Static Analysis of Device Drivers: We Can Do Better!

Sidney Amani[‡§] Leonid Ryzhyk[‡§] Alastair F. Donaldson[¶]
Gernot Heiser[‡§] Alexander Legg[‡‖] Yanjin Zhu[‡§]
[‡]NICTA[*] [§]University of New South Wales [¶]University of Oxford[†] [‖]University of Sydney
sidney.amani@nicta.com.au

ABSTRACT

We argue that the device driver architecture enforced by current operating systems complicates both manual and automatic reasoning about driver behaviour. In particular, it makes it hard and in some cases impossible to statically verify that the driver correctly interacts with the rest of the kernel. This limitation cannot be addressed solely via better verification tools. We maintain that qualitative improvement in the effectiveness of static driver verification must rely on an improved driver architecture, leading to drivers that are easier to write, understand, and verify.

To support our claims, we present a device driver architecture, called active drivers, that satisfies these requirements. We outline our methodology for specifying and verifying active driver protocols using existing model checking tools and describe initial experimental results.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Input/Output; B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices

General Terms

Reliability, Verification

Keywords

Device Drivers, Software Protocols, Model Checking

1. INTRODUCTION

Faulty device drivers are a major source of operating system (OS) failures. In this paper we focus on a specific class of driver bugs—violations of protocols between the driver and the rest of the OS. These bugs are becoming increasingly common due to the growing complexity of internal OS interfaces [1]. Our earlier study [9] showed that OS protocol violations account for 20% of all driver bugs.

Automatic verification tools have proved useful in detecting driver protocol violations. In fact, some of the most advanced verification tools for C, including SLAM [1], SatAbs [3], and Blast [7] were either designed with the purpose of verifying driver protocols or used driver verification as their primary case study.

Despite significant effort invested in improving these tools, they remain limited in the size and complexity of drivers that they can handle and properties that they can verify for these drivers without generating a large number of false positives.

We argue that to a large extent these limitations are due to the device driver architecture enforced by current operating systems. In this architecture the driver does not have its own thread of control and instead its functions are called directly by OS threads. If the driver makes incorrect assumptions about the order in which the OS can invoke its entry points, it will be unable to handle these invocations correctly. As these assumptions are implicit in the source code of the driver, such bugs can only be detected via their indirect consequences, which can be difficult or impossible to identify automatically.

The following code fragment, showing two driver entry points, illustrates the problem:

```
int suspend() {... free (p); ...}
void remove() {... p->data=0; ...}
```

This code assumes that the `remove()` entry point cannot be called after `suspend()` and therefore it is safe to deallocate pointer `p` inside `suspend()`. While uncommon, this sequence of driver invocations can occur in practice if the user unplugs the device while it is in the suspended state, leading to an invalid pointer dereference.

The problem in this example is that the root cause of the bug—the fact that the driver is not prepared to handle `remove()` after `suspend()`—is implicit in the driver code and cannot be discovered using control flow analysis. The model checker can still find the bug by employing a handwritten C model of the OS kernel that randomly generates all possible sequences of driver invocations by the OS [1] and detecting that one of these sequences leads to a use-after-free error; however such analysis quickly gets intractable for code involving complex pointer manipulation. For instance, if the `remove()` function accessed the content of `p` via an alias pointer, this could lead to very long verification time.

[*]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
[†]Alastair F. Donaldson is supported by EPSRC grant EP/G051100.

Furthermore, if the problematic sequence of invocations caused the driver to issue invalid commands to the device instead of performing an invalid pointer dereference, such a bug would even in principle be impossible to detect automatically without providing a formal model of the device to the model checker.

We believe that qualitative improvement in the effectiveness of automatic driver verification must rely on an improved driver architecture. In this architecture, interactions with the OS must be explicit in the code of the driver, which allows verifying the driver's compliance with OS protocols using control flow analysis.

In our earlier work [10], we proposed such an architecture, called *active drivers*. It was originally introduced with the goal of improving driver reliability by making drivers easier to write and understand.

In the present paper we show that the active driver architecture also facilitates automatic analysis of driver correctness. To this end, we specify interaction protocols between active drivers and the OS using finite state machines. The driver implementation is then automatically checked for compliance with the protocol state machine using an existing model checker, namely SatAbs. Our initial experiments show that this methodology allows verification of driver properties that are hard or impossible to check for conventional drivers.

In the rest of the paper we give an outline of the active driver architecture (Section 2), present our methodology for writing formal specifications of active driver protocols (Section 3), and describe how these protocols can be verified using an existing C model checker (Section 4). We summarise our experimental results in Section 5 and discuss related work in Section 6.

2. ACTIVE DRIVERS

Active device drivers [10] were introduced to address two issues with the conventional driver architecture, where a driver is a passive object that is only activated when an external thread invokes one of its entry points. First, since multiple threads can invoke the driver concurrently, it must take care to synchronise the invocations to avoid race conditions. Second, since the driver does not have its own thread of control, it cannot rely on programming-language constructs to maintain its control flow and instead records its execution state using state variables. Both issues make the logic of the driver difficult to implement and even harder to understand and modify.

In the active driver architecture, every driver runs in the context of its own thread. Communication between the driver thread and other OS threads occurs via message passing. The driver-OS interface consists of a set of *mailboxes* where each mailbox is used for a particular type of message. The OS sends I/O requests and interrupt notifications to the driver by placing them in appropriate mailboxes. The driver receives a message by performing a blocking wait on one or more mailboxes. The driver notifies the OS about a completed request via a reply mailbox. A message can carry a payload consisting of one or more arguments. The number and types of arguments is determined by the message type.

An active driver is structured as a sequential program, with the order in which the driver handles OS requests explicitly defined in the structure of the program. In addition, since the driver handles all requests in the context of its own thread, it does not have to worry about thread synchronisation.

An active driver can register several interfaces with the OS. For each interface, the OS creates a set of mailboxes for communication with the driver.

Support for active drivers can be added to an existing OS kernel. To this end, the OS must be extended with interface adapters that perform the conversion between native driver interfaces supported by the OS and message-based interfaces implemented by the driver.

The driver exchanges messages with the OS via `EMIT` and `AWAIT` primitives. The `EMIT` function places a message in a mailbox. The sending thread continues without blocking. The `AWAIT` function takes references to one or more mailboxes. If there is a message queued at one of these mailboxes, `AWAIT` returns immediately. Otherwise, it blocks until a message arrives to one of the mailboxes. In either case, it returns a reference to the mailbox containing the message. A mailbox can queue multiple messages. `AWAIT` always dequeues exactly one message, which is accessible via a pointer in the returned mailbox.

We illustrate how the active driver architecture facilitates static analysis of device drivers using a fragment of active driver code that matches the example from Section 1:

```
1 mb = AWAIT(suspend, remove, ...);
2 if (mb == suspend) {
3     ...
4     free(p);
5     mb = AWAIT(resume);
6     ...
7 } else if (mb == remove) {
8     p->data = 0; ...
9 }
```

Here, `suspend`, `resume`, and `remove` are pointers to driver mailboxes. In line 1 the driver waits for both suspend and remove requests. After receiving a suspend request in line 2, the driver deallocates pointer `p` and waits for a resume request in line 5.

This implementation has an equivalent bug to the one found in the original, passive, version of the driver: the driver does not handle remove requests in the suspended state. A correct implementation must wait on both `resume` and `remove` mailboxes in line 5. Otherwise the driver deadlocks if `remove` rather that `resume` arrives while it is blocked at line 5.

The key difference between the passive and the active versions of the driver is that the active version better captures the programmer's intention and achieves cleaner separation of control and data flows. In this version, the bug is evident in the control flow of the driver and can be discovered without resorting to pointer analysis. Upon discovering the bug, the driver developer adds the `remove` message to the `AWAIT` call in line 5 and implements code to handle this message. It is clear from the control flow of the driver that this code is invoked in the suspended state and therefore it is likely to differ from the `remove` handler for the full-power state in line 8.

As can be seen from the above example, active driver code is more verbose compared to passive drivers. This results in a small increase in the code size for a complete driver (see Section 5). We believe that this overhead is justified by the improved clarity of active drivers.

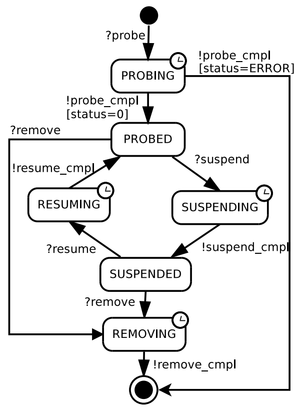

Figure 1: PCI protocol fragment.

3. SPECIFYING DRIVER PROTOCOLS

In order to enable formal analysis of the driver-OS interaction, we associate a behavioural protocol with each active driver interface. The protocol specifies messages from the OS that the driver must be prepared to handle in every state as well as messages that the driver is allowed to send to the OS. Driver protocols are defined by the I/O framework designer and are generic in the sense that every driver that implements the given interface must comply with the associated protocol.

We specify driver protocols in the form of finite state machines (FSMs) where every state transition corresponds to a message sent or received by the driver. The protocol state machine conceptually runs in parallel with the driver: whenever the driver sends or receives a message that belongs to the given protocol, this triggers a matching state transition in the protocol state machine.

The use of FSMs to formalise driver behaviour was proposed in the Dingo [9] driver framework. While Dingo aims to make driver protocols easier to understand and to check at runtime, our focus is on static verification. Therefore we use a subset of the Dingo specification language that lends itself naturally to static analysis.

Figure 1 shows a simplified version of the protocol that must be implemented by any driver for a PCI-based device. This protocol describes the handling of initialisation, shutdown, and power management requests by the driver. Each protocol state transition is labelled by the name of the mailbox through which the driver sends or receives a message, with exclamation marks (“!”) denoting send transitions and question marks (“?”) denoting receive transitions. The label can also contain an optional guard condition in square brackets, which constrains values of message arguments.

According to Figure 1, in the initial state the driver must wait for a `probe` message from the OS. In response to this message, the driver initialises the device and notifies the OS about the completion of this operation via a `probe_cmpl` message. If initialisation is successful (the `status` argument of `probe_cmpl` message is 0), the protocol state machine moves to the `PROBED` state; otherwise it reaches its final state. The remaining protocol state transitions in Figure 1 can be interpreted analogously.

In some states of the protocol the OS is waiting for the driver to complete a request. The driver cannot remain in such a state indefinitely, but must eventually leave the state by sending a response message to the OS. Such states are called *timed* states and are labelled with the clock symbol in Figure 1.

We use the Statecharts [6] syntax to achieve compact representation of complex protocol state machines. Statecharts organise states into a hierarchy, where several primitive states can be clustered into a super-state. Super-states can be composed sequentially or in parallel. The latter is useful for specifying several independent activities within a protocol, e.g. sending and receiving of network packets.

In order to enable automatic verification of drivers' protocol compliance, we define a set of rules that must be followed by any driver that implements a protocol:

1. *EMIT:* The driver is allowed to emit a message to a mailbox if and only if this message triggers a valid state transition in the protocol state machine.
2. *AWAIT:* Whenever the driver performs an `AWAIT` operation, it must wait on all incoming mailboxes enabled in the current state of the protocol.
3. *Timed:* The driver must not remain in a timed state forever.
4. *Termination:* When the main driver function returns, the protocol state machine must be in a final state.

The first rule forbids the driver to send a message that the OS does not expect in the current state. The second rule prevents the driver from waiting for only a subset of enabled incoming messages. Violation of this rule can lead to a deadlock, as shown in Section 2. The third rule forces the driver to respond to all OS requests by eventually exiting each timed state of the protocol. Finally, the fourth rule makes sure that the driver does not terminate leaving some of its protocols in intermediate states.

Rule 2 is too restrictive in practice. It requires the driver to wait for all enabled messages of all its protocols. As a result, the driver is unable to deliberately delay handling of some of the messages, which is useful for instance if the driver wants to complete the current request before checking for more requests from the OS. In order to enable delayed message handling without sacrificing correctness, we use a relaxed version of rule 2 that allows the driver to wait for a subset of enabled protocol messages as long as one of these messages is guaranteed to be received eventually in the current state. The protocol designer must mark such safe sets of messages in the protocol state machine.

4. VERIFYING DRIVER PROTOCOLS

In order to automatically verify driver protocols, we must first convert them into a format understood by verification tools. Most verification tools for C handle properties expressed as source code assertions. Therefore we encode rules 1 and 2 into assertions embedded in modified versions of `EMIT` and `AWAIT` used for verification. These modified functions keep track of the state of the protocol and check that each `EMIT` and `AWAIT` call issued by the driver complies with rules 1 and 2. At the moment, these checks must be written manually based on protocol specifications. Automating this task is part of ongoing work.

In order to verify rule 4, we implement a wrapper function around the driver's main function, which checks that upon return from main the protocol state machine is in one of its final states.

The above approach to driver verification does not require modifications to driver source code. Furthermore, it verifies drivers compositionally, one protocol at a time.

The main limitation of assertion-based verification is that it does not handle liveness properties, i.e. properties whose violation cannot be demonstrated using a finite trace of the program. Rule 3 falls into this category. It requires the driver to eventually respond to any OS request. This rule cannot be violated in any finite sequence of steps; however an infinite run of the driver can violate it.

Liveness properties can be formalised using temporal logic and verified with the help of a temporal logic model checker [5]. Results presented in this paper were obtained using the SatAbs model checker, which only supports assertion-based verification. Therefore we did not verify rule 3 in our experiments. Further work on the project will address this limitation.

Speeding up verification We have experimented with active driver protocol verification using SatAbs [3], a counterexample-guided abstraction refinement (CEGAR) model checker for C.

Although manual analysis of driver code confirms that the control structure of active drivers is largely independent of the data path and therefore can in principle be verified efficiently, our initial experiments showed poor verification performance, with most verification runs not finishing within reasonable time. Analysis of performance issues led to an improved encoding of protocol state machines and tuning of internal SatAbs heuristics.

First, we discovered that verifying a complete protocol in one go leads to overly complex abstractions. As the model checker tries to verify various assertions that encode the protocol, it adds new predicates to the abstract model of the driver. While the number of predicates involved in verifying an individual protocol state transition is typically small, the overall number of predicates generated with this approach overwhelms the model checker.

We address the problem by decomposing each driver protocol state machine into a set of much simpler protocols. The decomposition is equivalent to the original protocol, i.e. the driver satisfies the original protocol if and only if it satisfies each protocol in the decomposition. In our experience, even complex driver protocols allow decomposition into simple protocols with no more than 4 states and only a few transitions.

At the moment, we perform the decomposition manually using the following heuristic rule: every protocol in the decomposition must capture a single constraint on the driver behaviour. One example of such a rule that can be extracted from the PCI protocol (Figure 1) is that after sending a `probe_complete` message the driver must be prepared to handle `remove` and `suspend` messages from the OS. Protocol decomposition only needs to be performed once for a class of device drivers. Nevertheless it is a tedious and error-prone task; therefore we plan to automate it in the future.

Verifying each protocol in the decomposition requires only a small subset of predicates involved in checking the monolithic protocol, leading to exponentially faster verification. By checking each individual protocol in parallel, the monolithic protocol can be verified in the time required by the slowest protocol in the decomposition.

Second, we found that in analysing a spurious counterexample, predicates extracted from program points close to the failed assertion tended to be useful in allowing verification to progress, while predicates less close to the error were often irrelevant to the property being checked. Thus, we added to SatAbs a feature to introduce no more than a user-specified number of new predicates (usually between 1 and 4) during each CEGAR iteration, favouring predicates derived from program points close to the failed assertion.

Third, we noticed that by default SatAbs introduced a large number of pointer-related predicates. Most of these predicates do not affect driver's control flow and are therefore irrelevant to verifying driver protocols. Thus we added to SatAbs a heuristic to favour non-pointer predicates, that is predicates which are not equality tests between pointer expressions, when adding new predicates.

5. EVALUATION

In order to evaluate the active driver architecture and its impact on driver verification, we specified and implemented protocols for several classes of device drivers, including Ethernet, SCSI, and ATA driver protocols. We based our protocol specifications on equivalent Linux driver interfaces, to simplify porting of existing Linux drivers to the active architecture. The drivers we ported are the RTL8169 Ethernet controller driver, the generic SCSI-to-ATA converter driver (`libata`), and the AHCI SATA controller driver.

Our initial verification results are based on the AHCI controller driver. We chose this driver due to the extreme complexity of its protocol. The Linux ATA protocol that this driver implements emerged as a result of splitting a monolithic ATA controller driver into a device-independent layer (`libata`) and device-specific drivers for various ATA controllers. The two layers interact via numerous nested callbacks most of which can only be invoked in a specific context. Our specification of the ATA protocol state machine consists of 39 states and 61 transitions. We decomposed this protocol into an equivalent set of 22 protocols, each consisting of 2 to 4 states. The native Linux AHCI driver contains 2268 lines of C code; the equivalent active driver is 2427 loc.

Using the modified version of SatAbs described in the previous section, we were able to verify all safety properties of ATA and PCI protocols. The longest checker took 12 hours.

In all cases, analysis of SatAbs output revealed that most predicates required to verify driver protocols were control-flow related, with only a small number of data-related predicates. This shows that the active driver architecture effectively separates the control logic of the driver responsible for interaction with the OS from its data flow, which enables efficient verification of driver protocols.

We evaluate the performance overhead of active drivers by comparing the performance of the native Linux driver and the active driver for the AHCI controller using the `iozone` benchmark suite running on a system with a 2.33GHz Intel Core 2 Duo CPU with one enabled core, Marvell 88SE9123 PCIe 2.0 SATA controller, and WD Caviar SATA-II 7200 RPM hard disk. While both drivers achieved the same I/O throughput on all tests, the active driver's CPU utilisation (measured with oprofile) was slightly higher (Figure 2). This overhead is due to the higher cost of message-based communication compared to direct function calls used by the native driver. It can be reduced using an optimised implementation of the message passing mechanism and improved protocol design. Our ATA driver protocol, based on the equivalent

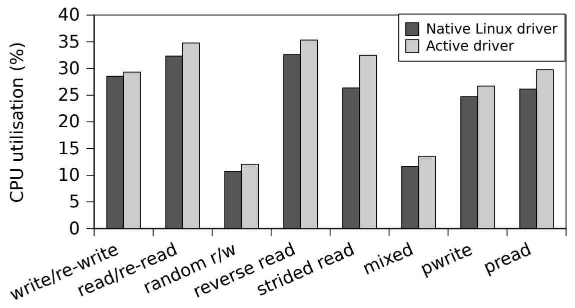

Figure 2: Native vs. active driver performance on the iozone benchmark.

Linux interface, requires 10 messages for each I/O operation. A clean-slate redesign of this protocol would involve much fewer messages.

6. RELATED WORK

Several previous systems, including Singularity [4], RMoX [2], and Dingo [9], implement variations of the active device driver architecture. Singularity and RMoX also support static verification of driver protocols. To this end, they rely on specialised language features and radically different OS architectures.

Active drivers were introduced as an OS and language-independent concept in [10], with the goal of simplifying driver development and avoiding common types of driver bugs. In the present work we demonstrate that active drivers also facilitate static verification of driver-OS protocols for drivers written in C and for conventional OSes.

Tools like SLAM [1], SatAbs [11], and Coccinelle [8] have been used for static analysis of Windows and Linux device drivers. As discussed in Section 1, the power of these tools is limited by the conventional driver architecture. In this work we demonstrate that the same tools can be used to verify complete driver-OS protocols for active drivers.

7. CONCLUSION

We argued that the effectiveness of automatic verification tools in finding driver bugs can be increased with the help of an improved device driver architecture where interactions with the OS are explicit in the source code of the driver. We presented such an architecture and showed that it enables the use of existing model checking tools to verify properties that are hard or impossible to check on conventional drivers.

Acknowledgement

We are grateful to Michael Tautschnig for assistance with the use of SatAbs.

8. REFERENCES

[1] 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 *1st EuroSys Conf.*, pages 73–85, Leuven, Belgium, Apr 2006.

[2] F. Barnes and C. Ritson. Checking process-oriented operating system behaviour using CSP and refinement. *Operat. Syst. Rev.*, 43(4):45–49, Oct 2009.

[3] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[4] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *1st EuroSys Conf.*, pages 177–190, Leuven, Belgium, Apr 2006.

[5] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna — A Static Model Checker. In *11th FMICS*, pages 297–300, Bonn, Germany, Aug 2006.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, Jun 1987.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *14th Int. Conf. Comp. Aided Verification*, pages 526–538, Copenhagen, Denmark, 2002.

[8] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: ten years later. In *16th ASPLOS*, pages 305–318, Newport Beach, CA, USA, 2011.

[9] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *4th EuroSys Conf.*, Nuremberg, Germany, Apr 2009.

[10] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *1st APSys*, pages 25–30, New Delhi, India, Aug 2010.

[11] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *22nd ASE*, pages 501–504, Atlanta, Georgia, USA, 2007.