# Towards Automatic Software Model Checking of Thousands of Linux Modules - A Case Study with Avinux

Hendrik Post, Carsten Sinz, Wolfgang Küchlin [1]

*Symbolic Computation Group*
*University of Tübingen*
*Tübingen, Germany*

**Abstract**

Modular software model checking of large real-world systems is known to require extensive manual effort in environment modelling and preparing source code for model checking. Avinux is a tool chain that facilitates the automatic analysis of Linux and especially of Linux device drivers. The tool chain is implemented as a plugin for the Eclipse IDE, using the source code bounded model checker CBMC as its backend. Avinux supports a verification process for Linux that is built upon specification annotations with SLICx (an extension of the SLIC language), automatic data environment creation, source code transformation and simplification, and the invocation of the verification backend. In this paper technical details of the verification process are presented: Using Avinux on thousands of drivers from various Linux versions lead to the discovery of six new errors. In these experiments, Avinux also reduced the immense overhead of manual code preprocessing that other projects incurred.

*Keywords:* Software Model Checking, Integration and Application of Formal Methods, Operating Systems.

## 1 Introduction

Engler et al. explain in a detailed review [11], why model checking real-world software is problematic, and that it fails to discover the large numbers of errors that light-weight static analysis methods are able to report. However, in mission critical software (including operating systems), model checking may provide valuable results that cannot be produced with light-weight verification methods. The Static Driver Verifier (SDV) project [1] for Windows device drivers is an example for this approach. The Linux operating system kernel still poses a challenge to software model checking approaches because of its size and complexity. Linux lacks a strict and uniform driver framework similar to the Windows Driver Model covered by SDV. The tool chain Avinux targets the problems described by Engler et al., such that almost automatic, effective model checking of Linux source code can be

[1] Email: {post, sinz, kuechlin}@informatik.uni-tuebingen.de

achieved. This paper describes technical details and the experiences from applying Avinux to thousands of Linux source files from different kernel versions.

Avinux[2] [18] is an integrated software verification tool chain. It comes as an Eclipse[3] plugin which significantly improves the automation in Linux code analysis and error checking. Post and Küchlin [18] describe how Avinux can be used to verify various functional and non-functional properties of Linux code. In this paper, technical details on how the Avinux tool chain finds errors in Linux device drivers are given. The verification backend of Avinux is the bounded software model checker CBMC [8]. Experiments show that large manual source code preparation can be avoided in contrast to other case studies [11,1,7,14]. Several experiments are performed on various Linux kernel versions, each with over three thousand processed translation units.

Avinux has been used to rediscover several known errors in Linux. These errors have been reported to require extensive manual code simplification and transformation [14] when checked with BLAST. BLAST requires that this manual adaptation is performed once per driver. With Avinux, the manual effort is reduced to the construction of an operating system model, which basically implements an abstract use case that simulates the interactions between the operating system and modules or device drivers. The manual work required to employ Avinux is limited on changes per kernel version. The checking of each driver is then done automatically. Examples of re-discovered errors can be obtained through the project home page. This paper also reports six new bugs discovered by Avinux.

In this article, the notions of *completeness* and *soundness* are used with respect to both verification methods (like bounded model checking) and error specifications. A verification method is called complete, if each error of a program (of a certain error class) is really detected. It is called sound, if no false alarms (*false positives*) are produced. A false positive is a reported error that in fact can never happen during program execution. A specification is called complete, if it encompasses all errors of a certain class (e.g. memory access violations). It is called sound, if no false positives can occur, i.e., it must capture the error cases, but not more.

Bounded model checking [3] (BMC) is a sound verification technique. If its bound is chosen high enough—which is rarely the case for real world examples—it is also complete. For the sake of presentation, this work treats BMC as a sound and complete technique, though it is in fact incomplete (with feasibly chosen bounds) for almost all device drivers that were analyzed. As a consequence, a specification rule may be called complete, although it may not be checked for all possible program traces using bounded model checking. As this is a limitation of the verification backend—rather than the rule itself—this distiction is not made explicit.

Modular analysis, as implemented in this work, faces the problem that the external environment may restrict the set of input parameters for the module under test. When checking the module on its own, these restrictions are typically not available. Thus, false positives—even when using a sound verification backend—can hardly be avoided. On the other hand, the external environment may also cause false negatives: callbacks and other unmodelled side effects, e.g. the parallel update

[2] Project home page: `http://www-sr.informatik.uni-tuebingen.de/∼post/avinux`

[3] `www.eclipse.org`

of a shared variable, can lead to program traces that are not included in a straightforward modular analysis. Note that in the context of system code written in C, an over-approximation of the environment is infeasible to compute due to callbacks, pointer arithmetic and the manifold interfaces between different subsystems and modules.

This work contains sound and complete rules. Checking these rules with bounded model checking leads to an (possibly incomplete) set of error reports. This set still contains spurious reports due to the problems described above stemming from the unknown environment. Some techniques are presented to treat the environment problem pragmatically, such that a feasible ratio of false positives to real errors can be achieved.

In Section 2 the general architecture of Avinux is described. A summary of the verification process is given in Subsection 2.1. Results applying Avinux to Linux drivers are described in Subsection 4.1, followed by a review of specifications that are checked for Linux drivers (Section 3). Section 4.2 reviews current limitations and planned extensions. Section 5 gives an overview about related projects before main aspects of Avinux are summarized in the last section.

## 2 Components and Verification Process

The Avinux tool chain integrates five components that are orchestrated by a plugin for the Eclipse IDE: CBMC, CIL, DEC, CLEANC and SLICx. The first two, CBMC and CIL, originate from other research groups. SLICx is an extension and reimplementation of the Microsoft specification language SLIC [2]. DEC and CLEANC are within the contribution of this paper.

CBMC [8] is a bounded model checker for C that extracts models directly from source code. Function calls are inlined, and loops and recursion are unwound up to a user provided bound. Types are reduced to a bit-level representation that models the execution on bit-level hardware according to the ANSI C standard. CBMC has built-in support for memory safety checking and recognizes a specification language in the form of `assert` and `assume` statements.

CIL [16] is a code transformation framework that translates several C dialects and non-standard code constructs into ANSI-C programs. CIL has already been used for software model checking [7].

DEC is a Data Environment Construction module. It treads the problem that modular analysis looses constraints on input variables [17], e.g. parameters. It scans a C translation unit for global identifiers and parameters of entry functions that are of pointer type or contain a pointer type member. Up to a user provided bound the object graph is unrolled by creating appropriate objects for each pointer. These pointers are then initialized such that at the start of every control flow every pointer points to a valid object. Assuming that such an environment is indeed provided by the operating system, the verification backend guarantees that no errors occur in the module under test. DEC provides a tradeoff that reduces the number of false positives[4] while missing potential bugs from violated assumptions about external

[4] In this work the term *false positive* refers to problem reports that are caused by the modularity of the analysis.

usage of the module. For DEC, generic tools for data environment creation are adapted. CUTE [21] is one example for such a tool, though not all features of CUTE are covered by DEC.

CLEANC is an additional code cleaning facility. Although CIL transforms most code dialects into ANSI-C, several additional simplifications could be identified in order to prepare the code for CBMC. Examples include empty structs, compiler attributes and some forms of nested static function pointer initializers. A list of eliminated hazards is presented on page 10.

SLICx is a novel specification language [18], which extends the SLIC [2] language for interface specification used in the Microsoft Static Driver Verifier. SLIC allows to define a safety automaton. State transitions can be defined in the form of transfer functions that may read program variables and may read and update state variables. In contrast to SLIC, SLICx allows the definition of general transfer functions including multiple assignments, function calls and arbitrary C expressions and statements, i.e. SLICx allows the modification of program variables. This work encompasses a SLICx compiler that transforms SLICx specifications into C code, which is then merged with the Linux sources to be checked. The language SLICx and the detection of race conditions, deadlocks and API violations using SLICx are described in [18].

### *2.1 The Verification Process*

The following steps are necessary for analyzing a Linux device driver with Avinux:

(i) Configuration of the Linux kernel so that the relevant modules are built.

(ii) Formulation of an interface rule as in SLICx.

(iii) Automatic annotation of all modules with the above rule (SLICx Compiler).

(iv) Transformation of the Linux kernel with CIL.

(v) Merging of all relevant source code files with CIL.

(vi) Automatic code simplification with CLEANC.

(vii) Manual creation of a `main` function simulating the operating system's use of the driver.

(viii) Automatic creation of a data environment for `main` with DEC.

(ix) Running CBMC on `main`.

The performance bottleneck is located in step (vii) because the creation of `main`, e.g. an abstract unit driver, has to be done for every driver. This problem has been solved for some Windows driver architectures (WDM and KMDF [1]), but Linux drivers have less standardized architectures and interfaces. For Linux, such an operating system's model cannot be implemented in a generic way. Figure 1 gives a graphical representation of the process that transforms a plain Linux driver into a checkable unit.

The analysis which lead to the discovery of a new bug is now described in detail. Additionally an overview of other experiments is given.

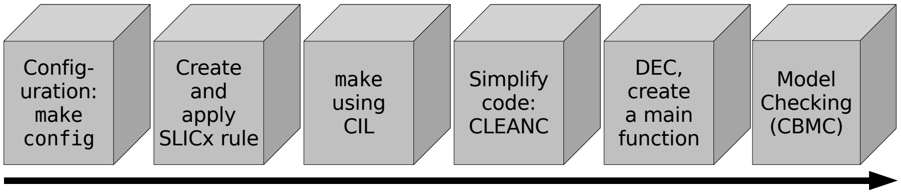
Config-uration: make config
Create and apply SLICx rule
make using CIL
Simplify code: CLEANC
DEC, create a main function
Model Checking (CBMC)

Fig. 1. The creation of a checkable unit requires multiple transformations. The creation of the SLICx rule and the construction of a test harness, i.e. main function for the module under test, must be performed manually. All other steps are performed automatically.

## 3 Specification of Common Error Types

In this section typical errors in Linux are listed. These errors can be identified using Avinux and especially the new specification language SLICx. Note that the built-in checking capabilities of CBMC alone do not cover these error classes.

(i) Proving the absence of memory leaks.

(ii) Sequential simulation of preemption.

(iii) Absence of deadlocks.

(iv) Race condition detection.

Some SLICx rule implementations for these error types are available for download from the project website.

### *3.1 Memory*

#### *3.1.1 Memory Safety with CBMC*

Memory safety—i.e. the avoidance of invalid accesses to memory areas—is already implemented in CBMC. However, CBMC only supports generic memory-related specifications that must be true for all programs written in C:

- Dereferences of `NULL` pointers.
- Dereferences of pointers to deallocated objects.
- Dereferences of pointers to objects that were not initialized within the scope of analysis.
- Accesses beyond an object's bounds within memory—e.g. after the last element of an array.
- Calls to function pointers with an offset.

#### *3.1.2 Memory Leaks (ML)*

Dynamic objects are allocated by means of `kmalloc()` and some minor variations of it. Finally they must be deallocated by `kfree()`. Otherwise memory leaks may occur which poses a hazard for server operating systems like Linux. The following description is treated as a definition of the absence of this hazard: *After* `kmalloc` *eventually* `kfree` *must be called.*

This definition can be adapted for Linux modules. Most modules in Linux implement a life-cycle that begins when the module is loaded for the first time and ends when the module is finally unloaded, for example when the device is unplugged

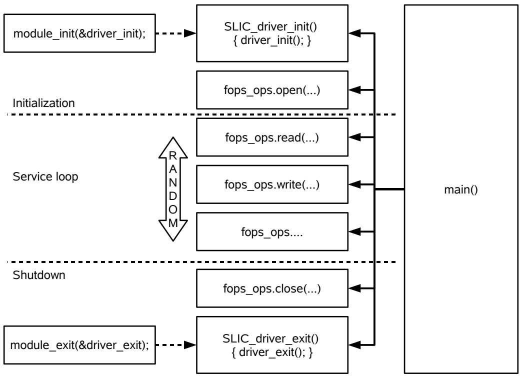

Fig. 2. An abstract lifecycle of a Linux Device Driver module. The `module_init` and `module_exit` macros mark the functions that are called first and last. Preprocessor macros are used to transform them into uniformly named functions that may be called from a main function.

and the driver is no longer needed.

The specification concerning memory leaks for modules may hence be transformed: previously allocated memory must be freed by `kfree()` when the last function in a module's life-cycle (cf. Figure 2) returns. Basically the liveliness property from the definition is simulated by a safety property specification: *After* `main` *the object's state must be* `freed`.

### *3.2 Preemption Simulation (PS)*

Concerning the parallel execution of different threads of control, many successful techniques have been developed. In this section an approach is presented that models preemption instead of a complete parallel thread interleaving.

The following example illustrates a common interleaving for device drivers on a multiprocessor system: A dispatch function is servicing a request while an interrupt occurs and preempts the running function. The interrupt handler may in turn be interrupted by other interrupt handlers. Both, the dispatch functions and interrupt handlers, may be interrupted by exception handlers that handle page faults or divisions by zero. Exception handlers, however, may not be preempted. The execution of an interrupt handler may be prevented by disabling interrupts, i.e. precise API modelling is necessary to prohibit false traces.

Two aspects are illustrated by the example:

- Preemption or context switches are controlled by calls to API functions.
- A complete coverage of context switches in not necessary under all circumstances.

The implementation of sequential simulation is facilitated by SLICx. As annotation rules may change the program state, it is possible to add additional calls to interrupt handlers or other preempting functions before and after any relevant function calls. A more detailled description and examples are given by Post and Küchlin in [18]. Note that this transformation provides the infrastructure for any rule, but especially for checking race conditions and deadlocks as indicated below, in a parallel context.

### *3.3 Sound Locking (AL)*

The commonly cited locking rule 'Alternating Locking' (AL) refers to the requirement that for each lock instance, `lock` and `unlock` operations must be performed in an alternating manner. If a lock object is requested twice without unlocking a deadlock occurs. This specification is sound, but clearly incomplete with respect to deadlocks. This rule can be implemented in a similar way as in the original SLIC language [2]. The common base rule is extended in so far as it coveres all different locking and unlocking operations occurring in the Linux API.

### *3.4 Complete Locking (LO)*

The sound rule AL is complemented by a second rule that is complete with respect to deadlocks [5]. The four Coffman Conditions [9] describe necessary requirements that must be true in order to produce a deadlock.

Three of the Coffman Conditions are true due to the Linux locking implementation. Therefore the only option to avoid deadlocks is to enforce that the fourth condition is not true: deadlocks can only occur if there is a circular wait for locks. To prevent this, it is required that locks are requested in a strict locking order. A SLICx rule can monitor this requirement and is therefore complete with respect to deadlocks (in the locking API).

The rule is named 'Lock Order' (LO) and is implemented via a non-deterministically chosen pair of lock objects similar to the universal quantification trick. For each watched lock a status flag is introduced that monitors the locked / unlocked state of this lock. Moreover, a flag is introduced that stores the order of acquisitions for the chosen pair of locks. If it never happens that any pair of locks is acquired in more than one order, it may be inferred that a circular wait is not induced by the locks.

If the lock order were specified in the API, each thread could be checked separately. As this is not the case for Linux drivers, the problem cannot be decomposed. The Avinux implementation checks this rule only for preemption simulation (cf. Figure 3). Therefore its treatment of locking is still incomplete, but the LO-rule itself provides a complete locking specification.

### *3.5 Race conditions (UA)*

A race condition may occur if two threads of control access a shared memory location and at least one of them writes a new value into it. Instead of finding race conditions directly, a conservative rule is proposed that is complete but unsound with respect to races. A notable advantage is that this rule implements race checking by the standard means provided by CBMC and SLICx, while each tool does not offer a solution by itself. The rule is called 'Unprotected Access' (UA).

The solution covers the following common setting: *A dynamically allocated* `struct` *shall be protected from accesses without prior acquisition of a lock that protects it.* This requirement is reversed: an `unlock` operation prohibits all further accesses to this struct and its members up to the next `lock` operation. Memory

[5] Completeness refers to the spinlock part of the API.

```
spin_lock.exit{
  // case 1: parameter is the first
  // monitored lock
  if(which_first_lock!=NULL &&
     ($1 == which_first_lock)){
     if (first_lock_locked==0){
      if (second_lock_locked==1) {
        if (!order_set) {
          first_before_second = 0;
          order_set = 1;
        } else
          if (first_before_second) {
          abort "Lock order viol.";
          }
      }
     first_lock_locked = 1;
  } else {
      abort "Double acquire!";
  }
  // case 2: parameter is the second
  // monitored lock
  ...
}
```

```
irq_return_t interrupt_handler() {
       spin_lock(lock2);
       spin_lock(lock1);
       ...
       spin_unlock(lock1);
       spin_unlock(lock2);
}

void device_read() {
       spin_lock(lock1);
       if (nondet_bool())
           interrupt_handler();
       spin_lock(lock2);
       if (nondet_bool())
           interrupt_handler();
       // ...
       spin_unlock(lock2);
       if (nondet_bool())
           interrupt_handler();
       spin_unlock(lock1);
       if (nondet_bool())
           interrupt_handler();
}
```

Fig. 3. (l) Excerpt from the specification implementation of the locking restriction (LO). The `which_*_lock` pointers track the pair of locks that is currently monitored. `order_set` tracks if one lock order has been determined on the current trace. (r) This example shows a common locking situation between an interrupt handler and a driver service function. Both functions operate on a pair of locks, but the interrupt handler uses a different locking order. Using partial simulation of the preemption, this deadlock is discovered by Avinux.

```
 // annotation omitted
 1: spin_lock(&lock);
 2: driver->request_nr++;
 3: spin_unlock(&lock);
// nondet_bool() implements *
  if (which_lock==&lock) {
    if (nondet_bool()) {
         object_destroyed = 1;
         free(driver);
    }
  }
// invalid access:
 4: driver->request_nr++;
 5: spin_lock(&lock);
  if (which_lock==&lock) {
     if (object_destroyed) while(1);
  }
 6: driver->request_nr++;
 7: spin_unlock(&lock);
// annotation omitted
```

```
spin_lock.entry {
  if (which_lock==$1) {
    if (object_destroyed==1)
       // Terminate trace
      __CPROVER_ASSUME(0);
  }
}

spin_unlock.exit {
  if(which_lock==$1) {
    if(*) {
      object_destroyed = 1;
      kfree(which_object);
    }
  }
}
```

Fig. 4. (l) A code example for race condition detection. The code in unnumbered lines on the left side is inserted by the SLICx rule on the right side (r).

accesses cannot be directly annotated with either SLIC or SLICx. CBMC also offers no way to insert additional checks for each memory access. Therefore built-in memory checks from CBMC are exploited.

Consider the code excerpt in Figure 4. The `struct driver` is protected by the spinlock `lock`. Possible driver code is presented on the left. Line 4 contains an unprotected access that may lead to data races if the code is reentrant. The application of the rule (right) inserts the unnumbered lines into the driver's code where

- `which_lock` refers to the lock that is monitored.
- `$1` refers to the first parameter of the lock / unlock functions. In the example it is replaced on the left side by `&lock` as it is the only lock instance that occurs.
- `__CPROVER_ASSUME(0);` is a CBMC statement to terminate the execution.

In order to make CBMC report the access in line 4, `free` is non-deterministically called on `driver`. CBMC‘s memory checking ensures that all memory accesses occurring after free will be reported (line 4). Additionally false positives shall be removed that arise when the previously deallocated struct is locked—i.e. protected—again in line 5. This restoration of the object’s state is achieved by a termination of the current execution if `driver` had been deallocated before. An additional complication is that reallocations of `driver` could occur between line 3 and 4. This problem can be solved by annotating allocation functions. The relationship between locks and protected structs can be heuristically inferred from Linux conventions as locks are commonly embedded in the structs they protect.

# 4 Producing Results

## *4.1 Checking thousands of Device Drivers*

In order to demonstrate the work Avinux automates an execution example is given. This process that has exposed a new bug in `drivers/ide/ide-probe.c`. The source file is part of the IDE subsystem, i.e. `ide-probe.c` contains functions for the detection and identification of IDE devices. The tested Linux version is the vanilla kernel 2.6.19. The bug persists at least from kernel version 2.6.0, but has not been detected by testing and code reviews.

Linux is configured with the `allyesconfig` make target which includes most modules and subsystems that may be used on the Pentium 4 (64 bit) host. Note that CBMC is compiled as a 32 bit application running on the same host. The size of Linux simple data types is set to values typical for a 64 bit system, i.e. CBMC interprets integers as 64 bit vectors. As a next step all source files in the `drivers` folder are annotated with the following specification.

### *4.1.1 SLICx Rule*

Several SLICx rules have been formulated and checked. To demonstrate the process and the language one of the is illustrated in greater detail.

Figure 5 shows the SLICx rule that is checked for all source files separately. The rule expresses that it is not allowed to call `kfree` on a pointer without prior reallocation through `kmalloc`. Two calls to `kfree` without intermediate allocation are called *double-free* error.

The state tracking of a possibly infinite number of heap memory objects is implemented by non-deterministic selection. This mechanism is called *Universal Quantification Trick* and was first applied in software model checking in [10].

The SLICx compiler converts the rule into global state parameters and C-style transfer functions with a `__SLIC` prefix (Fig. 5, right). In each driver, Avinux replaces all occurrences of `kmalloc` and `kfree` by calls to wrapper functions that trigger the SLICx transfer functions. If a trace exists that violates the specification, CBMC will report that the `assert(0)` in `__SLIC_ERROR` statement is reachable.

### *4.1.2 Transforming the driver*

Avinux internally invokes the make build process using the code transformation wrapper `cilly` (CIL). The kernel transformation takes approximately three hours

```
state{
  int allocated = 0;
  int has_target = 0;
  void *which_chunk = NULL;
}

kmalloc.exit{
  if(has_target==0){
    if(*){
      which_chunk = $return;
      has_target = 1;
    }
  }
  if(has_target==1 &&
     which_chunk==$return &&
     $return>0) {
    allocated = 1;
  }
}

kfree.entry{
  if(has_target==1 &&
     $1 == which_chunk){
    if (allocated==0) {
      abort "Invalid free!";
    }
    allocated = 0;
  }
}
```

```
// state variables
int __SLIC_state_allocated=0;
int __SLIC_state_has_target=0;
int __SLIC_state_failed=0;
void * __SLIC_state_which_chunk = NULL ;
// other declarations
void __SLIC_ERROR(char* error_string){assert(0);}
// kmalloc transfer functions
...
// kfree transfer and wrapper functions
void __SLIC_fun_kfree_entry(const void* __SLIC_arg_1){
 if (__SLIC_state_has_target == 1 &&
    (__SLIC_arg_1 == __SLIC_state_which_chunk ) ){
    if (__SLIC_state_allocated == 0 ) {
      __SLIC_ERROR( "Invalid free!" );
    }
    __SLIC_state_allocated = 0;
 }
}
// wrapper function
void __SLIC_kfree(const void* __SLIC_arg_1) {
  __SLIC_fun_kfree_call( __SLIC_arg_1);
  __SLIC_fun_kfree_entry( __SLIC_arg_1);
  kfree( __SLIC_arg_1);
  __SLIC_fun_kfree_exit( __SLIC_arg_1);
  __SLIC_fun_kfree_return( __SLIC_arg_1);
}
```

Fig. 5. The SLICx rule for double-free errors is shown on the left. `has_target` is set to 1 if an address is chosen to be monitored. `which_chunk` refers to the currently monitored address. `allocated` captures the current status concerning allocation respectively deallocation. The SLICx compiler translates the rule into C-style transfer functions (right) that are woven into every driver. Note that empty functions and the `kfree` transfer functions are omitted on the right side.

and the result is a collection of a few thousand `cil.c` files. Theses files have the following characteristics:

- The file contains the code from the original driver or subsystem and from all included kernel headers. External function definitions and all information that would normally be linked to the driver is not included.
- After the CIL transformation all statements and expressions are side-effect free, e.g. expressions like `*p++` are split up using temporary variables.
- Each occurrence of the functions `kmalloc` and `kfree` are replaced by wrapper functions that contain the translated SLICx transfer functions.

After processing the source files additional transformations must be made in order to make the source files acceptable as input for CBMC. CLEANC takes the following actions:

- Replace `enums` by `int` constants.
- Remove `__attribute__((...))` tags.
- Remove all variations of `asm` and `volatile`. Inline assembler blocks are removed without replacement (may lead to both, false positives and false negatives).
- Remove `inline` and `register` declarations.
- Replace empty structs: `struct struct_type_name { };` is replaced by `struct struct_type_name { int tmp; };` .
- Replace `__alignof__(expr)` by 1.
- Remove `(char*)` casts in front of constant strings.
- Extend declaration of empty arrays: `int x[0];` becomes `int x[1];`.

- Truncate unsigned long long constants: `0x0ULL --> 0x0UL`
- Replace incomplete types:
  - · Change `extern void x;` into `extern int x;`.
  - · Convert `extern int x[];` into `extern int * x;`.

Some of these issues are fixed with CBMC version 2.5. CLEANC is implemented as a Perl script except for the removal of `enums`. This feature is implemented using the Eclipse CDT code parsing framework to avoid complex regular expressions.

### *4.1.3 Analyzing the driver collection*

It is infeasible to manually create control logic environments for thousands of device drivers. The minimal information that must be provided for CBMC is the enumeration of possible entry points (functions) for each module, but even this abstraction poses an obstacle for automatic analysis. Simply enumerating all defined functions per file is incorrect as it is not known whether functions are actually called from outside the module. Fortunately Linux limits the visibility of symbols such that only symbols that are explicitly passed to a macro `EXPORT_SYMBOL` may be referenced from outside the module. Minimal control logic environments are therefore obtained by creating one environment per function symbols passed to `EXPORT_SYMBOL`. Each control logic environment only contains one call the function.

The heuristic can be refined as described in Subsection 4.1.6.

CBMC checks whether for each of the obtained drivers and for each exported function symbol, the error state is reachable. The whole process took approximately 36 hours for 1514 translation units. 95 counter-examples could be obtained of which only one could be proven to be a bug. The CBMC loop unwind depth was set to one, e.g. every loop skipped or executed exactly once.

In order to avoid problems arising from incomplete environments and potentially uninitialized interface objects, CBMC is configured to check `assert` claims only. Memory safety violations and other built-in checks were disabled. The analysis was performed using a patched version of CBMC 2.4 as newer versions of CBMC are not distributed as source code.

### *4.1.4 Error Example*

Upon detection of a potential IDE device, the kernel invokes the function `do_probe` in `ide-probe.c`. It is possible that in `do_probe` a previously allocated memory chunk `drive->id` is deallocated. This is done via calls to `try_to_identify`, `actual_try_to_identify`, `do_identify` and finally `kfree`. The deallocated memory is used without safety check in line 579:

```
strstr(drive->id->model, "E X A B Y T E N E S T")
```

Instead of reporting this *use-after-free* error, CBMC reported a *double-free* error. CBMC's memory checks were disabled, therefore the *use-after-free* could not be detected directly. However, checking for *double-free* errors exposed a trace where the deallocated memory was passed to `kfree`. In concrete executions, the *double-free* bug cannot occur due to the previous *use-after-free* error.

### 4.1.5 More Empirical Results

*BLASTing Linux code* [14] is a technical case study that summarizes the transformations necessary to use the model checker BLAST on real Linux Device Drivers. Their error samples provide an informal micro-benchmark in order to test how many errors Avinux exposes.

Rediscovery of the given error examples could be achieved without the described manual code transformations (cf.[14]). The construction of the environment model was the only manual task for each driver. Other work such as the creation of specifications and header files had to be done once per kernel version. The time needed to analyze the drivers was clearly dominated by the compilation of the Linux kernel using CIL (several hours). The CBMC runtime for a single driver was in the range of minutes[6] .

In this way six out of eight memory safety errors were reproduced. So far, only three out of eight race condition and deadlock examples could be reproduced as for these cases a detailed operating system model needs to be provided.

While testing Avinux on functions with variable parameters a second bug in the file `drivers/char/tpm/tpm.c` was identified:

```
line 1130: devname = (char *)__SLIC_kmalloc(7U, 208U);
line 1131: scnprintf(devname, 7UL, "%s%d", "tpm", chip->dev_num);
```

Taking into account that memory allocation via `kmalloc` may fail—the return value may be `NULL`—the code excerpt violates the preconditions of `scnprintf`. In `scnprintf` the buffer devname is read without sanity checks. The source stems from kernel version 2.6.19.

In addition to the above examples the following new bugs were found (kernel version 2.6.20.4):

- Usage of a lock without initialization (`drivers/char/rtc.c`).
- A deadlock may occur due to a race between an interrupt handler and the module initialization in `/drivers/net/wan/sbni.c`.
- A possible dereference of a null pointer may occur in `fs/xfs/xfs_da_btree.c`.
- Invalid use of the `kmem_cache_free` interface in file `drivers/infiniband/mad.c`.

More details for these errors can be obtained through the project website.

### 4.1.6 Coverage

For a complete analysis one needs to cover all possible traces that lead to a possible violating call to `kmem_cache_free`. In order to get a full path coverage for all possible scenarios in which the function under test may be invoked, a transitive closure on the static call tree is computed. The closure starts at the function under test and the used operator is *called-by*. Thereby it may be safely assumed that all possible entry points into the module are modelled.

If a function `foo` is included in more than one driver by means of preprocessing directives, a verification task is created for the first occurrence of this function

[6] However, this does not include the manual creation of the main function. This task took days as internal workings of subsystems are poorly documented .

Table 1
Summary of the performed experiments.

| **Description** | **Kernel** | **Result** |
|---|---|---|
| Rediscovery of memory safety related errors [14], | 2.6.11-14 | 6 out of 8 errors discovered with minor manual changes. |
| Rediscovery of race and deadlocks [14], | 2.6.13-15 | 3 out of 8 errors discovered. |
| Double-free error analysis of all files, | 2.6.19 | 1 Error found in `ide-probe.c`. |
| Test effectiveness of DEC when analyzing `kmem_cache_free` calls in all files. | 2.6.20.4 | False positive rate reduced by more than 50%. |

definition.

### *4.1.7 Testing DEC*

Though the creation of an environment may underapproximate the real systems behavior, it is commonly a very effective mean to rank error reports. Errors reports that occur when the module is called without an environment, but not when every input variable is initialized, are false positives with high probability. A reasonable treatment of these cases is to set up a new specification encompassing the requirement that the input is initialized. This inferred property could then be checked for all callsites on the module entry point.

In order to test the effectiveness of the environment creation, a second specification was analyzed: is `kmem_cache_free` called with its second parameter being `NULL`? Note that the aim of this second experiment is to evaluate whether DEC may reduce the number of false positives.

DEC was tested on a set of 589 functions that call `kmem_cache_free` directly or through intermediate functions. Without DEC, CBMC reports possible violations for 194 test samples. Using DEC, this number is reduced to 81 error reports. The inspection of 81 bug reports is still a matter of days. One of the most promising planned improvements is the automatic abstraction of unknown interface functions that may return potentially invalid interface objects.

DEC was configured with an unwinding depth of two. The unwinding of the object graph was limited to entry function parameters—globally uninitialized pointers were not initialized.

Table 2 shows approximate runtimes for all verification stages. Table 1 gives a summary of the performed experiments.

### *4.2 Limitations*

In this case study Avinux suffers the following limitations:

- Automatic source code linking of strongly interdependent translation units.
- Usability problems for the downloadable Eclipse plugin.
- Some GCC flags in newer kernels break CIL and must be removed manually - for example `-fno-stack-protection`.
- Uncovered syntactic problems - for example parameter declarations like `proc_handler proc_handler`, where the first occurrence refers to a type, while the latter is a parameter name.
- Limited coverage of concurrent executions.

Table 2
Comparative list of approximate runtimes of all verification stages (Kernel under test: Linux 2.6.20.4, `allyesconfig` configuration; Host: Pentium 4 with 3 Ghz, 1 GB RAM). For a given SLICx rule and a given operating system model the process bottleneck is the verification by CBMC.

| **Stage** | **Runtime** | **Comment** |
|---|---|---|
| Configuration | 1 min | Enabling all features |
| Create SLICx rule | 1 min - weeks | Concurrency and subsystem modelling may be necessary. |
| Automatic annotation | 1h | Annotation of drivers. |
| Automatic annotation | +2h | Annotation of all sources (network, file systems,...). |
| Kernel build using CIL | 1h - 5h | Highly configuration and architecture dependent |
| Merging of manually selected translation units. | 1 min per merge | Merging actions must be specified manually. |
| CLEANC | 30 min | |
| Environment modelling | 1 min - weeks per model | Dependent on subsystem complexity and documentation. |
| DEC | 2h | |
| Verification by CBMC | 6h - weeks | Checking one SLICx rule in all source files, e.g. 36h for checking the `kmem_cache_free` rule. |

Avinux is currently extended such that it automatically fixes all of the above concerns. Nevertheless each new kernel version may produce new problems that may require manual inspection and fixes.

# 5 Related Work

Several other groups have applied formal methods to Linux related software.

MOPS [6] is a software model checker that has been applied to checking security properties of various Unix applications. Though MOPS detected some security flaws, the authors report problems concerning error reporting and integration into the software build process. The authors indicate that at the final project stage the analysis of software packages took weeks.

Engler et al. [11] provide various case studies applying model-checking and lightweight techniques to Linux code. Their detailed descriptions lead them to the following conclusion: modular model checking often requires too much manual effort concerning environment and specification modelling. They abandoned classical modular model checking and developed a custom model checker (CMC) that was run on complete Linux kernels. However, most of the time CMC did not terminate [15].

*BLASTing Linux code* [14] is a detailed case study that demonstrates problems applying the model checker BLAST on Linux drivers. The authors indicate that BLAST cannot even rediscover known bugs in Linux drivers.

The Static Driver Verifier project [1] is most successful in checking Windows device drivers automatically. The authors describe that the success of the whole project crucially depends on a very detailed model of the operating system using the driver under test. SDV revealed many bugs, but is currently not available for custom specifications. Moreover it does not provide a Linux operating model and GNU C extension support. It is unknown if modular model checking for Linux can be automated such that it is valuable from a practitioners point of view.

KISS and TCBMC are two examples for the treatment of parallel program executions. In addition to how Avinux simulates interrupt handler code, KISS [19] annotates each statement with code that simulates a possible termination of the interrupting code. In general, KISS simulates strictly more traces than pure preemption simulation, but in the example these additional traces may only occur in multi-processor settings. TCBMC [20] is an extension of CBMC that simulates a bounded number of possible context switches. This modelling of parallel executions contains many more traces than modelled by Preemption Simulation, but it is unclear how TCBMC would scale in a setting of thousands of lines of code. In the limited domain of preemption by interrupt handlers, the presented approach is computationally feasible, as indicated before.

Witkowski et al. [23] provide a predicate abstraction based tool—DDVerify—for the automatic analysis of concurrent Linux device drivers. DDVerify is strictly more expressive as it provides full model checking capabilities combined with unrestricted shared memory parallelism. The tool also covers the creation of environment models for a small set of drivers by providing templates for common driver architectures. These templates are more elaborate than the generic models of Avinux, but they provide less coverage than manually created ones. The authors report the discovery of two new bugs arising in a sequential execution of a driver. It is unclear, to how many drivers DDVerify has been applied. The available version of DDVerify contains 18 examples encompassing a total of 33k lines of code. Avinux provides fewer coverage than DDVerifiy but it has successfully been applied to several versions of Linux with several millions lines of code.

## 6 Summary

Prior experiments with several software model checkers were disappointing. Not even one software model checker for C was able to accept single device drivers as input [14]. Manually transforming drivers made them processable but still even known bugs could not be rediscovered. In contrast to many other projects large quantities of Linux modules with millions of source code lines should be checked. The need for an integrated solution lead to the development of Avinux. Presented results show that thereby thousands of device drivers could be checked where others works limit their scope to a few examples [14,4].

The motivation for the development of Avinux was the large manual overhead necessary for every single driver to prepare them for bounded model checking. In the authors personal view, this overhead per driver has been significantly reduced. Once generic specifications and operating systems models are provided, bounded model checking results can be obtained in a few minutes for a single driver. However, the unsoundness of modular analysis used in this work induces a large overhead when inspecting error reports from the model checker.

Verification examples can be found on the project website. A preliminary version of Avinux is available for download.

The focus of Avinux lies in the automatic preparation and discovery of verification tasks. Other state-of-the-art verification approaches for C—e.g. Cascade [22], TCBMC [20], F-Soft [13] or the novel extension of the model checker MAGIC by Chaki et al. [5]—could be easily integrated.

The use of Avinux is currently limited by a lack of automatic inference mechanisms for complete operating system models. DEC currently implements an effective strategy for data environment creation that reduced the number of false positives in the experiments. A lack of preconditions on entry points in modules leads to many false positives that make cases studies a time-consuming task. Avinux facilitates across-the-board case studies in the Linux domain.

Future work will concentrate on automatic creation of complete environment models for Linux. Ball et al. provide such test drivers manually for Windows drivers [1]. Inspired by the success of the tool SDV it seems reasonable that Avinux has great potential in improving the quality of Linux device drivers. Six novel bugs have already been discovered. One of those has already been patched in the mainline Linux kernel.

Avinux has produced large sets of error reports which might contain many more real errors. Each report must be analyzed manually to check whether it is real or spurious. The authors have performed this task for hundreds of counter-example traces, but thousands of inspections are still due. A possible solution to this problem would be help from the Linux community.

## References

[1] Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani and A. Ustuner, *Thorough static analysis of device drivers*, in: *EuroSys Conf., Proc.* (2006), pp. 73–85.

[2] Ball, T. and S. K. Rajamani, *SLIC: A specification language for interface checking*, Technical report, Microsoft Research, MSR-TR-2001-21, [21 August 2007] (2001).
URL http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2001-21

[3] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs.*, in: *Proc. TACAS'99*.

[4] Chaki, S., E. Clarke, A. Groce, S. Jha and H. Veith, *Modular verification of software components in C*, in: *25th Intl. Conf. on Software Engineering (ICSE), Proc.* (2003), pp. 385–395.

[5] Chaki, S., E. M. Clarke, N. Kidd, T. W. Reps and T. Touili, *Verifying concurrent message-passing C programs with recursive calls.*, in: H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 12th Intl. Conf., Proc.*, LNCS **3920** (2006), pp. 334–349.

[6] Chen, H., D. Dean and D. Wagner, *Model checking one million lines of C code*, in: *Network and Distributed System Security Symposium (NDSS), Proc.* (2004), pp. 171–185.

[7] Chen, H. and D. Wagner, *MOPS: an infrastructure for examining security properties of software*, in: *9th ACM Conf. on Computer and Communications Security (CCS), Proc.* (2002), pp. 235–244.

[8] Clarke, E., D. Kroening and F. Lerda, *A tool for checking ANSI-C programs*, in: K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS **2988** (2004), pp. 168–176.

[9] Coffman, E. G., M. Elphick and A. Shoshani, *System deadlocks*, ACM Computing Surveys **3** (1971), pp. 67–78.

[10] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby and H. Zheng, *Bandera: extracting finite-state models from Java source code*, in: *Software Engineering (ICSE),22nd Intl. Conf. ,Proc.* (2000), pp. 439–448.

[11] Engler, D. R. and M. Musuvathi, *Static analysis versus software model checking for bug finding*, in: B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI), 5th Intl. Conf., Proc.*, LNCS **2937** (2004), pp. 191–210.

[12] Etessami, K. and S. K. Rajamani, editors, "Computer Aided Verification (CAV), 17th Intl. Conf., Proc." LNCS **3576**, Springer Verlag, Edinburgh, Scotland, UK, 2005.

[13] Ivancic, F., Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter and P. Ashar, *F-Soft: Software verification platform*, in: Etessami and Rajamani [12], pp. 301–306.

[14] Mühlberg, J. T. and G. Lüttgen, *BLASTing Linux Code*, in: *11th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS), Proc.*, 2007, pp. 211–226.

[15] Musuvathi, M. and D. R. Engler, *Model checking large network protocol implementations*, in: *1st Symposium on Networked Systems Design and Implementation (NSDI), Proc.* (2004), pp. 155–168.

[16] Necula, G. C., S. McPeak, S. P. Rahul and W. Weimer, *CIL: Intermediate language and tools for analysis and transformation of C programs*, in: *11th Intl. Conf. on Compiler Construction (CC), Proc.* (2002), pp. 213–228.

[17] Post, H. and W. Küchlin, *Automatic data environment construction for static device drivers analysis (poster abstract)*, in: *Conf. on Specification and verification of component-based systems (SAVCBS), Proc.* (2006), pp. 89–92.

[18] Post, H. and W. Küchlin, *Integration of static analysis for linux device driver verification*, in: J. Davies and J. Gibbons, editors, *Integrated Formal Methods (IFM), 6th Intl. Conf. , Proc.*, LNCS **4591** (2007), pp. 518–537.

[19] Qadeer, S. and D. Wu, *KISS: keep it simple and sequential*, , **39** (2004), pp. 14–24.
URL http://portal.acm.org/citation.cfm?id=996845

[20] Rabinovitz, I. and O. Grumberg, *Bounded model checking of concurrent programs*, in: Etessami and Rajamani [12], pp. 82–97.

[21] Sen, K., D. Marinov and G. Agha, *CUTE: A concolic unit testing engine for C*, in: *ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE)*, 2005, pp. 263–272.

[22] Sethi, N. and C. Barrett, *Cascade: C assertion checker and deductive engine*, in: T. Ball and R. B. Jones, editors, *Computer Aided Verification (CAV), 17th Intl. Conf., Proc.*, LNCS **4144** (2006), pp. 166–169.

[23] Witkowski, T., N. Blanc, D. Kroening and G. Weissenbacher, *Model checking concurrent Linux device drivers*, in: R. E. K. Stirewalt, A. Egyed and B. Fischer, editors, *22nd IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)* (2007), pp. 501–504.