Active Device Drivers

Sidney Amani[‡§] Peter Chubb[‡§] Alastair F. Donaldson[¶]
Alexander Legg[‡] Leonid Ryzhyk[‡§] Yanjin Zhu[‡§]
[‡]*NICTA* [§]*University of New South Wales* [¶]*Imperial College London*
sidney.amani@nicta.com.au

Abstract

We develop a practical solution to the problem of automatic verification of the interface between device drivers and the operating system. Our solution relies on a combination of improved driver architecture and verification tools. Unlike previous proposals for verification-friendly drivers, our driver development and verification methodology supports drivers written in C and can be implemented in any existing OS. Our Linux-based evaluation shows that this methodology amplifies the power of existing model checking tools in detecting driver bugs, making it possible to verify properties that are beyond the reach of traditional techniques.

1 Introduction

Faulty device drivers are a major source of operating system (OS) failures [12, 6]. Recent studies of Windows and Linux drivers show that over a third of driver bugs result from the complex interface between driver and OS [19, 2]. The OS defines complex rules on the ordering and arguments of driver invocations, rules that often are neither well documented nor are stable across OS releases. Worse, the OS can invoke driver functions from multiple concurrent threads, and so driver developers must implement complex synchronisation logic to avoid races and deadlocks.

Automatic verification has proved useful in detecting OS interface violations in device drivers. Research on automatic driver verification has followed two main avenues. The first avenue, represented by tools like SLAM [2], Terminator [8], SATABS [7], and Blast [13] focuses on verifying existing Windows and Linux drivers. Despite significant effort invested in improving these tools, they remain limited in the complexity of properties that can be efficiently verified without generating a large number of false positives.

The second line of research, represented by the Singularity [10] and RMoX [3] OSs, focuses on making drivers more amenable to automatic verification via improved software architecture. In this architecture each driver has its own thread and communicates with the OS using message passing, which makes the driver control flow and its interactions with the OS easier to understand and analyse. We refer to such drivers as *active drivers*, in contrast to conventional, *passive*, drivers that are structured as collections of entry points invoked by OS threads.

Singularity and RMoX rely on OS and language support for improved verifiability. They are both process-based OSs, where all system components (not just device drivers) communicate via message passing. Driver development requires getting to grips with new programming languages that provide first-class support for message-based communication and formal specification and verification of communication protocols.

In this paper we show that the benefits of active drivers can be achieved while writing drivers in familiar C for a conventional OS. To this end, we present an implementation of an active driver framework for the Linux kernel. The framework does not require any modifications to existing kernel code and allows active drivers to co-exist with conventional drivers.

We develop a new verification method that enables efficient, automatic checking of active driver protocols. Our method leverages existing verification tools for C, extended with several novel optimisations geared towards making active driver verification tractable. Like other existing automatic verification techniques, the method is not complete—it helps to find bugs, but does not guarantee their absence.

Through experiments involving verification of several complex drivers for Linux, we demonstrate that our driver design and verification methodology amplifies the power of verification tools in finding driver bugs. In particular, many properties that are hard or impossible to verify in conventional drivers can be easily checked on active drivers.

2 Passive vs active drivers

In this section we argue that the active driver architecture should be the preferred one for modern OSs. To this end, we discuss the shortcomings of the conventional driver architecture and show how active drivers address these shortcomings.

2.1 Passive drivers

The passive driver architecture supported by all mainstream OSs suffers from two problems that complicate verification of the driver-OS interface: *stack ripping* and *concurrency*.

Stack ripping A passive device driver comprises a collection of entry points invoked by the OS. When writing the driver, the programmer makes assumptions about possible orders in which its entry points are going to be activated. These assumptions are not explicit in the driver source code and can only be discovered by combining the driver with a hand-written model of the OS kernel that systematically generates all possible sequences of driver invocations [2]. This phenomenon, when the control flow of the program is scattered across multiple entry points and cannot be reconstructed from its source code is known as stack ripping [1].

Building accurate OS models has proved difficult, as even OS designers often lack a good understanding of what driver-OS interactions are allowed in practice [2].

Even in the presence of an accurate OS model, analysis of the driver control flow can be tricky. The following code fragment, showing two driver entry points, illustrates the problem:

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

This code incorrectly assumes that the `unplug()` entry point cannot be called after `suspend()` and therefore it is safe to deallocate pointer `p` inside `suspend()`.

The bug can be discovered with the help of an OS model that simulates, among others, the problematic sequence of invocations and by using pointer analysis to detect the use-after-free pattern on pointer `p`. This approach does not scale well, because pointer analysis quickly gets intractable for code involving complex pointer manipulation.

In this example the *root cause* of the bug—the fact that the driver does not expect `unplug()` to happen after `suspend()`—is implicit in the source code of the driver. Instead we can only detect the bug's indirect consequence, e.g., an invalid pointer dereference.

Concurrency The OS can invoke driver entry points from multiple concurrent threads, forcing driver developers to implement complex synchronisation logic to avoid races and deadlocks. To detect concurrency bugs, the OS model used in driver verification must be extended to simulate the multithreaded environment of an OS kernel [20]. Even with such a model, concurrent verification is far less scalable than sequential verification, as thread interleaving leads to dramatic state explosion.

Case study We demonstrate the adverse effects of stack rippping and concurrency on driver verification through a real-world case study involving the Linux driver for the RTL8169 Ethernet controller. We analyse the history of bug fixes made to this driver, and identify those fixes that address OS interface violation bugs, where the driver incorrectly responds to certain sequences of OS requests. We found 12 such bugs. We apply SATABS, a state-of-the-art model checker for C, to detect these bugs. SATABS has been successfully applied to Linux drivers in the past [20]. In this paper, we also use it for analysis of active drivers and, as reported in Section 6, this enables efficient, automatic verification of RTL8169 and other drivers. Using SATABS as a model checker for both active and traditional drivers provides a fair comparison.

Detecting driver bugs with SATABS requires a model of the OS. We built a series of such models of increasing complexity so that each new model reveals additional errors but introduces additional execution traces and is therefore harder to verify. This way we explore the best-case scenario for the passive driver verification methodology: using our knowledge of the error we tune the model for this exact error. In practice more general and hence less efficient models are used in driver verification.

In addition, to make sure that our study is not biased, we optimised our OS models for the best SATABS performance. To this end we analysed spurious counterexamples generated by SATABS and restructured the models to avoid such counterexamples or added static predicates to eliminate the counterexamples whenever possible (see Section 4 for more details on SATABS).

Our initial model is purely sequential. It generates one call to a driver entry point at a time and waits for the invocation to complete before choosing the next entry point to call. By gradually improving this simple model, we were able to find 7 out of 12 bugs within 5 minutes each.

The remaining errors are race conditions that are only triggered by concurrent driver invocations. In order to model concurrency while minimising state explosion, we simulate a limited form of concurrency where driver entry points can be invoked in a nested fashion whenever the driver calls an OS callback. While this simplified concurrency model is powerful enough to trigger the remaining 5 errors, SATABS was only able to find one of them before being interrupted after 12 hours.

This analysis illustrates that (1) building an accurate OS model suitable for driver verification is a difficult task, and (2) an accurate OS model can be prohibitively expensive to verify.

2.2 Active drivers

In contrast to passive drivers, an active driver runs in the context of its own thread or threads. Communication between driver threads and other OS threads occurs via message passing. The OS sends I/O requests and interrupt notifications to the driver using messages. A message can carry a payload consisting of a number of typed arguments, determined by the *message type*. When the driver is ready to receive a message, it performs a blocking wait specifying one or more message types that it can accept in the current state. It notifies the OS about a completed request via a reply message. Hence, the order in which the driver handles and responds to OS requests is defined explicitly in its source code and can be analysed without requiring an OS model and without running into state explosion due to thread interleaving.

Active driver framework We present our instantiation of the active driver architecture. Our design is based on the Dingo active driver framework for Linux [19], improving upon it in two ways. First, Dingo's message passing primitives are implemented as C language extensions. In contrast, our framework supports drivers in pure C. Second, Dingo does not support automatic driver protocol verification.

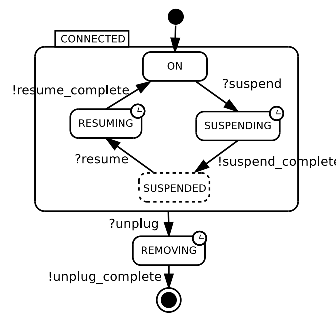
In our framework, the driver-OS interface consists of a set of *mailboxes*, where each mailbox is used for a particular type of message. The driver exchanges messages with the OS via `EMIT` and `AWAIT` primitives, that operate on messages and mailboxes. The `EMIT` function takes a pointer to a mailbox, a message structure, and a list of message arguments. It places the message in the mailbox and returns control to the caller without blocking. The `AWAIT` function takes references to one or more mailboxes and blocks until a message arrives in one of them. It returns a reference to the mailbox containing the message. A mailbox can queue multiple messages. `AWAIT` always dequeues the first message in the mailbox. This message is accessible via a pointer in the returned mailbox.

An active driver can consist of several threads that handle different activities. For example, our active driver for a SATA controller (see Section 5) creates a thread per SATA port. Each driver thread registers one or more message-based interfaces, along with associated protocols, with the OS. This allows us to verify protocol compliance for each thread in isolation. In doing so, we ignore potential race conditions and deadlocks between threads inside the driver. This is acceptable, as our goal is bug finding and not complete formal verification.

Previous research [19] has shown that active drivers can benefit from cooperative thread scheduling. The performance of most drivers is bound by I/O bandwidth rather than CPU speed, therefore they do not require true multiprocessor parallelism. Cooperative scheduling limits the number of possible thread interleavings, making drivers easier to write and simplifying verification of properties involving multiple threads. Our framework supports cooperative scheduling of threads within a driver (however, driver threads are scheduled preemptively with respect to other drivers and the rest of the kernel). In the future the framework can be easily extended to enable preemptive scheduling for those drivers that can take advantage of true parallelism.

```
1  mb=AWAIT(suspend,unplug,..);
2  if (mb==suspend) {
3    dev_suspend();
4    free(p);
5    EMIT(suspend_complete,msg);
6    //Bug! Uncomment to fix
7    mb=AWAIT(resume/*,unplug*/);
8    ...
9  } else if (mb==unplug) {
10   p->data = 0;
11   ...
12 }
```

(a) Faulty code



(b) Protocol

Figure 1: Active driver code with a bug similar to the one in the example in Section 2 and the matching protocol specification.

Example We use an example to illustrate how the active driver architecture facilitates driver development and verification. Figure 1(a) shows a fragment of driver code that matches the example of a driver bug in Section 2.1.

In Figure 1, `suspend`, `unplug`, `suspend_complete`, and `resume` are pointers to driver mailboxes. In line 1 the driver waits for both suspend and unplug requests. After receiving a suspend request (checked by the condition at line 2) the driver puts the device in a low-power mode (line 3), deallocates pointer `p` (line 4) and notifies the OS about completion of the request by sending a message to the `suspend_complete` mailbox (line 5). It then waits for a resume request at line 7.

This implementation has an equivalent bug to the one found in the passive version of the driver: it does not handle hot-unplug notifications after receiving a suspend request. A correct implementation must wait on both `resume` and `unplug` mailboxes at line 7. Otherwise the driver can deadlock waiting for a `resume` message that never arrives.

In the active version of the driver, requests that the driver accepts or ignores in each state are explicitly listed in the driver source code. As a result, the root cause of the bug can be discovered using control flow analysis, without resorting to pointer analysis.

The code in Figure 1(a) is longer than the original passive implementation, because all OS interactions are initiated by the driver explicitly. In our experience, this verbosity makes the logic of the driver easier to follow while having only modest effect on the overall driver size.

3 Specifying driver protocols

This section presents our visual formalism for specifying active driver protocols. The formalism is similar to protocol state machines of Dingo [19] and Singularity [10]; however it provides additional means to capture liveness and fairness constraints, which enable the detection of additional types of driver bugs.

The active driver framework associates a protocol with each driver interface. The protocol specifies legal sequences of messages exchanged by the driver and the OS.

Protocols are defined by the driver framework designer and are generic in the sense that every driver that implements the given interface must comply with the associated protocol. In the case when the active driver framework is implemented within an existing OS, active driver protocols are derived from the existing driver interface supported by the OS. The framework includes wrapper components that perform the translation between the native function-based interface and message-based active driver protocols.

We specify driver protocols using finite state machines (FSMs) with additional liveness and fairness constraints. 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.

Figure 1(b) shows a state machine for the protocol used by the example driver, describing the handling of power management and hot unplug requests. Each protocol state transition is labelled with the name of the mailbox through which the driver sends ('!') or receives ('?') a message.

We represent complex protocol state machines compactly using Statecharts, which organise states into a hierarchy so that several primitive states can be clustered into a super-state. For example, the `CONNECTED` super-state in Figure 1(b) groups four primitive states with a common property that the `?unplug` transition is enabled in all of them.

In some protocol states 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(b).

In order to ensure that the driver does not deadlock in an `AWAIT` statement, the developer must rely on an additional assumption that if the driver waits for all incoming OS messages enabled in the current state, then one of them will eventually arrive. This is a form of *weak fairness* constraint [14] on the OS behaviour, which means that if some event (in this case, arrival of a message) is continuously enabled, it will finally occur. Not all protocol states have the weak fairness property. In the protocol state machine, we show weakly fair states with dashed border. For example, the `SUSPENDED` state in Figure 1b is weakly fair, which guarantees that at least one of `resume` and `unplug` messages will eventually arrive in this state.

A protocol-compliant device driver must obey the following 5 rules.

Rule 1. *(EMIT) The driver is allowed to emit a message to a mailbox iff this message triggers a valid state transition in the protocol state machine.*

The next rule ensures the driver does not ignore an incoming message forever.

Rule 2. *(AWAIT) When in a state where there is an enabled incoming message, the driver must eventually issue an `AWAIT` on the corresponding mailbox or transition into a state where this message is not enabled.*

Rule 3. *(Deadlock-freedom) All `AWAIT` operations eventually terminate.*

To show that Rule 3 holds for an `AWAIT` statement, one must check that at this point in the program *at least one* protocol of the driver is in a weakly fair state and the `AWAIT` waits for all enabled messages of this protocol. This check requires simultaneous consideration of all driver protocols. For efficiency reasons we want to be able to verify one protocol at a time. Therefore, we replace Rule 3 with the weaker rule, which does not guarantee deadlock freedom, but detects most deadlocks in practice:

Rule 3'. *For each protocol of the driver and for each `AWAIT` operation, if the `AWAIT` waits for at least one message of this protocol, it waits for all of its messages enabled in the current state.*

Rule 4. *(Timed) The driver must not remain in a timed state forever.*

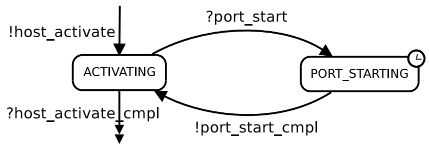


Figure 2: Example of a protocol with a strongly fair transition (shown with a double arrow).

Rule 5. *(Termination) When the main driver function returns, the protocol state machine must be in a final state.*

Note that Rule 5 does not require that every driver run terminates, merely that if it does terminate then all protocols must be in their final states.

Rules 1, 3 and 5 describe *safety* properties, whose violation can be demonstrated by a finite execution trace. Rules 2 and 4 are *liveness* rules, for which counterexamples are infinite runs. Rule 2 is violated by an infinite run which, after a finite number of steps, reaches a point where an incoming message `m` is enabled, and remains enabled in all subsequent steps, but is never waited for by the driver. Rule 4 is violated by an infinite run if, after a finite number of steps, the protocol state machine enters a timed state, and then remains in that timed state in all subsequent steps.

Checking liveness properties often requires making further assumptions on the OS behaviour. Figure 2 illustrates this using a fragment of the SATA driver protocol. According to this protocol, the driver can send a `host_activate` message to the OS, which responds with a `host_activate_cmpl` message. While handling the `host_activate` request, the OS can issue one or more `port_start` requests to the driver. The OS guarantees that the `host_activate_cmpl` message is eventually received if the driver keeps waiting for it infinitely often in the `ACTIVATING` state. This ensures that the driver does not get stuck in the loop between `ACTIVATING` and `PORT_STARTING` states forever. This type of fairness, where some event is guaranteed to happen as long as it is enabled infinitely often (but not necessarily continuously), is called *strong fairness* [14]. The strongly fair transition is marked with double arrows in the state diagram.

4 Verifying driver protocols

Active driver architecture opens the way to more efficient verification of driver protocols. This section outlines the challenges that had to be overcome to achieve this and the resulting verification methodology.

The goal of driver protocol verification is to check whether the driver meets all safety and liveness requirements assuming fair OS behaviour. We use two tools to this end: SATABS [7], geared towards safety analysis, and GOANNA [11], geared towards liveness analysis. These tools provide complementary capabilities that, when combined, enable full verification of driver protocols. Given an active device driver and the set of protocols it implements, we use SATABS to check safety rules 1, 3, and 5 and GOANNA to check liveness rules 2 and 4. This combination works well in practice, yielding a low overall false positive rate.

Our methodology is compatible with other similar tools. We use SATABS and GOANNA because our team is familiar with their internals and has the expertise required to implement novel optimisations to improve performance on active driver verification tasks.

4.1 Checking safety

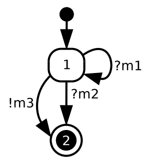
SATABS is an abstraction-refinement based model checker for C and C++ for checking *safety* properties. It is designed to perform best when checking control-flow dominated properties with a small number of data dependencies. Active driver protocol-compliance safety checks fall into this category.

Given a program to verify, SATABS iteratively computes and verifies its finite-state abstraction with respect to a set of predicates over program variables. At each iteration it either terminates (by discovering a bug or proving that the program is correct) or generates a spurious counterexample. In the latter case, the counterexample is analysed by the tool to discover new predicates, used to construct a refined program abstraction. Abstraction and refinement are both fully automatic.

We use a simple driver protocol shown in Figure 3a and a fragment of driver code that implements this protocol in Figure 3b as a running example to illustrate the use of SATABS.

SATABS verifies program properties expressed as source code assertions. We encode rules 1 and 3' as assertions embedded in modified versions of `AWAIT` and `EMIT`. Figure 3c shows the driver code with `AWAIT` and `EMIT` functions encoding Rule 1 inlined. These functions keep track of the protocol state using the global `state` variable. The `AWAIT` function simulates the receiving of a message by randomly selecting one of incoming mailboxes enabled in the current state (line 5) and updating the state variable based on the current state and the message selected. The `assume(0)` statement in line 11 tells SATABS that this branch can never be reached; hence no other messages are allowed by the protocol.

Similarly, the `EMIT` function updates the state variable based on the current state and the message being sent. It contains an assertion that triggers an error when the driver is trying to send a message that is not allowed in the current state. Note that the `m3==m3` tautology in line 16 is a result of inlining the body of `EMIT`, which compares its first argument against `m3`.



(a) Driver protocol

```
mbox_t *m;
m = AWAIT(m1,m2);
if(m==m1) {
  EMIT(m3,msg);
}
```

(b) Driver source code

```
/*initial state*/
int state=1;
mbox_t *m;
//AWAIT{
 m=random_mb(m1,m2);
 if(state==1&&m==m1)
   state=1;
 else
 if(state==1&&m==m2)
   state=2;
 else assume(0);
//}
if(m==m1) {
  //EMIT{
   /*m3 in state 1*/
   if(m3==m3&&state==1)
     state=2;
   else
     assert(0);
  //}
}
```

(c) Driver with `AWAIT` and `EMIT` functions inlined.

```

p1=1;


p2=*;
if(p1&&p2)
  p1=1;
else
if(p1&&!p2)
  p1=0;
else assume(0);

if(p1) {


  if(true&&p1)
    p1=0
  else
    assert(0);

}
```

(d) Abstraction w. r. t. predicates $p1$ and $p2$

```
int state=1;
m=random_mb(m1,m2);
if (state==1&&m==m1) {
  state=1;
  EMIT(m3,msg);
} else if(state==1&&m==m2) {
  state=2;
} else assume(0);
```

(e) Driver code after CFG transformation (with the `EMIT` function body collapsed).

Figure 3: Safety verification example

To verify rule 5, we append to the driver's main function a check to ensure that, if the driver does terminate, the protocol state machine is in a final state.

In our running example, the abstraction refinement loop terminates after discovering predicates $p1 \equiv (state == 1)$ and $p2 \equiv (m == m1)$. The abstraction of the program in Figure 3c with respect to these two predicates is shown in Figure 3d. The abstract program has the same structure as the concrete one; however it only keeps track of the predicate variables, abstracting away the rest of the driver state. Using this pair of predicates (but not any one them separately), SATABS is able to verify that this abstract program can not trigger the assertion; hence the original concrete program is correct with respect to the safety property being checked.

Our preliminary experiments show that straightforward application of SATABS to active drivers results in very long verification times. This is in part due to the complexity of driver protocols being verified and in part because predicate selection heuristics implemented in these tools introduce large numbers of unnecessary predicates, leading to overly complex abstractions. The problem is not unique to SATABS. Our preliminary experience with SLAM, another state-of-the-art abstraction-refinement tool produced similar results.

We describe several novel strategies that exploit the properties of active drivers to make their safety verification feasible. We believe that these techniques will also be useful in other software protocol verification tasks.

Protocol decomposition The abstraction-refinement technique is highly sensitive to the size of the property being checked. Complex properties require many predicates. Since verification time grows exponentially with the number of predicates, it is beneficial to decompose complex properties into simple ones that can be verified independently.

We decompose each driver protocol state machine into a set of much simpler subprotocols as a preprocessing step. Every subprotocol captures a simple rule derived from the main protocol.

For instance, given the protocol in Figure 1(b), we can define, among others, the following rules, related to the `unplug` message: (1) the driver must be prepared to handle the `unplug` message at any time; (2) the driver must respond to the `unplug` message by sending `unplug_complete` to the OS. These rules are captured by the two subprotocol state machines at the far left in Figure 4. The next rule (the third protocol from the left) describes the occurrence of the `suspend` message: `suspend` can arrive in the initial state, is reenabled by the `resume_complete` message, and is permanently disabled by the `unplug` message.

The complete decomposition consists of six subprotocols shown in Figure 4. This decomposition is constructed in such a way that each subprotocol describes the occurrence of a single type of message, shown in bold italics in the diagram. Any other message is allowed to occur in any state of the subprotocol, as it is constrained by a separate subprotocol.

The decomposition has an important property that the driver satisfies safety constraints of the original protocol if and only if it does so for each protocol in the decomposition. Formally, this is achieved by ensuring that parallel product of subprotocol state machines is equivalent to the original protocol state machine.

In our experience, even complex driver protocols allow decomposition into simple subprotocols with no more than four states and only a few transitions. Verifying each subprotocol requires a small subset of predicates involved in checking the monolithic protocol, leading to exponentially faster verification.

Automatically provide key predicates One way to speed-up the abstraction-refinement algorithm is to seed it with a small set of key predicates that allow refuting large families of counterexamples. Guessing such key predicates *in general* is extremely difficult. In case of active driver verification, an important class of key predicates can be provided to SATABS automatically.

As illustrated in Figure 3c, when checking a driver protocol, we introduce a global variable that keeps track of protocol state. In verifying the protocol, SATABS eventually discovers predicates over this variable of the form `(state==1)`, `(state==2)`, ..., one for each state of the protocol (e.g., predicate $p1$ in Figure 3d). These predicates are important to establishing the correspondence between the driver control flow and the protocol state machine. We therefore provide these predicates to SATABS on startup, which accelerates verification significantly.

Control-flow transformations We found that it often takes SATABS many iterations to correlate dependent program branches. For example, the else-if branch in line 9 and the if-branch in line 13 in Figure 3c cannot both be taken in the same run of the driver. SATABS does not know about this correlation initially, leading to a spurious counterexample trace that takes both branches and triggers the assertion in line 19. This counterexample can be refuted using predicate $p2 \equiv (m == m1)$. In practice, however, the driver can contain hundreds lines of code between the condition in line 13 and the failed assertion. This leads to a large number of similar spurious counterexample traces. In analysing these traces, SATABS introduces many predicates that only refute a subset of these counterexamples before discovering $p2$, which allows refuting all of them.

This problem frequently occurs in active drivers when the driver `AWAIT`s on multiple mailboxes and then checks the returned value.

To remedy the problem, we have implemented a novel control-flow graph transformation that uses static analysis to identify correlated branches, and merges them. The analysis identifies, through inspecting the use of the `AWAIT` function, where to apply the transformation. Then infeasible paths through each candidate region are identified by generating Boolean satisfiability queries which are discharged to a SAT solver. The CFG region is then rewritten to eliminate infeasible paths. The effect of the rewriting on the CFG is shown in Figure 5. Figure 3e shows the driver after the transformation. This version of the driver can be verified using only predicate `p1`.

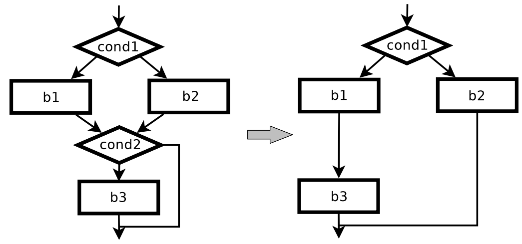


Figure 5: CFG transformation example.

This technique effectively avoids the expensive search for additional predicates using much cheaper static program analysis. In our experiments, SATABS performs orders of magnitude more effectively over the new program structure, being able to quickly infer key predicates that could previously only be inferred after many abstraction refinement iterations and the inference of many redundant predicates.

4.2 Checking liveness

As SATABS is restricted to analysis of safety properties, the GOANNA tool comes into play for analysis of liveness properties. GOANNA is a C and C++ bug finding tool that supports user-defined rules written in the CTL temporal logic [9], which allows natural specification of safety and liveness properties.

Unlike SATABS, GOANNA is intended as a fast compile-time checker and therefore does not perform data-flow analysis.

Properties to be checked for each protocol are extracted from the protocol specification. In particular, we apply the *AWAIT* rule to every incoming mailbox and the *Timed* rule to every timed state of the protocol.

Describing a temporal property using the GOANNA specification language involves two steps. First, we identify a set of important program events related to the property being verified, such as sending and receiving of messages. We use syntactic pattern matching to label program locations that correspond to these events. Second, we encode the property to be checked as a temporal logic formula in a dialect of CTL, defined over events identified at the previous step. Due to limited space, we omit the details of this encoding.

4.3 Automation

Verifying active driver protocols requires transforming protocol state machines into a representation supported by the verification tools. These transformations include: protocol decomposition, encoding of safety properties into C assertions, and encoding of liveness properties into the GOANNA specification language.

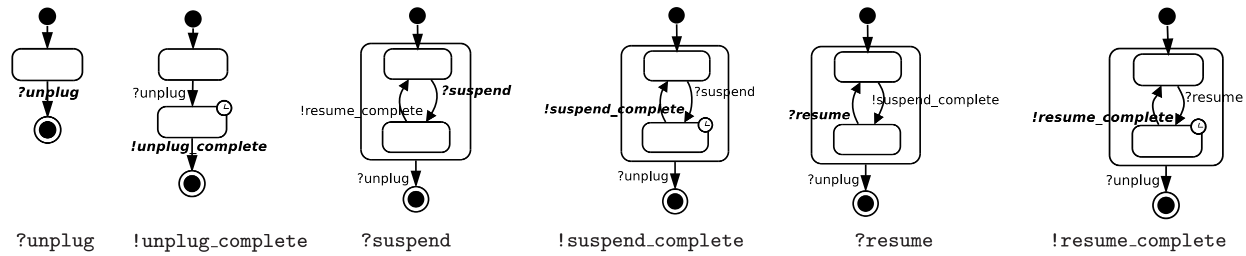


Figure 4: Decomposition of the protocol in Figure 1(b).

These transformations can be automated in a straightforward way, but their automation will require significant additional implementation effort. Because our resources are limited, in all experiments cited in this paper protocol transformations were performed manually, providing a large proof-of-concept for our approach.

5 Implementation

We implemented the active driver framework along with several active device drivers in Linux 2.6.38. The framework consists of loadable kernel modules and does not require any changes to other kernel components.

The generic part of the framework shared by all active drivers provides support for scheduling and message passing. It implements the *cooperative domain* abstraction, which constitutes a collection of cooperatively scheduled kernel threads hosting an active driver. Threads inside the domain communicate with the kernel via a shared message queue. The framework guarantees that at most one thread in the domain is runnable at any time. The thread keeps executing until it blocks in the `AWAIT` function. `AWAIT` checks whether there is a message available in one of the mailboxes specified by the caller and, if so, returns without blocking. Otherwise it calls the thread dispatcher function, which finds a thread for which a message has arrived. The dispatcher uses the kernel scheduler interface to suspend the current thread and make the new thread runnable. In the future this design can be optimised by implementing native support for light-weight threads in the kernel.

`EMIT` and `AWAIT` functions do not perform memory allocation and therefore never fail. This simplifies driver development, as the driver does not need to implement error handling logic for each invocation of these ubiquitous operations. On the other hand this means that the driver is responsible for allocating messages sent to the OS and deallocating messages received from the OS. By design of driver protocols, most mailboxes can contain at most one message, since the sender can only emit a new message to the mailbox after receiving a completion notification for the previous request. Such messages can be pre-allocated statically.

Interrupt handling in active drivers is separated into top and bottom halves. The driver registers with the framework a top-half function that is invoked by the kernel in the primary interrupt context (outside the cooperative domain). A typical top-half handler reads the interrupt status register, acknowledges the interrupt in the device, and sends an IRQ message to the driver. The actual interrupt handling happens inside the cooperative domain in the context of the driver thread that receives the IRQ message. IRQ delivery latency can be minimised by queueing interrupt messages at the head of the message queue; alternatively interrupts can be queued as normal messages, which avoids interrupt livelock an ensures fair scheduling of interrupts with respect to other driver tasks.

In addition to the generic functionality described above, the active driver framework defines protocols for supported classes of drivers and provides wrappers to perform the translation between the Linux driver interface and message-based active driver protocols. Wrappers enable conventional and active drivers to co-exist within the kernel.

Active driver protocols are derived from the corresponding Linux interfaces by replacing every interface function with a message or a pair of request/response messages. While multiple function calls can occur concurrently, messages are serialised by the wrapper.

Since Linux lacks a formal or informal specification of driver interfaces, deriving protocol state machines often required tedious inspection of the kernel source. On the positive side, we found that, compared to building an OS model as a C program, state machines provide a natural way to capture protocol constraints and are useful not only for automatic verification, but also as documentation for driver developers.

Table 1 lists protocols we have specified and implemented wrappers for. For each protocol, it gives the number of protocol states and transitions, and the number of subprotocols in its decomposition (see Section 4.1). The PCI protocol provides access to OS services used to manage a device on a PCI bus, including configuration, hot plugging, and power management. Ethernet, SATA, and SCSI protocols describe services that network and stor-

protocol	#states	#transitions	#subprotocols
PCI	13	41	11
Ethernet	17	36	6
SCSI	42	67	-
SATA	39	70	22
DAI	8	20	6

Table 1: Implemented active driver protocols.

driver	supported protocols	LOC (native)	LOC (active)
RTL8169 1Gb Eth	PCI, Ethernet	4,220	4,317
ATA framework	SCSI, SATA(client)	9,287	9,718
AHCI SATA	PCI, SATA	2,268	2,487
OMAP DAI audio	DAI	583	705

Table 2: Active device driver case studies, protocols that each driver implements, and the size of the native Linux and active versions of the driver in lines of code (LOC) measured using `sloccount`.

age drivers provide to the OS. Finally, the Digital Audio Interface (DAI) protocol is part of the Linux ALSA audio framework and must be implemented by drivers for audio controller devices. We do not give the size of decomposition of the SCSI protocol as we did not verify this protocol in our experiments.

Table 2 lists active device drivers we have implemented along with protocols that each driver supports. All four drivers control common types of devices found in virtually every computer system. These drivers were obtained by porting native Linux drivers to the active architecture, which allows direct comparison of their performance and verifiability against conventional drivers.

Our first active driver case study is based on the RTL8169 Gigabit Ethernet controller driver, which is interesting due to its stringent performance requirements. The second case study is based on two drivers layered on top of each other in the Linux storage stack: the ATA framework driver and the AHCI Serial ATA controller driver. This case study demonstrates that active drivers support the stacked driver architecture found in most OSs. Our final case study demonstrates that the active driver architecture is suitable for the embedded space using an audio controller driver for the OMAP SoC.

6 Evaluation

6.1 Verification

We applied the verification methodology described in Section 4 to RTL8169, AHCI, and OMAP DAI drivers. We did not verify the ATA framework driver due to time constraints. Verification was performed on machines with 2GHz quad-core Intel Xeon CPUs.

driver	avg(max) time(minutes)	avg(max) refinements	avg(max) predicates
RTL8169	29 (103)	3 (7)	3 (8)
AHCI	123 (335)	2 (6)	2 (19)
OMAP DAI	5 (13)	2 (5)	2 (0)

Table 3: Statistics for checking safety properties using SATABS.

Verification using SATABS and GOANNA For each of the three drivers we were able to verify all safety properties defined by their protocols using SATABS with zero false positives. Table 3 shows statistics for verifying safety properties using SATABS. For each driver, it gives average and maximum time, the number of abstraction refinement loop iterations and the number of predicates required for verification to succeed, across all subprotocols for each driver. The number of predicates reflects predicates discovered dynamically by the abstraction refinement loop and does not include candidate predicates with which SATABS is initialised (see Section 4.1).

The small number of predicates involved in checking these properties indicates that the control skeleton of an active driver responsible for interaction with the OS has few data dependencies. This confirms that the active driver architecture achieves its goal of making the driver-OS interface amenable to efficient automatic verification. At the same time, the fact that several refinements are required in most cases indicates that the power of the abstraction refinement method is necessary to avoid false positives when checking safety.

Despite the small number of predicates required, verification times are relatively high for our benchmarks. This is due to the large size of our drivers, and the fact that SMV, the model checker used by SATABS, was not designed primarily for model checking boolean programs. We experimented with the BOOM model checker [4], which is geared towards boolean program verification. While in many cases verification using BOOM was several times faster than with SMV, we did not use it in our final experiments due to stability issues.

All optimisations described in Section 4.1 proved essential to making verification tractable. Disabling any one of them led to overly large abstractions that could not be analysed within reasonable time.

We used GOANNA to verify liveness properties of drivers as explained in Section 4.2. GOANNA performs a less precise analysis than SATABS and is therefore much faster. It verified all drivers in less than 1 minute while generating 8 false positives.

These results demonstrate that active drivers' protocol compliance can be verified using existing tools. At the same time they suggest that an optimal combination of accuracy and verification time requires a trade-off between full-blown predicate abstraction of SATABS and purely syntactic analysis of GOANNA.

Comparison with conventional driver verification An important remaining question is: how does our verification methodology compare against the conventional approach to driver verification in terms of its ability to detect real driver bugs? In Section 2 we showed, using the Linux RTL8169 driver case study, that the scalability of the traditional verification methodology for passive drivers is limited by the complexity of building an accurate OS model and the state explosion resulting from concurrency.

We carried out an equivalent case study on the active version of the RTL8169 driver. To this end, we simulated the 12 OS protocol violations found in the native Linux driver in the active driver. To reproduce concurrency-related errors in the active driver we considered message sequences that simulate thread interleavings of the conventional driver. We were able to detect each of the 12 protocol violation bugs within 3 minutes per bug.

This result confirms that the active driver architecture along with the verification methodology presented above lead to device drivers that are more amenable to automatic verification than passive drivers.

Comparison with SLAM SLAM [2] is a state-of-the-art driver verification tool used in industry to find bugs in Windows device drivers. It defines hundreds of safety rules that capture common driver safety errors. Combined with the Terminator [8] liveness checker, it can also detect liveness errors.

Analysis of the SLAM rule database shows that most of the rules are similar to subprotocols obtained after decomposition of active driver protocols (see e.g. Figure 4). They describe simple properties such as “event A must happen after event B and before event C”. With the exception of rules that are not applicable to active drivers, such as spinlock usage rules, all of SLAM rules can be defined as part of active driver protocols.

On the other hand, not every active driver protocol rule can be defined for conventional drivers. Rules that require the driver to wait for certain protocol messages in a state do not have analogues in the SLAM rule database. This is not a limitation of SLAM, but rather a conceptual limitation of the passive driver architecture, as discussed in Section 2. Out of the 45 subprotocols in Table 1, 26 subprotocols encode such rules and thus would not be amenable to SLAM-based verification.

6.2 Performance

Microbenchmarks The performance of active drivers depends on the overhead introduced by thread switching and message passing. We measure this overhead on a machine with 2 quad-core 1.5GHz Xeon CPUs.

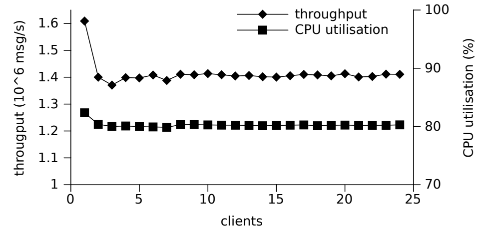


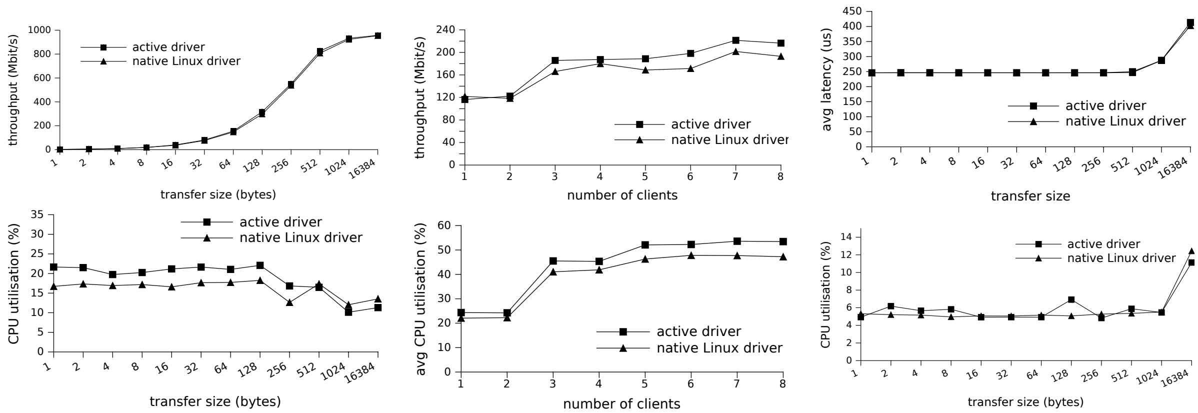
Figure 6: Message throughput and aggregate CPU utilisation over 8 CPUs for varying number of clients.

In the first set of experiments, we measure the communication throughput by sending a stream of messages from a normal kernel thread to a thread inside a cooperative domain. Messages are buffered in the message queue and delivered in batches when the cooperative domain is activated by the scheduler. This setup simulates streaming of network packets through an Ethernet driver. The achieved throughput is $2 \cdot 10^6$ messages/s (500 ns/message) with both threads running on the same core and $1.2 \cdot 10^6$ messages/s (800 ns/message) with the two threads assigned to different cores on the same chip.

Second, we run the same experiment with varying number of kernel threads distributed across available CPU cores (without enforcing CPU affinity), with each Linux thread communicating with the cooperative thread through a separate mailbox. As shown in Figure 6, we do not observe any noticeable degradation of the throughput or CPU utilisation as the number of clients contending to communicate with the single server thread increases (the drop between one and two client threads is due to the higher cost of inter-CPU communication). This shows that our implementation of message queueing scales well with the number of clients.

Third, we measure the communication latency between a Linux thread and an active driver thread running on the same CPU by bouncing a message between them in a ping-pong fashion. The average measured roundtrip latency is 1.8 μs. For comparison, the roundtrip latency of a Gigabit network link is at least 55μs.

Macrobenchmarks We compare the performance of the active RTL8169 Ethernet controller driver against equivalent native Linux driver using the Netperf benchmark suite on a 2.9GHz quad-core Intel Core i7 machine. Results of the comparison are shown in Figure 7. In the first set of experiments we send a stream of UDP packets from the client to the host machine, measuring achieved throughput (using Netperf) and CPU utilisation (using `oprofile`) for different payload sizes. The client machine is equipped with a 2GHz AMD Opteron CPU and a Broadcom NetXtreme BCM5704 NIC. The active driver achieved the same throughput as the native Linux driver on all packet sizes, while using 20% more CPU in the worst case (Figure 7(a)).



(a) UDP throughput for varying packet sizes for a single client. The top graph shows achieved throughput; the bottom graph shows CPU utilisation.

(b) UDP throughput for multiple clients (packet size=64 bytes). The top graph shows aggregate throughput; the bottom graph shows average CPU utilisation across 8 cores.

(c) UDP latency for varying packet sizes for a single client. The top graph shows average round-trip latency; the bottom graph show CPU utilisation.

Figure 7: Performance of the RTL8169 Ethernet driver measured with Netperf.

In the second set of experiments, we fix payload size to 64 bytes and vary the number of clients generating UDP traffic to the host between 1 and 8. The clients are distributed across four 2GHz Intel Celeron machines with an Intel PRO/1000 MT NIC. The results (Figure 7(b)) show that the active driver sustains up to 10% higher throughput while using proportionally more CPU. Further analysis revealed that the throughput improvement is due to slightly higher IRQ latency, which allows the driver to handle more packets per interrupt, leading to lower packet loss rate.

The third set of experiments measures the round trip communication latency between the host and a remote client with 2GHz AMD Opteron and NetXtreme BCM5704 NIC. Figure 7(c) shows that the latency introduced by message passing is completely masked by the network latency in these experiments.

We evaluate the performance of the AHCI SATA controller driver and the ATA framework driver using the `iozone` benchmark suite running on a system with a 2.33GHz Intel Core 2 Duo CPU, Marvell 88SE9123 PCIe 2.0 SATA controller, and WD Caviar SATA-II 7200 RPM hard disk. We run the benchmark with working set of 500MB on top of the raw disk.

We benchmark both drivers, stacked on top of each other, against equivalent Linux drivers. Both setups achieved the same I/O throughput on all tests, while the active drivers' CPU utilisation was slightly higher (Figure 8). This overhead can be reduced through improved protocol design. Our SATA driver protocol, based on the equivalent Linux interface requires 10 messages for each I/O operation. A clean-slate redesign of this protocol would involve much fewer messages.

We did not benchmark the DAI driver, as it has trivial performance requirements and uses less than 5% of CPU.

7 Related work

Active drivers Singularity [10] is a research OS written in the Sing# programming language. It comprises a collection of processes communicating over message channels. Sing# supports a state-machine-based notation for specifying communication protocols between various OS components, including device drivers. The Sing# compiler checks protocol compliance at compile time. Sing# extends its memory safety guarantees to message-based communication. For example, the compiler is able to verify that the program never dereferences a pointer whose ownership was passed to another process in a message. In contrast, our C-based implementation of active drivers does not assign any special meaning to pointers passed between the driver and the OS.

RMoX [3] is a process-based OS written in occam-pi. RMoX processes communicate using synchronous rendezvous. Communication protocols are formalised using the CSP process algebra and verified using the FDR tool.

The Dingo [19] active driver framework for Linux aims to simplify driver programming in order to help driver developers avoid errors. It relies on a C language extension to provide language-level support for messages and threads. Dingo uses a Statechart-based language to specify driver protocols; however it only supports runtime protocol checking and does not implement any form of static verification.

The CLARITY [5] programming language is designed to make passive drivers more amenable to automatic verification. To this end it provides constructs that allow writing event-based code in a sequential style, which reduces stack ripping. It simplifies reasoning about concurrency by encapsulating thread synchronisations inside *coord* objects that expose well-defined se-

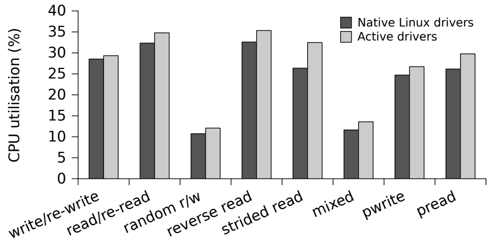


Figure 8: Native vs. active AHCI and ATA framework driver performance on the iozone benchmark.

quential protocols to the user.

User-level drivers User-level driver frameworks for microkernel-based [17] OSs encapsulate each device driver in a separate process that communicates with other OS processes using some form of message passing. The driver thread executes an event loop that handles incoming messages by invoking appropriate driver entry points. Thus, even though the driver has its own thread of control and uses messages for external communication, internally it is based on the passive programming model and suffers from stack ripping.

Verification tools Automatic verification tools for C [2, 8, 7, 13, 18] is an active area of research, which is complementary to our work on making drivers amenable to formal analysis using such tools. Any improvements to these tools are likely to further improve the speed and accuracy of active driver verification.

Several verification tools, including SPIN [16], focus on checking message-based protocols in distributed systems. These tools work on an abstract model of the system that is either written by the user or extracted from the program source code [15]. Such a model constitutes a fixed abstraction of the system that cannot be automatically refined if it proves too coarse to verify the property in question. Our experiments show that abstraction refinement is essential to avoiding false positives in active driver verification; therefore we do not expect these tools to perform well on active driver verification tasks.

8 Conclusion

We argue that improvements in automatic device driver verification cannot rely solely on smarter verification tools and require an improved driver architecture. Previous proposals for verification-friendly drivers were based on specialised language and OS support and therefore were not compatible with existing systems. Based on ideas from this earlier research, we developed a driver architecture and verification methodology that support drivers written in C and can be implemented in any existing OS. Our experiments confirm that this methodology enables more thorough verification of the driver-OS interface than what is possible for conventional drivers.

9 Acknowledgements

We would like to thank Michael Tautschnig for his help in troubleshooting SATABS issues. We thank the GOANNA team, in particular Mark Bradley and Ansgar Fehnker, for explaining GOANNA internals and providing us with numerous ideas and examples of verifying active driver properties using GOANNA. We thank Toby Murray for his feedback on a draft of the paper.

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.

References

[1] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W., AND DOUCEUR, J. Cooperative task management without manual stack management. In *2002 USENIX* (Monterey, CA, USA, Jun 2002), pp. 289–302.

[2] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *1st EuroSys Conf.* (Leuven, Belgium, Apr 2006), pp. 73–85.

[3] BARNES, F., AND RITSON, C. Checking process-oriented operating system behaviour using CSP and refinement. *Operat. Syst. Rev. 43*, 4 (Oct 2009), 45–49.

[4] BASLER, G., HAGUE, M., KROENING, D., ONG, C.-H. L., WAHL, T., AND ZHAO, H. Boom: Taking boolean program model checking one step further. In *TACAS* (2010), vol. 6015 of *Lecture Notes in Computer Science*, Springer, pp. 145–149.

[5] CHANDRASEKARAN, P., CONWAY, C. L., JOY, J. M., AND RAJAMANI, S. K. Programming asynchronous layers with CLARITY. In *6th ESEC* (Dubrovnik, Croatia, 2007), pp. 65–74.

[6] CHOU, A., YANG, J.-F., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *18th SOSP* (Lake Louise, Alta, Canada, Oct 2001), pp. 73–88.

[7] CLARKE, E. M., KROENING, D., SHARYGINA, N., AND YORAV, K. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design 25*, 2-3 (2004), 105–127.

[8] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *2006 PLDI* (Ottawa, Ontario, Canada, 2006), pp. 415–426.

[9] EDMUND M. CLARKE, ORNA GRUMBERG, D. P. *Model Checking*. MIT Press, 1999.

[10] FÄHNDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G. C., LARUS, J. R., AND LEVI, S. Language support for fast and reliable message-based communication in Singularity OS. In *1st EuroSys Conf.* (Apr 2006), pp. 177–190.

[11] FEHNKER, A., HUUCK, R., JAYET, P., LUSSENBURG, M., AND RAUCH, F. Goanna — A Static Model Checker. In *11th FMICS* (Bonn, Germany, Aug 2006), pp. 297–300.

[12] GANAPATHI, A., GANAPATHI, V., AND PATTERSON, D. Windows XP kernel crash analysis. In *20th LISA* (Washington, DC, USA, 2006), pp. 101–111.

[13] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., NECULA, G. C., SUTRE, G., AND WEIMER, W. Temporal-safety proofs for systems code. In *14th CAV* (2002), pp. 526–538.

[14] HOLZMANN, G. *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston, MA, USA, 2003.

[15] HOLZMANN, G. J. Logic verification of ANSI-C code with SPIN. In *7th SPIN* (2000), pp. 131–147.

[16] HOLZMANN, G. J. *The SPIN Model Checker: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[17] LIEDTKE, J., BARTLING, U., BEYER, U., HEINRICHS, D., RULAND, R., AND SZALAY, G. Two years of experience with a μ-kernel based OS. *Operat. Syst. Rev. 25*, 2 (Apr 1991), 51–62.

[18] PADIOLEAU, Y., LAWALL, J., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in Linux device drivers. In *3rd EuroSys Conf.* (Glasgow, UK, Apr 2008), pp. 247–260.

[19] RYZHYK, L., CHUBB, P., KUZ, I., AND HEISER, G. Dingo: Taming device drivers. In *4th EuroSys Conf.* (Apr 2009).

[20] WITKOWSKI, T., BLANC, N., KROENING, D., AND WEISSENBACHER, G. Model checking concurrent Linux device drivers. In *22nd ASE* (Atlanta, Georgia, USA, 2007), pp. 501–504.