2:34	0:27
*smc91x**	Net	0	2,256	0	10:41	0:22
pluto2	Media	2	591	3	1:45	1:01
econet	Proto.	2	818	0	0:11	0:11
ens1371	Sound	0	2,112	5	27:07	4:48
*a1026**	Voice	1	1,116	1	0:34	0:03
ed	Net	0	5,014	0	0:49	0:13
re	Net	0	3,440	3	16:11	0:21
rl	Net	0	2,152	1	2:00	0:08
es137x	Sound	1	1,688	2	57:30	0:09
maestro	Sound	1	1,789	2	17:51	0:27

Table 5.2: Drivers tested. Those in *italics* run on Android-based phones, those followed by an asterisk are for embedded systems and do not use the PCI bus. Drivers above the line are for Linux and below the line are for FreeBSD. Line counts come from CLOC [1]. Times are in minute:second format, and are an average of three runs.

Bug Type	Bugs	Kernel / Checker	Cross EntPt	Paths	Ptrs
Hardware Dep.	7	6 / 1	4	6	6
API Misuse	15	7 / 8	6	5	1
Race	3	3 / 0	3	2	3
Alloc. Mismatch	3	0 / 3	3	0	3
Leak	7	0 / 7	6	1	7
Driver Interface	3	0 / 3	0	2	0
Bad pointer	1	1 / 0	0	0	1
Totals	39	17 / 22	22	16	21

Table 5.3: Summary of bugs found. For each category, we present the number of bugs found by kernel crash/warning or a checker and the number that crossed driver entry points (“Cross EntPt”), occurred only on specific paths, or required tracking pointer usage.

5.3.2 Bug Finding

Across the 26 drivers listed in Table 5.2, we found the 39 distinct bugs described in Table 5.3. Of these bugs, S^2E detected 17 via a kernel warning or crash, and the checkers caught the remaining 22. Although these bugs do not necessarily result in driver crashes, they all represent issues that need addressing and are difficult to find without visibility into driver/kernel interactions.

These results show the value of symbolic execution. Of the 39 bugs, 56% spanned multiple driver invocations. For example, the `akm8975` compass driver calls `request_irq` before it is ready to service interrupts. If an interrupt occurs immediately after this call, the driver will crash, since the interrupt handler dereferences a pointer that is not yet initialized. In addition, 41% of the bugs occurred on a unique path through a driver other than one that returns success, and 54% involved pointers and pointer properties that may be difficult to detect statically.

Bug Validation

Of the 39 bugs found, at least 17 were fixed between the 2.6.29 and 3.1.1 kernels, which indicates they were significant enough to be addressed. We were unable to establish the current status of 7 others because of significant driver changes. We have submitted bug reports for 5 unfixed bugs in mainline Linux drivers, all of which have been confirmed as genuine by kernel developers. The remaining bugs are from drivers outside the main Linux kernel that we have not yet reported.

5.3.3 Developer Effort

One of the goals of SymDrive is to minimize the effort to test a driver. The effort of testing comes from three sources: (i) annotations to prepare the driver for testing, (ii) testing time, and (iii) updating code as kernel interfaces change.

To measure the effort of applying SymDrive to a new driver, we tested the `phantom` haptic driver from scratch. The total time to complete testing was 1h:45m, despite having no prior experience with the driver and not having the hardware. In this time, we configured the symbolic hardware, wrote a user-mode test program that passes symbolic data to the driver's entry points, and executed the driver four times in different configurations. Of this time, the overhead of SymDrive compared to testing with a real device was an additional

pass during compilation to run SymGen, which takes less than a minute, and 38 minutes to execute. Although not a large driver, this test demonstrates SymDrive's usability from the developer's perspective.

Annotations

The only per-driver coding SymDrive requires is annotations on loops that slow testing and annotations that prioritize specific paths. Table 5.2 lists the number of annotation sites for each driver. Of the 26 drivers, only 6 required more than two annotations, and 9 required no annotations. In all cases, SymDrive printed a warning indicating where an annotation would benefit testing.

Testing Time

Symbolic execution can be much slower than normal execution. Hence, we expect it to be used near the end of development, before submitting a patch, or on periodic scans through driver code. We report the time to load, initialize, and unload a driver (needed for detecting resource leaks) in Table 5.2. Initialization time is the minimum time for testing, and thus presents a lower bound.

Overall, the time to initialize a driver is roughly proportional to the size of the driver. Most drivers initialize in 5 minutes or less, although the `ens1371` sound driver required 27 minutes, and the corresponding FreeBSD `es137x` driver required 58 minutes. These two results stem from the large amount of device interaction these drivers perform during initialization. Excluding these results, execution is fast enough to be performed for every patch, and with a cluster could be performed on every driver affected by a collateral evolution [107].

Kernel Evolution

Near the end of development, we upgraded SymDrive from Linux 2.6.29 to Linux 3.1.1. If much of the code in SymDrive was specific to the kernel interface, porting SymDrive would be a large effort. However, SymDrive's use of static analysis and code generation minimized the effort to maintain tests as the kernel evolves: the only changes needed were to update a few checkers whose corresponding kernel functions had changed. The remainder of the system, including SymGen and the test framework, were unchanged. The number of lines of code changed was less than 100.

Driver	Touched Funcs.	Coverage	CPU	Time Latency
8139too	93%	83%	2h36m	1h00m
a1026	95%	80%	15m	13m
apds9802	85%	90%	14m	7m
econet	51%	61%	42m	26m
ens1371	74%	60%	*8h23m	*2h16m
lp5523	95%	83%	21m	5m
me4000	82%	68%	*26h57m	*10h25m
mmc31xx	100%	83%	14m	26m
phantom	86%	84%	38m	32m
pluto2	78%	90%	19m	6m
tle62x0	100%	85%	16m	12m
es137x	97%	70%	1h22m	58m
rl	84%	71%	13m	10m

Table 5.4: Code coverage.

Furthermore, porting SymDrive to a new operating system is not difficult. We also ported the SymDrive infrastructure, checkers excluded, to FreeBSD 9. The entire process took three person-weeks. The FreeBSD implementation largely shares the same code base as the Linux version, with just a few OS-specific sections. This result confirms that the techniques SymDrive uses are compatible across operating systems.

5.3.4 Coverage

While SymDrive primarily uses symbolic execution to simulate the device, a second benefit is higher code coverage than standard testing. Table 5.4 shows coverage results for one driver of each class, and gives the fraction of functions executed ("Touched Funcs.") and the fraction of basic blocks *within* those functions ("Coverage").[4] In addition, the table gives the total CPU time to run the tests on a single machine (CPU) and the latency of the longest run if multiple machines are used (Latency). In all cases, we ran drivers multiple times and merged the coverage results. We terminated each run once it reached a steady state and stopped testing the driver once coverage did not meaningfully improve between runs.

Overall, SymDrive executed a large majority (80%) of driver functions in most drivers, and had high coverage (80% of basic blocks) in those functions. These results are below 100% for two reasons. First, we could not invoke all

[4] * Drivers with an asterisk ran unattended, and their total execution time is not representative of the minimum.

entry points in some drivers. For example, `econet` requires user-mode software to trigger additional driver entry points that SymDrive is unable to call on its own. In other cases, we simply did not spend enough time understanding how to invoke all of a driver's code, as some functionality requires the driver to be in a specific state that is difficult to realize, even with symbolic execution. Second, of the functions SymDrive did execute, additional inputs or symbolic data from the kernel were needed to test all paths. Following S^2E's relaxed consistency model by making more of the kernel API symbolic could help improve coverage.

As a comparison, we tested the `8139too` driver on a real network card using `gcov` to measure coverage with the same set of tests. We loaded and unloaded the driver, and ensured that transmit, receive, and all ethtool functions executed. Overall, these tests executed 77% of driver functions, and covered 75% of the lines in the functions that were touched, as compared to 93% of functions and 83% of code for SymDrive. Although not directly comparable to the other coverage results due to differing methodologies, this result shows that SymDrive can provide coverage better than running the driver on real hardware.

5.3.5 Patch Testing

The second major use of SymDrive is to verify driver patches similar to a code reviewer. For this use, we seek high coverage in every function modified by the patch in addition to the testing described previously. We evaluate SymDrive's support for patch testing by applying all the patches between the 3.1.1 and 3.4-rc6 kernel releases that applied to the `8139too` (net), `ks8851` (net) and `lp5523` (LED controller) drivers, of which there were 4, 2, and 6, respectively. The other drivers lacked recent patches, had only trivial patches, or required upgrading the kernel, so we did not consider them.

In order to test the functions affected by a patch, we used favor-success scheduling to fast-forward execution to a patched function and then enabled high coverage mode. The results, shown in Table 5.5, demonstrate that SymDrive is able to quickly test patches as they are applied to the kernel, by allowing developers to test nearly all the changed code without any device hardware. SymDrive was able to execute 100% of the functions touched by all 12 patches across the 3 drivers, and an average of 98% of the code in each function touched by the patch. In addition, tests took an average of only 12 minutes to complete.

Driver	Touched Funcs.	Coverage	Serial	Time Parallel
8139too	100%	96%	9m	5m
ks8851	100%	100%	16m	8m
lp5523	100%	97%	12m	12m

Table 5.5: Patched code coverage.

Execution Tracing

Execution tracing provides an alternate means to verify patches by comparing the behavior of a driver before and after applying the patch. We used tracing to verify that SymDrive can distinguish between patches that change the driver/device interactions and those that do not, such as a collateral evolution. We tested five patches to the `8139too` network driver that refactor the code, add a feature, or change the driver's interaction with the hardware. We executed the original and patched drivers and record the hardware interactions. Comparing the traces of the before and after-patch drivers, differing I/O operations clearly identify the patches that added a feature or changed driver/device interactions, including which functions changed. As expected, there were no differences in the refactoring patches.

We also apply tracing to compare the behavior of drivers for the same device across operating systems. Traces of the Linux `8139too` driver and the FreeBSD `rl` driver show differences in how these devices interact with the same hardware that could lead to incorrect behavior. In one case, the Linux `8139too` driver incorrectly treats one register as 4 bytes instead of 1 byte, while in the other, the `rl` FreeBSD driver uses incorrect register offsets for a particular supported chipset. Developers fixed the Linux bug independently after we discovered it, and we validated the FreeBSD bug with a FreeBSD kernel developer. We do not include these bugs in the previous results as they were not identified automatically by SymDrive.

These bugs demonstrate a new capability to find hardware-specific bugs by comparing independent driver implementations. While we manually compared the traces, it may be possible to automate this process.

5.3.6 Comparison to other tools.

We compare SymDrive against other driver testing/bug-finding tools to demonstrate its usefulness, simplicity, and efficiency.

S^2E

In order to demonstrate the value of SymDrive's additions to S^2E, we executed the `8139too` driver with only annotations to the driver source guiding path exploration but without the test framework or SymGen to prioritize relevant paths. In this configuration, S^2E executes using *strict consistency*, wherein the only source of symbolic data is the hardware, and maximizes coverage with the MaxTbSearcher plugin. This mode is the default when a developer does not write API-specific plugins; results improve greatly when these plugins are available [33]. We ran S^2E until it ran out of memory to store paths and started thrashing after 23 minutes.

During this test, only 33% of the functions in the driver were executed, with an average coverage of 69%. In comparison, SymDrive executed 93% of functions with an average coverage of 83% in 2½ hours. With S^2E alone, the driver did not complete initialization and did not attempt to transmit packets. In addition, S^2E's annotations could not be made on the driver source, but must be made on the binary instead. Thus, annotations must be regenerated every time a driver is compiled.

Adding more RAM and running the driver longer would likely have allowed the driver to finish executing the initialization routine. However, many uninteresting paths would remain, as S^2E has no automatic way to prune them. Thus, the developer would still have considerable difficulty invoking other driver entry points, since S^2E would continue to execute failing execution paths.

In order for S^2E to achieve higher coverage in this driver, we would need a plugin to implement a relaxed consistency model. However, the `8139too` driver (v3.1.1) calls 73 distinct kernel functions, which would require developer effort to code corresponding functions in the plugin.

Static-Analysis Tools

Static analysis tools are able to find many driver bugs, but require a large effort to implement a model of operating system behavior. For example, Microsoft's Static Driver Verifier (SDV) requires 39,170 lines of C code to implement an operating system model [101]. SymDrive instead relies on models only for the I/O bus implementations, which together account for 715 lines of code for 5 buses. SymDrive supports FreeBSD with only 491 lines of OS-specific code, primarily for the test framework, and can check drivers with the debugging facilities already included in the OS.

In addition, SDV achieves much of its speed through simplifying its analysis, and consequently its checkers are unable to represent arbitrary state. Thus, it is difficult to check complex properties such as whether a variable has matched allocation/free calls across different entry points.

Kernel Debug Support

Most kernels provide debugging to aid kernel developers, such as tools to detect deadlock, track memory leaks, or uncover memory corruption. Some of the test framework checkers are similar to debug functionality built into Linux. Compared to the Linux leak checker, kmemleak, the test framework allows testing a single driver for leaks, which can be drowned out when looking at a list of leaks across the entire kernel. Furthermore, writing checkers for SymDrive is much simpler: the Linux 3.1.1 kmemleak module is 1,113 lines, while, the test framework object tracker, including a complete hash table implementation, is only 722 lines yet provides more precise results.

5.4 Conclusions

SymDrive uses symbolic execution combined with a test framework and static analysis to test Linux and FreeBSD driver code without access to the corresponding device. Our results show that SymDrive can find bugs in mature driver code of a variety of types, and allow developers to test driver patches deeply. Hopefully, SymDrive will enable more developers to patch driver code by lowering the barriers to testing. In the future, we plan to implement an automated testing service for driver patches that supplements manual code reviews, and investigate applying SymDrive's techniques to other kernel subsystems.

6 SYMDRIVECLUSTER: AUTOMATING SYMDRIVE

SymDrive has proven effective at allowing developers to test individual drivers. By providing developers with feedback on where annotations are needed, SymDrive simplifies the process of getting an individual driver to run. Moreover, the test framework finds a variety of bugs and provides useful correctness guarantees. In its current implementation, however, SymDrive is not scalable.

Automatically testing many drivers with SymDrive is difficult for five major reasons. First, the process of configuring symbolic hardware is repetitive and error-prone. A single mistake means the driver will not initialize. Second, SymGen is incompatible with the existing kernel build process, forcing developers to compile each driver individually. Third, developers must supply annotations to most drivers. Although SymDrive provides developers with instructions on where to supply the annotations, it reports only one missing annotation at a time. Consequently, if a driver requires five annotations, developers must re-execute the same driver five times in order to find all five locations. Fourth, SymDrive runs only one driver at a time, which limits scalability. Finally, SymDrive itself provides no default means to test drivers: it does not invoke driver entry points without developer effort.

To overcome these limitations, we present *SymDriveCluster* (*SDC*), a system designed to test Linux device drivers *automatically*, with no per-driver developer effort. In contrast to SymDrive, which runs only a single driver at a time and requires developer interaction, SDC runs on a *cluster* of machines and invokes driver entry points automatically.

SDC addresses the major problems with executing a single driver automatically. First, SDC uses an enhanced static analysis to establish an appropriate symbolic hardware configuration for each driver. This process, which is included in an enhanced SymGen tool, removes the tedious manual configuration used in SymDrive. Second, SymGen now integrates with the existing kernel build process, and allows the developer to instrument all Linux kernel drivers automatically. Third, SDC no longer requires per-driver annotation. Instead, SymGen instrumentation notifies SDC when a problematic code segment is executing, which allows SDC to relax execution constraints and bypass the problematic code fragment. Although this approach may introduce false positive bug reports, we have found it highly effective at executing drivers.

SDC also introduces the SymControl tool, which allows developers to run

SDC on many machines in a cluster. SymControl combines the fact that drivers are largely independent of each other with SDC's capacity to test individual drivers automatically. The developer first specifies a list of machines on which to run SDC, a list of drivers to test, and a list of test cases to invoke. SymControl then copies the necessary files and executes SDC on all the machines. It may execute each driver repeatedly, each time with different symbolic hardware configurations and with different test cases.

Once SDC tests a set of drivers under different configurations, it aggregates the results, and provides analysis tools to help developers read bug reports. Since SDC may encounter the same bug along multiple execution paths, it uses a basic "bucketing" heuristic to group similar bug reports. It then presents the developer with a bug report representative of each group.

SDC makes two major contributions relative to existing work: (1) it scales symbolic execution as a testing tool to support many Linux device drivers, and (2) it tests drivers with no per-driver code changes or annotations, making it more similar to static bug-finding tools. Preliminary results indicate that it can find 164 bugs in 107 network drivers with no per-driver effort.

We first discuss the difficulties of automatically testing many drivers in section 6.1. We then examine the approach SDC uses to automate the execution of a single driver in section 6.2. Then, in section 6.3, we examine how to automate testing many drivers. Finally, we present some preliminary experimental results in section 6.4, and conclude.

6.1 Motivation

Motivating the need for SDC may be easiest with an example. To test the LP5523 LED controller driver with SymDrive, one must: (a) configure SymDrive to run with a "dummy device" equivalent to a device the driver actually supports, (b) re-compile the driver using SymDrive's compilation process, (c) invoke the virtual machine with the appropriate dummy device, and then copy the necessary files to it, (d) enable symbolic execution and begin loading the driver, (e) observe the driver's behavior (f) annotate the driver appropriately if SymDrive warns the developer that it has stopped making progress and go back to step (b) as needed, and (g) run different workloads on the driver as it executes symbolically to test additional driver entry points.

This process takes considerable time and effort. We decompose the process of testing drivers into five parts, namely: (1) configuring symbolic hardware,

(2) guaranteeing forward execution progress, (3) invoking driver entry points, (4) running SymGen on many drivers, and (5) testing many drivers in parallel. Solving the first three problems allow developers to test individual drivers automatically, while additionally solving the last two allow the testing of many drivers.

6.1.1 Configuring Symbolic Hardware

Setting up a driver to test with SymDrive first requires identifying the bus that the driver uses. In many cases, the developer can find this information relatively easily by looking at the driver source code. Knowing the bus is important because the developer must be able to set up appropriately configured symbolic hardware for that bus.

Second, the developer must identify the hardware that the driver supports. This process is tedious and requires examination of the source code to understand what devices the driver supports. Once the developer finds this information, he or she must encode the information into a format that enables SymDrive to create the necessary symbolic hardware.

This process is surprisingly error-prone, because of the variety of different parameters that the developer must set. For example, configuring a basic PCI device requires defining roughly ten different parameters, such as the device and vendor IDs, and the size and type of I/O that the device supports. A mistake in any one of these parameters means the symbolic device will not match the driver's expectation, and results either in the kernel not loading the driver, or in a false positive bug report as the driver crashes unrealistically. Other buses are similarly complex, and each mistake can take many minutes to find and correct.

Third, in the case of PCI devices, the developer must establish how the driver communicates with the device. PCI drivers use either port or memory-mapped I/O to access the device, and some support both techniques. Moreover, the developer needs to establish the size of the I/O regions. This information is critical, because if the developer configures the symbolic hardware with too small of an I/O region, the driver will likely crash as it tries to communicate with the device.

Spear et al. propose explicit specifications for driver I/O and configuration parameters to improve reliability and avoid resource conflicts [120]. If implemented more widely, this approach would largely eliminate the difficulty of

configuring symbolic hardware.

6.1.2 Guaranteeing Forward Execution Progress

One difficulty with symbolic execution is that it may not be able to execute loops efficiently. In this case, SymDrive becomes "stuck" in the loop: each iteration generates new paths, and SymDrive is unable to break out of it quickly. SymDrive alerts developers to the problematic code fragment, which requires manual annotation. The developer must provide SymDrive with an alternative implementation that does repeatedly generate new execution paths on each iteration, or that terminates the loop more quickly. When multiple annotations are necessary, the developer must re-run the driver to find the location of each. This process is time-consuming, and incorrectly supplying an annotation requires re-testing the driver to identify the problem.

6.1.3 Invoking Driver Entry Points

SymDrive also requires constant developer attention during use, because testing a driver involves running software to invoke the driver's entry points. If the developer does not invoke a driver entry point, the code remains untested. Moreover, a test might stall while executing a particular entry point, because the execution may be unable to proceed without another annotation. Although SymDrive warns the developer if testing a particular entry point requires an annotation, the developer must then stop everything, write the annotation, and then re-execute the driver from the beginning. This process could easily require 30 minutes.

6.1.4 Running SymGen

After setting up the appropriate symbolic hardware, the developer must also recompile the driver using SymGen, the static analysis tool that instruments the driver for use with SymDrive. This process is straightforward, but still requires the developer to copy the driver files to a special directory, add the driver and its files to an existing build script, and then compile it. Undertaking this process for every driver in the Linux kernel would require many hours, and would require constant intervention each time a new driver version became available.

6.1.5 Testing Many Drivers

Finally, SymDrive includes no mechanisms to scale the symbolic execution across multiple machines. Symbolic execution is a highly-parallelizable technology, and newer versions of S^2E enable parallelized symbolic execution on a single machine. For example, new versions allow a single machine to explore multiple execution paths in parallel. Cloud9 [35], another symbolic execution tool based on KLEE, scales symbolic execution across machines, but does not support testing device drivers or kernel code.

6.2 Automatically Testing one Driver

SymDrive itself has a variety of limitations that prevent it from automatically testing a driver. This section examines the techniques SDC uses to address these limitations. By providing SymDrive with the capacity to test a single driver automatically, it is much easier to add the ability to test many drivers automatically. The key challenges are in automatically: (a) identifying the driver's symbolic hardware configuration, (b) executing loops in drivers efficiently without manual annotation, and (c) invoking the driver's entry points.

6.2.1 Configuring Symbolic Hardware

In order to run a driver, it is first necessary to configure the symbolic hardware correctly. This information consists of (a) the bus that the driver uses, (b) all supported hardware details, because some drivers support many distinct devices, and (c) the size and type of any I/O regions used to communicate with the device.

If the symbolic hardware is not set up correctly, the Linux Plug and Play mechanism will not attempt to load the driver. This problem can happen if SDC accidentally tries to test a driver for which it has no underlying symbolic bus support. Similarly, if the size or type of any I/O region is incorrect, the driver either will not initialize or will crash. A crash can occur because the driver could attempt to read a register that is not mapped into memory, which triggers an invalid page fault.

To solve this problem, SDC uses a separate static analysis built into the existing SymGen tool. This static analysis runs on the entire Linux kernel at compile time, or on individual out-of-tree drivers. We next examine how

SymGen identifies the information necessary to configure the symbolic hardware in more detail, starting with the device bus.

Device Bus

Using static analysis, SymGen must first determine the bus that the driver uses, which it does by examining how the driver registers itself with the kernel. For example, PCI drivers use the `pci_register_driver` function, while SPI drivers use the `spi_register_driver` function. Linux treats other buses similarly: drivers for devices on these other buses use a corresponding `register_driver` function. Currently, SDC supports only the PCI bus, so it must be able to classify drivers according to whether they use this bus.

A few drivers, such as the `matrox` video card driver, support multiple buses. This driver uses both the PCI and I^2C buses and calls both `pci_register_driver` as well as `i2c_add_driver`. SDC does not currently support these kinds of drivers.

Device Identifiers

PCI devices include a feature called "Plug and Play." The purpose of this feature, among other things, is to provide the operating system with the capacity to find out what driver to use with the device. These devices provide the operating system with identifiers, such as a vendor ID and a device ID, which the operating system can use to find a corresponding driver. If the device does not supply these identifiers, or supplies incorrect ones, the operating system will not attempt to load the driver.

Fortunately, finding the identifiers necessary to load a given driver is easy to do by looking at the driver's source code. First, SymGen finds a call to `pci_register_driver`. Almost all Linux PCI drivers statically declare an array of `pci_device_id` structures. They then store a pointer to this array in a static `pci_driver` object, which they then pass to the kernel via the `pci_register_driver` function call. The idea is to inform the kernel that the driver supports a certain set of PCI devices. The kernel then invokes the driver if a PCI device with the appropriate identifiers is present in the system.

Once SymGen finds the appropriate structure definition, it extracts all `device`, `vendor`, `subdevice`, `subvendor`, `class`, and `class_mask` fields. The tool then writes this information into a set of scripts, with one script per supported device. SDC uses the scripts to automate the process of starting the underlying virtual machine with appropriately configured symbolic hardware.

I/O Configuration

Perhaps the most complex and error-prone part of this process is the identification of the device I/O configuration. PCI devices include several `base address registers`, or BARs. Each BAR includes an address, which is essentially an integer, as well as the size and type of I/O.

Each BAR in a PCI device may support either port or memory-mapped I/O. Device drivers using port I/O must use special CPU instructions to communicate with the device. In contrast, drivers using memory-mapped I/O use the CPU's existing memory access instructions. The driver first maps the I/O memory region into the kernel's address space, and then can read and write data in that region according to the device specification.

Currently, SymGen does not establish the size of each I/O region automatically. Instead, it assumes that the device has the maximum of six regions, each of large size (8MB). Although devices may exist with larger regions than this, we have not found any. If a driver expects the region to be larger, it will crash when it attempts to read or write a value beyond the end of it.

Deriving the size of this region statically may be complex because the device itself stores this information and reports it to the operating system. The driver may use this reported value when setting up the region for access, and in doing so, can use the region without prior knowledge of its size. By interposing on functions such as `ioremap`, which sets up a region of I/O memory for use, and running the driver on real hardware, it is possible to establish the region size at runtime. However, this approach requires the device, and makes SymDrive inconvenient to use.

SymGen also does not establish the type of each region independently. Instead, it assumes that all the regions are either of one type or the other. The technique SymGen uses is to examine the types of I/O access calls the driver makes. In the Linux kernel, the standard approach for reading a 32-bit value via memory-mapped I/O is the `readl` function. This function essentially dereferences the memory at the specified address. Similarly, reading a 32-bit value via port I/O requires the `inl` function, which uses the necessary CPU instructions. SymGen scans the driver source code looking for calls to these functions, and if the driver has a predominance of one or the other, it assumes the driver interacts with the device using the associated mechanism.

Unfortunately, some devices use more complex interfaces that SymGen cannot establish with precision. For example, the 8139too PCI device uses

two base address registers. One describes a port-I/O region, and the other a memory-mapped I/O region. The device itself treats both these regions as equivalent. In other words, on this device, a write to the first port is equivalent to a write to the first byte of the MMIO space in the sense that both operations act on the same underlying device register. The 8139too driver chooses which region to use at runtime, and if the appropriate region is unavailable or of the wrong type, the driver will not initialize. This approach is problematic for SymGen because of its assumption that the device uses either port or memory-mapped I/O exclusively.

Another source of complexity is that some drivers use the `ioreadX` and `iowriteX` family of functions to access the device. These functions work with both I/O types, and provide no useful information to SymGen about the underlying access mechanism. In these cases, SymGen uses calls to functions such as `ioport_map` and `pci_ioremap_bar` to disambiguate the access type.

6.2.2 Guaranteeing Forward Execution Progress

Once the symbolic hardware environment for the driver is configured, SDC can attempt to load the driver. Driver initialization code is branch-heavy. It often contains a variety of conditional statements related both to data read from the device and to kernel return values. If the driver returns an unexpected value, for example, the driver will likely return an error and initialization will fail. This problem is significant because if the driver fails initialization, testing other entry points is not possible.

Initializing a driver symbolically requires addressing two challenges that would otherwise prevent initialization. First, paths that initialize the driver successfully are of the most value, because without successfully loading the driver, testing its other entry points is impossible. Therefore, we need a way to distinguish successful paths from failing ones. Second, loops that generate additional execution paths can easily halt forward progress if SDC is unable to bypass them.

The previous version of SymDrive included techniques to address these problems. First, SymDrive used favor-success scheduling to prioritize successfully-executing paths. It uses driver function return values to establish whether a given path is executing successfully, and then deprioritizes those paths that return failure codes. Second, SymDrive warned developers when loops began to generate additional execution states, so they could supply annotations.

```
unsigned short eeprom_data, i;
for (i = 0; i < (EEPROM_CHECKSUM_REG + 1); i++) {
    e1000_read_eeprom(..., &eeprom_data);
    checksum += eeprom_data;
}
if (checksum == (u16)EEPROM_SUM)
    return E1000_SUCCESS;
else return -E1000_ERR_EEPROM;
```

Figure 6.1: This figure is a copy of Figure 4.3. SDP uses the device trace to execute these loops quickly, whereas SymDrive requires developers to annotate these loops manually and SDC breaks from them automatically.

Although favor-success scheduling appears to be a scalable solution to driver initialization, annotating driver loops is not. To address these challenges, SDC combines favor-success scheduling with a new technique to bypass loops and other problematic code automatically.

Loop Elision

One major problem with SymDrive was the need to annotate some loops, such as checksum loops, in order to permit forward progress. Figure 6.1 shows an example loop that reads a set of values from the hardware and calculates a checksum. Executing checksum loops is effectively impossible symbolically because it is similar to finding a set of data that hashes to a given value. In general, the problematic loops are those that generate additional paths on each iteration, and provide no usable mechanism for early termination, such as a `break` statement.

Instead of requiring annotations for these loops, SDC breaks out of them after one iteration. Afterward, it stores unconstrained symbolic data in any variables written to during that iteration. In the example shown in Figure 6.1, SDC breaks from the loop after one iteration and replaces the contents of the `checksum` variable with unconstrained symbolic data. After the loop, there is a conditional statement that compares the unconstrained checksum tally against some constant, and if the check fails, SDC will immediately switch to executing the other, successful path.

This approach guarantees that all such loops terminate quickly. By introducing additional symbolic data and relaxing constraints, SDC is making an

approximation in the driver's behavior because it cannot model it precisely. In doing so, SDC is making a compromise in favor of scalability and at the expense of precision. Static analysis tools often make similar approximations. The goal of SDC is to maintain the highest precision possible while requiring no per-driver effort.

Unfortunately, as expected, this approximation may introduce false positive bug reports. False positives may stem from (a) relaxing constraints on the variables written in the loop, and (b) breaking from the loop early. We have not yet observed false positives stemming from relaxed constraints. However, we have seen false positives resulting from terminating the loop early. For example, some loops that create more paths also interact with the kernel. In one case, a loop created multiple device files, to export multiple entry points to the kernel, while also testing some device input. By breaking from the loop early, the driver would create only one device file. Fortunately, most loops either: (a) do not generate more paths, (b) do generate paths but do not interact with the kernel, or (c) provide an existing break statement that allows for early termination. We have found only a few loops that both create more paths and interact with the kernel.

6.2.3 Invoking Driver Entry Points

Symbolic execution by itself does not test software. Once a driver loads, SDC must invoke its entry points. In SymDrive, the developer needed to run their own set of test programs to exercise the driver. In contrast, SDC provides tests that invoke as many driver entry points as possible automatically. Currently, SDC supports tests for network and sound drivers. Testing a network driver involves invoking all the `ethtool` functions, network packet transmit routines, and power management code. Similarly, testing a sound driver requires testing power management, sound playback, and sound recording functionality.

SDC does not yet include a general solution to exercising all of a driver's entry points, though we have conducted a variety of experiments in this area. For example, several common ways to invoke a driver are through the `/dev` and `/sys` file systems. One approach we have examined involves enumerating all entries in these two directory hierarchies before and after loading the driver, and then noting the new entries. The idea is first to find all device files that the driver creates, and then to open and access these files with symbolic data. We suspect that this technique would be most effective at testing each entry point

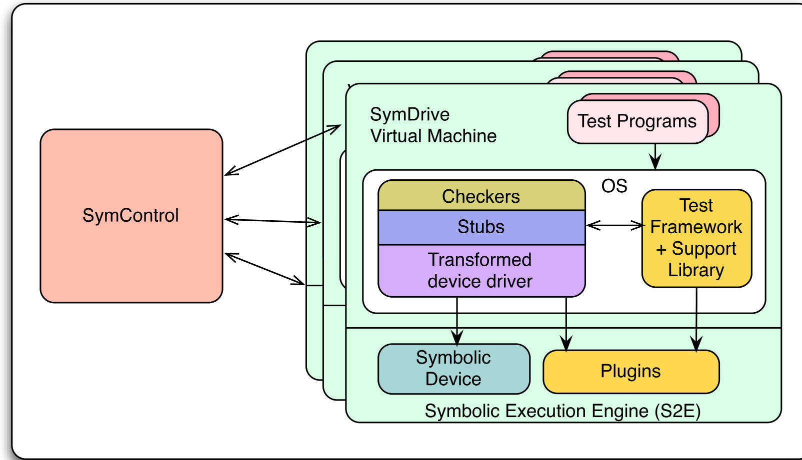

Figure 6.2: The SDC architecture. SDC consists primarily of an enhanced version of SymDrive, combined with the SymControl tool, which controls each of the running instances of SymDrive over a network.

independent of the others to allow for maximum code coverage; however, we have not yet implemented this functionality, and instead rely on tests specific to driver classes.

6.3 Testing Many Drivers

The previous section outlined the problems inherent in testing a single device driver automatically using symbolic execution. This section describes the challenges of testing drivers in parallel, on multiple machines. First, SDC must run SymGen on many drivers at once, which it does through integration with the kernel build process. Second, SDC must assign workloads to each machine, and make scheduling decisions about what work to provide each machine. We use a tool called *SymControl* to control the machines that run SDC. The architecture is shown in Figure 6.2. Third, SDC must provide facilities for collecting and analyzing the results. This section discusses each of these problems and their associated solutions in turn.

6.3.1 Automating SymGen

In order to test a driver with SymDrive, the developer must re-compile the driver with SymGen. This step is necessary to instrument the driver source code so that it can inform the underlying execution engine of useful code features. However, SymDrive requires the developer to use its own build process, which is inconvenient.

To address this limitation, SDC extends SymGen so that it can integrate with the existing Linux kernel build process, which allows developers to instrument and compile hundreds of drivers at once. We modified SymGen so that it acts as a replacement for GCC, which makes it easy to integrate with the existing build infrastructure. SDC also allows developers to compile individual drivers independently of the kernel source tree, if desired, just as SymDrive does.

6.3.2 SymControl

SymControl is a set of Python scripts that allow the developer to specify a set of machines to use for testing device drivers, as well as a list of drivers to test. When executed, SymControl connects to each test machine, copies the necessary files, and begins testing the driver under a variety of different conditions.

The design of this tool incorporates several key ideas. First, SymControl defines a set of test cases with which it can run drivers of each class. Second, it executes many drivers in parallel on multiple machines with these test cases. Third, it aggregates the results for later analysis. We next examine each of these features of SymControl in turn.

Test Scenarios

In order to test drivers of a given class, SDC develops the notion of a *test scenario*, which is simply a set of steps to execute in order to test a particular driver in some useful way. SymControl defines test scenarios essentially as lists of commands and conditions. SymControl includes the following test scenarios for network drivers:

- Load the driver, and then unload it.
- Load the driver, and attempt to transmit a packet.
- Load the driver, and invoke the power management functionality (suspend or resume).

- Load the driver, and execute a specific ethtool function, or all ethtool functions.

SymControl could, in principle, execute each of these scenarios with any supported symbolic hardware configuration. In doing so, SymControl can ensure that SDC achieves the maximum possible code coverage, because this approach is similar to running the driver with all supported hardware.

The last point in the list, regarding ethtool functions, illustrates a trade-off in the design of test scenarios. Network drivers in Linux often include a variety of ethtool entry points, which allow the user to configure functionality such as the MAC address. By defining each ethtool function as a separate scenario, SymControl can employ SDC's high-coverage mode to ensure that each entry point is exercised as comprehensively as possible. Currently, the Linux kernel includes 53 ethtool functions, although most drivers do not actually implement them all. Therefore, because power management includes two entry points, SDC could re-execute a driver up to 57 times in order to test all its functionality (the sum of 53, two power management scenarios, and the other two scenarios). Moreover, SDC could, in principle, conduct each of these 57 tests under a different supported hardware configuration. If a driver supported 20 PCI devices, the maximum number of runs would increase to the product of these two numbers, or 1,140 runs. Running the driver this many times would likely yield few additional insights.

In order to reduce this figure, SymControl currently executes each test scenario with a random supported hardware configuration, with the expectation that SDC will be able to test the initialization code more thoroughly as a result. It also executes all ethtool functions in a single pass. This approach reduces the maximum coverage of individual ethtool functions but also keeps the total number of tests manageable.

One feature currently missing from SymControl is the ability to feed information from one test into the next. For example, if the driver does not initialize successfully because of a bug, SymControl will still attempt to test the driver with each scenario. In the case of the ethtool functions, it would make sense first for SymControl to establish what functionality the driver actually implements, which may only be a few ethtool functions. Then, SymControl could target those few functions for test. We believe the ability to guide subsequent tests on the basis of results from earlier ones would be straightforward to implement.

Test Machine Interaction

In order to fulfill the goal of distributing workloads across many machines, SymControl needs the ability to interact with these machines. To facilitate this interaction, SymControl uses a Python remote procedure call library [56] in a client-server architecture. The machines used for running SDC serve as clients. This approach enables SymControl to (a) process all test-related messages and output in real-time, as the tests take place, and (b) delegate drivers and tests to any machine in need of work.

By running a client process on every test machine, SymControl can intercept all output from both the SDC runtime and all log messages from the kernel under test. Each client machine sends this output to the SymControl server. In this way, SymControl centralizes collection of all SDC output on all machines, and can make decisions on the basis of this output.

This approach enables SDC to make decisions based on the results of each test. For example, if a test scenario hangs for some reason, SymControl can abort that test and try the next one. Similarly, SymControl can quickly detect if the client has crashed for some reason, or if it has finished executing a test. As soon as a client finishes a test, SymControl checks to see if there are more tests to run, and if so, immediately assigns one. This approach keeps resource utilization high and prevents machines from ever sitting idle.

SymControl stores a list of all the drivers to test as well as each driver's class. The driver class defines the set of tests to use with that driver. At present, SDC supports the sound and network driver classes. In the future, we hope to add support for testing other types of drivers. In addition, SDC could use the same SymGen static analysis to find these pieces of information, but we have not yet done so.

To test drivers, SymControl first copies all drivers and SDC files to each remote machine. Then, SymControl loops over a set of (driver, test scenario, hardware configuration) combinations, and delegates each to a remote machine. The remote machine then runs the appropriate test, and returns the results. This process typically results in executing a given driver 5-10 times.

Other systems such as Cloud9 [35] parallelize symbolic execution at a much finer grain than does SDC. The benefit of Cloud9's approach is a better capacity to test an individual piece of code more deeply, by executing more paths. However, SDC instead aims to test many drivers more broadly, making fine-grained parallelization less important.

6.3.3 Results Analysis

Analyzing results for a driver in SymDrive is a manually process. The developer runs the driver, and watches for any bug reports to appear. All output is recorded, so the developer can also examine the result later. This process is straightforward when testing only a single driver at a time, but does not scale. If the developer runs 100 drivers, each with multiple hardware configurations and test scenarios, it becomes impractical to analyze the output manually. To address this problem, SDC includes a script for analyzing the results.

The script reads all driver output from all runs, and uses regular expressions to find any bug reports in the output. Bug reports in SDC all use a standard Linux stack trace format, which simplifies the process of extracting them from the execution output.

One problem that arises in this process is the need to bucket similar bug reports, to reduce redundancy. Many driver bugs occur along multiple paths, and it does not make sense to flood the developer with duplicate bug reports in these cases. To bucket bug reports effectively, SDC uses a basic heuristic that treats each call stack entry independently. If two stack traces associated with the same driver share more than 30% of their entries, then SDC considers them to be duplicates. This policy is very conservative and could unnecessarily bucket multiple dissimilar reports, with the benefit of minimizing developer effort at identifying duplicates.

Other work has examined more effective strategies at bucketing bug reports [43]. Because of time constraints, we have not yet implemented these techniques; however, we believe them to be compatible with SDC's design.

6.4 Evaluation

This section provides a discussion of SDC's code coverage, bug finding results, and scalability. All results are preliminary because SDC development is ongoing.

6.4.1 Methodology

We currently run SDC itself inside VMWare Workstation virtual machines. In this way, we can easily provision a single virtual machine, and then create copies as needed. We ran all experiments in this section on a 6-core Intel 3930K machine with 32GB RAM, with one VM to run SymControl, and three more

Driver Class	Num. Drivers	Touched Funcs.	Coverage
Network	107	66%	40%
Sound	66	77%	59%

Table 6.1: SDC results. The table shows the number of drivers in each class, the median percentage of functions touched in each driver, and the median coverage of the functions touched.

VMs to run SDC. Each VM was configured with 8GB of RAM and two CPU cores, and ran with Ubuntu 11.10 x64.

To test SDC, we collected a set of PCI network (107) and sound drivers (66) from the Linux v3.1.1 kernel, released 11-11-2011. This set represents nearly all network and sound drivers in Linux that support x86 and run on a PCI bus.

6.4.2 Code Coverage

To calculate code coverage, we find the median percentage of all functions that SDC executed at all, as well as the median code coverage of the functions that it did execute. Table 6.1 shows the results.

We have identified three likely reasons the results are not higher. The first reason is that SDC may trigger a bug along many paths, which prevents the driver from initializing. For example, many drivers have bugs that prevent them from servicing interrupts properly at the time they register their interrupt handler. The most common reason is that they have not initialized all necessary data structures. A spurious interrupt immediately after registering the interrupt handler can easily crash these drivers, because it attempts to access uninitialized memory. Because the `request_irq` function typically must complete before the driver is usable, and because SDC may invoke the interrupt handler many times before initialization completes, SDC is unable to test these drivers further until the developer fixes the bug. We worked around the problem in SymDrive itself because we manually corrected bugs as we found them, but this approach is not feasible in SDC.

The second reason for low coverage is that SDC occasionally becomes “stuck” in some part of driver initialization. One of the most common problems that we have not addressed involves drivers that execute a complex series of operations before returning an error. For example, a driver might encounter an error, and then execute a large number of cleanup routines before returning the error. SDC fails to deprioritize the failing path until the cleanup code completes, because the driver does not actually return the error until that point. The problem is

that executing the cleanup itself takes considerable time, and when multiple paths are affected, SDC wastes time executing this code before it can find a successful path. To solve this problem effectively, we need to develop additional heuristics that can identify paths that inevitably fail, which would allow SDC to prioritize successful paths earlier.

Finally, SDC may fail to invoke all driver entry points. This problem is not as significant as the previous two because we have written tests specifically tailored to network and sound drivers. However, testing other classes of drivers may have this problem, and further work is necessary to find and exercise all driver entry points automatically.

6.4.3 Bug Finding

Using our current bucketing heuristic, we have identified 164 bugs in the 107 network drivers. However, this result is preliminary for several reasons.

First, we have not yet examined the bugs to confirm whether they are genuine. We suspect false positives stem from bugs in the test framework, mistakes in symbolic hardware configuration, and breaking out of loops early. With additional time, we expect to work around most of these problems.

Second, we are not yet certain that our bucketing heuristic is successfully removing all duplicate bugs. Duplicate bug reports would artificially inflate this result. However, based on our continuing examination of the results, we do not expect this problem to be significant.

Despite these caveats, it is clear that SDC is capable of finding many bugs in many drivers with low developer effort.

6.4.4 Scalability

SymControl currently aggregates all results continuously. The machines running SDC transmit their output in real time to the VM running SymControl, which suggests a clear scalability bottleneck. However, SymControl currently uses very little CPU time because it does almost no calculations. Network bandwidth and disk throughput are more likely to be limiting factors, but SymControl uses data compression throughout to minimize limitations in these areas. We have not evaluated SDC's scalability, largely because SDC does not yet support enough drivers to make running it on more machines worthwhile.

We do not have precise performance results, but estimate that it takes 36 hours to test all 173 drivers on our testbed. We could improve this result considerably by using several dozen test machines instead of only three VMs.

6.4.5 Discussion

These preliminary results are encouraging, and suggest that with further development, SymDrive can operate at a significantly larger scale. We estimate that by adding support for a few more driver classes and buses, we could increase the number of supported drivers by another several hundred.

6.5 Summary

SDC automates the testing of many drivers in parallel in a cluster-based architecture. To accomplish this effectively, SDC combines new static analyses and runtime infrastructure to prepare and test many drivers on multiple machines in parallel. The system uses generic test scenarios to invoke relevant driver entry points on each driver. Using SDC, we can execute a large amount of driver code automatically, and find many bugs with no per-driver developer effort.

7 LESSONS LEARNED AND FUTURE WORK

7.1 Lessons Learned

After developing Decaf Drivers and SymDrive, several common themes emerged. These themes relate to the parts of the development process that posed the greatest challenges, and were often common to both projects. This chapter discusses these challenges in more detail, and considers some alternative approaches that would mitigate the challenges.

7.1.1 Driver Interfaces

The Decaf Drivers and Microdrivers projects both required interposing on the driver/kernel interface. Many other existing and potential driver projects also have this requirement. For example, Shadow Drivers [125] interposes on this interface to improve driver reliability. Boyd-Wickizer et. al explicitly states that, "A well-defined driver interface, such as (Termite [117]), would make it easier to move drivers to user-space..." [20]. This section outlines some of the challenges arising from this design, and the lessons learned.

7.1.1.1 Challenges

By using static analysis tools such as CIL [103] to analyze driver code automatically, working with the driver/kernel interface becomes more tractable. For example, these techniques allow developers to generate code that wraps driver and kernel functions automatically. However, the driver/kernel interface is broad, fine-grained, non-uniform, and constantly changing, and even static analysis was not always sufficient. We next consider each of these properties in turn.

Broad

In our experience, Linux and FreeBSD drivers typically invoke dozens of kernel functions, and export as many as several dozen driver functions to the kernel as callbacks. Transferring data across this interface typically involves a variety of pointers and interconnected data structures. Static analysis resolves many of these problems, but some design patterns create problems.

Function pointers are one area where static analysis is insufficient. Drivers often invoke the kernel through function pointers. It may not be possible to

determine what kernel function a given pointer points to at compile time, so interposing on that call is problematic. As an example, consider the kernel function `dvb_dmx_init`. This function initializes a structure with several function pointers to other kernel functions. The driver then calls the pointers, and neither Decaf Drivers nor SymDrive are able to interpose on the call properly. Decaf Drivers requires an annotation in this case, and in SymDrive, the runtime will not be able to correctly prioritize some paths and check correctness properties.

Resolving this limitation completely at compile time may be impossible, because the contents of function pointers can depend on data only available at runtime. Assuming the presence of a dynamic component, such as SymDrive's test framework, it may be possible to store all possible kernel functions in a large hash table, and then lookup unknown function pointers in the table as needed, but this approach is cumbersome and error-prone.

Interconnected, recursive, nested, and opaque data structures are another source of complexity. The Linux kernel includes a variety of linked lists, for example, and copying these properly can be a significant challenge without detailed knowledge of their specific implementations, such as whether they are circular or null-terminated. Nested data structures are also problematic. For example, Linux kernel and driver code often casts pointers from one structure type into another via the `container_of` macro. Marshaling this data in general can be impossible without significant numbers of code annotations.

Fine-Grained

If the only interface between drivers and the kernel was the function-call interface, then a combination of static and dynamic analyses would likely be more successful. However, Linux and FreeBSD drivers routinely read and write kernel data directly, as opposed to through accessor functions.

Although there are several arguments in favor of a fine-grained driver interface, they are generally unconvincing. These arguments include: (a) C programming language limitations, which does not encourage object-oriented design or the use of accessor functions, and (b) potential performance impacts, which might dissuade kernel and driver developers from using them. Although C does little to protect the contents of structures, it would nevertheless be reasonable to design the driver/kernel interface to prevent direct access to kernel data structures. Similarly, the second concern is becoming less relevant as compiler technology improves. Short, statically-defined accessor functions

today are likely to be just as fast as accessing the data directly.

This fine-grained interface imposes additional difficulties on both Decaf Drivers and SymDrive. Decaf Drivers contains significant complexity to marshal and demarshal data structures that drivers access directly. Similarly, SymDrive's function-level checkers cannot verify that the driver is correctly modifying individual fields in kernel data structures at the time of access, causing it to miss potential bugs.

Inconsistent

The interface is also inconsistent. In the case of driver entry points, drivers might export some functions via function pointers stored in kernel data structures, some functions as parameters to functions, and some functions via the `EXPORT_SYMBOL` macro. In the worst case, detecting all driver entry points is nearly impossible without an analysis of the full kernel, precisely because of the long chains of data structures in which driver function pointers are sometimes stored.

Return values are another source of inconsistency, for several reasons. First, many kernel and driver functions return `void`. In these cases, detecting whether the call is successful relies on checking some other data structure, such as a parameter. Alternatively, the caller may simply assume the call is successful. Second, a few kernel and driver functions return positive values on an error, or small negative values on success. These functions violate kernel design guidelines, but are present anyway.

One of SymDrive's primary features, namely favor-success scheduling, relies on the presence of a consistent way to detect whether the driver is executing successfully. Similarly, high-coverage mode depends on being able to inject failures in a consistent way. When the driver violates SymDrive's assumptions about what constitutes success, SymDrive is often unable to initialize the driver successfully, which prevents further testing. Similarly, high-coverage mode injects small negative numbers into return values, to see how the driver behaves. When kernel functions use other mechanisms for error reporting or other values to report an error, this approach fails.

Constantly Changing

In Linux, the driver/kernel interface constantly changes [67]. Thus, interposing on this interface manually is inadvisable given the recurring effort it

entails.

SymDrive has some issues in this area because of the test framework. Although the test framework is optional and developers can write checks as necessary, we found that upgrading the kernel we used for testing caused difficulties because the checkers in the test framework would need revision. This problem is inherent to any system that makes any assumptions about the driver/kernel interface, because it changes with effectively every kernel revision. Moreover, with no standard list of changes, the developer is left with trial and error as the only viable approach to establishing which checkers need updating.

7.1.1.2 Lessons

The challenges listed previously leave us with several lessons and goals for future kernel designs.

First, future kernel designs should assume that interacting with device drivers incurs a high cost, because the resulting implementation would be more amenable to analysis. The tight integration between drivers and the kernel today provides high performance, but makes Decaf Drivers, SymDrive, and similar tools more complex. With a narrow interface, Decaf Drivers could use simpler marshaling code generation, and SymDrive could verify driver execution properties more completely. Using only one mechanism to export driver functions to the kernel would also simplify the analyses used in Decaf and SymDrive. Similarly, eliminating function pointers as a means of invoking the kernel would help with reasoning about the interface. Requiring all kernel functions to return an integer error code would also simplify SymDrive. Ideally, the kernel/driver interface would consist exclusively of function calls with simple data structures as parameters, which would facilitate analysis of drivers independently of the kernel.

Second, given that Linux is written as it is, the best possible approach to analyzing the driver interface seems to be a combination of static and dynamic analysis when possible. In the case of Decaf, a dynamic approach was largely impossible because it was necessary to determine what data to marshal statically. The approach SymDrive takes to interposing on the driver/kernel is much simpler than that of Decaf because it was not necessary to determine what data was being transferred between the kernel and the driver. The approach used in SymDrive may be useful as an independent tool for any application that requires dynamically interposing on the interface.

By limiting the size and complexity of the driver/kernel interface, and treating device drivers more like applications with a well-defined API, static and dynamic analyses such as SymDrive and Decaf Drivers would be simplified considerably.

7.1.2 Driver Design Uniformity

The previous section touched on the issue of code uniformity, in the sense that each driver has unique properties. This issue is fundamental to open source projects with hundreds of contributors and a programming API and language that allow for multiple solutions to a given problem. Although it seems obvious, the notion that each driver is unique often causes additional development hurdles when analyzing driver code or interfaces.

One major difficulty that arose in both Decaf Drivers and SymDrive was the varying structure of device drivers. Some device drivers are written as event handlers, in which a `switch` statement dominates control flow. These kinds of drivers often read a value from the hardware, and then dispatch the driver along the appropriate path in response to that event. The Linux UHCI-HCD USB host controller driver is an example of this type, as are some storage device drivers.

This design is problematic for both Decaf Drivers and SymDrive for unrelated reasons. In the case of Decaf Drivers, DriverSlicer is unable to move most of the driver code out of the kernel. When one function can potentially touch most driver code, and if that function executes at high priority as in the UHCI-HCD driver, then Decaf must leave it in the kernel. Similarly, SymDrive has considerably more difficulty initializing a driver properly when it is written as an event handler because it is unable to infer the order in which driver functions must execute to leave the driver in a state useful for further testing. For example, the `carm_fsm_task` function in the `sx8` driver includes a large switch statement. During driver initialization, the driver must proceed through a number of distinct execution states, as represented in the `host->state` variable. While moving through these states, the driver must also invoke various functions in the correct order. SymDrive has no way to infer what the correct order is, and has difficulty initializing this driver as a result.

Other drivers, in contrast, rely more heavily on the kernel to invoke the driver's entry points in the correct order. That is, these drivers do not use large `switch` statements to decide what code to execute next; instead, the kernel

assumes this responsibility. For example, these drivers load, wait for interrupts or user actions to trigger some behavior, and then unload. Many network and sound drivers fall into this category. Drivers written in this style are easier to reason about, because the order in which driver functions execute does not depend on hardware-specific information. SymDrive has less difficulty because it "knows" implicitly what code to execute next. In event handlers, the order of operations is more difficult to infer, which frequently leaves SymDrive unable to initialize the driver properly.

7.1.2.1 Lessons

Linux clearly use drivers written in a variety of different styles, likely stemming in part from the number of developers involved. Consequently, writing additional static and dynamic analyses is made more complex. If all drivers were written using a specific style or format, driver analysis tools such as SymDrive would likely be more generalizable and we would see more work in this area. Using a better driver model would be a step in the right direction, and there are a variety of potential improvements Linux could make that would facilitate development of new analysis tools.

First, the driver/kernel interface should encode more information about the driver's state. For example, interrupt handlers in many drivers often read a value from the hardware, and then invoke different pieces of functionality depending on the value read. Drivers do not inform the kernel of these different code paths, so analysis tools have no clear means by which to infer their underlying purpose. This problem arose in SymDrive when attempting to test network packet receive functionality: this code often executes only along specific code paths reachable from the interrupt handler, which often prevented SymDrive from testing it automatically. If the interrupt handler were instead split into several independent functions, and if the driver notified the kernel of these functions, SymDrive would have an easy time invoking more driver code.

Second, the driver/kernel interface should include more enforcement to ensure the driver is in the expected state. For example, some drivers claim that initialization has completed, when, in fact, they have deferred some initialization to a background task that will not execute until some time elapses. Network drivers occasionally use this approach to defer checking whether a cable is plugged in, and only afterward do they enable transmit and receive. This approach has few apparent practical benefits, and significantly complicates

static and dynamic code analysis. Identifying the purpose of this kind of code is difficult, because it does not execute as part of the standard driver initialization, which precludes a variety of possible correctness checks.

Given that Linux allows considerable flexibility on the part of driver developers, we can also draw some conclusions about the design of SymDrive. Given that favor-success scheduling cannot work in all cases, because of insufficient source-level information about the driver's state, it may make sense to bring back the device trace/replay feature. By using concrete input that the device could actually generate, SymDrive would again have the information needed to drive execution through the required state transitions. This approach has the disadvantage of requiring some device-specific information, but because the optional trace files would be readily stored and transmitted, there are few apparent downsides.

In contrast, Decaf Drivers's design has few apparent directions for improvement given the existing driver/kernel interface. Because some drivers are designed to allow a single function to touch most others, it is unclear how Decaf could safely move most of this code out of the kernel without significantly rewriting the driver or without developing new analysis techniques.

7.1.3 Marshaling Data Structures

In addition to interposing on the control transfer between the kernel and driver, Decaf Drivers suffered from the complexity of marshaling complex kernel data structures. This problem stemmed from opaque pointers, the interconnectedness of kernel structures, and the broad driver/kernel interface.

In general, the problem of marshaling C data structures is impossible without significant annotations. Tools such as RPC [18, 63] work well for marshaling data structures that meet specific needs and adhere to strict requirements. However, C data structures in general are too complex to specify with the classic RPC interface description languages. For example, a common kernel design pattern is to embed one structure within another, and then use a macro to cast a pointer to the inner structure into the type of the outer structure. This technique provides a clever way of implementing linked lists, but violates a variety of type-safety guidelines and makes marshaling these data structures exceptionally complex.

7.1.3.1 Lessons

Although Microdrivers and Decaf Drivers serve as effective prototypes, scaling them up to the level of the full kernel would require significantly redesigning kernel and driver data structures to simplify marshaling them. We touched on one solution previously: by assuming that crossing the driver/kernel interface is expensive, the amount of data that would need marshaling would likely diminish. As it is, however, this interface is so tightly coupled that it makes marshaling data between the two halves exceedingly difficult.

Another approach is to limit the complexity of kernel data structures to those that having straightforward encodings using the RPC IDL. This approach has considerable appeal: if the RPC specification was available, it would be easy to generate the appropriate marshaling code. However, this approach would require a significant kernel redesign because many data structures are too complex to represent using this approach. A more advanced IDL and communication protocol might preserve the richness of kernel data structures but would be correspondingly more difficult to specify.

The kernel could also make more use of opaque data pointers to limit the size of the interface to well-defined fields and parameters. In contrast, the kernel currently allows direct access to nearly all kernel state though sequences of pointers. By hiding kernel data structures, DriverSlicer could simply stop when it reached an opaque pointer.

Finally, the design of Microsoft's user- and kernel-mode driver frameworks may make more sense than that of Linux's driver model in the long term for several reasons. First, these driver frameworks keep the driver/kernel interface stable across revisions of the kernel. Consequently, marshaling data structures is simpler because it does not require as much automation. Developers could amortize the cost of verifying the correctness of the marshaling code over time: the marshaling code would not necessarily need constant revision as the driver and kernel changed. Second, the user-mode driver framework already supports a number of driver classes, and can make simplifying assumptions about drivers in each class. DriverSlicer, in contrast, makes few assumptions about the driver's class and requires additional complexity as a result.

7.2 Future Work

Decaf Drivers and SymDrive effectively solved a number of problems, but clear limitations to both pieces of work remain. We next consider some of the directions these two projects could take.

7.2.1 Improving Driver Programmability

We have concentrated so far on converting existing kernel drivers to Java. Using Decaf Drivers involves taking an existing driver and splitting it into user- and kernel-mode components. Once split, the developer can migrate the driver to Java. This approach is satisfactory for migrating existing drivers to Java, but does not address the fact that many new drivers are written each year. It does not make sense to write these drivers using programming paradigms from the 1970s, only to rewrite them.

To that end, one clear direction for future work is to develop a framework for writing a decaf driver from scratch. Such a framework would require support for writing the driver nucleus–decaf driver interface. One way to accomplish this goal would be to write a static analysis tool, similar to DriverSlicer, that can analyze the decaf driver as well as the driver nucleus and generate the appropriate marshaling code.

Another approach would be to abandon the split-driver model introduced in Microdrivers, and instead execute the entire driver in user mode as in other systems [20, 98, 49, 85]. This approach has the benefit of allowing the entire driver to be written in another programming language, at the cost of reduced performance. However, because many devices require relatively low-throughput, such as user-input devices, this approach may still be appropriate for large numbers of drivers. The benefit would be a simplified development process for the whole driver, rather than just parts of it as in Decaf Drivers.

Message-signaled interrupts, which remove the need to share an interrupt with other devices, may also reduce the penalty of executing drivers in user-mode. This style of interrupt is available on all new PCI-E devices. With shared interrupts, a driver's interrupt handler checks that its associated device was the source of the interrupt. If not, it returns early. Therefore, running a driver in user mode with a shared interrupt imposes a performance penalty on any other devices sharing that interrupt, because of the additional context switch necessary for the user-mode driver to check whether it should handle the

interrupt. With message-signaled interrupts, the context switch performance penalty only applies to the user-mode driver.

7.2.2 Symbolic Execution Scalability

Although SDC makes progress in moving symbolic execution into the same realm of scalability as static analysis tools, considerable work remains. We next consider some of the limits to scalability we encountered.

Different Platforms

At present, SDC supports only drivers that compile on x86 machines. This limitation prevents us from testing drivers most often found on embedded devices, because these drivers only compile on other platforms, such as ARM. Consequently, SDC would benefit from support for symbolic execution with other instruction sets. With SymDrive, we tested several drivers that ran only on non-x86 hardware by manually porting them to x86, which is not possible at scale.

S^2E itself has preliminary support for executing ARM instructions symbolically, but this support is not complete. Using it would require further modifications to SymDrive and SDC, but is possible.

SDC also does not yet support FreeBSD drivers. We suspect that adding support for it would be relatively straightforward because SymDrive itself already provides the necessary support. Moreover, SymControl itself does not make many Linux-specific assumptions.

Symbolic Hardware

In order to initialize a driver, one must always first set up the appropriate symbolic hardware. In the case of PCI devices, extracting the details of the hardware supported is a relatively straightforward static analysis: simply scan the relevant driver files for the necessary information, and use it to set up the symbolic hardware.

However, this approach fails for large classes of devices. For example, I^2C, platform, and SPI devices all rely on a special `platform_data` structure. Because these buses do not support Plug 'n Play, developers instead manually encode specific hardware details into the kernel. These buses are present most often in embedded devices, so the kernel uses whichever `platform_data` information is appropriate for the embedded device on which it is executing. As an example,

an LED controller device present in several different SoC implementations might have different numbers of LEDs connected to it. The kernel encodes this value into per-platform structures, and then passes the appropriate structure to the driver depending on the platform. As it is, extracting this `platform_data` information statically is highly complex because of the manner in which the kernel associates it with a driver.

Other issues that arose with extracting symbolic hardware information include: (a) accurately determining the sizes of port I/O and I/O memory regions, and (b) what PCI configuration space options should be enabled to ensure successful driver execution. Solving these problems would increase the variety of devices that SDC supports. However, given the variety of ways in which drivers access I/O memory registers and the number of configuration space options available, establishing all of this information statically may be complex.

Several potential solutions to these difficulties exist. One unsatisfactory approach is to manually extract all of this hardware information from the source code. Of course, this approach is not scalable. A second idea is to provide a well-defined language in which to specify platform-specific hardware details for each driver, similar to [120]. The kernel currently relies on a variety of macros and data structures to encode this information, but a more uniform approach would enable better driver analysis. By using SymGen or a similar tool, we could generate the necessary code under `/arch` automatically, as well as communicate the information to SymDrive so that it could set up the appropriate symbolic hardware. Being able to extract or specify the symbolic hardware details necessary to initialize drivers of all classes automatically would be an important step toward running SDC more broadly.

Symbolic Kernel Input

Currently, SymDrive supports a high-coverage mode in which the runtime returns symbolic values from kernel functions. In doing so, SymDrive provides a simple way to test a driver's error paths.

However, this approach has several issues, because it can cause driver and kernel failures that would never occur in reality. As an example of the first case, a kernel function that returns an integer might always return 0. However, by supplying symbolic return values, the driver might execute another path that would never occur. This kind of error is nevertheless relatively important

because it suggests the driver is brittle in the face of possible driver/kernel interface changes, and developers would likely still benefit from these reports.

Injecting additional symbolic data can also lead to unrealistic kernel failures. SymDrive does terminate paths that return to the kernel in high-coverage mode, but unexpected failures can still occur along paths in which the driver is invoking the kernel with unusual symbolic data. For example, if SymDrive reads a symbolic kernel return value and then passes it to another kernel function, SymDrive may emit a false-positive bug warning.

To address this problem, high-coverage mode would benefit from the work done in the EIO project by Gunawi et al [68]. By first doing a static analysis of the kernel to find out the possible return values from every function, SymDrive could constrain the symbolic return values for each kernel function accordingly. In this way, high-coverage mode would much better reflect reality, because the driver would only take paths that might actually occur.

Even this approach has limitations, however. For example, a kernel function could execute successfully and allocate an object. Because execution forks into two paths after this call takes place, the driver might then take the error path because of the symbolic return value. The driver then would not free the object because of the error. The test framework would then report a memory leak because along that path the object was, in fact, allocated. One obvious approach is to avoid calling the kernel function along the failure path. This approach seems workable in practice and has the benefit of better mimicking the kernel's likely behavior. That is, on a failure path, many kernel functions "undo" any temporary work they may have done anyway. We have not yet implemented this behavior.

Test Scenarios

Currently, SDC uses an ad-hoc approach to reduce the number of potential test scenarios by invoking each scenario with a random supported hardware configuration. This technique reduces the number of runs, but is likely not optimal. It would be useful to know when selecting a different hardware configuration would increase coverage meaningfully. For example, some drivers execute special code paths, but only in the presence of a specific supported device. The 8139too driver executes some pieces of code only when an 8129 chip is present, which represents one of 24 different supported configurations. Currently, SDC has no way to know that only that specific symbolic hardware

would lead to executing those code paths, which means it would need to execute each test scenario under all possible symbolic hardware configurations in order to test this code, which is impractical without a very large cluster.

Developing a generic way of invoking driver entry points would be a useful direction in which to scale SDC, because per-class tests would no longer be necessary. Since SymDrive itself does support more driver classes, we believe the amount of effort necessary to extend SDC correspondingly is modest.

7.2.3 Device Driver Comparison

We conducted a number of experiments related to comparing device drivers implemented in both Linux and FreeBSD for equivalent hardware. Using SymDrive to run the drivers, we were able to interpose on all kernel and hardware interfaces. The goal was to see if we could compare the drivers in an interesting way. Several ideas presented themselves, but we were unable to find clear paths forward.

Hardware operations

First, by comparing the hardware operations performed by two drivers, it is possible to find bugs in the driver/device interface. For example, if the initialization path for one driver writes to a particular device register, and the initialization path for the equivalent driver in another operating system does not, then we can draw a conclusion: either there is a bug, or one driver is not using all available device functionality.

The challenge with this problem is determining whether a given difference is interesting or not. Unfortunately, drivers written for different operating systems may not provide equivalent feature sets, leaving one driver less complete than the other. The result is that the less complete driver, while correct, uses the hardware differently. We found that by focusing on device write operations, we could narrow the list of differences significantly while still producing meaningful results.

One challenge we had was the need to review device data sheets to determine whether a difference was meaningful. The problem is that these documents are difficult to find without signing non-disclosure agreements and working closely with the device vendor. Because of time limitations, we found this limitation to be a significant impediment to forward progress.

Performance

A second approach to comparing drivers may involve comparing the number of instructions along particular execution paths. In doing so, developers may be able to identify whether one driver implementation is more efficient than the other. The idea is to look at the instructions executed along a critical driver path, such as the network packet transmit path, and see if one driver spends more time on this operation than the other.

In our experiments, we found the largest challenge to be ensuring that the execution paths in each operating system were, in fact, comparable. Because different operating systems make different design choices about how to execute tasks, we found it challenging to compare execution paths in a meaningful way. For example, when transmitting a packet, the Linux kernel passes a single packet to the driver, whereas FreeBSD appears to batch multiple packets into a single driver invocation. Measuring the number of instructions required to transmit a single packet at this level becomes more complex as a result, because of the different design choices.

Although the idea of comparing device drivers across operating systems holds considerable theoretical appeal, we found the problem too challenging to make meaningful progress. Nevertheless, it may be that effective solutions to these problems exist, and that one could uncover deeper insights into device drivers with the right approaches.

7.2.4 Detecting Driver State

In order to test any device driver, one must have some indication as to what state the driver is in. This problem involves examining runtime state of many kinds, including error codes returned from kernel and driver functions, kernel function calls made, and internal driver state.

Kernel and Driver Error Codes

SymDrive treats driver and kernel return values as a key element in detecting whether the driver is making forward execution progress. If a driver function returns an error code, SymDrive assumes that the symbolic hardware has driven it into an erroneous state. Testing the driver further often involves finding an alternate path that leaves the driver running successfully.

The problem here stems from the return values that driver and kernel functions use. Most Linux drivers adhere to the kernel's standard of returning

small negative values on error conditions. However, a number of drivers do not follow this guideline, and detecting the driver's execution state is far more complex as a result. One approach is manual annotation: if a developer specifies the successful and failing return values in advance, SymDrive would have no problem, but this approach requires significant additional effort. As it stands, no automatic solution to this design limitation of Linux may be possible. Inferring the driver's state in the absence of standard error codes in the general case appears too complex.

Ideally, functions in the driver and kernel would always: (a) return an integer, and (b) return small negative values on an error, and any other value on success. Functions that need to return small negative values as part of their operation should instead use "out parameters" and return a 0. Although requiring non-void return values might diminish kernel performance by consuming additional CPU registers, it would simplify SymDrive and other analyses [68] considerably.

Operational State

If drivers included explicit details about the driver's state, such as whether execution is proceeding correctly, SymDrive could be generalized further. Currently, it is not possible to test all drivers without having a complete understanding of the various states a driver might be in. This information is necessary because otherwise SymDrive might drive execution into a state that appears to be successful, but precludes execution of major parts of the driver.

As an example of the issue, network drivers need to detect a "carrier," such as a cable or wireless network, before they will transmit packets. However, the lack of a carrier is not an error condition per-se, so even if the kernel consistently returned error codes when necessary, SymDrive might still initialize the driver in such a way that it does not detect a carrier. The result is that SymDrive is unable to invoke the transmit and receive functionality.

As a second example, some drivers include complex functions that operate as event handlers. They are called when an interrupt occurs, and then dispatch execution depending on the value received from the hardware. Some times, these drivers expect sequences of values to be read during several distinct interrupts. However, SymDrive cannot currently detect the order in which these events must occur in order to initialize the driver completely. The result is a driver that has not encountered an error condition, but has not initialized either: it simply waits for the hardware to return the correct value during some interrupt,

and eventually times out.

These two examples illustrate the difficulty of testing drivers completely when the hardware is not available. Inferring the specification of the hardware from the driver is more complex in these cases, and additional heuristics are necessary in order to execute these drivers properly.

8 CONCLUSIONS

In many respects, computer operating systems have not changed significantly in the last forty years. They are still written in the same programming language, they are still difficult for developers to debug and test, and they still include a number of bugs as a result. Kernel driver development suffers from these same difficulties, and requires having the device hardware available in order to conduct any meaningful testing. The result in all cases is buggy drivers that developers have difficulty writing and maintaining.

In order to improve device driver development and testing, this dissertation presents two new systems. The first, Decaf Drivers, provides developers with a framework with which they can gradually migrate C code out of the kernel and into user-mode Java. As a result, developers can leverage modern programming features such as exceptions and garbage collection, which simplifies the process of driver development. Moreover, Decaf Drivers provides full backward compatibility with the existing Linux driver/kernel interface, so the substantial investment made in these drivers so far is not wasted.

The second system we present, SymDrive, provides a new approach to testing drivers even when the hardware is not available. By supplying symbolic data in place of device input, SymDrive can check the behavior of Linux and FreeBSD kernel device drivers along multiple execution paths. To facilitate testing, SymDrive uses static analysis and code generation to improve the efficiency of the underlying symbolic execution. Our results show that SymDrive is effective at finding driver bugs and testing driver patches.

Together, these systems provide new approaches to writing and testing device drivers. However, as outlined previously, there are a variety of other approaches to improving driver development and testing, which leaves plenty of work to do.

Appendices

A CHECKERS

This section describes how to implement a checker in the SymDrive test framework, and presents a list of all the checkers used.

A.1 Writing a Checker

Writing checkers for Linux and FreeBSD is straightforward. To write a checker, we define a function with the following properties:

- It must always return `int`.
- The first parameter is `const char *fn`. This parameter is the name of the function that called this checker, and is generally not used.
- The second parameter is `int prepost`. This parameter contains either a 0 or 1, and indicates whether the runtime is invoking the checker to test a pre condition or post condition. The body of the checker often contains a conditional statement to check this variable and conduct the appropriate tests depending on its value.
- The type of the third parameter of the checker matches the corresponding function return type, with an extra level of indirection. For example, if the function being checked returns `char`, this parameter must be `char *retval`. Its value is undefined when the checker runs as a precondition, but contains the actual function return value when run as a postcondition.
- The remaining parameters match the types of the parameters passed to the original function, but with an extra level of indirection.

As an example, the Linux `__kmalloc` function has the signature `void *__kmalloc (size_t size, gfp_t flags)`. To write a checker, we would use the signature, `int __kmalloc_MJRcheck (const char *fn, int prepost, void **retval, size_t *arg0, gfp_t *arg1)`. We use the MJRcheck suffix because it is a unique suffix in Linux and FreeBSD.

One important feature of checkers is the additional level of indirection the provide for all function parameters and return values. This indirection allows checkers to modify driver and kernel state to facilitate further testing, if desired. The result is that each checker can modify any function parameters if needed. We

use this feature in several places to improve SymDrive's effectiveness. Specific places in which we use this feature are outlined in section A.2.

The bodies of checkers can contain arbitrary code and data, can perform arbitrary calculations, and can use any desired data structures. For example, the `kmalloc` checker stores the a pointer to the allocated object as well as its size, to verify that the object is eventually passed to `kfree`. This approach provides the checkers with maximum expressive power.

A.2 Checker List

This section presents a brief overview of the checkers currently used in SymDrive. The set of checkers is extensible, and writing new checkers is straightforward.

Driver Initialization and Cleanup

These checkers record some data, such as the driver's bus and execution state, for use in other checkers. The support library stores this information.

- Verify the driver calls an appropriate driver registration function, such as `pci_register_driver` during the call to `init_module`.
- Verify the return value from `init_module` corresponds with the return value from the registration function.
- Record the driver's execution state: these entry points allow the driver to block if desired.

register_netdev

Calls to this function inform the test framework that SymDrive is testing a network driver.

- Record that this driver is a network driver. Other checkers use this information to check network-specific properties.
- Record the result of this call, to verify other functions such as `netif_carrier_off` execute appropriately.
- The `unregister_netdev` call verifies that `register_netdev` was called successfully. Drivers should never call `unregister_netdev` without first calling `register_netdev` successfully.

Many kernel functions follow this pattern of “matched calls.” Here, calls to `register_netdev` and `unregister_netdev` must match. The object tracker itself provides a useful mechanism for ensuring calls match when allocating and deallocating objects. If no objects are involved, the checker can use a simpler interface for verifying the calls match.

netif_carrier_off

This checker verifies that `register_netdev` was called successfully. In older versions of the kernel, the ordering here was important. Newer versions remove the restriction.

This checker illustrates that drivers must invoke many kernel functions in the correct order, or problems arise. The checker interface is expressive enough to verify ordering properties even when multiple threads and driver entry points are involved. All access to support library and checker state is synchronized, so checkers can check any desired ordering properties, provided the developer can express them as pre and post condition assertions.

netif_stop_queue

See `netif_carrier_off`: the checker is identical.

Work Queues and Timers

This section applies to:

- `schedule_timeout_uninterruptible`
- `schedule_delayed_work`
- `queue_delayed_work`
- `mod_timer`
- `schedule_timeout`

The test framework optionally multiplies the timeout value parameter by a constant specified on the test framework command line. This feature is not checking any property, but instead allows the developer to slow down the rate at which the work queue runs. In doing so, the developer can avoid wasting time testing work queue code repeatedly.

These functions illustrate a convenient use-case for the checkers. They not only can check driver behavior, but can re-define the driver/kernel interface. For example, if a developer were sufficiently motivated, they could replace the entire interface with a kernel model by using the existing checker infrastructure. Using a kernel model may provide better test coverage than when using the real kernel, but would be difficult to write and maintain. Thus, while it does not make sense to implement a kernel model using the checkers, the mechanism is almost expressive enough to allow it (with some minor caveats).

`usb_submit_urb`, `usb_control_msg`

These checkers only execute in SDP currently, because the latest version of SymDrive lacks USB support. They check to ensure the driver successfully allocated the buffer parameters using the appropriate functions.

`copy_from_user`

This checker ensures the buffer into which the data is being copied is large enough. Because the object tracker is only aware of heap allocations, this checker devolves into a no-op if the buffer's address is unrecognized, which is the case for any stack-based buffers.

No check is currently possible if the driver passes a pointer to a stack variable because SymDrive has no way to find out the size of the buffer. This limitation is unfortunate because it is conceivable that SymDrive could detect stack overflow errors resulting from the use of this function. It may be possible to extend the SymGen static analysis to communicate the size of stack-based objects to the object tracker, in which case we could write more sophisticated assertions here.

device_register

This checker simply verifies that calls to `device_register` match with `device_unregister`.

Allocators

All these functions allocate some object, return a pointer or an error, and then require the driver to deallocate the object. They all use the underlying `generic_allocator` library API, and their implementations are all similar. Their primary purpose is to store information about the object allocated, such as

its size and origin. Other checkers can use this information to verify correct behavior. In addition, when the driver unloads, the test framework can verify that all allocated objects are freed.

- `free_netdev`, `alloc_etherdev_mqs`, `alloc_netdev_mqs`
- `kmalloc`, `kfree`, `__kmalloc`, `kstrdup`
- `vmalloc`, `vfree`
- `__netdev_alloc_skb`, `__alloc_skb`, `dev_alloc_skb`, `dev_kfree_skb_any`, `consume_skb`, `kfree_skb`
- `ioremap`, `ioremap_nocache`, `iounmap`
- `usb_alloc_urb`, `usb_free_urb`
- `__get_free_pages`, `free_pages`
- `dma_alloc_coherent`, `dma_free_coherent`
- `snd_dma_alloc_pages`, `snd_dma_free_pages`
- `skb_dma_map`, `skb_dma_unmap`
- `device_create_file`, `device_remove_file`

Several allocator/deallocator pairs have associated checkers but are incomplete because the driver/kernel interface is unclear. These incomplete checkers are as follows:

- `__request_region`, `__release_region`
- `dma_alloc_from_coherent`, `dma_release_from_coherent`
- `request_firmware`, `release_firmware`

This list is not representative of all kernel functions satisfying the allocate/free semantics; rather, these are simply the ones we have identified. Ideally, we would implement similar functions for all kernel interfaces using these semantics.

Locks

These functions all use the underlying `generic_lock_allocator` and `generic_lock_state` APIs to track lock acquisition and release. These checkers were most useful in SDP. In contrast, SymDrive and SDC achieve most of the same functionality by enabling lock debugging in the kernel configuration. All unlock functions are omitted from the list below for brevity.

- `__spin_lock_init`, `spin_lock_init`, `__raw_spin_lock_init`
- `__mutex_init`, `mutex_init`, `mutex_lock_nested`, `mutex_lock_interruptible_nested`, `mutex_lock_killable_nested`
- `__init_rwsem`, `down_write`, `down_read`
- `__rwlock_init`, `_write_lock_irq`, `_read_lock_irq`, `_read_lock`, `_write_lock_irqsave`, `_read_lock_irqsave`
- `_spin_lock`, `spin_lock`, `_raw_spin_lock`
- `_spin_lock_bh`, `spin_lock_bh`, `_raw_spin_lock_bh`
- `_spin_lock_irqsave`, `spin_lock_irqsave`, `_raw_spin_lock_irqsave`
- `_spin_lock_irq`, `spin_lock_irq`, `_raw_spin_lock_irq`
- `_spin_trylock`, `_raw_spin_trylock`

The purpose of these checkers is to ensure the driver uses locks correctly. They check for problems such as mismatched calls.

Driver Registration

These functions register a driver of the given type with the kernel. The test framework ensures that drivers register themselves during initialization, and call the corresponding "unregister" function during cleanup.

- `misc_register`, `misc_deregister`
- `proto_register`, `proto_unregister`
- `i2c_add_driver`, `i2c_del_driver`
- `__class_create`, `class_destroy`

- platform_driver_register, platform_driver_unregister
- spi_register_driver, spi_unregister_driver

GPIO

We implemented the entire symbolic GPIO "bus" using the SymDrive checker interface. The functions replaced with symbolic equivalents are:

- gpio_is_valid
- gpio_request
- gpio_request_one
- gpio_request_array
- gpio_free
- gpio_free_array
- gpio_direction_input
- gpio_direction_output
- gpio_set_debounce
- gpio_get_value
- __gpio_get_value
- gpio_set_value
- __gpio_set_value
- gpio_cansleep
- gpio_get_value_cansleep
- gpio_set_value_cansleep
- gpio_export
- gpio_export_link
- gpio_sysfs_set_active_low

- `gpio_unexport`
- `gpio_to_irq`
- `irq_to_gpio`

Using the checkers themselves to implement the desired symbolic bus model has the benefit that one need not learn complex Linux interface for setting up a GPIO bus. However, it is easier to miss functionality using this approach, because the compiler and runtime will not warn the developer if a function is missing from the model.

B SYMDRIVE SUPPORT LIBRARY

This section presents a complete description of the library API used in the test framework to support the implementation of checkers. Many checkers use the features in this library to simplify their implementation.

B.1 Structures

The test framework library API exports a number of structures for use with checkers. These structures keep track of global driver state across entry points. The current implementation is incomplete, and a number of valid scenarios remain unimplemented, but these limitations are not inherent to the design.

`global_state_struct` This structure includes data relevant to drivers of any type, running on any bus. It includes information on the following driver properties: (a) the current execution context and whether scheduling is allowed, (b) the device IRQ and whether it is enabled or disabled (if applicable), and (c) the object tracker.

`XYZ_state_struct` The remaining structures store information specific to drivers of specific types. For example, some structures are specific to network drivers, regardless of the bus, while other drivers are specific to drivers for devices running on a particular bus, irrespective of type.

`hkey / hvalue` These structures are used in the object tracker, which is effectively a hash table. The object tracker maps `hkey` objects to `hvalue` objects. The `hkey` is always simply a memory address corresponding to an object allocated by or provided to the driver. The `hvalue` structure contains additional information on the object, such as its size, and how it was allocated. The object tracker also tracks lock state, and this information is stored within the `hvalue` object as well.

Checkers use these structures to establish what properties to verify. In the future, we would like to encapsulate these structures behind an API, instead of directly exposing their fields. The current design is a prototype intended to demonstrate SymDrive's capabilities, and it would not make sense to preserve direct access to these structures from checkers in a more complete implementation.

B.2 Functions

The primary interface of the library API is through a set of exported functions. This section describes these functions.

`set_driver_bus` Notify the test framework what kind of bus the driver uses. This test framework can then ensure the driver calls appropriate kernel functions.

`get_driver_bus` Returns the current driver bus.

`remove_driver_bus` Usually used during the driver's shutdown path. For example, `misc_deregister`, `usb_deregister` and similar functions all invoke this.

`set_driver_class` Specify the class of the driver. Currently supported classes are: (a) `DRIVER_NET` and (b) `DRIVER_SND`.

After implementing the initial test framework, it became clear that this library functionality is not very scalable and does not help find many real bugs. It may make sense to remove this functionality in the future.

`get_driver_class` Returns the current driver class, as specified with `set_driver_class`.

`set_device_state` Specifies whether the device is enabled (`DEVICE_STATE_ENABLED`) or disabled (`DEVICE_STATE_DISABLED`).

`set_call_state` This function is used when invoking kernel functions. There are three states for a given function: (a) `NOT_CALLED` (b) `CALLED_OK` and (c) `CALLED_FAILED`. The idea is to track whether a given kernel function has succeeded or failed.

As future work, it would make sense to track all kernel function calls automatically, according to their return values. For example, using a data structure similar to the object tracker, we could store the results of each kernel function, mapping function names to results. In this way, a variety of manually-written test framework structures and variables would not be necessary. For example, it would no longer be necessary to track whether a particular kernel function succeeded or failed, it would simply be available in this enhanced object tracker.

The driver/kernel interface is highly complex, and we found a variety of bugs in which the driver was misusing this interface. For this reason, we believe it makes sense to extend the test framework to track usage of this interface more completely and transparently, since we expect that a richer set of checks would result.

`push_blocking_state` Specify that the driver is now in a state the either allows or disallows blocking (scheduling). Allowed states are: (a) `BLOCKING_UNDEFINED`, (b) `BLOCKING_YES` and (c) `BLOCKING_NO`.

The test framework tracks this information using a stack. Entries on the stack should become progressively more restrictive. For example, it never makes sense for a driver to be in a state that does not allow blocking, and then enter a new state that allows blocking. The reverse condition, however, is reasonable.

A variety of checkers use this functionality to ensure the driver does not attempt to block along specific code paths. For example, with this information, the test framework can verify that calls to `kmalloc` use the appropriate `GFP_` flags. Use of this API is optional, but it is helpful in detecting this class of bug.

The current implementation does not correctly track the state in all cases. Instead, the state is often left as `BLOCKING_UNDEFINED`. With additional effort, we could track this information more completely and likely find more bugs as a result.

`pop_blocking_state` Revert to the blocking state the driver was in before it issued a call to `push_blocking_state`.

`kernel_blocking_state` Return the current blocking state: either blocking is allowed, it is not, or it is unknown.

`push_user_state` This function and `pop_user_state` are analogous to `push_blocking_state` and `pop_blocking_state`, respectively.

Instead of tracking whether the driver can block, this stack tracks whether an unprivileged user can invoke this code path. In this way, the test framework can verify additional security properties. For example, if an unprivileged user can invoke a particular entry point, then that entry point should not allocate resources without freeing them. Otherwise, the user could crash the kernel.

We do not currently use this feature as comprehensively as might be desired, largely because it is difficult to know what driver entry points are user-accessible. As is the case with much of the test framework, having a more complete understanding of what precise behavior is and is not allowed by the kernel interface would be helpful.

`pop_user_state` See `push_user_state`.

`kernel_user_state` This function is analogous to `kernel_blocking_state` except it returns whether an unprivileged used can directly invoke this code.

`push_kernel_state` This function, along with `pop_user_state`, is essentially a combination of `push_blocking_state` and `push_user_state`.

`pop_kernel_state` See `push_kernel_state`.

`set_device_interrupts` This function specifies whether it is reasonable to invoke the driver's interrupt handler along this path.

This function is currently used only via the lock tracking functionality. It is not directly related to checkers because it does not help verify any correctness properties; however, it does allow SymDrive to model execution more accurately by disallowing interrupts along paths that would not allow interrupts in reality. For example, the test framework calls this function when the driver holds a spinlock acquired via `spin_lock_irqsave`, because the driver's interrupt handler would never execute when this lock is held. SymDrive would produce false positive bug reports if it did not track this information and invoked the interrupt handler at inappropriate times.

`can_call_interrupt_handlers` Returns true if the driver's interrupt is enabled, false otherwise.

`assert_allocated_weak` Verifies that an allocated object is at least as large as specified. If the object tracker does not have a record of the object, then the function succeeds. This function is useful when the object in question may be allocated on the stack and the test framework is unaware of its size. See `assert_allocated`.

`assert_allocated` Verifies that an object has been allocated previously, is present in the object tracker, and is at least the appropriate size.

Suggested future work: account for objects allocated on the stack. Currently, only objects stored on the heap are present in the object tracker, and this limitation precludes a variety of bug checks. If all stack objects were also included in the object tracker, then it would be reasonable to use this function (as opposed to `assert_allocated_weak`) much more broadly.

`generic_allocator` Allocate an object of the specified size and type. This function is most commonly used in conjunction with checkers for kernel functions that allocate objects, such as `kmalloc`. After gaining experience with SymDrive, it became clear that this mechanism could be generalized to most kernel functionality that requires the driver to invoke a pair of functions, such as the creation and removal of a device file.

`generic_free` The corresponding function for freeing or deallocating a kernel object.

`generic_lock_allocator` This function allocates a lock object. Lock objects are never freed, so there is no corresponding `free` function.

`generic_lock_state` Alter the state of the specified lock. Given an existing lock object, this function notifies the object tracker that the lock is now either acquired or released. This function is useful to ensure that a lock operation is paired correctly with its corresponding unlock operation.

`mem_flags_test` Verifies that the given `GFP_XYZ` flag is correct given the current execution context, as specified with `push_kernel_state` and related functions. For example, this function ensures that `GFP_ATOMIC` is used only along paths that disallow blocking.

C BUGS FOUND

This appendix describes the bugs we found in Linux and FreeBSD using SymDrive. This list includes all bugs found with SymDrive at any time with any version. Bugs for which additional information is available include footnotes referring to relevant URLs. Not all of the bugs in the following lists are necessarily "bugs" in the traditional sense; instead, the list also includes references to other unusual code or design features that SymDrive uncovered and may warrant further investigation. We also include and note a few bugs that were too complex to verify, and could stem from bugs in SymDrive.

C.1 Linux v3.1.1

lp5523

Bug #1 A NULL terminator is missing on the `lp5523_attributes` array. The compiler placed this array next to `lp5523_led_attribute_group` in the binary, and the kernel keeps these arrays adjacent when loading the driver into memory. This second array is NULL-terminated, so the the driver nearly works as a result. However, problems arise if the driver is unloaded and reloaded: the missing NULL terminator causes these two arrays to "run together" and results in passing the corresponding array elements to incorrect kernel functions. Greg Kroah-Hartman verified this bug is legitimate,[1] though the patch remains un-merged. The bug is visible in the driver code.[2]

Bug #2 In this driver, `lp5523_probe` calls `lp5523_init_led` repeatedly. After this call to `lp5523_init_led`, `lp5523_probe` calls `INIT_WORK` on the corresponding `brightness_work` structure. Suppose that, during a call to `lp5523_init_led`, the call to `sysfs_create_group` fails. This call takes place immediately after the call to `led_classdev_register`. In this case, `lp5523_init_led` calls `led_classdev_unregister`. But because `led_classdev_register` completes successfully, `led_classdev_unregister` calls `led_brightness_set`, which invokes the driver func-

[1] `http://driverdev.linuxdriverproject.org/pipermail/devel/2012-April/025476.html`

[2] `http://lxr.linux.no/#linux+v3.1.1/drivers/leds/leds-lp5523.c#L747`

tion `lp5523_set_brightness`. Unfortunately, this function calls schedule_-work on the uninitialized work queue and the kernel panics. The bug is fixed[3], and visible in the driver code.[4]

This bug led to a separate kernel patch that prevents the kernel from panicking when uninitialized work queues are used accidentally.[5]

ks8851

Bug #1 A missing `mutex_lock/unlock` can potentially cause a crash. We authored and submitted a patch to correct it. The patch moves the `ks8851_rdreg16` call above the call to `request_irq` and caches the result for subsequent repeated use. A spurious interrupt may otherwise cause a crash. The bug is patched in the latest kernels[6] and remains visible in the old driver code.[7]

We would like to express thanks to Stephen Boyd, Flavio Leitner, and Ben Hutchings for feedback while developing the patch.

Bug #2 The `request_irq` and `free_irq` calls are matched incorrectly. The `dev_id` parameter passed to `free_irq` needs to match the one passed to the corresponding `request_irq`. The bug is fixed[8] and is visible in the old driver code.[9]

Bug #3 There appears to be a hardware-dependence bug in `ks8851_rx_pkts`:

```
rxh = ks8851_rdreg32(ks, KS_RXFHSR);
rxlen = rxh >> 16;
rxlen -= 4;
rxalign = ALIGN(rxlen, 4);
skb = netdev_alloc_skb_ip_align(ks->netdev, rxalign);
```

---

[3] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=5391dd0a9d084633e20e6583cfed233581c452f9

[4] http://lxr.linux.no/#linux+v3.1.1/drivers/leds/leds-lp5523.c#L943

[5] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=f5b2552b4ebbeadcadde1532d7bbd3f850719046

[6] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=51c61a2838c33dab7b6659b9a3e008bb1b40bc9b

[7] http://lxr.linux.no/#linux+v3.1.1/drivers/net/ks8851.c#L1664

[8] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=e8195b24feb208f6e944e7542779f4397776794d

[9] http://lxr.linux.no/#linux+v3.1.1/drivers/net/ks8851.c#L1684

A hardware error could lead to the allocation of a very large packet but it is not clear this counts as a bug. The allocated skb could be up to 65KB in size, which is bounded, but significantly larger than normal.

The bug is visible[10] but unfixed.

Bug #4 The driver does not cancel the work queue properly. The result is that `ks8851_irq_work` may execute during driver unload and can crash the system. We added `cancel_work_sync` as an experiment, but this approach only shrinks the window of vulnerability. This bug was validated by Stephen Boyd,[11] assuming the premise that hardware-dependence bugs should be fixed is valid. However, without a good fix, it remains unpatched. The bug is visible in the driver's source code.[12]

hostap

Bug #1 The driver accidentally used the `GFP_ATOMIC` allocation flag when `GFP_KERNEL` would suffice. Although the code is correct as it is, the `GFP_KERNEL` flag is a better choice in this execution context. The bug is patched,[13] and visible in original driver's source code.[14]

C.2 Linux v2.6.29

This section lists bugs we found in Linux v2.6.29. Although many of these were fixed independently in subsequent kernel versions, we were not aware of them before using SymDrive to find them.

[10] `http://lxr.linux.no/#linux+v3.1.1/drivers/net/ks8851.c#L466`

[11] `http://lists.openwall.net/netdev/2012/04/13/139`

[12] `http://lxr.linux.no/#linux+v3.1.1/drivers/net/ks8851.c`

[13] `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=579b0637647de6e03e5707b32eb7c2ce56cc0f27`

[14] `http://lxr.linux.no/#linux+v3.1.1/drivers/net/wireless/hostap/hostap_main.c`

8139too

Bug #1 This driver includes a subsequently-corrected[15] hardware-dependence bug. The driver will crash if it reads an unexpected value from the hardware and a debug flag is enabled. The bug is visible in an archive.[16]

Bug #2 During driver unload, a race condition can occur when when the driver is run with an RTL8129 chip. The bug is fixed[17] and remains visible in the unpatched driver.[18]

be2net

Bug #1 Calling `netif_stop_queue` before calling `register_netdev` is an error in this version of the kernel. The bug was subsequently fixed,[19] and is visible in the v2.6.29 driver code.[20].

Bug #2 Similar to the previous bug, this driver calls `netif_carrier_off` before calling `register_netdev`. This bug is also fixed[21] and visible.[22].

Bug #3 `be_open` returns errors but the transmit queue is still marked as running. Instead, it should be stopped. The bug is fixed[23] and visible in the v2.6.29 driver code.[24]

Bug #4 `be_probe` returns positive numbers when an error occurs under some circumstances. It is unclear whether this bug is fixed, though it likely

---

[15] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=3fd7fa4a89f0b85b9b33e922f15a2289c0fb8499

[16] http://lxr.linux.no/#linux+v2.6.29/drivers/net/8139too.c#L858

[17] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=23f333a2bfafba80339315b724808982a9de57d9

[18] http://lxr.linux.no/#linux+v2.6.29/drivers/net/8139too.c#L1102

[19] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=9b6cefd6593c2b661e0052d53f2fff6fc5463975

[20] http://lxr.linux.no/#linux+v2.6.29/drivers/net/benet/be_main.c#L1579

[21] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=d09f698056a33c8b078497fb23e3304b6f8a908f

[22] http://lxr.linux.no/#linux+v2.6.29/drivers/net/benet/be_main.c#L1579

[23] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=889cd4b2e529db4988525b0b3e6fb2c095760848

[24] http://lxr.linux.no/#linux+v2.6.29/drivers/net/benet/be_main.c#L1415

is because this driver has been significantly modified since the v2.6.29 implementation. The file in which the problem originates, `be_cmds.c`, has also been modified heavily.

Bug #5 `be_open` also returns positive numbers during an error condition under some circumstances, and this bug has also been fixed.[25] The bug is visible,[26] though the locations at which the driver could generate the positive numbers are not obvious. The driver propagates the invalid return values throughout the call stack.

Bug #6 This driver contains at least one hardware-dependence bug. `be_tx_compl_get` returns an unexpected value from the hardware in `be_poll_tx`, and this causes `be_tx_compl_process` to be called and fail with the `BUG_ON`. This bug has likely not been fixed, though the bug is visible.[27]

Bug #7 During cleanup: `be_exit_module -> be_remove -> be_close -> napi_disable -> be_poll_tx -> netif_wake_queue`. This code path starts the transmit queue during the shutdown sequence after the driver had called `netif_stop_queue` to disable it. It appears to be a race condition because `be_poll_tx` is clearly not called by `napi_disable` – the stack is mixing up two threads. It is unclear whether this is fixed. The bug is visible.[28]

Bug #8 `cleanup_module -> be_exit_module -> be_remove -> be_close -> be_rx_queues_destroy -> be_rx_q_clean -> get_rx_page_info -> pci_unmap_page`. The problem here is that `pci_unmap_page` is freeing something it never allocated. We were unable to verify this bug independently, though it was probably fixed.[29] Because of this bug's complexity, it remains unclear how precisely this problem occurred.

---

[25] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=889cd4b2e529db4988525b0b3e6fb2c095760848

[26] http://lxr.linux.no/#linux+v2.6.29/drivers/net/benet/be_main.c#L1415

[27] http://lxr.linux.no/#linux+v2.6.29/drivers/net/benet/be_main.c#L910

[28] http://lxr.linux.no/#linux+v2.6.29/drivers/net/benet/be_main.c#L1478

[29] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=cdab23b7017693c00dd69fa28bcdf5b0434b3838

dl2k

Bug #1 There is a hardware dependence bug in the shutdown routine `rio_close`. The problem is that it calls `pci_unmap_single` and passes in the parameter `desc_to_dma(&np->rx_ring[i])`. `np->rx_ring` is DMA memory allocated in `rio_probe1` with the `pci_alloc_consistent` function. Consequently, `np->rx_ring[i].fraginfo` could contain arbitrary data if the device misbehaves. Because `desc_to_dma` uses this unvalidated data, it can crash. This bug is unfixed but remains visible.[30]

e1000

Bug #1 The driver incorrectly calls `netif_stop_queue` before calling `register_netdev`. The bug was fixed[31] and remains visible in the v2.6.29 code.[32]

Bug #2 This driver calls `netif_carrier_off` before `register_netdev`. As before, the bug was fixed,[33] and remains visible in the v2.6.29 code.[34]

Bug #3 There appears to be a `pci_map_page/pci_map_single/pci_unmap_page/pci_unmap_single` mismatch. SymDrive found this using the object tracker mechanism. The bug was likely fixed[35] but we are not certain. The bug is distributed throughout the `e1000_main.c` file, which is also available.[36]

econet

This driver contains two security vulnerabilities. Although we knew about these errors in advance, we were able to detect them with SymDrive and reproduce the vulnerability.

---

[30] http://lxr.linux.no/#linux+v2.6.29/drivers/net/dl2k.c#L1746

[31] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=eb62efd287fe6e12d18083287e38e4a811c28256

[32] http://lxr.linux.no/#linux+v2.6.29/drivers/net/e1000/e1000_main.c#L1222

[33] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=eb62efd287fe6e12d18083287e38e4a811c28256

[34] http://lxr.linux.no/#linux+v2.6.29/drivers/net/e1000/e1000_main.c#L1222

[35] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=b16f53bef9be0a756a0672e27d0a526686040e02

[36] http://lxr.linux.no/#linux+v2.6.29/drivers/net/e1000/e1000_main.c

Bug #1 The details of this bug are available.[37,38,39,40]

Bug #2 The details of the second bug are also available.[41,42,43,44]

et131x

Bug #1 This driver calls `netif_carrier_off` before `register_netdev`. The problematic calls are in `et131x_link_detection_handler`, which itself is called from `et131x_pci_setup`. The bug is fixed,[45] and the details are available in the v2.6.29 driver code.[46]

Bug #2 Similar to dl2k, this driver has a hardware-dependence bug. The function call of interest is `pci_unmap_single` in `et131x_free_send_packet`. The driver passes this kernel function addresses stored in DMA memory. Although it is unclear whether the bug is fixed, the bug is visible in the v2.6.29 driver.[47]

forcedeth

Bug #1 There is a `pci_map_single/map_page/unmap_single/unmap_page` mismatch in `nv_release_txskb`. The forcedeth driver does not use the DMA API properly in its transmit completion path and in `nv_loopback_test()`. `pci_map_single()` should be paired with `pci_-`

---

[37] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3849

[38] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=349f29d841dbae854bd7367be7c250401f974f47

[39] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=fa0e846494792e722d817b9d3d625a4ef4896c96

[40] http://lxr.linux.no/#linux+v2.6.29/net/econet/af_econet.c#L263

[41] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3850

[42] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=349f29d841dbae854bd7367be7c250401f974f47

[43] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=16c41745c7b92a243d0874f534c1655196c64b74

[44] http://lxr.linux.no/#linux+v2.6.29/net/econet/af_econet.c#L648

[45] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=0f18f767e5f9a4c5a9fa976df168d0f3e33c91da

[46] http://lxr.linux.no/#linux+v2.6.29/drivers/staging/et131x/et131x_initpci.c#L494

[47] http://lxr.linux.no/#linux+v2.6.29/drivers/staging/et131x/et1310_tx.c#L1238

`unmap_single()` and `pci_map_page()` should be paired with `pci_unmap_page()`. However, the forcedeth transmit path uses `pci_map_single()` and `pci_map_page()`, but the transmit completion path only uses `pci_unmap_single()`. Similarly, `nv_loopback_test()` uses `pci_map_single()` and `pci_unmap_page()`.

This bug is fixed[48] and the relevant driver code is available.[49]

me4000

Bug #1 The `me4000_open` driver includes a bug that appears to stem from a copy/paste error, in which the developer accidentally passed the wrong parameter to `spin_lock`. The driver contains the call `spin_lock(&cnt_context->use_lock)`, but this code is incorrect. Instead, it should use `ext_int_context`. It appears that the developer copy/pasted this code from elsewhere and did not update it appropriately when used here.

This bug was fixed in the copy of the driver available from the manufacturer.[50] However, the Linux kernel dropped the me4000 driver some time after the release of v2.6.29, and merged its functionality with the comedi driver. Consequently, this bug is largely obsolete. The bug is, nevertheless, visible in the v2.6.29 code.[51]

pcnet32

Bug #1 The `pcnet32_set_ringparam` function calls `spin_lock_irqsave(lp->lock)`. Then, the function calls `pcnet32_realloc_rx_ring` and `pcnet32_realloc_tx_ring`, which both call `pci_free_consistent`. However `pci_free_consistent` is a "might sleep" function, and so produces kernel warnings (via `WARN_ON(irqs_disabled());` in `dma_free_coherent`). Assuming the function can sleep, this bug could lead to a kernel deadlock under the right conditions.

---

[48] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=73a370795371e41f72aeca61656d47adeadf28e5

[49] http://lxr.linux.no/#linux+v2.6.29/drivers/net/forcedeth.c

[50] http://www.meilhaus.de/en/service/download/inhalte/me-46xx-me-foxx-me-jekyll/

[51] http://lxr.linux.no/#linux+v2.6.29/drivers/staging/me4000/me4000.c#L1533

Neither of these bugs is fixed, and the bug is visible in the driver code.[52]

pluto2

Bug #1 In this driver, data structures are not initialized in right order, which may lead to a crash if a spurious interrupt occurs. pluto2 calls `request_irq` before is ready to service interrupts. If the device generates an interrupt immediately, it may invoke `pluto_dma_end`. `pluto_dma_end` calls `dvb_dmx_swfilter_packets`, and this function assumes struct `dvb_demux(demux)->lock` is initialized. However, pluto2 does not initialize this variable until after `request_irq` finishes, via a call to `dvb_dmx_init(dvbdemux)` in `pluto2_probe`.

This bug is not fixed and is visible in the driver.[53]

Bug #2 The driver never calls `tda1004x_release` anywhere. `tda10046_attach` allocates memory with `kmalloc` but never frees it. Pluto2 does not call the release routine (`ops.release`), but does call it in `frontend_init` on an error path. However, it does not call it during proper device shutdown. Moreover, `dvb_frontend_detach` is never called, which might have the effect of calling `ops.release()`. In other words, the problem is that this driver is calling `tda10046_attach` but not releasing it.

Kernel developers appear to have updated the kernel interface involved here, to make this kind of bug less likely, but this driver has not been updated. The new solution is to use `dvb_frontend_detach`. Moreover, the correct way to attach appears to be via `dvb_attach`, using `tda10046_attach` as an argument. Interestingly, the patch that adds `dvb_frontend_detach` says: "Remove buggy `dvb_detach()` macro and replace with unified `dvb_frontend_detach()` call."

This bug is not fixed and is visible in the driver.[54]

---

[52] `http://lxr.linux.no/#linux+v2.6.29/drivers/net/pcnet32.c#L791`

[53] `http://lxr.linux.no/#linux+v2.6.29/drivers/media/dvb/pluto2/pluto2.c`

[54] `http://lxr.linux.no/#linux+v2.6.29/drivers/media/dvb/pluto2/pluto2.c`

C.3 FreeBSD 9

FreeBSD 9 drivers have no corresponding test framework in SymDrive, so the bugs SymDrive found in these drivers were all reported via a kernel panic or warning.

es137x

There appears to be a hardware-dependence bug in `es_pci_detach`. A spurious interrupt could occur, for example, before the call to `bus_teardown_intr` but after the call to `callout_stop`. The interrupt handler then calls `chn_intr`, which attempts to acquire the `c->lock` lock. However, the `pcm_channel(c)` call is invalid at this point because of calls that take place in `pcm_unregister`, which itself is called at the start of `es_pci_detach`. This bug is not fixed.

maestro

This driver has a hardware dependence bug in `agg_attach`. The driver calls `snd_setup_intr` before it is ready to service interrupts. If a spurious interrupt arrives immediately after this call, it will crash. It appears the crash can be fixed by moving this call to immediately before the call to `pcm_setstatus`, which is essentially at the end of the function (`agg_attach`). This bug is not fixed.

C.4 Other Drivers

This section describes bugs found in drivers from various other kernels. We have verified that some of these drivers run on Android-based phones.

akm8975

The driver we tested is part of the Android 3.0. For reference, a similar copy of the driver is available online[55] that exhibits all of the same bugs.

Bug #1 The driver executes `request_irq` before it is necessarily ready to service interrupts. `request_irq` is called in `akm8975_init_client` but `akm->this_client` is not set until after that call. The result is that the interrupt handler can crash because it contains this line: `disable_irq_nosync(akm->this_client->irq);`

[55] `http://os1a.cs.columbia.edu/lxr/source/drivers/misc/akm8975.c`

Bug #2 The driver does not delete a device file created during probe via `device_create_file`, which is essentially a resource leak.

Bug #3 The driver does not appear to stop the work queue during module unload. The driver schedules the workqueue via `schedule_-work(&akm->work)` during invocations of the interrupt handler, but there is no corresponding flush or cancel work call. In addition, the driver could call `enable_irq` while executing the work queue even though the interrupt has already been freed during module unload.

Bug #4 `akm8975_remove` unconditionally calls `free_irq` even though the IRQ may not necessarily have been allocated. This circumstance arises if probe does not complete successfully for some other reason.

mmc31xx

This driver is available via a third-party Android kernel.[56] We verified that this driver runs on a real phone.

Bug #1 This driver includes several resource leaks during driver unload. `device_create_file` is called twice but the file is never destroyed. Similarly, `class_create` is called but the class is never removed.

Bug #2 There are three calls to `device_create_file` (in `mmc31xx_init` and `mmc31xx_probe`), but no calls to remove the files on failure or successful paths.

a1026

This driver is available as part of CyanogenMod v2.6.37.6.[57]

`kmalloc` may be called with a zero-size parameter. Doing this is not a bug,[58] but we would argue that this code is very fragile and should be modified. This issue stems from there being no lower bounds check on an `unsigned int` being read from user-mode in `a1026_bootup_init`. Note that the `kmalloc` call passes `img->img_size` but `img_size` is an `unsigned int` constrained to be less than a

[56] `https://github.com/EnJens/android-tegra-2.6.36-adam/commit/c0fb9fd5fa6527edf9ef7f56f67cc2456000be40#diff-4`

[57] `https://github.com/CyanogenMod/cm-kernel/blob/android-msm-2.6.37/drivers/misc/a1026.c`

[58] `http://lwn.net/Articles/236920/`

constant. Thus, this code passes an invalid kernel pointer to `copy_from_user`. This code does not appear to have any bug resulting from this error though but this approach is dangerous because any future (accidental) dereference operation would result in a kernel crash.

tle62x0

This driver is available as part of the Android 3.0 kernel. A similar copy is available online for reference.[59]

Bug #1 The driver is is missing `device_remove_file` for a file created during probe. `ret = device_create_file(&spi->dev, &dev_attr_status_show);` has no corresponding remove. Loading/unloading/re-loading the module results in an error because the file remains present.

Bug #2 The probe function's error handling code is incorrect. This loop creates some files using the following loop: `for (ptr = 0; ptr < pdata->gpio_count; ptr++)`. This loop might use indices 0 through 3. Suppose the creation of file index 3 fails. The error handling loop reads as follows: `for (; ptr > 0; ptr--)`. This code would free files 4, 3, 2, and 1, which is a mismatch.

C.5 Other Bugs

In addition to the bugs discussed previously, SymDrive found two hardware-specific driver bugs. These bugs involved the driver using the hardware incorrectly. We found them by comparing the hardware operations used by two drivers for equivalent hardware, and noting discrepancies.

We reported one of these bugs to the FreeBSD developers, who subsequently fixed it.[60] We found a similar hardware-specific driver bug in the corresponding Linux driver, but other developers fixed it independently before we could report it.[61] These two bugs are related only in that they are present in corresponding drivers for the same hardware. They are otherwise distinct.

---

[59] `http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/drivers/spi/tle62x0.c`

[60] `http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/dev/re/if_re.c#rev1.95.2.85`

[61] `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=7d03f5a48e4d90854275b06433626243b3b3db17`

To find these bugs, we compared execution traces of the two drivers manually.

[1] Al Danial. CLOC: Count lines of code. http://cloc.sourceforge.net/, 2010.

[2] Albrecht, H. Remote Tea. http://remotetea.sourceforge.net/.

[3] Allchin, J. Windows Vista team blog: Updating a brand-new product, Nov. 2006. http://windowsvistablog.com/blogs/windowsvista/archive/2006/11/17/updating-a-brand-new-product.aspx.

[4] Amani, S., Ryzhyk, L., Donaldson, A., Heiser, G., Legg, A., and Zhu, Y. Static analysis of device drivers: We can do better! In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems* (2011).

[5] Anand, S., Păsăreanu, C. S., and Visser, W. Jpf-se: a symbolic execution extension to java pathfinder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems* (Berlin, Heidelberg, 2007), TACAS'07, Springer-Verlag, pp. 134–138.

[6] Apple Inc. Introduction to I/O kit fundamentals, 2006.

[7] Armand, F. Give a process to your drivers! In *Proceedings of the EurOpen Autumn 1991* (Sept. 1991).

[8] Arthur, S. Fault resilient drivers for Longhorn server. Tech. Rep. WinHec 2004 Presentation DW04012, Microsoft Corporation, May 2004.

[9] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., et al. Thorough static analysis of device drivers. In *EuroSys* (2006).

[10] Ball, T., Levin, V., and Rajamani, S. K. A decade of software model checking with SLAM. In *Commun. of the ACM* (New York, NY, USA, July 2011), vol. 54, pp. 68–76.

[11] Ball, T., and Rajamani, S. K. The SLAM project: Debugging system software via static analysis. In *POPL* (2002).

[12] Ball, T., and Rajamani, S. K. SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[13] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugenbauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM SOSP* (2003).

[14] Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. Specification and verification: The Spec# experience. In *Commun. of the ACM* (New York, NY, USA, June 2011), vol. 54, pp. 81–91.

[15] Bellard, F. QEMU, a fast and portable dynamic translator. In *USENIX ATC* (Berkeley, CA, USA, 2005).

[16] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53* (February 2010), 66–75.

[17] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39–59.

[18] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.

[19] Bissyandé, T. F., Réveillère, L., Lawall, J. L., and Muller, G. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 60–69.

[20] Boyd-Wickizer, S., and Zeldovich, N. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX ATC'10, USENIX Association, pp. 9–9.

[21] Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT—A formal system for testing and debugging programs by symbolic execution. In *Intl. Conf. on Reliable Software* (1975), pp. 234–245.

[22] Brewer, E., Condit, J., McCloskey, B., and Zhou, F. Thirty years is long enough: getting beyond c. In *Proceedings of the Tenth IEEE HOTOS* (2005), pp. 14–14.

[23] Bucur, S., Ureche, V., Zamfir, C., and Candea, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 183–198.

[24] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[25] Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engler, D. EXE: Automatically generating inputs of death. *In ACM Transactions on Information and System Security* (2008).

[26] Cadar, C., and Sen, K. Symbolic execution for software testing: three decades later. *Commun. ACM 56*, 2 (Feb. 2013), 82–90.

[27] 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 *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 45–58.

[28] Catorcini, A., Grunkemeyer, B., and Grunkemeyer, B. CLR inside out: Writing reliabile .NET code. *MSDN Magazine* (Dec. 2007). `http://msdn2.microsoft.com/en-us/magazine/cc163298.aspx`.

[29] Chandrashekaran, P., Conway, C., Joy, J. M., and Rajamani, S. K. Programming asynchronous layers with CLARITY. In *Proceedings of the 15th Annual Symposium on Foundations of Software Engineering* (Sept. 2007).

[30] Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., and Kaashoek, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems* (2011).

[31] Chipounov, V., Georgescu, V., Zamfir, C., and Candea, G. Selective symbolic execution. In *HotDep* (2009).

[32] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS* (New York, NY, USA, 2011), pp. 265–278.

[33] Chipounov, V., Kuznetsov, V., and Candea, G. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 2:1–2:49.

[34] Chubb, P. Get more device drivers out of the kernel! In *Ottawa Linux Symp.* (2004).

[35] Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., and Candea, G. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 5–10.

[36] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. Ccured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM, pp. 232–244.

[37] Conway, C. L., and Edwards, S. A. Ndl: a domain-specific language for device drivers. *SIGPLAN Not. 39*, 7 (2004), 30–36.

[38] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux device drivers*. O'Reilly Media, Incorporated, 2005.

[39] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, Feb. 2005.

[40] Cova, M., Felmetsger, V., Banks, G., and Vigna, G. Static detection of vulnerabilities in x86 executables. In *ACSAC* (Washington, DC, USA, 2006), pp. 269–278.

[41] Coverity. Analysis of the Linux kernel, 2004. Available at `http://www.coverity.com`.

[42] Crameri, O., Bianchini, R., and Zwaenepoel, W. Striking a new balance between program instrumentation and debugging time. In *EuroSys* (2011), pp. 199–214.

[43] Dang, Y., Wu, R., Zhang, H., Zhang, D., and Nobel, P. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 1084–1093.

[44] David, F. M., Chan, E. M., Carlyle, J. C., and Campbell, R. H. CuriOS: Improving reliability through operating system structure. In *OSDI 8* (Dec. 2008).

[45] Dike, J. User-mode linux. `http://user-mode-linux.sourceforge.net`, June 2005.

[46] Division, I. N. Netperf: A network performance benchmark. `http://www.netperf.org`.

[47] Eicken, T. V., chao Chang, C., Czajkowski, G., Hawblitzel, C., Hu, D., and Spoonhower, D. J-Kernel: A capability-based operating system for java. In *Lecture Notes in Computer Science* (1999), pp. 369–393.

[48] Eisler, M. XDR: External data representation standard. RFC 4506, Internet Engineering Task Force, May 2006.

[49] Elson, J. FUSD: A Linux framework for user-space devices, 2004. User manual for FUSD 1.0.

[50] Engler, D., and Ashcraft, K. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003).

[51] Engler, D., Chen, D. Y., and Chou, A. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM SOSP* (Lake Louise, Alberta, Oct. 2001), pp. 57–72.

[52] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001).

[53] Engler, D. R., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI* (2000).

[54] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: software guards for system address spaces. In *Proceedings of the 7th USENIX OSDI* (2006), pp. 6–6.

[55] Erlingsson, Ú., Roeder, T., and Wobber, T. Virtual environments for unreliable extensions. Tech. Rep. MSR-TR-05-82, Microsoft Research, June 2005.

[56] Filiba, T. Rpyc - transparent, symmetric distributed computing. http://rpyc.readthedocs.org/en/latest/, March 2013. Version 3.2.3.

[57] Flanagan, C., and Freund, S. N. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN/SIGSOFT PASTE* (June 2001), pp. 90–96.

[58] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. Safe hardware access with the Xen virtual machine monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure* (2004).

[59] Ganapathi, A., Ganapathi, V., and Patterson, D. Windows xp kernel crash analysis. In *LISA '06: Proceedings of the 20th conference on Large Installation System Administration* (Berkeley, CA, USA, 2006), USENIX Association, pp. 12–12.

[60] Ganapathy, V., Balakrishnan, A., Swift, M. M., and Jha, S. Microdrivers: A new architecture for device drivers. In *Proceedings of the Eleventh IEEE HOTOS* (2007).

[61] Ganapathy, V., Renzelmann, M. J., Balakrishnan, A., Swift, M. M., and Jha, S. The design and implementation of microdrivers. In *ASPLOS 13* (Mar. 2008), pp. 168–178.

[62] 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 *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 103–116.

[63] GNU Project. Rpcgen. http://www.gnu.org/software/libc/.

[64] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *PLDI* (2005), pp. 213–223.

[65] Godefroid, P., Levin, M., Molnar, D., et al. Automated whitebox fuzz testing. In *NDSS* (2008).

[66] Goetz, B. Plugging memory leaks with weak references. http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html, 2005.

[67] Greg Kroah-Hartman. The Linux kernel driver interface. http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt, 2011.

[68] Gunawi, H., Rubio-González, C., Arpaci-Dusseau, A., Arpaci-Dusseau, R., and Liblit, B. EIO: Error handling is occasionally correct. In *6th USENIX FAST* (2008).

[69] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review 40*, 3 (2006), 80–89.

[70] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Failure resilience for device drivers. In *Proceedings of the 2007 IEEE International Conference on Dependable Systems and Networks* (June 2007), pp. 41–50.

[71] Hirzel, M., and Grimm, R. Jeannie: Granting Java native interface developers their wishes. In *Proceedings of the ACM OOPSLA '07* (Oct. 2007), pp. 19–38.

[72] Hsieh, W., Fiuczynski, M., Garrett, C., Savage, S., Becker, D., and Bershad, B. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software* (Feb. 1996).

[73] Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., and Wobber, T. An overview of the singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research (MSR), Oct. 2005.

[74] Hunt, G., Larus, J., M.Abadii, andPP. Barham, M. A., Fähdrich, M., Hawblitzel, C., Hodson, O., andNi. Murphy, S. L., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[75] IBM. Linux test project. http://ltp.sourceforge.net/, May 2010.

[76] Intel Corporation. Based on new microarchitecture, intel itanium processor, doubles performance and boosts resiliency.

http://newsroom.intel.com/community/intel_newsroom/blog/2012/11/08/new-intel-itanium-processor-9500-delivers-breakthrough-capabilities-for-mission-critical-computing, 2012.

[77] Intel Corporation. The evolution of a revolution. http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf, 2012.

[78] Jungo. Windriver cross platform device driver development environment. Tech. rep., Jungo Corporation, Feb. 2002. http://www.jungo.com/windriver.html.

[79] Kadav, A., Renzelmann, M. J., and Swift, M. M. Tolerating hardware device failures in software. In *SOSP* (New York, NY, USA, 2009), pp. 59–72.

[80] Kadav, A., and Swift, M. M. Understanding modern device drivers. *SIGARCH Comput. Archit. News 40*, 1 (Mar. 2012), 87–98.

[81] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[82] Kuznetsov, V., Chipounov, V., and Candea, G. Testing closed-source binary device drivers with DDT. In *USENIX ATC* (2010).

[83] Larson, E., and Austin, T. High coverage detection of input-related security facults. In *USENIX Security* (2003), p. 9.

[84] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[85] Leslie, B., Chubb, P., Fitzroy-Dale, N., Gotz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y., Elphinstone, K., and Heiser, G. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech. 20*, 5 (2005).

[86] LeVasseur, J., Uhlig, V., Stoess, J., and Gotz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX OSDI* (2004).

[87] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th USENIX OSDI* (2004), pp. 289–302.

[88] Love, R. Kernel locking techniques. *Linux Journal* (Aug. 2002). `http://www.linuxjournal.com/article/5833`.

[89] Majumdar, R., and Sen, K. Hybrid concolic testing. In *ICSE* (2007), pp. 416–426.

[90] Menon, A., Schubert, S., and Zwaenepoel, W. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ACM, pp. 301–312.

[91] Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. Devil: An IDL for hardware programming. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation* (Berkeley, CA, USA, 2000), USENIX Association, pp. 2–2.

[92] Microsoft. Windows device testing framework design guide. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff539645%28v=vs.85%29.aspx`, 2011.

[93] Microsoft. Evolving response and the march 2013 bulletin release. `http://blogs.technet.com/b/msrc/archive/2013/03/12/evolving-response-and-the-march-2013-bulletin-release.aspx`, 2013.

[94] Microsoft Corp. PREfast for drivers. `http://www.microsoft.com/whdc/devtools/tools/prefast.mspx`.

[95] Microsoft Corporation. Windows Server 2003 DDK. `http://www.microsoft.com/whdc/DevTools/ddk/default.mspx`, 2003.

[96] Microsoft Corporation. How to use driver verifier to troubleshoot windows drivers. http://support.microsoft.com/kb/q244617/, Jan. 2005. Knowledge Base Article Q244617.

[97] Microsoft Corporation. Architecture of the kernel-mode driver framework. http://www.microsoft.com/whdc/driver/wdf/KMDF-arch.mspx, September 2006.

[98] Microsoft Corporation. Architecture of the user-mode driver framework. http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx, May 2006. Version 0.7.

[99] Microsoft Corporation. Introduction to the WDF user-mode driver framework. http://www.microsoft.com/whdc/driver/wdf/umdf_intro.mspx, May 2006.

[100] Microsoft Corporation. Interoperating with unmanaged code. http://msdn.microsoft.com/en-us/library/sd10k43k(VS.71).aspx, 2008.

[101] Microsoft Corporation. Static Driver Verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, May 2010.

[102] Musuvathi, M., and Engler, D. R. Model checking large network protocol implementations. In *In Proceedings of 1st NSDI* (2004), pp. 155–168.

[103] Necula, G. C., Mcpeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Constr.* (2002).

[104] Nelson, S., and Waskiewicz, P. Virtualization: Writing (and testing) device drivers without hardware. www.linuxplumbersconf.org/2011/ocw/sessions/243. In Linux Plumbers Conference, 2011.

[105] Nethercode, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI* (2007).

[106] Okumura, T., Childers, B. R., and Mosse, D. Running a Java VM inside an operating system kernel. In *Proceedings of the 4th ACM VEE* (Mar. 2008), pp. 161–170.

[107] Padioleau, Y., Lawall, J., Hansen, R. R., and Muller, G. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys* (2008), pp. 247–260.

[108] Padioleau, Y., Lawall, J. L., and Muller, G. Understanding collateral evolution in Linux device drivers. In *EuroSys 2006*, pp. 59–71.

[109] Person, S., Yang, G., Rungta, N., and Khurshid, S. Directed incremental symbolic execution. In *PLDI 2011* (2011), pp. 504–515.

[110] Project UDI. Uniform Driver Interface: Introduction to UDI version 1.0. `http://udi.certek.cc/Docs/pdf/UDI_tech_white_paper.pdf`, Aug. 1999.

[111] Păsăreanu et al, C. S. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA* (2008), pp. 15–26.

[112] Renzelmann, M. J., Kadav, A., and Swift, M. M. Symdrive: Testing drivers without devices. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 279–292.

[113] Renzelmann, M. J., and Swift, M. M. Decaf: Moving device drivers to a modern language. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 14–14.

[114] Ritchie, D. The Unix tree: The 'nsys' kernel, Jan. 1999. `http://minnie.tuhs.org/UnixTree/Nsys/`.

[115] Ritchie, D. M. The development of the c language. *SIGPLAN Not. 28*, 3 (Mar. 1993), 201–208.

[116] Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. Dingo: Taming device drivers. In *Proceedings of the 2009 EuroSys Conference* (Apr. 2009).

[117] Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., and Heiser, G. Automatic device driver synthesis with termite. In *SOSP '09* (2009), pp. 73–86.

[118] Ryzhyk, L., Kuz, I., and Heiser, G. Formalising device driver interfaces. In *Workshop on Programming Languages and Systems* (Oct. 2007).

[119] Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13* (2005), pp. 263–272.

[120] Spear, M., Roeder, T., Hodson, O., Hunt, G., and Levi, S. Solving the starting problem: Device drivers as self-describing artifacts. In *Proceedings of the 2006 EuroSys Conference* (Apr. 2006), pp. 45–58.

[121] Stallings, W. *Operating Systems: Internals and Design Principles*, 6th ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[122] Sun Microsystems. UNIX programmer's supplementary documents: rpcgen programming guide. `http://docs.freebsd.org/44doc/psd/22.rpcgen/paper.pdf`.

[123] Susskraut, M., and Fetzer, C. Automatically finding and patching bad error handling. In *DSN* (2006), pp. 13–22.

[124] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM SOSP* (Bolton Landing, New York, Oct. 2003), pp. 207–222.

[125] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems 23*, 1 (Feb. 2005).

[126] Torvalds, L. Linux kernel source tree. `http://www.kernel.org`.

[127] Torvalds, L. UIO: Linux patch for user-mode I/O, July 2007.

[128] von Eicken, T., Chang, C.-C., Czajkowski, G., Hawblitzel, C., Hu, D., and Spoonhower, D. J-Kernel: a capability-based operating system for Java. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, vol. 1603 of *LNCS*. Springer-Verlag, 1999, pp. 369–393.

[129] Williams, D., Reynolds, P., Walsh, K., Sirer, E. G., and Schneider, F. B. Device driver safety through a reference validation mechanism. In *OSDI* (2008).

[130] Witchel, E., Rhee, J., and Asanovic, K. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM SOSP* (2005).

[131] Xie, T., Marinov, D., Schulte, W., and Notkin, D. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS* (2005), pp. 365–381.

[132] Yamauchi, H., and Wolczko, M. Writing solaris device drivers in java. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems* (New York, NY, USA, 2006), ACM, p. 3.

[133] Yamauchi, H., and Wolczko, M. Writing Solaris device drivers in Java. Tech. Rep. TR-2006-156, Sun Microsystems, Apr. 2006.

[134] Yang, J., Sar, C., Twohey, P., Cadar, C., and Engler, D. Automatically generating malicious disks using symbolic execution. In *IEEE Symp. on Security and Privacy* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 243–257.

[135] Young, M., Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., and Tevanian, A. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference* (1986).

[136] Zamfir, C., and Candea, G. Execution synthesis: A technique for automated software debugging. In *EuroSys* (2010), pp. 321–334.

[137] 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 *Proceedings of the 7th USENIX OSDI* (2006).