Beefy Drivers for Flaky Hardware

Asim Kadav and Tara Mohan
CS 706 Project Proposal, Fall 2008

Abstract

The current implementation of device drivers assumes and relies on the correctness of the underlying hardware. Any unresponsive hardware can cause the application and/or the system to crash or hang. However, current technology trends indicate that devices are becoming more susceptible to hardware failures, both permanent and transient in nature. Hence, the current implementation of drivers is poorly suited to run on increasingly unreliable device hardware.

This project intends to solve this problem by building reliable device drivers that can tolerate hardware failures and recover as necessary. This project intends to statically analyze device drivers and identify driver code that assumes correct hardware behaviour as a basis for forward progress of the driver. These drivers will then be *hardened* by modifying the code to alleviate this reliance. Time permitting, we also intend to build a fault injection framework to test the robustness of our methodology. Based on our results, we also plan to discuss possible recovery strategies that can be performed in cases where the driver cannot continue to perform normally due to corruption of the driver's state.

1 Introduction

Even though many applications are resistant to operate with minor errors [8, 16], device drivers in commodity operating systems *as-is* are unable to recover from most device errors and can potentially cause operating system hangs or crashes. In the current scheme of drivers in modern operating systems, the majority of system crashes are attributed to drivers; for example, 89% of the crashes of the Windows operating system are due to drivers [9]. Another related study on the Windows operating system shows that systems with fault-tolerant drivers suffer from a signficantly lower crash rate [1].

Even worse, as device manufacturing processes continue to advance to meet the growing demands of functionality, performance, and specifications, we are increasingly hitting bottlenecks in the process of developing reliable device hardware. Due to problems such as transistor variability, aging and transient errors, it will become increasingly difficult to maintain the reliability of manufactured device hardware even as manufacturing costs continue to increase [4]. Because most drivers are reliant on correct device behaviour, device errors when traveling up the software stack can cause abnormal system behaviour, application crashes, or entire system crashes or hangs, depending on which layer of the software stack gets affected. Drivers are worst affected by device unreliability because they operate inside the operating system in priveleged mode. Any errors in drivers can corrupt kernel data structures or lockup the entire operating system kernel. One such example is shown on the left hand side of Table 1. The unsafe driver shown there will wait indefinitely for the device. A device failure here can cause the entire system to hang.

This research accepts these transient device errors as a part of the device specification and tries to build reliable device drivers from existing drivers which can tolerate transient hardware mis-behaviour and/or fail gracefully when such a problem occurs. This document proposes our design methodology, expected project timeline, and evaluation methodology, and discusses the related work.

2 Design

This section describes the design of the different components of our project as follows in different subsections.

2.1 Driver Hardening

We intend to use static analysis to *harden* drivers for hardware faults. An example of a *hardened* driver is illustrated on the right hand side of Table 1. The given piece of code does not rely or hang by waiting indefinitely on the driver but instead waits for the device and then gracefully flags an error message. To *harden* existing broken drivers to behave more reliably, we intend to build a tool that will statically analyze all driver code and use program slicing techniques to identify control paths where the driver may terminate (or hang) without an error message after reading some state of the driver. Some examples of this include (a) control flow paths such as a `while` loop or a `for` loop where the driver waits on device registers, (b) a device register value being used as an index of an array or (c) a driver going off to sleep until an interrupt occurs.

We will identify all such control paths that may have data or control dependencies on the correct device operation. If a driver uses a value generated by a device in

Unsafe Driver (3c59x.c)	**Safe Driver** (pcnet32.c)

while (ioread16(ioaddr + Wn7_MasterStatus) & 0x8000) { ; }	ticks = 0;
	while (!(a->read_csr(ioaddr, 5) & 0x0001)) {
	. . .
	if (++ticks < 200) {
	printk(KERN_DEBUG "%s: Error getting into suspend!",
	dev->name);
	break;
	}
	}

Table 1: An illustration of a hardware unsafe and a hardware safe driver.

Dependency Type	**Description**	**Possible Fix**
Control	Waiting for the device in infinite `while` loop.	Add a simple counter.
Control	Driver waiting for an interrupt.	Insert timeouts or detect dynamically.
Data	Using device register to perform indexed array access.	Check array bounds.
Data	Waiting on a device resource.	Use callbacks.
Data	Data re-reads.	Avoid reading the data twice when not required.
Data	DMA into incorrect address.	Invoke checks on send address in DMA calls.
Data	Corrupt data from device.	Use retry more often to handle transient errors.

Table 2: Possible classes of failures due to unreliable hardware

a calculation, for example, as an array index, then the resulting value is data-dependent on the device. When a driver makes a control decision, such as a branch or function call, based on either data or an interrupt from a device, the driver is control-dependent on the correct functioning of that device. We intend to perform a complete dependency-finding analysis for the dependency classes listed in Table 2. This table may expand as we examine more and more drivers.

For example, for case (c) described above, where a driver takes action only during an interrupt handler, the driver depends on the device to generate an interrupt. Here, we will need to analyze drivers to determine which code executes only in response to interrupts. We can identify interrupt handlers based on their registration with the kernel in the driver code. Similarly, reading data generated by the device, through explicit I/O instructions or memory operations on a region shared with the device, depends on the correct operation of the device. We can identify such shared memory regions by the APIs used to allocate them. For example, Linux drivers map I/O device registers into virtual memory with the `ioremap` function and allocate shared memory with the `pci_alloc_consistent` kernel call. From these APIs, we will track which memory accesses refer to shared memory. After identifying such memory accesses, we can identify the driver code that depends on device behaviour. We may need to perform a *points-to* analysis for the above memory dependency requirement. We intend to use the facility provided in CIL to perform this analysis.

The second challenge, after the detection of failure-intolerant code, is generating failure detection code. After identifying driver code that depends on device correctness, we will insert a failure detector that identifies when the device has failed, i.e., when it violates the driver's assumptions about its behaviour. For address calculations that depend on device correctness, we plan to insert range checks that detect when array bounds are exceeded [17] for arrays whose sizes are known at compile-time. For liveness checks, counters in loops may suffice in many cases. More complicated tests, such as those to detect when a packet has not been sent, may also be necessary. For these tests, we will generate timer callbacks from the kernel that test whether any long-lived requests are still pending.

2.1.1 Implementation of Analyses

We intend to use the Berkely CIL tool [11] for our analysis. CIL operates on C language source code and performs source transformation to produce suitable data structures. By operating on these data structures, we can analyze the code and make suitable modifications to produce *hardened* kernel module binaries. CIL is written in the OCaml language in which we are both novices. To learn OCaml, which is necessary for our analysis, we will go through the OCaml tutorials available at `http://www.ocaml-tutorial.org`. We intend to

perform our analysis on the Linux source code on the stable `linux-2.6.18` kernel.

We intend to proceed in the following manner to perform our analysis:

1. Preprocess the device driver source files in the `drivers` directory of the kernel tree and read them via CIL.

2. Once loaded in CIL, analyze each driver source file to identify code which depends on the device control information.

3. Fortify this code by removing or minimizing this dependency and generating failure detection code in the intermediate CIL form.

4. Build the hardened kernel module for the driver from the intermediate form.

As obvious from this description, we do not intend to produce any hardened original C syntax. Also, we do not intend to limit ourselves to any particular device or class of drivers. Also, our analysis is largely intra-procedural except in cases where we need to find registration of interrupts with the kernel, which may be done inside a different procedure than where it is used.

This section is the primary focus of this class project.

2.2 Fault Injection Framework

To suitably evalute the effectiveness of our tool, we intend to develop a fault injection framework based on existing driver fault injection frameworks [18]. The driver fault injection framework will intersperse itself between the driver and the device and will simulate device failures, such as reads or writes of device registers, randomly. We also plan to simulate device failure by repressing valid events or generating false events like device interrupts. The thoroughness involved in this section of the project will depend on the time that will be remaining with us after satisfactorily completing the previous section of the project.

2.3 Recovery Systems

Performing proper recovery when permanent failures occur raises additional challenges. Unlike software failures, hardware failures may require additional operations by the OS to recover, such as failing over to redundant hardware. The driver should either recover to a functioning state or shut down to limit failure propagation. Past work on failure handling has shown the difficulties of adding error handling to existing code [14].

There also might be cases of transient errors where our beefy drivers become unusable, even though they are able to avoid a crash or hang. We intend to discuss what recovery techniques will best suit in cases where the driver state is corrupted based on the results of our fault injection framework.

3 Project Timeline

The project intends to earnestly stick to the timeline as given in Table 3.

4 Methodology of Evaluation

Our evaluation will examine the performance overhead of using the *hardened* device driver code. We intend to perform this for some of the devices depending on the availibility of the devices using the drivers. To evaluate the effectiveness of our tool, we will consider the various driver dependencies on correct device operation that our tool is able to identify and *harden*. The metric for this evaluation will be the number of such cases we are able to identify. We will also measure the ability of the resulting *hardened* drivers to tolerate failures caused by the fault-injection framework.

5 Related work

There has been significant research in applying program analysis techniques to systems research. For applying runtime checks on driver and OS code, many tools exist, such as the Driver Verifier(DV) and PURIFY, which can analyze the running code for violations due to array bounds, memory leaks, etc. There are also many static verification tools targeting device driver reliability, such as the Static Driver Verifier(SDV) tool [2]. SDV is a bug finding tool that checks for and reports misuse of kernel API calls in drivers. SDV has an analysis engine that verifies the driver against a set of API usage rules and determines usage bugs for the programmer to fix. This engine is based on SLAM [3] which is a generic static-checker which tries to detect all possible ways a driver can disobey a set of API usage rules.

Our tool, on the other hand, is designed to fortify the driver interaction with the hardware. Also, while SDV is aimed at preventing kernel misuse, our tool is directed towards device misuse. Bug finding tools like SDV generally assume that devices behave correctly and rely on programmers to fix the problems found [2, 3, 5, 7, 12]. Our described mechanism, however, relies on static analysis to detect code that is not tolerant of device failures, fixes it and also dynamically uses recovery techniques to manage system violations arising from device misbehaviour. This improves the coverage of our tool and removes the burden on programmers to fix the failures that occur. There are also past efforts at driver hardening provide fault injection tools to discover through testing where a driver may fail [10, 13]. In this work, as mentioned before, we will use static analysis to harden driver code that is not tolerant of device failures.

Dates	Work Item	Status
Oct 20th - Oct 24th	Get Proposal approved	Incomplete
Oct 25th - Oct 27th	Setup the test and build environment with required kernel and other CIL tools. Familarize selves with CIL and OCAML.	Incomplete
Oct 27th - Nov 1st	Develop simple basic CIL infrastructure.	Incomplete
Nov 1st - Nov 15th	Complete CIL development and generate fortified driver code.	Incomplete
Nov 15th - Dec 1st	Develop testbench and evaluate the developed drivers depending on available devices.	Incomplete
Dec 1st - Dec 10th	Brainstorm recovery techniques.	Incomplete

Table 3: Proposed schedule of the project

To improve device driver safety, prior work has also investigated putting non-performance critical code in user-space so that a driver failure does not wreak havoc on the system [6]. There also has been prior work to maintain the driver in a safe state using finite state automata generated from device specifications to prevent the driver from mis-behaving [15]. However, none of the above work is targeted at saving the driver/system from unreliable hardware.

6 Expected Conclusions

With this project, we intend to be able to classify the common bugs due to hardware failures and beef up our drivers to handle these failures. With a suitable test framework, we intend to introduce transient hardware failures and determine the effectiveness of our approach. We will also discuss recovery scenarios and the application of suitable recovery techniques. Finally, we will examine how this work will be useful in efforts to improve the fault tolerance of legacy code by automating failure detection and recovery, which may apply beyond device drivers.

7 Project Participants

The student participants in this class project are strictly limited to Asim Kadav and Tara Mohan. Asim's advisor, Prof. Mike Swift is involved in this research and this project idea originates from him. There may be other non CS-706 students involved in this research, but only after January 2009.

References

[1] Sandy Arthur. Fault resilient drivers for Longhorn server. Technical Report WinHec 2004 Presentation DW04012, Microsoft Corporation, May 2004.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference*, Leuven, Belgium, April 2006.

[3] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

[4] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. In *Proc. of the 38th Annual IEEE/ACM International Symp. on Microarchitecture*, November 2005.

[5] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX OSDI*, pages 1–16, October 2000.

[6] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. Microdrivers: a new architecture for device drivers. In *Proc. of the 13th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[7] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 526–538. Springer-Verlag, July 2002.

[8] X. Li and D. Young. Application-level correctness and its impact on fault tolerance. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, February 2007.

[9] Microsoft Corporation. Windows XP embedded with service pack 1 reliability. `http://msdn2.microsoft.com/en-us/library/ms838661.aspx`, January 2003.

[10] Microsoft Corporation. Windows device testing framework.

http://msdn2.microsoft.com/en-us/library/aa973530.aspx, 2007.

[11] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, April 2002.

[12] Hendrik Post and Wolfgang Kuchlin. Integrated static analysis for Linux device driver verification. In *Proceedings of the 6th International Conference on integrated Formal Methods*, July 2007.

[13] Sun Microsystems. *Solaris Express Software Developer Collection: Writing Device Drivers*, chapter 13: Hardening Solaris Drivers. Sun Microsystems, 2007.

[14] Martin Süßkraut and Christof Fetzer. Automatically finding and patching bad error handling. In *Proceedings of the 6th European Dependable Computing Conference (EDCC'06)*, pages 13–22, October 2006.

[15] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX OSDI*, 2008.

[16] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *In The 2nd Workshop on System Effects of Logic Soft Errors (SELSE)*, 2006.

[17] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th USENIX OSDI*, November 2006.

[18] Louis Zhuang, Stanley Wang, and Kevin Gao. Fault injection test harness. In *Proceedings of the Ottawa Linux Symposium*, June 2003.