Understanding Modern Device Drivers

Asim Kadav and Michael M. Swift
Computer Sciences Department, University of Wisconsin-Madison
{kadav, swift} @cs.wisc.edu

Abstract

Device drivers are the single largest contributor to operating-system kernel code with over 5 million lines of code in the Linux kernel, and cause significant complexity, bugs and development costs. Recent years have seen a flurry of research aimed at improving the reliability and simplifying the development of drivers. However, little is known about what constitutes this huge body of code beyond the small set of drivers used for research.

In this paper, we study the source code of Linux drivers to understand what drivers actually do, how current research applies to them and what opportunities exist for future research. We determine whether assumptions made by driver research, such as that all drivers belong to a class, are indeed true. We also analyze driver code and abstractions to determine whether drivers can benefit from code re-organization or hardware trends. We develop a set of static-analysis tools to analyze driver code across various axes. Broadly, our study looks at three aspects of driver code (i) what are the characteristics of driver code functionality and how applicable is driver research to *all* drivers, (ii) how do drivers interact with the kernel, devices, and buses, and (iii) are there similarities that can be abstracted into libraries to reduce driver size and complexity?

We find that many assumptions made by driver research do not apply to all drivers. At least 44% of drivers have code that is not captured by a class definition, 28% of drivers support more than one device per driver, and 15% of drivers do significant computation over data. From the driver interactions study, we find that the USB bus offers an efficient bus interface with significant standardized code and coarse-grained access, ideal for executing drivers in isolation. We also find that drivers for different buses and classes have widely varying levels of device interaction, which indicates that the cost of isolation will vary by class. Finally, from our driver similarity study, we find 8% of all driver code is substantially similar to code elsewhere and may be removed with new abstractions or libraries.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Measurement, Design

Keywords Device Drivers, Measurement

ASPLOS'12, March 3–7, 2012, London, England, UK.

1. Introduction

Modern computer systems are communicating with an increasing number of devices, each of which requires a driver. For example, a modern desktop PC may have tens of devices, including keyboard, mouse, display, storage, and USB controllers. Device drivers constitute 70% of the Linux code base [32], and likely are a greater fraction of the code written for the Windows kernel, which supports many more devices. Several studies have shown that drivers are the dominant cause of OS crashes in desktop PCs [14, 28]. As a result, there has been a recent surge of interest in techniques to tolerate faults in drivers [12, 38, 39, 47], to improve the quality of driver code [20]; and in creating new driver architectures that improve reliability and security [4, 15, 21, 23, 24, 31, 44].

However, most research on device drivers focuses on a small subset of devices, typically a network card, sound card, and storage device, all using the PCI bus. These are but a small subset of all drivers, and results from these devices may not generalize to the full set of drivers. For example, many devices for consumer PCs are connected over USB. Similarly, the devices studied are fairly mature and have standardized interfaces, but many other devices may have significant functionality differences.

Thus, it is important to study all drivers to review how the driver research solutions being developed are applicable to all classes of drivers. In addition, a better understanding of driver code can lead to new abstractions and interfaces that can reduce driver complexity and improve reliability.

This paper presents a comprehensive study of all the drivers in the Linux kernel in order to broadly characterize their code. We focus on (i) what driver code does, including where driver development work is concentrated, (ii) the interaction of driver code with devices, buses, and the kernel, and (iii) new opportunities for abstracting driver functionality into common libraries or subsystems. We use two static analysis tools to analyze driver code. To understand properties of driver code, we developed DrMiner, which performs data-flow analyses to detect properties of drivers at the granularity of functions. We also developed the DrComp tool, which uses geometric shape analysis [3] to detect similar code across drivers. DrComp maps code to points in coordinate space based on the structure of individual driver functions, and similar functions are at nearby coordinates.

The contributions of this paper are as follows:

- First, we analyze what driver code does in order to verify common assumptions about driver code made by driver research. We show that while these assumptions hold for most drivers, there are a significant number of drivers that violate these assumptions. We also find that several rapidly growing driver classes are not being addressed by driver research.
- Second, we study driver *interactions* with the kernel and devices, to find how existing driver architecture can adapt to a world of multicore processors, devices with high-power processors and virtualized I/O. We find that drivers vary widely

by class, and that USB drivers are more efficient in supporting multiple chipsets than PCI drivers. Furthermore, we find that XenBus drivers may provide a path to executing drivers outside the kernel and potentially on the device itself.

- Third, we study driver code *contents* to find opportunities to reduce or simplify driver code. We develop new analysis tools for detecting similar code structures and their types that detect over 8% of Linux driver code is very similar to other driver code, and offer insights on how this code can be reduced.

In the remainder of this paper, we first discuss device driver background and develop a taxonomy of drivers. We then present the three broad classes of results on driver behavior in Sections 3 and 4. In Section 5 we present results showing the extent of repeated code in drivers. Section 6 discusses our findings.

2. Background

A device driver is a software component that provides an interface between the OS and a hardware device. The driver configures and manages the device, and converts requests from the kernel into requests to the hardware. Drivers rely on three interfaces: (i) the interface between the driver and the kernel, for communicating requests and accessing OS services; (ii) the interface between the driver and the device, for executing operations; and (iii) the interface between the driver and the bus, for managing communication with the device.

2.1 Driver/Device Taxonomy

The core operating system kernel interacts with device drivers through a set of interfaces that abstract the fundamental nature of the device. In Linux, the three categories of drivers are *character* drivers, which are byte-stream oriented; *block* drivers, which support random-access to blocks; and *network* drivers, which support streams of packets. Below these top-level interfaces, support libraries provide common interfaces for many other families of devices, such as keyboards and mice within character drivers.

In order to understand the kinds of drivers Linux supports, we begin by taxonomizing drivers according to their interfaces. We consider a single driver as a module of code that can be compiled independently of other code. Hence, a single driver can span multiple files. We consider all device drivers, bus drivers and virtual drivers that constitute the driver directories (`/sound` and `/drivers`) in the Linux 2.6.37.6 kernel, dated April, 2011. We perform our analyses on all drivers that compile on the x86 platform, using the kernel build option to compile all drivers. Overall, we consider 3,217 distinct drivers. While there are a significant number of Linux drivers that are distributed separately from the kernel, we do not consider them for this work.

We detect the class of a driver not by the location of its code, but by the interfaces it registers: *e.g.*, `register_netdev` indicates a driver is a network device. We further classify the classes into sub-categories to understand the range of actual device types supported by them through manual classification, using the device operations they register, the device behavior and their location. While Linux organizes related drivers in directories, this taxonomy is not the same as the Linux directory organization: network drivers are split under *drivers/net*, *drivers/atm* and other directories. However, block drivers are split by their interfaces under *drivers/scsi*, *drivers/ide* and other directories

Figure 1 shows the hierarchy of drivers in Linux according to their interfaces, starting from basic driver types i.e. char, block and net. We identify 72 unique classes of drivers. The majority (52%) of driver code is in character drivers, spread across 41 classes. Network drivers account 25% of driver code, but have only 6 classes. In contrast to the rich diversity of Figure 1, Table 1 lists the

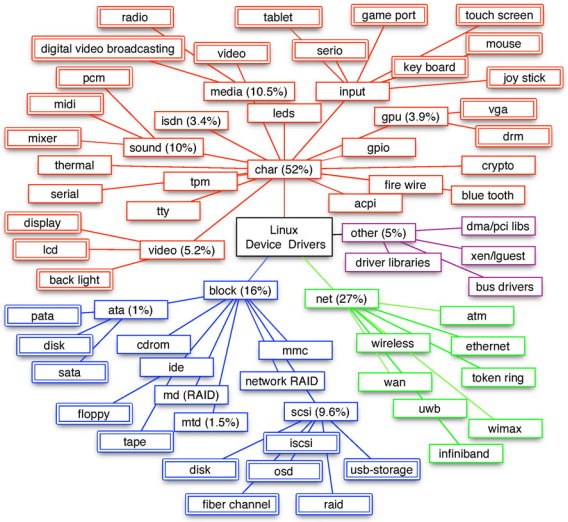

Figure 1. The Linux driver taxonomy in terms of basic driver classes. The size (in percentage of lines of code) is mentioned for 5 biggest classes. Not all driver classes are mentioned.

driver types used in research. Most driver research (static-analysis tools excepted) neglects the heavy tail of character devices. For example, video and GPU drivers contribute significantly towards driver code (almost 9%) due to complex devices with instruction sets that change each generation, but these devices are largely ignored by driver research due to their complexity.

We also looked at low-level buses, which provide connectivity to devices and discover the devices attached to a system. The majority of devices are either PCI (36% of all device identifiers) or USB (35%), while other buses support far fewer: I2C represents 4% of devices, PCMCIA is 3% and HID is 2.5% (mostly USB devices). The remaining devices were supported by less popular or legacy buses such as ISA or platform devices. We also found that 8% of devices perform low-level I/O without using a bus, interconnect, or support virtual devices. Higher-level protocols such as SCSI (8.5%) and IDE (2%) use one of these underlying low-level buses such as PCI and USB. While PCI drivers still constitute the greatest fraction of drivers, the number of devices supported by USB is similar to PCI. Hence, driver research should validate their performance and reliability claims on USB devices as well.

2.2 Driver Research Assumptions

Most research makes some simplifying assumptions about the problem being solved, and driver research is no different. For example, Shadow Drivers [38] assume that all drivers are members of a class and there are no unique interfaces to a driver. Similarly, the Termite driver-synthesis system assumes that drivers are state machines and perform no computations [33]. Table 2 lists the assumptions made by recent research into device drivers.

We separate these assumptions into two categories: (i) *interactions* refers to assumptions about how drivers interact with the kernel, and (ii) *architecture* refers assumptions about the role of the driver: is it a conduit for data, or does it provide more substantial processing or services? Interaction assumptions relate to how the kernel and driver interact. For example, systems that interpose on driver/device communication, such as Nooks [39], typically assume that communication occurs over procedure calls and not shared memory. Nooks' isolation mechanism will not work otherwise. Similarly, Shadow Drivers assume that the complete state of the device is available in the driver by capturing kernel/driver in-

Improvement type	System	Driver classes tested	Drivers tested
New functionality	Shadow driver migration [19]	net	1
	RevNIC [6]	net	4
Reliability (H/W Protection)	CuriOS [9]	serial port, NOR flash	2
	Nooks [39]	net, sound	6
	Palladium [7]	custom packet filter	1
	Xen [12]	net, sound	2
Reliability (Isolation)	BGI [5]	net, sound, serial, ramdisk, libs	16
	Shadow Drivers [38]	net, sound, IDE	13
	XFI [40]	ram disk, dummy	2
Specification	Devil [25]	scsi, video	2
	Dingo [32]	net	2
	Laddie [45]	net, UART, rtc	3
	Nexus [44]	net, sound, USB-mouse, USB-storage	4
	Termite [33]	net, mmc	2
Static analysis tools	Carburizer [18]	All/net	All/3
	Cocinelle [29]	Various	All
	SDV [2]	basic ports, storage, USB, 1394-interface, mouse, keyboard, PCI battery	126
Type safety	Decaf Drivers [31]	net, sound, USB controller, mouse	5
	Safedrive [47]	net, USB, sound, video	6
	Singularity [37]	Disk	1
User-level device drivers	Micro-drivers [15]	net, sound, USB controller	4
	SUD [4]	net, wireless, sound, USB controllers, USB	6/1
	User-level drivers [21]	net, disk (ata)	2

Table 1. Research projects on drivers, the improvement type, and the number and class of drivers used to support the claim end to end. Few projects test all driver interfaces thoroughly. Static analysis tools that do not require driver modifications are available to check many more drivers. Also, some solutions, like Carburizer [18], and SUD [4] support the performance claims on fewer drivers.

Driver interactions
Class membership: Drivers belong to common set of classes, and the class completely determines their behavior.
Procedure calls: Drivers always communicate with the kernel through procedure calls.
Driver state: The state of the device is completely captured by the driver.
Device state: Drivers may assume that devices are in the correct state.
Driver architecture
I/O: Driver code functionality is only dedicated to converting requests from the kernel to the device.
Chipsets: Drivers typically support one or a few device chipsets.
CPU Bound: Drivers do little processing and mostly act as a library for binding different interfaces together.
Event-driven: Drivers execute only in response to kernel and device requests, and to not have their own threads.

Table 2. Common assumptions made in device driver research.

teractions [38]. However, network cards that do TCP-offload may have significant protocol state that is only available in the device, and cannot be captured by monitoring the kernel/driver interface.

Several recent projects assume that drivers support a single chipset, such as efforts at synthesizing drivers from a formal specification [33]. However, many drivers support more than one chipset. Hence, synthesizing the replacement for a single driver may require many more drivers. Similarly, enforcing safety properties for specific devices [44] may be cumbersome if many chipsets must be supported for each driver. Other efforts at reverse engineering drivers [6] similarly may be complicated by the support of many chipsets with different hardware interfaces. Furthermore, these synthesis and verification systems assume that devices always behave correctly, and their drivers may fail unpredictably with faulty hardware.

Another assumption made by driver research is that drivers are largely a conduit for communicating data and for signaling the device, and that they perform little processing. Neither RevNIC [6]) nor Termite [33] support data processing with the driver, because it is too complex to model as a simple state machine.

While these assumptions all hold true for many drivers, this research seeks to quantify their real generality. If these assumptions are true for all drivers, then these research ideas have broad applicability. If not, then new research is needed to address the outliers.

3. What Do Drivers Do?

Device drivers are commonly assumed to primarily perform I/O. A standard undergraduate OS textbook states:

> A device driver can be thought of a translator. Its input consists of high-level commands such as "retrieve block 123." Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system."
>
> *Operating Systems Concepts* [36]

However, this passage describes the functions of only the small portion of driver code that which actually performs I/O. Past work revealed that the bulk of driver code is dedicated to initialization and configuration for a sample of network, SCSI and sound drivers [15].

We seek to develop a comprehensive understanding of what driver code does: what are the tasks drivers perform, what are the interfaces they use, and how much code does this all take. The goal of this study is to verify the driver assumptions described in the previous section, and to identify major driver functions that could benefit from additional research.

3.1 Methodology

To study the driver code, we developed the *DrMiner* static analysis tool using CIL [27] to detect code properties in individual drivers. DrMiner takes as input unmodified drivers and a list of driver data-structure types and driver entry points. As drivers only execute when invoked from the kernel, these entry points allow us to determine the purpose of particular driver functions. For example, we find the different devices and chipsets supported by analyzing the `device_id` structures registered (*e.g.*, `pci_device_id`, `acpi_device_id` etc.) with the kernel. We also identify the driver entry points from the driver structure registered (*e.g.*, `pci_driver`, `pnp_device`) and the device operations structure registered (*e.g.*, `net_device`). DrMiner analyzes the function pointers registered by the driver to determine the functionality of each driver. We then construct a control-flow graph of the driver that allows us to determine all the functions reachable through each entry point, including through function pointers.

We use a *tagging* approach to labeling driver code: DrMiner tags a function with the label of each entry point from which it is reachable. Thus, a function called only during initialization will be labeled initialization only, while code common to initialization and shutdown will receive both labels. In addition, DrMiner identifies specific code features, such as loops indicating computation.

Figure 2. The percentage of driver code accessed during different driver activities across driver classes.

We run these analyses over the entire Linux driver source and store the output in a SQL database. The database stores information about each driver as well as each function in the driver. The information about the driver consists of name, path, size, class, number of chipsets, module parameters, and interfaces registered with the kernel. The information about each driver function consists of function name, size, labels, resources allocated (memory, locks etc.), and how it interacts with the kernel and the device. From the database, determining the amount of code dedicated to any function is a simple query. In our results, we present data for about 25 classes with the most code.

3.2 What is the function breakdown of driver code?

Drivers vary widely in how much code they use for different purposes; a simple driver for a single chipset may devote most of its code to processing I/O requests and have a simple initialization routine. In contrast, a complex driver supporting dozens of chipsets may have more code devoted to initialization and error handling than to request handling.

Figure 2 shows the breakdown of driver code across driver classes. The figure shows the fraction of driver code invoked during driver initialization, cleanup, ioctl processing, configuration, power management, error handling, `/proc` and `/sys` handling, and most importantly, core I/O request handling (*e.g.*, sending packet for network devices, or playing audio for sound card) and interrupt handling across different driver classes.

The largest contributors to driver code are initialization and cleanup, comprising almost 36% of driver code on average, error handling (5%), configuration (15%), power management (7.4%) and ioctl handling (6.2%). On average, only 23.3% of the code in a driver is dedicated to request handling and interrupts.

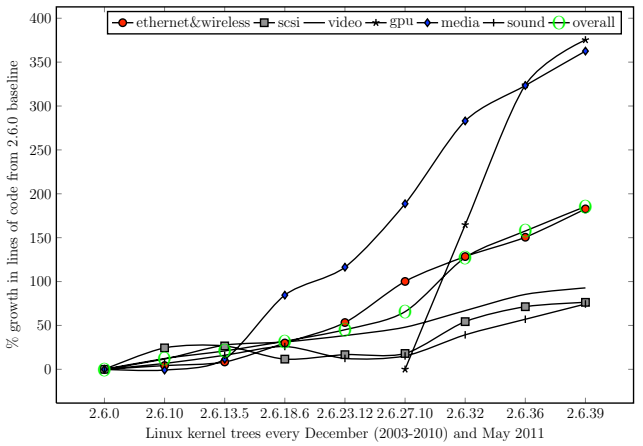

Figure 3. The change in driver code in terms of LOC across different driver classes between the Linux 2.6.0 and Linux 2.6.39 kernel.

Implications: These results indicate that efforts at reducing the complexity of drivers should not only focus on request handling, which accounts for only one fourth of the total code, but on better mechanisms for initialization and configuration. For example, as devices become increasingly virtualization aware, quick ways to initialize or reset are critical for important virtualization features such as re-assignment of devices and live migration [19]. Drivers contain significant configuration code (15%), specifically in network (31%) and video (27%) drivers. As devices continue to become more complex, driver and OS research should look at efficient and organized ways of managing device configuration [35].

3.3 Where is the driver code changing?

Over time, the focus of driver development shifts as new device classes become popular. We compared the breakdown of driver code between the 2.6.0 and 2.6.39 for new source lines of code added annually to different driver classes. We obtain the source lines of code across different classes in 9 intermediate revisions (every December since 2.6.0) using sloccount [43].

Figure 3 shows the growth in driver across successive years from the 2.6.0 baseline for 8 major driver classes. Overall, driver code has increased by 185% over the last eight years. We identify three specific trends in this growth. First, there is additional code for new hardware. This code includes wimax, GPU, media, input devices and virtualization drivers. Second, there is increasing support for certain class of devices, including network (driven by wireless), media, GPU and SCSI. From 2.6.13 to 2.6.18, the devices supported by a vendor (QLogic) increased significantly. Since, they were very large multi-file SCSI drivers, the drivers where coalesced to a single driver, reducing the size of SCSI drivers in the driver tree. In Section 5, we investigate whether there are opportunities to reduce driver code in the existing code base. Third, there is minor code refactoring. For example, periodically, driver code is moved away from the driver or bus library code into the respective classes where they belong. For example, drivers from the i2c bus directory were moved to misc directory.

Implications: While Ethernet and sound, the common driver classes for research, are important, research should look further into other rapidly changing drivers, such as media, GPU and wireless drivers.

```
drivers/ide/ide-cd.c:
static int cdrom_read_tocentry(...) {
  // Read table of contents data
  for (i = 0; i <= ntracks; i++) {
    if (drive->atapi_flags &
        IDE_AFLAG_TOCADDR_AS_BCD) {
      if (drive->atapi_flags &
          IDE_AFLAG_TOCTRACKS_AS_BCD)
        toc->ent[i].track =
          bcd2bin(toc->ent[i].track);
      msf_from_bcd(&toc->ent[i].addr.msf);}
    toc->ent[i].addr.lba =
      msf_to_lba(toc->ent[i].addr.msf.minute,
                 toc->ent[i].addr.msf.second,
                 toc->ent[i].addr.msf.frame);
  }
}
```

Figure 4. The IDE CD-ROM driver processes table-of-contents entries into a native format.

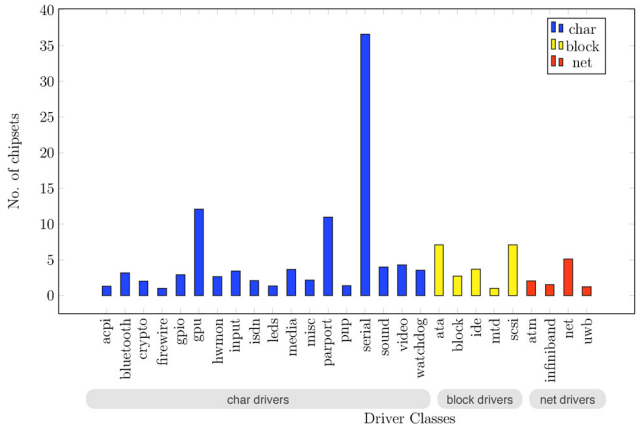

Figure 5. The average number of chipsets supported by drivers in each class.

3.4 Do drivers belong to classes?

Many driver research projects assume that drivers belong to a class. For example, Shadow Drivers [38] must be coded with the semantics of all calls into the driver so it can replay them during recovery. However, many drivers support proprietary extensions to the class interface. In Linux drivers, these manifest as private `ioctl` options, `/proc` or `/sys` entries, and as load-time parameters. If a driver has one of these features, it may have additional behaviors not captured by the class.

We use DrMiner to identify drivers that have behavior outside the class by looking for load-time parameters and code to register `/proc` or `/sys` entries. We do not identify unique `ioctl` options. Overall, we find that most driver classes have substantial amounts of device-specific functionality. Code supporting `/proc` and `/sys` is present in 16.3% of drivers. Also, 36% of drivers have load-time parameters to control their behavior and configure options not available through the class interface. Overall, 44% of drivers use atleast one of the two non-class features. Additionally, `ioctl` code comprises 6.2% of driver code, can also cause non-class behavior.

As an example of how these class extensions are used, the e1000 gigabit network driver has 15 load-time parameters that allow control over interrupt processing and transmit/receive ring sizing, and interrupt throttling rate. This feature is not part of any standard network device interface and is instead specific to this device. Similarly, the i915 DRM GPU driver supports load parameters for down-clocking the GPU, altering graphic responsibilities from X.org to the kernel, and power saving. These parameters change the code path of the driver during initialization as well as during regular driver operations. While the introduction of these parameters does not affect the isolation properties of the reliability solutions, as the interfaces for setting and retrieving these options are standard, it limits the ability to restart and restore the driver to a previous state since the semantics of these options are not standardized.

Implications: While most driver functionality falls into the class behavior, many drivers have significant extensions that do not. Attempts to recover driver state based solely on the class interface [38] or to synthesize drivers from common descriptions of the class [6, 33] may not work for a substantial number of drivers. Thus, future research should explicitly consider how to accommodate unique behaviors efficiently.

3.5 Do drivers do significant processing?

As devices become more powerful and feature processors of their own, it is often assumed that drivers perform little processing and simply shuttle data between the OS and the device. However, if drivers require substantial CPU processing, for example to compute parity for RAID, checksums for networking, or display data for video drivers, then processing power must be reserved. Furthermore, in a virtualized setting, heavy I/O from one guest VM could substantially reduce CPU availability for other guest VMs.

DrMiner detects processing in drivers by looking for loops that (i) do no I/O, (ii) do not interact with the kernel, and (iii) are on core data paths, such as sending/receiving packets or reading/writing data. This ensures that polling loops, common in many drivers, are not identified as performing processing.

We find that 15% of drivers have at least one function that performs processing, and that processing occurs in 1% of all driver functions. An even higher fraction (28%) of sound and network drivers do processing. Wireless drivers, such as ATH, perform processing to interpolate power levels of the device under different frequencies and other conditions. Many network drivers provide the option of computing checksums on the outgoing/incoming packets. Finally, even CD-ROM drivers, which largely read data off the device, do computation to analyze the table of content information for CD-ROMs, as shown in Figure 4.

Implications: A substantial fraction of drivers do some form of data processing. Thus, efforts to generate driver code automatically must include mechanisms for data processing, not just converting requests from the OS into requests to the device. Furthermore, virtualized systems should account for the CPU time spent processing data when this processing is performed on behalf of a guest VM. These results also point to new opportunities for driver and device design: given the low cost of embedded processors, can all the computation be offloaded to the device, and is there a performance or power benefit to doing so?

3.6 How many device chipsets does a single driver support?

Several driver research projects require or generate code for a specific device chipset. For example, Nexus requires a safety specification that is unique to each device interface [44]. If a driver supports only a single device, this requirement may not impose much burden. However, if a driver supports many devices, each with a different interface or behavior, then many specifications are needed to fully protect a driver.

We measure the number of chipsets or hardware packagings supported by each Linux driver by counting the number of PCI, USB or other bus device IDs (*i.e.*, i2c, ieee1394) that the driver recognizes. These structures are used across buses to identify (and match) different devices or packagings that are supported by the

```
static int __devinit cy_pci_probe(...)
{
 if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo) {
      ...
 if (pci_resource_flags(pdev,2)&IORESOURCE_IO){
       ..
 if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo ||
      device_id == PCI_DEVICE_ID_CYCLOM_Y_Hi) {
         ..
}else if (device_id==PCI_DEVICE_ID_CYCLOM_Z_Hi)
 ....
 if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo ||
     device_id == PCI_DEVICE_ID_CYCLOM_Y_Hi) {
  switch (plx_ver) {
   case PLX_9050:
   ...
  default:    /* Old boards, use PLX_9060 */
   ...
}
```

Figure 6. The cyclades character drivers supports eight chipsets that behaves differently at each phase of execution. This makes driver code space efficient but extremely complex to understand.

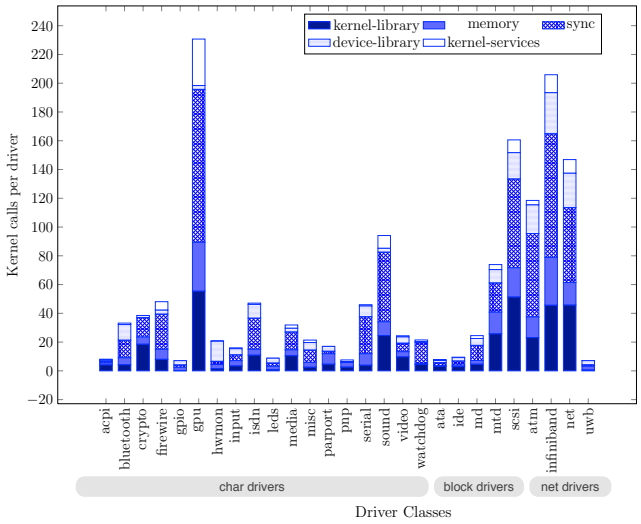

Figure 7. The average kernel library, memory, synchronization, kernel device library, and kernel services library calls per driver (bottom to top in figure) for all entry points.

driver. Figure 5 shows the average number of chipsets supported by each driver in each driver class. While most drivers support only a few different devices, serial drivers support almost 36 chipsets on average, and network drivers average 5. The Radeon DRM driver supports over 400 chipsets, although many of these may indicate different packagings of the same internal chipset. Generic USB drivers such as usb-storage and usb-audio support over 200 chipsets each, and the usb-serial driver supports more than 500 chipsets. While not every chipset requires different treatment by the driver, many do. For example, the `3c59x` 100-megabit Ethernet driver supports 37 chipsets, 17 sets of features that vary between chipsets, and two complete implementations of the core send/receive functionality. Overall, we find that 28% of drivers support more than one chipset and these drivers support 83% of the total devices.

In order to measure the effects of number of chipsets on driver code size, we measured the least-square correlation coefficient between the number of chipsets support by a driver and the amount of code in the driver and found them to be weakly correlated (0.25), indicating that drivers supporting more chipsets were on average larger than those that did not. However, this does not completely explain the amount of initialization code, as the correlation between the number of chipsets and the percentage of initialization code was 0.07, indicating that the additional chipsets increased the amount of code throughout the driver.

Implications: These results indicate that Linux drivers support multiple chipsets per driver and are relatively efficient, supporting 14,070 devices with 3,217 device and bus drivers, for an average of approximately 400 lines of code per device. Any system that generates unique drivers for every chipset or requires per-chipset manual specification may lead to a great expansion in driver code and complexity. Furthermore, there is substantial complexity in supporting multiple chipsets, as seen in Figure 6, so better programming methodologies, such as object-oriented programming [31] and automatic interface generation, similar to Devil [25], should be investigated.

3.7 Discussion

The results in this section indicate that while common assumptions about drivers are generally true, given the wide diversity of drivers, one cannot assume they always hold. Specifically, many drivers contain substantial amounts of code that make some of the existing research such as automatic generation of drivers difficult, due to code unique to that driver and not part of the class, code that processes data, and code for many chip sets.

4. Driver Interactions

The preceding section focused on the *function* of driver code, and here we turn to the *interactions* of driver code: how do drivers use the kernel, and how do drivers communicate with devices? We see three reasons to study these interactions. First, extra processing power on devices or extra cores on the host CPU provide an opportunity to redesign the driver architecture for improved reliability and performance. For example, it may be possible to move many driver functions out of the kernel and onto the device itself. Or, in virtualized systems, driver functionality may execute on a different core in a different virtual machine. Second, much of the difficulty in moving drivers between operating systems comes from the driver/kernel interface, so investigating what drivers request of the kernel can aid in designing more portable drivers. Third, the cost of isolation and reliability are proportional to the size of the interface and the frequency of interactions, so understanding the interface can lead to more efficient fault-tolerance mechanisms.

We examine the patterns of interaction between the driver, the kernel and the device, with a focus on (i) which kernel resources drivers consume, (ii) how and when drivers interact with devices, (iii) the differences in driver structure across different I/O buses, and (iv) the threading/synchronization model used by driver code.

4.1 Methodology

We apply the DrMiner tool from Section 3 to perform this analysis. However, rather than propagating labels down the call graph from entry points to leaf functions, here we start at the bottom with kernel and device interactions. Using a list of known kernel functions, bus functions, and I/O functions, we label driver functions according to the services or I/O they invoke. Additionally, we compute the number of invocations of bus, device and kernel invocations for each function in a driver. These call counts are also propagated to determine how many such static calls could be invoked when a particular driver entry point is invoked.

4.2 Driver/Kernel Interaction

Drivers vary widely in how they use kernel resources, such as memory, locks, and timers. Here, we investigate how drivers use these resources. We classify all kernel functions into one of five categories:

1. Kernel library (*e.g.*, generic support routines such as reporting functions,[1] timers, string manipulation, checksums, standard data structures)
2. Memory management (*e.g.*, allocation)
3. Synchronization (*e.g.*, locks)
4. Device library (*e.g.*, subsystem libraries supporting a class of device and other I/O related functions)
5. Kernel services (*e.g.*, access to other subsystems including files, memory, scheduling)

The first three are generic library routines that have little interaction with other kernel services, and could be re-implemented in other execution contexts. The fourth category, device library, provides I/O routines supporting the driver but does not rely other kernel services, and is very OS dependent. The final category provides access to other kernel subsystems, and is also OS dependent.

Figure 7 shows, for each class of drivers, the total number of function calls made by drivers in every class. The results demonstrate several interesting features of drivers. First, the majority of kernel invocations are for kernel library routines, memory management and synchronization. These functions are primarily *local* to a driver, in that they do not require interaction with other kernel services. Thus, a driver executing in a separate execution context, such as in user mode or a separate virtual machine, need not call into the kernel for these services. There are very few calls into kernel services, as drivers rarely interact with the rest of the kernel.

The number of calls into device-library code varies widely across different classes and illustrates the abstraction level of the devices: those with richer library support, such as network and SCSI drivers, have a substantial number of calls into device libraries, while drivers with less library support, such as GPU drivers, primarily invoke more generic kernel routines.

Finally, a number of drivers make very little use of kernel services, such as ATA, IDE, ACPI, and UWB drivers. This approach demonstrates another method for abstracting driver functionality when there is little variation across drivers: rather than having a driver that invokes support library routines, these drivers are themselves a small set of device-specific routines called from a much larger common driver. This design is termed a "miniport" driver in Windows. Thus, these drivers benefit from a common implementation of most driver functionality, and only the code differences are implemented in the device-specific code. These drivers are often quite small and have little code that is not device specific.

These results demonstrate a variety of interaction styles between drivers and the kernel: drivers with little supporting infrastructure demonstrate frequent interactions with the kernel for access to kernel services but few calls to device support code. Drivers with a high level of abstraction demonstrate few calls to the kernel over all. Drivers with a support library demonstrate frequent calls to kernel generic routines as well as calls to device support routines.

Implications: Drivers with few calls into device libraries may have low levels of abstraction, and thus are candidates for extracting common functionality. Similarly, drivers with many kernel interactions and device library interaction may benefit from converting to a layered on "miniport" architecture, where more driver functionality is extracted into a common library.

[1] We leave out `printk` to avoid skewing the numbers from calls to it.

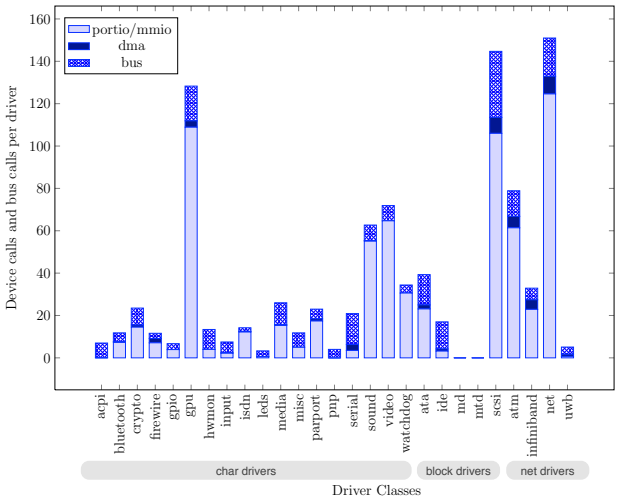

Figure 8. The device interaction pattern representing port I/O, memory mapped I/O, bus resources (bottom to top) invoked via all driver entry points.

Furthermore, a large fraction of driver/kernel interactions are for generic routines (memory, synchronization, libraries) that do not involve other kernel services. Thus, they could be implemented by a runtime environment local to the driver. For example, a driver executing in a separate virtual machine or on the device itself can make use of its local OS for these routines, and drivers in user space can similarly invoke user-space versions of these routines, such as the UML environment in SUD [4].

4.3 Driver/Device Interaction

We next look at interaction between the driver and the device. We analyzed all functions in all drivers, and if a function does I/O itself, or calls a function that results in an I/O, we label it as *perform I/O*. We categorize driver/device interactions around the type of interaction: access to memory-mapped I/O (MMIO) regions or x86 I/O ports (port IO) are labeled *mmio/portio*, *DMA* is DMA access, either initiated by the driver or enabled by the driver creating a DMA mapping for a memory region, and calls to a bus, such as USB or PCI (*bus*). We could not determine statically when a device initiates DMA, although we do count calls to map a page for future DMA (*e.g.*, `pci_map_single`) as a DMA action. Our analysis can detect memory mapped I/O through accessor routines such as `read/writeX` family, `ioread/iowrite` family of routines and port I/O using the `in/outX` family. DrMiner cannot identify direct dereferences of pointers into memory-mapped address ranges. However, direct dereference of I/O addresses is strongly discouraged and most non-conforming drivers have been converted to use accessor routines instead. We also note that all I/O routines on x86 eventually map down to either port or MMIO. Here, though, we focus on the I/O abstractions used by the driver.

Figure 8 shows, for each class of device, the number of device interactions in the entire driver. The results demonstrate that driver classes vary widely in their use of different I/O mechanisms. IDE and ATA drivers, both block devices, show very different patterns of interaction: IDE drivers do very little port or MMIO, because they rely on the PCI configuration space for accessing device registers. Hence, they show a greater proportion of bus operations. Additionally, virtual device classes such as md (RAID), do page-level writes by calling the block subsystem through routines like `submit_bio` rather than by accessing a device directly.

BUS	Kernel Interactions					Device Interactions			
	mem	*sync*	*dev lib.*	*kern lib.*	*kern services*	*port/mmio*	*dma*	*bus*	*avg devices/driver*
PCI	15.6	57.8	13.3	43.2	9.1	125.1	7.0	21.6	7.5
USB	9.6	25.5	5.6	9.9	3.0	0.0	2.2[2]	13.8	13.2
Xen	10.3	8.0	7.0	6.0	2.75	0.0	0.0	34.0	1/All

Table 3. Comparison of modern buses on drivers across all classes. Xen and USB drivers invoke the bus for the driver while PCI drivers invoke the device directly.

Second, these results demonstrate that the cost of isolating drivers can vary widely based on their interaction style. Direct interactions, such as through ports or MMIO, can use hardware protection, such as virtual memory. Thus, an isolated driver can be allowed to access the device directly. In contrast, calls to set up DMA or use bus routines rely on software isolation, and need to cross protection domains. Thus, drivers using higher-level buses, like USB, can be less efficient to isolate, as they can incur a protection-domain transition to access the device. However, as we show in the next section, access devices through a bus can often result in far fewer operations.

Implications: The number and type of device interactions vary widely across devices. Thus, the cost of isolating drivers, or verifying that their I/O requests are correct (as in Nexus [44]) can vary widely across drivers. Thus, any system that interposes or protects the driver/device interaction must consider the variety of interaction styles. Similarly, symbolic execution frameworks for drivers [20] must generate appropriate symbolic data for each interaction style.

4.4 Driver/Bus Interaction

The plurality of drivers in Linux are for devices that attach to some kind of PCI bus (*e.g.*, PCIe or PCI-X). However, several other buses are in common use: the USB bus for removable devices and XenBus for virtual devices [46]. Architecturally, USB and Xen drivers appear to have advantages, as they interact with devices over a message-passing interface. With USB 3.0 supporting speeds up to 5 Gbps [42] and Xen supporting 10 Gbps networks [30], it is possible that more devices will be accessed via USB or XenBus.

In this section, we study the structural properties of drivers for different buses to identify specific differences between the buses. We also look for indications that drivers for a bus may have better architectural characteristics, such as efficiency or support for isolation. We focus on two questions: (i) does the bus support a variety of devices efficiently, (ii) will it support new software architectures that move driver functionality out of the kernel onto a device or into a separate virtual machine? Higher efficiency of a bus interface results from supporting greater number of devices with standardized code. Greater isolation results from having less device/driver specific code in the kernel. If a bus only executes standardized code in the kernel, then it would be easier to isolate drivers away from kernel, and execute them inside a separate virtual machine or on the device itself such as on an embedded processor.

Table 3 compares complexity metrics across all device classes for PCI, USB, and XenBus. First, we look at the efficiency of supporting multiple devices by comparing the number of chipsets supporting by a driver. This indicates the complexity of supporting a new device, and the level of abstraction of drivers. A driver that supports many chipsets from different vendors indicates a standardized interface with a high level of common functionality. In contrast, drivers that support a single chipset indicate less efficiency, as each device requires a separate driver.

The efficiency of drivers varied widely across the three buses. PCI drivers support 7.5 chipsets per driver, almost always from the same vendor. In contrast, USB drivers average 13.2, often from many vendors. A large part of the difference is the effort at standardization of USB protocols, which does not exist for many PCI devices. For example, USB storage devices implement a standard interface [41]. Thus, the main USB storage driver code is largely common, but includes call-outs to device-specific code. This code includes device-specific initialization, suspend/resume (not provided by USB-storage and left as an additional feature requirement) and other routines that require device-specific code. While there are greater standardization efforts for USB drivers, it is still not complete

Unlike PCI and USB drivers, XenBus drivers do not access devices directly, but communicate with a back-end driver executing in a separate virtual machine that uses normal Linux kernel interfaces to talk to any driver in the class. Thus, a single XenBus driver logically supports *all* drivers in the class. in a separate domain so we report them as a single chipset. However, device-specific behavior, described above in Section 3.4, is not available over XenBus; these features must be accessed from the domain hosting the real driver. XenBus forms an interesting basis for comparison because it provides the minimum functionality to support a class of devices, with none of the details specific to the device. Thus, it represents a "best-case" driver.

We investigate the ability of a bus to support new driver architectures through its interaction with the kernel and device. A driver with few kernel interactions may run more easily in other execution environments, such as on the device itself. Similarly, a driver with few device or bus interactions may support accessing devices over other communication channels, such as network attached devices [26]. We find that PCI drivers interact heavily with the kernel unless kernel resources are provided by an additional higher-level virtual bus (*e.g.*, ATA). In contrast, Xen drivers have little kernel interaction, averaging only 34 call sites compared to 139 for PCI drivers. A large portion of the difference is that Xen drivers need little initialization or error handling, so they primarily consist of core I/O request handling.

The driver/device interactions also vary widely across buses: due to the fine granularity offered by PCI (individual bytes of memory), PCI drivers average more device interactions (154) than USB or XenBus devices (14-34). Thus, USB drivers are more economical in their interactions, as they batch many operations into a single request packet. XenBus drivers are even more economical, as they need fewer bus requests during initialization and as many operations as USB for I/O requests. Thus, USB and XenBus drivers may efficiently support architectures that access drivers over a network, because access is less frequent and coarse grained.

Implications: These results demonstrate that the flexibility and performance of PCI devices comes with a cost: increased driver complexity, and less interface standardization. Thus, for devices that can live within the performance limitations of USB or in a virtualized environment for XenBus, these buses offer real architectural advantages to the drivers. With USB, significant standardization enables less unique code per device, and coarse-grained access allows efficient remote access to devices [1, 10, 17].

[2] USB drivers invoke DMA via the bus.

XenBus drivers push standardization further by removing *all* device-specific code from the driver and executing it elsewhere. For example, it may be possible to use XenBus drivers to access a driver running on the device itself rather than in a separate virtual machine; this could in effect remove many drivers from the kernel and host processor.

The mechanism for supporting non-standard functionality also differs across these buses: for PCI, a vendor may write a new driver for the device to expose its unique features. For USB, a vendor can add functionality to the existing common drivers just for the features. For XenBus, the features must be accessed from the domain executing the driver and are not available to a guest OS.

4.5 Driver Concurrency

Another key requirement of drivers in all modern operating systems is the need to multiplex access to the device. 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, as it increases the need for synchronization among multiple independent threads. We investigate how drivers multiplex access across long-latency operations: do they tend towards threaded code, saving state on the stack and blocking for events, or toward event-driven code, registering callbacks either as completion routines for USB drivers or interrupt handlers and timers for PCI devices. If drivers are moved outside the kernel, the driver and kernel will communicate with each other using a communication channel and supporting event-driven concurrency may be more natural.

We determine that a driver entry point requires a threaded programming style if it makes blocking calls into the kernel, or busy-waits for a device response using `msleep()` which enables blocking. All other entry points are considered "event friendly", in that they do not suspend the calling thread. We did not detect specific routines that use event-based synchronization, as they often rely on the device to generate the callback via an interrupt rather than explicitly registering with the kernel for a callback .

The results, shown in Figure 9 in the bars labeled *event friendly* and *threaded*, show that the split of threaded and event-friendly code varies widely across driver classes. Overall, drivers extensively use both methods of synchronization for different purposes. Drivers use threaded primitives to synchronize driver and device operations while initializing the driver, and updating driver global data structures, while event-friendly code is used for core I/O requests. Interestingly, network drivers and block drivers, which are not invoked directly by user-level code, have a similar split of code to sound drivers, which are invoked directly from application threads. This arises because of the function of most driver code, as reported in Section 3.2: initialization and configuration. This code executes on threads, often blocking for long-latency initialization operations such as device self-test.

Implications: Threaded code is difficult to run outside the kernel, where the invoking thread is not available. For example, Microdrivers [16] executes all driver code in an event-like fashion, restricting invocation to a single request at a time. Converting drivers from threads to use event-based synchronization internally would simplify such code. Furthermore, events are a more natural fit when executing driver code either in separate virtual machine or on a device itself, as they naturally map to a stream of requests arising over a communication channel [34].

5. Driver Redundancy

Given that all the drivers for a class perform essentially the same task, one may ask why so much code is needed. In some cases, such as IDE devices, related devices share most of the code with

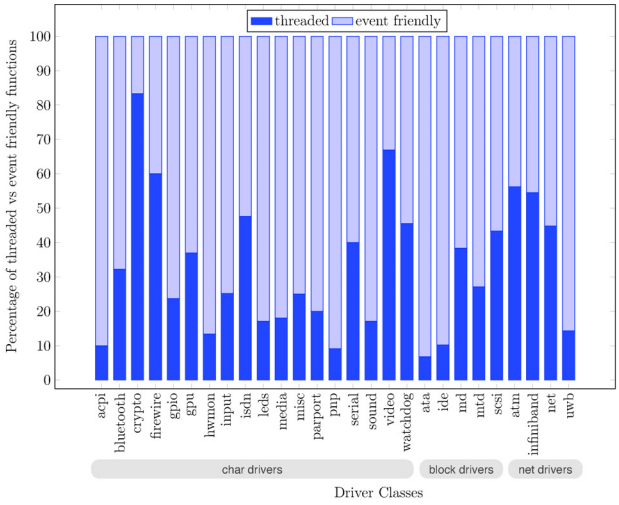

Figure 9. The percentage of driver entry points under coverage of threaded and event friendly synchronization primitives.

```
DrComp signature: 1.798865
static int hpt374_fn1_cable_detect(....)
{
  struct pci_dev *pdev = to_pci_dev(...);

  unsigned int mcrbase = 0x50
            + 4 * ap->port_no;

  u16 mcr3;
  u8 ata66;

  /*Do the extra channel work */
  pci_read_config_word(pdev,
                  mcrbase+2,&mcr3);
  /*Set bit 15 of 0x52 to enable ..*/
  pci_write_config_word(pdev,
                   mcrbase + 2,..);
  pci_read_config_byte(pdev,
                  0x5A,&ata66);
  /*Reset TCBLID/FCBLID to output */
  pci_write_config_word(pdev,
                  mcrbase+2,mcr3);
  if (ata66 & (2 >> ap->port_no))
      return ATA_CBL_PATA40;
  else
      return ATA_CBL_PATA80;
}
```

```
DrComp signature: 1.8
static int hpt37x_cable_detect(...)
{
  struct pci_dev *pdev = to_pci_dev(...);

  u8 scr2, ata66;

  pci_read_config_byte(pdev,
                  0x5B, &scr2);
  pci_write_config_byte(pdev,
                  0x5B,...);

  udelay(10); /* debounce */

  /* Cable register now active */
  pci_read_config_byte(pdev,
                  0x5A,&ata66);
  /* Restore state */
  pci_write_config_byte(pdev,
                  0x5B,scr2);
  if (ata66 & (2 >> ap->port_no))
      return ATA_CBL_PATA40;
  else
      return ATA_CBL_PATA80;
}
```

Figure 10. Similar code between two different HPT ATA controller drivers essentially performing the same action. These are among the least-similar functions that DrComp is able to detect these functions as related. The boxes show differentiating statements in the two functions that account for the close signature values.

a small amount of per-device code. Most device classes, though, replicate functionality for every driver. The problem of writing repeated/redundant code is well documented. It causes maintainability issues in software development [13], and is also a significant cause of bugs in the Linux kernel [8, 22, 29]. Providing the right abstractions also helps in code standardization and integrating kernel services such as power management in a correct fashion across all drivers. Without a global view of drivers, it can be difficult to tell whether there are opportunities to share common code.

To address this question, we developed a scalable, code similarity tool for discovering similar code patterns across related drivers and applied it to Linux drivers. The goal of this work is to find driver functions with substantially similar code, indicating that the common code could be abstracted and removed from all drivers to reduce driver code size and complexity.

5.1 Methodology

We developed a new code-similarity tool to handle the number of Linux drivers to find similarities rather than exact copies. We

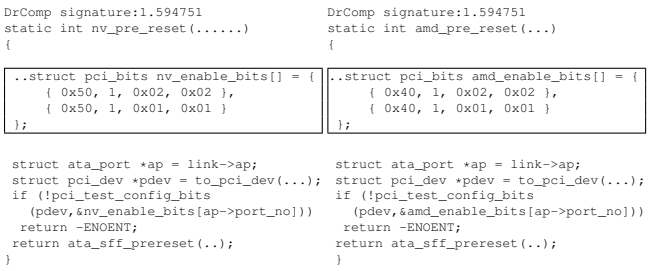

Figure 11. The above figure shows identical code that consumes different register values. Such code is present in drivers where multiple chipsets are supported as well as across drivers of different devices. The functions are copies except for the constants as shown in the boxes.

needed to parse through the entire driver source tree consisting of 5 million lines of code, with close to a million lines of code in large classes like network drivers. Most existing clone-detection tools develop a stream of tokens or tree/graphs and perform an $n \times n$ comparison of all code snippets to detect clones, which given the size of the driver classes, is not possible. In addition, we needed a *similarity* detector for finding code that that is closely related but not identical. With more control over how similar regions are detected using information from the semantics of drivers, we are able to detect more useful similar code. For example, while parsing function calls, we treat calls to the device and kernel differently, improving the accuracy of our similarity-detection tool.

Our similarity tool, *DrComp*, is based on *shape analysis*[3] [11]. This is a method to determine whether clusters of points have a similar shape and variants of these technique are often used to cluster data, to determine nature of data distribution, and to detect identical shapes in computer vision [3].

DrComp generates a set of multidimensional coordinates for every function in every driver. It then detects as similar two functions whose coordinate sets (*shape*) are similar. DrComp processes a driver function and adds a point to the function's shape for every statement or action in statement for loop, kernel interaction, conditionals, device interaction, variable assignment, break and return statements. The coordinates of the point are the offset into the function (line number) and the statement type. To improve accuracy, it is important that the generated shape of the code emphasizes the important characteristics of the function. Hence, we also reinforce the shape of the function by weighting statements that are important yet sparse, such as a function returns and calls to kernel functions. The shape of each driver function is a cloud of points on plane representing the structure of the program. While we consider only two dimensions of the code, the statement type and edit distance to generate the points, our tool can easily be extended to include additional dimensions based on code features (such as nesting depth) or driver features (such as interrupt frequency).

To eliminate complex comparison of two driver functions, we further reduce the shape of a driver down to a single *signature* value. We compute the signature as a function of Euclidean distance between all the points in the code cluster obtained above. The output of DrComp is a signature for every function in every driver. Thus, two functions with identical code will have identical signatures. Furthermore, code that is similar, in that it has a similar structure of loops and I/O operations, will have similar signatures.

Figure 10 shows an example of some of the *least similar* related code in drivers we found. These two functions have signatures within 0.05% of each other. DrComp only looks for the code structure from statement types (also distinguishing kernel and device invocations) and edit distance, so functions may use different arguments (register values), compare against different values or loop on different conditions, and still be grouped as similar code.

[3] We perform geometric shape analysis, not the program analysis technique of the same name.

5.2 Redundancy Results

DrComp detected that 8% of all driver code is very similar to other driver code. The results of our similarity study are shown in Table 4. For classes with many similarities, we show the number of fragment clusters (sets of similar code), as well as the total number of functions that are similar to another function. For the results in above table, we show results within individual driver classes and not across classes, as they are less likely to benefit from a shared abstraction.

Overall, we identified similarities within a single driver, across a subset of drivers in a class, and in some cases across most drivers in a class. Within a single driver, we found that the most common form of repeated code was wrappers around device I/O, driver library or kernel functions. These wrappers either convert data into the appropriate format or perform an associated support operation that is required before calling the routines but differ from one another because they lie on a different code path. These wrappers could be removed if the kernel interface supported the same data types as the device or if drivers provided appropriate abstractions to avoid such repeated code.

We also find swaths of similar functions across entire classes of drivers. The major difference between drivers for different chipsets of the same device are often constant values, such as device registers or flag values. For example, ATA disk drivers abstract most of the code into a core library, `libata`, and each driver implements a small set of a device-specific functionality. Commonly, these functions are short and perform one or two memory-mapped I/O reads or writes, but with different values for every driver. Figure 5 shows two functions from different ATA drivers with substantially similar code. This practice generates large bodies of very similar drivers with small differences. Further abstraction could additionally simplify these drivers, for example, replacing these routines with tables encoding the different constants. Similarly, a hardware specification language [25] may be able to abstract the differences between related devices into a machine-generated library.

Finally, we note similarities across subsets of drivers in a class. For example, another common class of similarities is wrappers around kernel functions and driver libraries for that class: the `release` method for frame buffers is virtually identical across many of the drivers, in that it checks a reference count and restores the VGA graphics mode. There are a few small differences, but refactoring this interface to pull common functionality into a library could again simplify these drivers.

Implications: Overall, these results demonstrate that there are many opportunities for reducing the volume of driver code by abstracting similar code into libraries or new abstractions. We visually inspected all function clusters to determine how a programmer could leverage the similarity by having a single version of the code. We see three methods for achieving this reduction: (i) procedural abstractions for driver sub-classes, (ii) better multiple chipset support and (iii) table driven programming.

The most useful approach is *procedural abstraction*, which means to move the shared code to a library and provide parameters covering the differences in implementation. There is significant code in single drivers or families of drivers with routines performing similar functions on different code paths. Creating driver-class or sub-class libraries will significantly reduce this code. Second, existing driver libraries can be enhanced with new abstractions that cover the similar behavior. There are many families of drivers that replicate code heavily, as pointed out in Table 4. Abstracting more code out these families by creating new driver abstractions that

Driver class	Driver subclass	Similar code fragments	Fragment clusters	Fragment size (Avg. LOC)	Redundancy results and action items to remove redundant code
char	acpi	64	32	15.1	Procedural abstraction for centralized access to kernel resources and passing get/set configuration information as arguments for large function pairs.
	gpu	234	108	16.9	Procedural abstractions for device access. Code replicated across drivers, like in DMA buffer code for savage, radeon, rage drivers, can be removed by supporting more devices per driver.
	isdn	277	118	21.0	Procedural abstraction for kernel wrappers. Driver abstraction/common library for ISDN cards in hisax directories.
	input	125	48	17.23	Procedural abstraction for kernel wrappers. Driver abstraction/common driver for all touchscreen drivers. Procedural abstraction in Aiptek tablet driver.
	media	1116	445	16.5	Class libraries for all Micron image sensor drivers. Procedural abstraction in saa 7164 A/V decoder driver and ALI 5602 webcam driver.
	video	201	88	20	Class libraries for ARK2000PV, S3Trio, VIA VT8623drivers in init/cleanup, power management and frame buffer operations. Procedural abstraction in VESA VGA drivers for all driver information functions.
	sound	1149	459	15.1	Single driver for ICE1712 and ICE1724 ALSA drivers. Procedural abstraction for invoking sound libraries, instead of repeated code with different flags. Procedural abstraction for AC97 driver and ALSA driver for RME HDSPM audio interface.
block	ata	68	29	13.3	Common power management library for ALI 15x3, CMD640 PCI, Highpoint ATA controllers, Ninja32, CIL 680, ARTOP 867X, HPT3x3, NS87415 PATA drivers and SIS ATA driver. Table driven programming for device access in these drivers.
	ide	18	9	15.3	Procedural abstraction for the few wrappers around power management routines.
	scsi	789	332	25.6	Shared library for kernel/scsi wrappers for Qlogic HBA drivers; pmc sierra and marvell mvsas drivers. Large redundant wrappers in mp2sas firmware, Brocade FC port access code.
net	Ethernet/ wireless	1906	807	25.1	Shared library for wireless drivers for talking to device/kernel and wireless routines. Lot of NICs share code for most routines like configuration, resource allocation and can be moved to a single driver with support for multiple chipsets. A driver sub-class library for all or vendor specific Ethernet drivers.
	infiniband	138	60	15.0	Procedural abstraction for Intel nes driver.

Table 4. The total number of similar code fragments and fragment clusters across driver classes and action items that can be taken to reduce them.

support multiple chipsets can simplify driver code significantly. Finally, functions that differ only by constant values can be replaced by table-driven code. This may also be applicable to drivers with larger differences but fundamentally similar structures, such as network drivers that use ring buffers to send and receive packets. By providing these abstractions, we believe there is an opportunity to reduce the amount of driver code, consequently reducing the incidence of bugs and improving the driver development process by producing concise drivers in the future.

6. Conclusions

The purpose of this study is to investigate the complete set of drivers in Linux, to avoid generalizing from the small set of drivers commonly used for research, and to form new generalizations.

Overall, we find several results that are significant to future research on drivers. First, a substantial number of assumptions about drivers, such as class behavior, lack of computation, are true for many drivers but by no means all drivers. For example, instead of request handling, the bulk of driver code is dedicated to initialization/cleanup and configuration, together accounting for 51% of driver code. A substantial fraction (44%) of drivers have behavior outside the class definition, and 15% perform significant computations over data. Thus, relying on a generic frontend network driver, as in Xen virtualization, conceals the unique features of different devices. Similarly, synthesizing driver code may be difficult, as this processing code may not be possible to synthesize. Tools for automatic synthesis of driver code should also consider driver support for multiple chipset as we find that Linux supports over 14,000 devices with just 3,217 bus and device drivers.

Second, our study of driver/device/kernel interactions showed wide variation in how drivers interact with devices and the kernel. At one end, miniport drivers contain almost exclusively device-specific code that talks to the device, leaving kernel interactions to a shared library. At the other end, some drivers make extensive calls to the kernel and very few into shared device libraries. This latter category may be a good candidate for investigation, as there may be shared functionality that can be removed. Overall, these results also show that the cost of isolating drivers may not be constant across all driver classes.

Third, our investigation of driver/device interaction showed that USB and XenBus drivers provide more efficient device access than PCI drivers, in that a smaller amount of driver code supports access to many more devices, and that coarse-grained access may support moving more driver functionality out of the kernel, even on the device itself. Furthermore, many drivers require very little access to hardware and instead interact almost exclusively with the bus. As a result, such drivers can effectively be run without privileges, as they need no special hardware access. We find that USB and Xenbus provide the opportunity to utilize the extra cycles on devices by executing drivers on them and can effectively be used to remove drivers from the kernel leaving only standardized bus code in the kernel.

Finally, we find strong evidence that there are substantial opportunities to reduce the amount of driver code. The similarity analysis shows that there are many instances of similar code patterns that could be replaced with better library abstractions, or in some cases with tables. Furthermore, the driver function breakdown in Section 3 shows that huge amounts of code are devoted to initialization; this code often detects the feature set of different chipsets. Again, this code is ripe for improvement through better abstractions, such as object-oriented programing technique and inheritance [31].

While this study was performed for Linux only, we believe many similar patterns, although for different classes of devices, will show in other operating systems. It may also be interesting to

compare the differences in driver code across operating systems, which may demonstrate subtle differences in efficiency or complexity. Our study is orthogonal to most studies on bug detection. However, correlating bugs with different driver architectures can provide insight on the reliability of these architectures in real life.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grants CNS-0915363 and CNS-0745517 and a grant from Google. We also thank Matt Renzelmann and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

References

[1] J. Andrus, C. Dall, A. Vant Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *SOSP*, 2011.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Eurosys*, 2006.

[3] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *Transactions on Pattern Analysis and Machine Intelligence*, 2002.

[4] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *USENIX ATC*, 2010.

[5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009.

[6] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *Eurosys*, Apr. 2010.

[7] T.-c. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *Operating Systems Review*, 33, 1999.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[9] F. M. David et al. CuriOS: Improving reliability through operating system structure. In *OSDI*, 2008.

[10] Digi International. AnywhereUSB. `http://www.digi.com/products/usb/anywhereusb.jsp`.

[11] I. Dryden and K. Mardia. *Statistical shape analysis*, volume 4. Wiley New York, 1998.

[12] K. Fraser et al. Safe hardware access with the Xen virtual machine monitor. In *OASIS Workhop*, 2004.

[13] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, 2010.

[14] A. Ganapathi, V. Ganapathi, and D. A. Patterson. Windows xp kernel crash analysis. In *LISA*, 2006.

[15] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *HOTOS11*, 2007.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS*, Mar. 2008.

[17] A. Hari, M. Jaitly, Y.-J. Chang, and A. Francini. The swiss army smartphone: Cloud-based delivery of usb services. In *Mobiheld*, 2011.

[18] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *SOSP*, 2009.

[19] A. Kadav and M. Swift. Live migration of direct-access devices. *Operating Systems Review*, 43(3):95–104, 2009.

[20] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC*, 2010.

[21] B. Leslie et al. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 2005.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[23] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *SOSP*, 2011.

[24] A. Menon, S. Schubert, and W. Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *ASPLOS*, 2009.

[25] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *OSDI*, 2000.

[26] Microsoft Corp. Web services on devices. `http://msdn.microsoft.com/en-us/library/aa826001%28v=vs.85%29.aspx`.

[27] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, 2002.

[28] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft WinHec Presentation DDT301, 2003.

[29] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys*, 2008.

[30] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *VEE*, 2009.

[31] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a modern language. In *USENIX ATC*, June 2009.

[32] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Eurosys*, 2009.

[33] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *SOSP*, 2009.

[34] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *APSys*, Aug. 2010.

[35] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A declarative language approach to device configuration. In *ASPLOS*, 2011.

[36] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, eighth edition, 2009.

[37] M. Spear et al. Solving the starting problem: Device drivers as self-describing artifacts. In *Eurosys*, 2006.

[38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *In TOCS*, 2005.

[40] Úlfar Erlingsson et al. Xfi: software guards for system address spaces. In *OSDI*, 2006.

[41] USB Implementors Forum. Universal serial bus mass storage class specification overview. `http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf`, 2003.

[42] USB Implementors Forum. Universal serial bus 3.0 specification. http://www.usb.org/developers/docs/usb_30_spec_071311.zip, 2011.

[43] D. A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`.

[44] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.

[45] L. Wittie. Laddie: The language for automated device drivers (ver 1). Technical Report 08-2, Bucknell CS-TR, 2008.

[46] Xen.org. Writing xen drivers: Using xenbus and xenstore. `http://wiki.xensource.com/xenwiki/XenBus`, 2006.

[47] F. Zhou et al. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.