

Automatic Verification of Active Device Drivers

Sidney Amani^{‡§} Peter Chubb^{‡§} Alastair F. Donaldson[¶]
Alexander Legg^{‡§} Keng Chai Ong[‡] 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 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 [15, 9]. Studies of Windows and Linux drivers show that over a third of driver bugs result from the complex interface between driver and OS [23, 2]. Automatic verification tools like SLAM [2], Terminator [11], SATABS [10], Blast [17], and Coccinelle [22] have been successfully used to find numerous OS interface violations in Windows and Linux drivers. However, 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.

Previous research has shown that dramatic improvements in automatic driver verification can be achieved with the help of an improved device driver architecture. The conventional driver architecture supported by all mainstream OSs suffers from two problems that impede verification: *concurrency* and *stack ripping*. Concurrency leads to state explosion, while stack ripping [1] complicates analysis of the driver control flow.

Both problems can be eliminated in a device driver architecture where each driver has its own thread and communicates with the OS using message passing. 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. The active driver architecture reduces the amount of concurrency that the driver must handle and makes the control flow of the driver and its interactions with the OS easier to understand and analyse. In particular, many properties that are hard or impossible to verify in conventional drivers can be easily checked on active drivers.

Previous implementations of the active driver architecture have been undertaken in research OSs such as Singularity [13] and RMoX [4]. These systems, designed for verifiability from the ground up, rely in essential ways on OS and language support to facilitate active driver verification. As a result, mainstream OSs have not been able to take advantage of innovations proposed in these systems.

In this paper we make two contributions. First, we demonstrate 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.

Second, 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 optimisations geared towards making active driver verification tractable.

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.

2 Passive vs active drivers

In this section we discuss the shortcomings of the conventional driver architecture and show how active drivers address these shortcomings.

2.1 Passive drivers

Passive drivers 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; however these assumptions remain implicit in the implementation. As a result, the control flow of the driver is scattered across multiple entry points and cannot be reconstructed from its source code. This phenomenon is known as stack ripping [1]. 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 by augmenting driver code with an OS model that simulates all possible legal sequences of driver 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 becomes intractable for code involving complex pointer manipulation.

To complicate things further, the OS can invoke driver entry points from multiple concurrent threads, forcing driver developers to implement intricate synchronisation logic to avoid races and deadlocks. Multithreading further complicates automatic verification of device drivers, as thread interleaving leads to dramatic state explosion.

Previous research [24] has shown that the vast majority of device drivers do not get any performance benefits from multithreading. The performance of most drivers is bound by I/O bandwidth rather than CPU speed, therefore they do not require true multiprocessor parallelism. Device drivers are multithreaded simply by virtue of executing within the multithreaded kernel environment and not because they require multithreading for performance or functionality.

Case study We demonstrate the adverse effects of stack ripping 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, over 9 years, since it was added to the Linux kernel tree, and identify those fixes that address OS interface violation bugs, where the driver incorrectly responds to certain sequences of OS requests. A typical example bug involves the driver attempting to use an OS resource such as timer after it has been destroyed by a racing thread. We found 12 documented OS interface violation bugs. We apply SATABS [5, 10], a state-of-the-art model checker for C, to detect these bugs.

SATABS has been successfully applied to Linux drivers in the past [25].

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).

By gradually improving the OS model, we were able to find 8 out of 12 bugs. However, when being provided a model accurate enough to trigger the remaining 4 errors, SATABS was not able to find the bugs before being interrupted after 12 hours. Our final model consisted of 333 lines of code and took several days to construct and refine. As stated above, this model is far from being complete and is just good enough to detect the target set of bugs.

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.

In this paper, we use SATABS 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.

2.2 Active drivers

In contrast to passive drivers, an active driver [13, 4, 23] runs in 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*. The driver notifies the OS about a completed request via a reply message. In an active device driver, the order in which the driver handles and responds to OS requests is defined explicitly in its source code and can be readily analysed automatically. Since the driver handles I/O requests sequentially, such analysis can be performed without running into state explosion due to thread interleaving.

Active driver framework We present our instantia-

tion of the active driver architecture. Our design is based on the design of the Dingo active driver framework for Linux [23]. In contrast to Dingo, which relies on C language extensions to implement message passing, our framework supports drivers in pure C.

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.

Previous research [23] 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 sure that a driver thread executes atomically with respect to other threads of the same driver, until it blocks waiting for a message. This scheduling discipline makes drivers easier to write and simplifies 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.

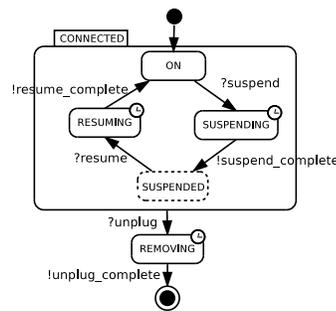
Example Figure 1(a) shows a fragment of driver code that matches the example of a driver bug in Section 2.1. Here, `suspend`, `unplug`, `suspend_complete`, and `resume` are pointers to driver mailboxes. In line 1 the driver waits for both suspend and unplug requests. After receiving a suspend request (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.

```

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.

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.

Note however that this time all event handling occurs in the context of a single thread. As a result the bug can be discovered simply by exploring the control skeleton of the driver and considering messages that the driver sends and waits for in each state, without resorting to pointer analysis. This example illustrates that eliminating stack ripping greatly simplifies analysis of the driver behaviour.

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 pro-

protocol state machines of Dingo [24] and Singularity [13], extended with 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. It is often useful to include constraints not only on the ordering of messages but also on their arguments in the protocol specification; however, in this work we focus on specifying and verifying ordering constraints only.

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, 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 deterministic finite state machines (FSMs). 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 [16], which organise states into a hierarchy so that several primitive states can be clustered into a super-state.

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 [19] 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 fair states with dashed border. For example, the `SUSPENDED` state in Figure 1b is 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 fol-

lowing 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.*

Rule 2. (AWAIT1) *The driver must not ignore an incoming message forever: when in a state where there is an enabled incoming message, the driver must eventually either issue an `AWAIT` on the corresponding mailbox or transition into a state where this message is not enabled.*

Rule 3. (AWAIT2) *All `AWAIT` operations eventually terminate. Equivalently, whenever the driver performs an `AWAIT` operation, at least one of its protocols must be in a fair state and the `AWAIT` must wait for all enabled messages of this protocol.*

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

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

Going back to the example in Figure 1, we can see that the `AWAIT` statement in line 6 violates Rule 3. This line corresponds to the `SUSPENDED` state of the protocol, where the driver can receive `unplug` and `resume` messages. By waiting for only one of these messages, the driver can potentially deadlock.

4 Verifying driver protocols

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` [5, 10], geared towards safety analysis, and `GOANNA` [14], geared towards liveness analysis. 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. In the rest of this section we focus on our methodology for checking safety properties with `SATABS`, which is where our main contributions lie.

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.

The main principles of operation of SATABS are similar to other abstraction-refinement model checkers [2, 17]. 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 refine the program abstraction, either by computing a finer abstraction using the current set of predicates [3] or by adding new predicates suggested by the counterexample. Abstraction and refinement are both fully automatic.

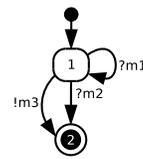
We use a simple driver protocol shown in Figure 2a and a fragment of driver code that implements this protocol in Figure 2b as a running example to illustrate the use of SATABS.

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 2c 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`.

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 2c with respect to these two predicates is shown in Figure 2d. 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



(a) Driver protocol

```

1 mailbox_t *m;
2 m = AWAIT(m1, m2);
3 if (m==m1) {
4   EMIT(m3, msg);
5 }

```

(b) Driver source code

<pre> 1 /*initial state*/ 2 int state=1; 3 mailbox_t *m; 4 //AWAIT{ 5 m=random_mb(m1, m2); 6 if (state==1&&m==m1) 7 state=1; 8 else 9 if (state==1&&m==m2) 10 state=2; 11 else assume(0); 12 //} 13 if (m==m1) { 14 //EMIT{ 15 /*m3 in state 1*/ 16 if (m3==m3&&state==1) 17 state=2; 18 else 19 assert(0); 20 //} 21 } </pre>	<pre> 1 2 p1=1; 3 4 5 p2=*; 6 if (p1&&p2) 7 p1=1; 8 else 9 if (p1&&!p2) 10 p1=0; 11 else assume(0); 12 13 if (p1) { 14 15 if (true&&p1) 16 p1=0 17 else 18 assert(0); 19 20 21 } </pre>
---	--

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

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

Figure 2: Safety verification example

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 SATABS introduce large numbers of unnecessary predicates, leading to overly complex abstractions. The problem is not unique to SATABS. Our preliminary experiments with SLAM [2], another state-of-the-art abstraction-refinement tool, produced similar results. We describe several 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.

4.1 Protocol decomposition

The abstraction-refinement technique is highly sensitive to the size of the property being checked. Checking complex properties requires 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, as simpler properties can often be verified with fewer predicates.

We automatically decompose each driver protocol state machine into a set of much simpler subprotocols as a preprocessing step. The decomposition is constructed in such a way that the driver satisfies safety constraints of the original protocol if and only if it does so for each protocol in the decomposition. Note that the driver itself remains unmodified; only the protocols it is checked against are replaced with collections of simpler protocols.

The following proposition (stated informally) gives a sufficient condition for correctness of decomposition.

Proposition 1. *Consider a protocol state machine M and its decomposition into state machines M_1, \dots, M_n . If the following conditions hold then a driver satisfies M if and only if it satisfies each of M_1, \dots, M_n :*

1. *The regular language generated by the protocol state machine of M is equivalent to the intersection of languages generated by M_1, \dots, M_n .*
2. *There exists a bijection between fair states of M and the union of fair states of M_1, \dots, M_n , such that for each fair state s of M and the corresponding fair state s' of M_i , the set of incoming messages enabled in s is equal to the set of incoming messages in s' .*

Protocol decomposition is performed automatically using Algorithm 1. Although it does not guarantee decomposition into smallest possible subprotocols¹, this heuristic algorithm works very well in practice, producing decompositions with no more than 4 states in each subprotocol for all driver protocols considered in our case studies.

The algorithm assumes a pre-processing step during which the protocol state machine is flattened by expanding all of its super-states. A flat protocol state machine $M = \langle Q, \Sigma, \delta, q_0, F, \varphi \rangle$ consists of a set of states Q , a set of messages Σ , a transition relation $\delta \subseteq Q \times \Sigma \times Q$, initial state q_0 , a set of final states F , and a set of fair states φ . We write $s \xrightarrow{c} t$ for $\langle s, c, t \rangle \in \delta$. We denote $M|R$, where $R \subseteq Q$, the restriction of M to states in R .

The algorithm exploits the fact that most messages in realistic driver protocols are enabled in only a small number of states within the protocol. We call a message of a

¹We have not been able to find an efficient precise algorithm in the literature.

Algorithm 1 Protocol state machine decomposition.

Input: Protocol state machine $M = \langle Q, \Sigma, \delta, q_0, F, \varphi \rangle$ and a subset of its states $R \subseteq Q$

Output: A set D of protocol state machines that form a decomposition of M

```

1: function DECOMPOSE( $M, R$ )
2:    $D \leftarrow \emptyset$ 
3:    $S \leftarrow R$ 
4:   while  $S \neq \emptyset$  do
5:     choose arbitrary  $s \in S$ 
6:     // compute partition containing state  $s$ 
7:      $P \leftarrow \text{PARTITION}(M|R, \emptyset, s)$ 
8:     //  $R$  could not be partitioned
9:     if  $P = R$  then
10:      return  $\{M\}$ 
11:     // merge states outside the partition
12:      $M' \leftarrow \text{MERGE}(M, R \setminus P)$ 
13:     // recursively decompose  $M'$ 
14:      $D \leftarrow D \cup \text{DECOMPOSE}(M', P)$ 
15:      $S \leftarrow S \setminus P$ 
16:   end while
17:   return  $D$ 
18: end function

```

// Merge states in A into a single state

```

19: function MERGE( $M = \langle Q, \Sigma, \delta, q_0, F, \varphi \rangle, A \subseteq Q$ )
20:    $Q' \leftarrow (Q \setminus A) \cup \{x\}$  //  $x$  is a fresh state
21:    $q'_0 \leftarrow \begin{cases} x & \text{if } q_0 \in A \\ q_0 & \text{otherwise} \end{cases}$ 
22:    $F' \leftarrow \begin{cases} F & \text{if } A \cap F = \emptyset \\ (F \setminus A) \cup \{x\} & \text{otherwise} \end{cases}$ 
23:    $\delta' \leftarrow \emptyset$ 
24:   for all  $s \xrightarrow{c} t \in \delta$  do
25:      $s' \leftarrow \begin{cases} x & \text{if } s \in A \\ s & \text{otherwise} \end{cases}$ 
26:      $t' \leftarrow \begin{cases} x & \text{if } t \in A \\ t & \text{otherwise} \end{cases}$ 
27:      $\delta' \leftarrow \delta' \cup \{s' \xrightarrow{c} t'\}$ 
28:   end for
29:   return  $\langle Q', \Sigma, \delta', q'_0, F', \varphi \setminus A \rangle$ 
30: end function

```

// Compute a partition of Q containing all states // in P and state p

```

31: function PARTITION( $M = \langle Q, \Sigma, \delta, q_0, F, \varphi \rangle, P, p$ )
32:   if  $p \in P \vee p \notin R$  then return  $P$ 
33:    $P' \leftarrow P \cup \{p\}$ 
34:   for all  $(s, c)$  such that  $s \xrightarrow{c} p \in \delta \vee p \xrightarrow{c} s \in \delta$  do
35:     if  $c \in \text{key}(M)$  then
36:        $P' \leftarrow \text{PARTITION}(M, P', s)$ 
37:   end for
38:   return  $P'$ 
39: end function

```

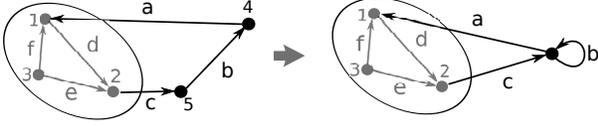


Figure 3: Merging states outside partition $\{1,2,3\}$.

protocol state machine a *key* message if all transitions labelled with this message end in the same state. For example all messages in the protocol in Figure 1(b) are key messages. Intuitively, after observing transmission of a key message, the observer knows the exact state of the protocol, even if they did not track previous protocol states. We denote $key(M) \subseteq \Sigma$ the set of key messages of M .

The algorithm breaks down the protocol state machine into non-overlapping partitions (line 7) such that *all transitions connecting different partitions are labelled with key messages*. Intuitively, this property guarantees that transitions within each partition can be checked by a separate observer who ignores all transitions outside the partition. The observer tracks entries to the partition by watching for relevant key messages. The algorithm converts each partition into a separate protocol state machine using the MERGE function, which merges states external to the partition into a single state that overapproximates protocol transitions outside the given partition (line 12). Figure 3 illustrates the merging step. As a result of the overapproximation, each protocol in the decomposition is weaker (i.e., more permissive) than the original monolithic protocol; however the way we construct the decomposition guarantees that the combination of protocols is equivalent to the original protocol.

The decomposition is repeated recursively for each partition (line 14). The algorithm returns a set of protocols that cannot be further partitioned using this procedure.

Figure 4 shows the decomposition of the protocol in Figure 1(b) produced by our algorithm. All partitions in this decomposition are of size one, with an additional state added to each partition to represent external states. In the figure, loop transitions labelled with “+” are shortcuts representing sets of loop transitions, one for each transition between external states.

Proposition 2. *Let $M = \langle Q, \Sigma, \delta, q_0, F, \varphi \rangle$ be a protocol state machine and let $\{M_1, \dots, M_n\} = \text{DECOMPOSE}(M, Q)$. Then the decomposition of M into protocols M_1, \dots, M_n satisfies conditions 1 and 2 of Proposition 1.*

In order to validate our implementation of the decomposition algorithm, we have implemented a tool to automatically verify protocol decompositions. The tool com-

putes the intersection of subprotocols in the decomposition and checks its equivalence to the original protocol state machine.

4.2 Automatically provide key predicates

Another 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 mentioned above, when checking a driver protocol, we introduce a global variable that keeps track of protocol state. During verification, SATABS eventually discovers predicates over this variable of the form $(state==1)$, $(state==2)$, \dots , one for each state of the protocol. 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.

4.3 Control-flow transformations

We found that it often takes SATABS many iterations to correlate dependent program branches. This problem frequently occurs in active drivers when the driver `AWAITS` on multiple mailboxes and then checks the returned value (e.g., line 2 in Figure 1(a)). If the driver performs the same check later in the execution, then both checks must produce the same outcome. SATABS does not know about this correlation initially, potentially leading to multiple spurious counterexample traces that take inconsistent branches. These counterexamples can be refuted using predicate $p \leftrightarrow (mb == suspend)$. In practice, however, SATABS may introduce many predicates that only refute a subset of these counterexamples before discovering p , which allows refuting all of them.

To remedy the problem, we have implemented a 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. Although we are not aware of this transformation being used in prior work, the idea of rewriting control-flow to reduce join points is a well-known method for improving the precision of static analysers.

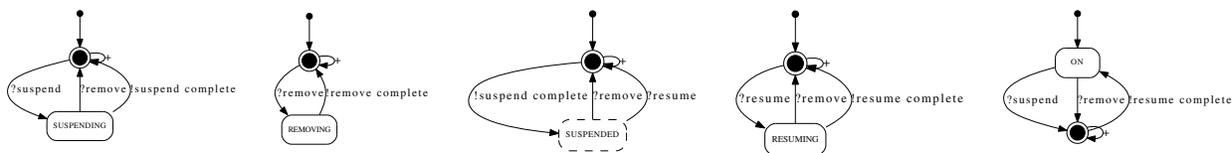


Figure 4: Decomposition of the protocol in Figure 1(b).

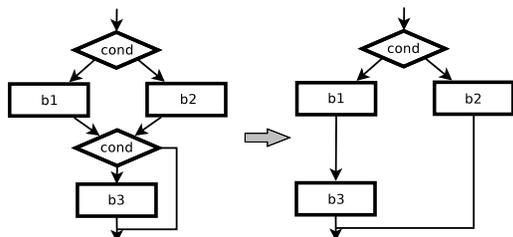


Figure 5: CFG transformation example.

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.

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 al-

location 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 queuing interrupt messages at the head of the message queue; alternatively interrupts can be queued as normal messages, which avoids interrupt livelock and 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. This one-off effort leads to the creation of reusable specifications that not only enable automatic driver verification, but are also useful as documentation for driver developers.

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`.

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). 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 storage 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.

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 (9)

Table 3: Statistics for checking safety properties using SATABS.

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.

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).

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 [6], 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 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. 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. A typical SLAM rule is of the form “event A must happen after event B and before event C ”. Combined with the Terminator [11] liveness checker, SLAM can also detect liveness errors.

Analysis of the SLAM rule database shows that, 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.

Consider again the driver protocol in Figure 1b. The protocol specifies that the `unplug` message can arrive in any state other than `REMOVING`. A failure to wait for this message when it is enabled will be automatically de-

tected by the verifier. In contrast, the same rule cannot be specified for passive drivers, which do not explicitly wait for incoming requests. 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.

The developers of SLAM report [2] that it took them several years to come up with a satisfactory set of rules formalising Windows driver interfaces. This is consistent with our experience. For example, it took us over a month to formalise just the SCSI and SATA protocols. The complexity of formalising existing driver protocols means that these protocols are equally hard to understand and implement correctly for the driver developer. We believe that the situation can be improved by designing new driver frameworks around the active driver architecture and formal driver protocol specifications from the ground up.

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.

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

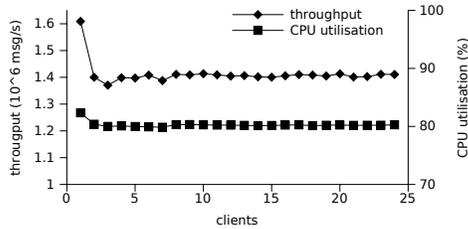


Figure 6: Message throughput and aggregate CPU utilisation over 8 CPUs for varying number of clients.

latency is $1.8 \mu\text{s}$. For comparison, the roundtrip latency of a Gigabit network link is at least $55 \mu\text{s}$.

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)).

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 [13] 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 [4] 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 [23] 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 [8] 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 sequential protocols to the user.

Event-based programming with state machines In active drivers we use the thread abstraction to structure event-based driver logic. Alternatively, event-based programs can often be naturally expressed in terms of state machines. This approach requires language support for state-machine based programming, which can be provided by a visual language, e.g., Statecharts [16], or a

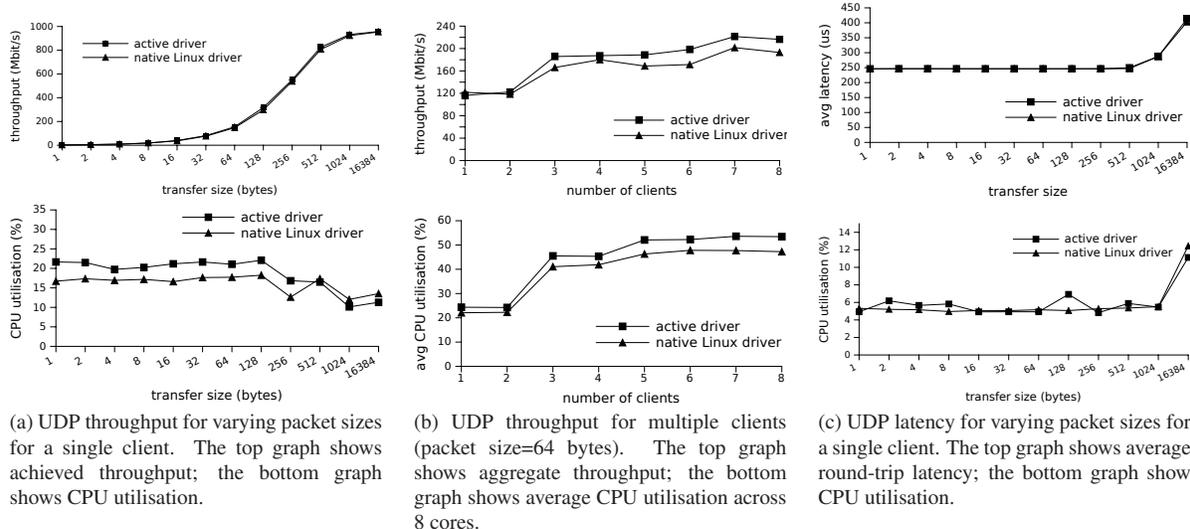


Figure 7: Performance of the RTL8169 Ethernet driver measured with Netperf.

text-based language like Esterel [7].

Recently, the P [12] programming language demonstrated the benefits of state-machine based programming for Windows device drivers. In P both driver protocols and implementation are described using collections of communicating state machines. In contrast, we only use state machines for specifying driver protocols, while the driver implementation is written in C.

P comes with a model checker that currently only checks safety properties. Interestingly, it uses explicit model checking to validate driver protocols. In contrast, in this work we use symbolic model checking for the same purpose. Directly comparing the two techniques and measuring the resulting tradeoffs between run time, completeness, and precision is an interesting future research direction.

User-level drivers User-level driver frameworks for microkernel-based [20] 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, 11, 10, 17, 21] 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 [19], focus on checking message-based protocols in distributed sys-

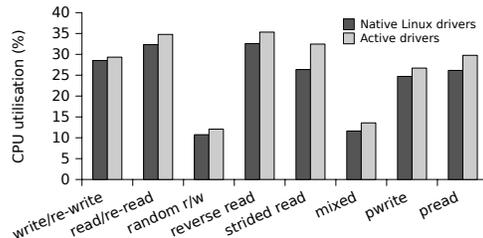


Figure 8: Native vs. active AHCI and ATA framework driver performance on the iозone benchmark.

tems. These tools work on an abstract model of the system that is either written by the user or extracted from the program source code [18]. 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. The main findings from this experiment are as

follows:

The active driver architecture mitigates problems related to stack ripping and cocurrency. It simplifies driver development (and hence reduces the number of bugs) and makes an important subset of correctness properties, related to the interaction between the driver and the OS, amenable to automatic verification.

The use of finite state machines to specify driver protocols allows capturing the entire protocol in a single formal model, which is passed as input to automatic verification tools, but also serves as a valuable design artifact that guides protocol design and implementation on both the OS and the driver side.

With several improvements existing verification tools can be applied to active drivers. Our experiments confirm that this approach enables more thorough verification of the driver-OS interface than what is possible for conventional drivers using the same tools.

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] BALL, T., COOK, B., DAS, S., AND RAJAMANI, S. K. Refining approximations in software predicate abstraction. In *TACAS* (Mar 2004), pp. 388–403.
- [4] BARNES, F., AND RITSON, C. Checking process-oriented operating system behaviour using CSP and refinement. *Operat. Syst. Rev.* 43, 4 (Oct 2009), 45–49.
- [5] BASLER, G., DONALDSON, A. F., KAISER, A., KROENING, D., TAUTSCHNIG, M., AND WAHL, T. SatAbs: A bit-precise verifier for C programs - (competition contribution). In *TACAS* (Mar 2012), pp. 552–555.
- [6] 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.
- [7] BERRY, G., GONTHIER, G., GONTHIER, A. B. G., AND LALTE, P. S. The estereel synchronous programming language: Design, semantics, implementation, 1992.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *2006 PLDI* (Ottawa, Ontario, Canada, 2006), pp. 415–426.
- [12] DESAI, A., GUPTA, V., JACKSON, E., QADEER, S., RAJAMANI, S., AND ZUFFEREY, D. P: safe asynchronous event-driven programming. In *PLDI* (Seattle, Washington, USA, 2013), pp. 321–332.
- [13] 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.
- [14] 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.
- [15] GANAPATHI, A., GANAPATHI, V., AND PATTERSON, D. Windows XP kernel crash analysis. In *20th LISA* (Washington, DC, USA, 2006), pp. 101–111.
- [16] HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3 (Jun 1987), 231–274.
- [17] 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.
- [18] HOLZMANN, G. J. Logic verification of ANSI-C code with SPIN. In *7th SPIN* (2000), pp. 131–147.
- [19] HOLZMANN, G. J. *The SPIN Model Checker: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.
- [20] 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.
- [21] 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.
- [22] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in Linux: ten years later. In *ASPLOS* (Newport Beach, CA, USA, Mar 2011), pp. 305–318.
- [23] RYZHYK, L., CHUBB, P., KUZ, I., AND HEISER, G. Dingo: Taming device drivers. In *4th EuroSys Conf.* (Apr 2009).
- [24] RYZHYK, L., CHUBB, P., KUZ, I., AND HEISER, G. Dingo: Taming device drivers. In *4th EuroSys Conf.* (Nuremberg, Germany, Apr 2009).
- [25] WITKOWSKI, T., BLANC, N., KROENING, D., AND WEISENBACHER, G. Model checking concurrent Linux device drivers. In *22nd ASE* (Atlanta, Georgia, USA, 2007), pp. 501–504.