# Verification of Linux Kernel Modules: Experience Report

Willem Penninckx, Bart Jacobs, Jan Smans, Jan Tobias Mühlberg

Department of Computer Science, K.U.Leuven, Belgium

**Abstract.** Modern operating systems such as Windows and Linux can be extended with kernel modules that can be loaded and unloaded at run-time. However, writing correct, memory safe kernel modules – in particular device drivers – has proven to be a notoriously difficult task for several reasons: concurrent callbacks, dynamic loading and unloading and a sophisticated, but complex API with many implicit rules.
In this paper, we report on our experience in modularly proving the absence of memory errors and API usage violations in a Linux kernel module using a generic, separation logic-based program verifier. More specifically, we describe how the absence of data races can be proven in the presence of concurrent callbacks, and how memory leaks and use-after-free bugs can be avoided when unloading. Moreover, we propose a formal specification for part of the Linux kernel API and prove that the module does not violate the API usage rules.

## 1 Introduction

The safety and security of today's omni-present computer systems critically depends on the reliability of operating systems (OS). Due to their complicated task of managing a system's physical resources, OSs are difficult to develop and to debug. As recent publications show, most defects causing operating systems to crash are not in the system's kernel but in the large number of operating system extensions available [19,6]. In Windows XP, for example, 85% of reported failures are caused by errors in device drivers [1]. As [6] explains, the situation is similar for Linux and FreeBSD: error rates reported for device drivers are up to seven times higher than error rates stated for the core components of these operating systems. In this paper we present a case study on applying the VeriFast [12] toolkit to verify memory safety and the absence of data races in a simple Linux kernel module.

***Linux Kernel Modules.*** Linux and its modules are written in C. The main documentation[1] to its APIs is provided with the freely available source code of the OS. The reader is also referred to [5,10] for an introduction to the Linux's internals. In brief, a Linux kernel module is a software component which extends the Linux OS kernel with certain features, e.g., the module may provide a cryptographic routine, or add support for a particular hardware to the OS. Linux

[1] Linux kernel sources and documentation: `http://www.kernel.org`.

kernel modules can be dynamically loaded and unloaded at runtime. To facilitate integration with the kernel, a module has to implement a set of kernel APIs. Most basically, each module has to provide an initialization function, which is invoked by the kernel right after the module has been loaded. This function is responsible for setting up the working environment of the module, usually involving hardware detection, allocation and initialization of data structures, and registering callback pointers to the driver's functions with the kernel. Afterwards the kernel may make use of the functionality provided by the driver by directly calling the driver's functions. Subsequently, each module also provides a cleanup function which is executed when the driver is unloaded. This function has to safely unregister all callbacks and release all resources claimed by the module. Of course there are further APIs which facilitate, e.g., memory management, communication with hardware and network access.

Developing error free kernel modules is particularly important because the module's code is executed as part of the Linux kernel itself. Hence, if a module performs actions such as de-referencing an invalid pointer, the behavior of the entire system is undefined. Since Linux does not provide garbage collection, it is also highly important that modules eventually release all resources they claim. Furthermore, being designed for a range of multiprocessor platforms, the Linux kernel is inherently concurrent. This concurrency is exposed to device drivers as well: each of the drivers callback functions may be invoked by an arbitrary number of concurrently executing Linux kernel threads. To avoid data races, Linux provides several mechanisms that implement mutual exclusion, protecting data structures shared between different kernel threads. To use these mechanisms correctly, however, is left to the module's developer.

***Our Contributions.*** Consequently, automated tools capable of verifying generic API contracts for OS extensions are highly desirable, yet not available to practitioners. In this paper we report on our experience in verifying a Linux kernel module. The contributions of our paper are threefold. Firstly, we present a toy Linux kernel module called *Helloproc* as an extensible case study on the feasibility of verifying Linux device drivers. Although Helloproc comprises only about 30 lines of C source code, it required us to meet all the challenges outlined above. Secondly, we propose a formal specification for part of the Linux kernel API in terms of separation logic [18]. Our specification ensures that a kernel module is free of data races in the presence of concurrent callbacks, and also that the module is free of memory leaks and use-after-free bugs when it is unloaded. Thirdly, we show how VeriFast is employed to efficiently prove that the Helloproc module satisfies the specification.

The remainder of this paper is organized as follows: In Sec. 2 we give a brief overview on VeriFast and separation logic, followed by an introduction to the Linux kernel's APIs and our specification of contracts for these APIs in Sec. 3. Our experience with developing a simple but correct Linux kernel module is presented in Sec. 4. In Sec. 5 we discuss related tools and case studies on the verification of OS components. Finally, in Sec. 6, we draw conclusions and outline future directions of research.

## 2 Background on Separation Logic

The tool we used to verify the kernel module is based on separation logic. Separation logic [16] is a logic for reasoning about concurrent, imperative programs with aliasing. A program that satisfies a separation logic specification is free from memory errors and data races. At the heart of separation logic lies the notion of permissions. That is, a function can only access a memory location if it has permission to do so. For example, *integer(p, 10)* denotes *(1)* that the integer pointed to by `p` is a valid memory location, *(2)* that the current function is allowed to access (read and write) that memory location and *(3)* that its current value is `10`.

To distinguish full (read and write) from read-only access, permissions are qualified with fractions [4] between 0 (exclusive) and 1 (inclusive), where 1 corresponds to full access and any other fraction denotes read-only access. For example, *[f]integer(p, 10)* denotes read-only access if *f* is less than 1 and full access if *f* equals 1. We typically omit *[1]* in full permissions. Permissions can be split and merged as required during the proof. For example, a thread that holds the permission *integer(p, 10)* can start two threads that both have permission to read `p` by passing *[1/2]integer(p, 10)* to each thread.

To abstract over the set of permissions required by a function, permissions can be grouped and hidden via predicates. For example, consider the predicate named *interval* shown below:

```
/*@
predicate interval(int* i, int low, int high) =
  integer(p, low) &*& integer(p+1, high) &*& low <= high;
@*/
```

This predicate groups the permissions to dereference `p` and `p + 1`, and additionally states that the value stored at `p` is less or equal to the value stored at `p+1`. That is, *interval(i, low, high)* can be considered to be a shorthand for the assertion *integer(p, low) &*& integer(p+1, high) &*& low <= high*. Predicates can be split and merged into fractions just like basic permissions as required during the proof.

Each function in the program has a corresponding contract, consisting of a precondition and a postcondition that respectively describe the permissions required and returned by the function. More specifically, the permissions described by the precondition conceptually move from the caller to the callee on entry to the function, and vice versa for the postcondition. As an example, consider the function `shift` shown below:

```
void shift(int *i, int v)
  //@ requires interval(i, ?low, ?high);
  //@ ensures interval(i, low + v, high + v);
{
  *i = *i + v;
  *(i + 1) = *(i + 1) + v;
}
```

shift's precondition states that the function may only be called if i points to a valid interval (where the meaning of "valid interval" is defined by the aforementioned predicate *interval*). The precondition imposes no restrictions on the interval's bounds, but binds the lower bound to the variable low and the upper bound to high. The postcondition guarantees that when the function returns i is still a valid interval, but its bounds have been shifted by v with respect to the function pre-state. Note our verification tool requires that annotations such as the predicate *interval* and shift's function contract are placed inside special comments (*/*@ ... @*/*) which are ignored by the C compiler but recognized by our verifier.

Programs typically rely on function type definitions to define the signature of function pointer parameters. In our verification approach, a function contract can be associated with each such function type definition. These function contracts can be written in an abstract manner via predicate families. Contrary to the regular predicates described above, predicate families can have multiple definitions, depending on an additional parameter called the predicate family index (typically the address of a function). For example, the function type definition equals shown below is specified in terms of the predicate family *equals_data*. Because the definition of *equals_data* depends on the function used as an index, any two data structures can be compared. For example, the function interval_equals satisfies the contract of equals (indicated by the *//@: equals* annotation), because *equals_data* is defined to hold in the context of this function for two pointers x and y if both pointers refer to valid intervals.

```
//@ predicate_family equals_data(void *index)(void *x, void *y);

typedef int equals(void *x, void *y);
  //@ requires equals_data(this)(x, y);
  //@ ensures equals_data(this)(x, y);

/*@
predicate_family_instance equals_data(interval_equals)(int *x, int *y) =
  interval(x, _, _) &*& interval(y, _, _);
@*/

bool interval_equals(int *x, int *y) //@: equals
{
  return *x == *y && *(x+1) == *(y+1);
}
```

VeriFast [12] is a verifier for C programs annotated with separation logic annotations. VeriFast modularly checks via symbolic execution that each function satisfies its contract. If the tool proves correctness of a C file, then all functions in that file are free from memory errors and data races, and all developer-defined assertions are guaranteed to hold (provided the functions defined in the file are only called in states satisfying their preconditions and all library functions satisfy their contract). The kernel module described in this paper was verified using VeriFast.

## 3 Specification of the Linux Kernel API

The kernel module we verify is written against a subset of the Linux Kernel API. In this section, we discuss the relevant kernel functions and provide a specification. This specification is not tailored to the specific module discussed in Section 4, but instead is generic and can be used for verifying arbitrary modules. Note that we did not formally verify whether the kernel's implementation conforms to this specification, as our main goal is checking correctness of modules, not of the Linux kernel itself.

The program verifier we use in this paper, VeriFast, cannot parse certain C constructs (such as macros) that are used in kernel header files. For that reason, we wrote a shallow wrapper around the kernel API and verified our module against this wrapper. The wrapper's header files do not contain any problematic constructs, and can hence be interpreted by VeriFast. The wrapper's implementation mostly delegates function calls to the corresponding kernel functions.

***Initialization and Cleanup.*** The Linux kernel assumes that each module has an initialization and a cleanup function. The former function is called by the kernel after the module is loaded into memory. The goal of this function is to initialize the module's data structures and to allocate resources. Similarly, the kernel calls the latter function right before unloading the module such that the resources it acquired can be freed.

The specification of the initialization and cleanup functions should enforce the following rules:

1. Kernel modules typically declare a number of global variables. These global variables share the lifetime of their module: they are allocated when the module is first loaded and freed when it is unloaded. The specification should *(a)* allow the module to access its global variables while is it is loaded into memory and *(b)* enforce that the module's globals, code and string literals are no longer accessed after unloading to prevent illegal memory accesses.
2. To accomplish its task, a kernel module can allocate and use kernel resources such as locks, files, etc. The specification should enforce that a module does not leak resources. That is, after the module is unloaded, all resources acquired by the module should have been released[2].
3. It should be possible to link up the initialization and cleanup functions. That is, the permissions that remain at the end of the initialization function should be passed to the cleanup function. This set of permissions is module-specific.

The code listing shown below defines the function contracts of the initialization and cleanup functions. Both functions have a ghost parameter[3] called `module` which denotes the module's unique identifier. The function contracts enforce the rules outlined above as follows.

[2] One can overrule the leak checking mechanism by explicitly using the `leak` command (but only for resources that do not endanger safety of unloading).

[3] Ghost variables are used only during verification, but ignored by the C compiler.

The predicate *module* groups the permissions to access the module's global variables. For each module, the definition of this predicate is generated by VeriFast based on the global variables defined by the module. If the parameter *initialValues* is true, then the globals still have their initial values (as given by their initializer). The first rule is enforced by placing *module(module, true)* in the precondition of the initialization function (such that it can access the global variables) and *module(module, _)* in the postcondition of the cleanup function (such that the permissions for the global variables all return to the kernel when the module is unloaded). The underscore (_) in the postcondition indicates that it is not necessary for the globals to have their initial values. Note that the initialization function can return a non-zero value to indicate initialization failed. If initialization fails, the kernel immediately unloads the module without calling the cleanup function. Therefore, the permissions *module(module, _)* must also be returned to the kernel in case initialization fails.

The predicate *cleanup_debt* is used to ensure that no resources are leaked after unloading (rule 2). That is, *cleanup_debt(i)* denotes that the module has allocated *i* resources. Initially, the debt is zero. Whenever the module allocates a resource, the debt is incremented by one and vice versa, when a resource is freed, the debt is decremented by one. The absence of resource leaks is enforced by checking that the debt is zero at the end of the cleanup function and in case initialization fails.

Finally, the predicate *module_state* groups the permissions that must be passed from the initialization function to the cleanup function. The definition of *module_state* is module-specific and can be defined by the implementer of the module. The predicate links up the initialization and cleanup functions as it both occurs in the postcondition of the former function and in the precondition of the latter. Note that the permissions stored inside *module_state* cannot be accessed by the module after the initialization function returns until the cleanup function is called.

```
/*@
predicate module(int moduleId, bool initialValues);
predicate module_state();
predicate cleanup_debt(int debt);
@*/

typedef int module_setup/*@(int module)@*/();
  //@ requires module(module, true) &*& cleanup_debt(0);
  /*@ ensures result == 0 ?
        module_state()
      :
        module(module, _) &*& cleanup_debt(0); @*/


typedef void module_cleanup/*@(int module)@*/();
  //@ requires module_state();
  //@ ensures module(module, _) &*& cleanup_debt(0);
```

***The procfs File-System.*** The proc API allows kernel modules to offer their services to user-space processes via files (under `/proc`). More specifically, a module can create a virtual file using the proc API. This file is virtual in the sense that the module can register callbacks such that open, read, write and close operations on the file are translated into invocations of the callback functions registered by the module.

In the wrapper API, each proc file must be placed inside a subdirectory of `/proc`. In specifications, directories are represented via the predicate *procfs_dir*. The predicate's first parameter is a pointer to a `procfs_dir` struct, while its second parameter denotes the number of files created by the current module in that directory. The definition of the predicate *procfs_dir* is hidden, as it is only needed when verifying the kernel, not when verifying kernel modules.

Directories can be created and destroyed using the functions `procfs_mkdir` and `procfs_rmdir`. The precondition of `procfs_mkdir` requires read permission to a character string. The postcondition returns this read permission. Moreover, if allocation succeeds (i.e. a non-zero value is returned), then a new `procfs_dir` is created. Additionally, the number of allocated resources is incremented by one with respect to the function's pre-state. The function `procfs_rmdir` destroys the directory and decreases the number of allocated resources by one. Note that a directory can only be destroyed if that directory contains no files.

```
struct procfs_dir;

//@ predicate procfs_dir(struct procfs_dir *entry, int nbFiles);

struct procfs_dir *procfs_mkdir(char *name);
  //@ requires [?f]string(name, ?cs) &*& cleanup_debt(?debt);
  /*@ ensures [f]string(name, cs) &*&
              result == 0 ?
                cleanup_debt(debt)
              :
                procfs_dir(result, 0) &*& cleanup_debt(debt + 1); @*/

void procfs_rmdir(struct procfs_dir *dir);
  //@ requires procfs_dir(dir, 0) &*& cleanup_debt(?debt);
  //@ ensures cleanup_debt(debt - 1);
```

`Procfs` files are represented in specifications via the predicate *procfs_file*. The predicate's first parameter is a pointer to a `procfs_file` struct, the second a pointer to its parent directory, and the last is a function pointer. This function pointer refers to a callback that is called by the kernel whenever a user process performs a read operation on the file.

Read callbacks that are registered with a file must satisfy the contract of `procfs_read_callback`. More specifically, the precondition of the callback function states that the function receives a fraction of the permissions described by *read_callback_data*. This predicate family typically holds permissions needed to access the module's global variables. The callback function gets only an unspecified fraction (denoted by *f*) of *read_callback_data* because multiple callbacks that each need similar permissions may be running concurrently in different threads. In addition to the predicate family, the callback function receives *procfs_buffer(handle)* which denotes the permission to access the buffer referred to by `handle`. The goal of the function is to fill this buffer which is made available to the user process performing the read operation. The callback's postcondition indicates that all permissions acquired in the precondition return to the callee when the function terminates.

```
struct procfs_file;

/*@ predicate procfs_file(struct procfs_file *entry,
                        struct procfs_dir *parent,
                        procfs_read_callback *read_callback);

predicate_family read_callback_data(procfs_read_callback *cb)();
predicate procfs_buffer(struct procfs_callback_handle *cb_hd); @*/

typedef int procfs_read_callback(struct procfs_callback_handle *handle);
  /*@ requires [?f]read_callback_data(this)() &*&
             procfs_buffer(handle); @*/
  /*@ ensures [f]read_callback_data(this)() &*&
             procfs_buffer(handle); @*/

struct procfs_file *procfs_create_file(char *name,
                                   struct procfs_dir *parent,
                                   procs_read_callback *callback);
  /*@ requires [?f]string(name, ?cs) &*& procfs_dir(parent, ?nbFiles) &*&
             is_procfs_read_callback(callback) == true &*&
             read_callback_data(callback)(); @*/
  /*@ ensures [f]string(name, cs) &*&
             result == 0 ?
               procfs_dir(parent, nbFiles) &*&
               read_callback_data(callback)()
             :
               procfs_dir(parent, nbFiles + 1) &*&
               procfs_file(result, parent, callback); @*/

void procfs_remove_file(struct procfs_file *entry);
  /*@ requires procfs_file(entry, ?parent, ?callback) &*&
             procfs_dir(parent, ?nbFiles); @*/
  /*@ ensures procfs_dir(parent, nbFiles - 1) &*&
             read_callback_data(callback)(); @*/
```

Files can be created and destroyed via the functions `procfs_create_file` and `procfs_remove_file`. The specification of both functions is similar to the one for directories. In addition however, the precondition of `procfs_create_file` requires that the given callback is a valid instance of `procfs_read_callback` (denoted *is_procfs_read_callback(callback)*). Moreover, the module-specific permissions (denoted as *read_callback_data(callback)()*) required by the callback must be passed to the kernel. As shown in the postcondition of the function `procfs_remove_file`, these permissions return to the module when the file is removed. Note that `procfs_remove_file` blocks until no clients remain for the file. Finally, the validity of the parent directory is required and ensured by `procfs_create_file` and `procfs_remove_file`, and the parent's number of children is updated appropriately.

The pre- and postcondition of the read callback include the predicate permission *procfs_buffer(handle)*, which is needed to access the buffer referred to by `handle`. More specifically, the read callback can fill the buffer by calling `procfs_put_int` and `procfs_put_string`. The module can check whether the buffer is full by calling `procfs_is_buffer_full`. If the buffer is full, the kernel calls the callback again passing in a larger buffer.

```
void procfs_put_int(struct procfs_callback_handle *handle, int i)
  //@ requires procfs_buffer(handle);
  //@ ensures procfs_buffer(handle);

void procfs_put_string(struct procfs_callback_handle *handle, char* text)
  //@ requires [?f]string(text, ?vs) &*& procfs_buffer(handle);
  //@ ensures [f]string(text, vs) &*& procfs_buffer(handle);

int procfs_is_buffer_full(struct procfs_callback_handle *handle)
  //@ requires procfs_buffer(handle);
  //@ ensures procfs_buffer(handle);
```

***Mutexes in Helloproc.*** Callbacks registered by kernel modules, such as the procfs read callback, can typically be called by multiple threads. It is therefore crucial that data structures manipulated by these callbacks are protected by a mutual exclusion mechanism to ensure the absence of race conditions. The module we verify in Section 4 uses mutexes for this purpose.

In specifications, a mutex is represented via the predicate `mutex`. More specifically, *mutex(m, inv)* holds if `m` is the address of a valid mutex that protects the memory locations described by the predicate *inv*. This predicate is typically called the lock invariant. The function `mutex_create` creates a new mutex with lock invariant *inv*. As *inv()* is included in the precondition but not in the postcondition (when creation succeeds), the caller can no longer directly access the memory locations in the lock invariant. If any thread (including the caller) wants to access memory locations in the lock invariant, it must acquire the mutex. As shown in the specifications below, a thread can try to acquire the mutex by calling `mutex_lock`. `mutex_lock` requires only a fraction of the mutex predicate meaning that multiple threads can try to acquire the mutex at the same time. However, the implementation of the mutex API guarantees that only a singly can hold the mutex at any time. When `mutex_lock` returns, the calling thread has successfully acquired the mutex and therefore receives the lock invariant, *inv()*, and a *mutex_held* permission. The latter permission denotes the right to release the mutex via `mutex_unlock`, thereby losing access again to the lock invariant. Finally, a mutex can be disposed by calling `mutex_dispose`. Disposing a mutex requires full permission to enforce that no other thread can try to acquire or hold on to the mutex during or after the dispose operation.

```
struct mutex;

//@ predicate mutex(struct mutex *mutex ; predicate() inv);
/*@ predicate mutex_held(struct mutex *mutex, predicate() inv,
                       int threadId, real f); @*/

struct mutex* mutex_create/*@(predicate() inv)@*/();
  //@ requires inv();
  /*@ ensures result == 0 ?
              inv()
             :
              mutex(result, inv); @*/

void mutex_lock(struct mutex* mutex);
  //@ requires [?f]mutex(mutex, ?inv);
  //@ ensures mutex_held(mutex, inv, currentThread, f) &*& inv();

void mutex_unlock(struct mutex* mutex);
  //@ requires mutex_held(mutex, ?inv, currentThread, ?f) &*& inv();
  //@ ensures [f]mutex(mutex, inv);

void mutex_dispose(struct mutex* mutex);
  //@ requires mutex(mutex, ?inv);
  //@ ensures inv();
```

## 4 Specification & Verification of a Linux Kernel Module

The kernel module subject to verification, Helloproc, provides a file in the virtual file-system `procfs`. The contents of this file are constructed on the fly every time the file is read by a user-space application. In Helloproc, the contents of the virtual file reflect the number of times it is read. Furthermore, the module can be dynamically loaded and unloaded. From a user point of view, this looks as follows:

```
# insmod helloproc.ko
# cat /proc/helloproc/myfile
This is helloproc. This file is read 1 times.
# cat /proc/helloproc/myfile
This is helloproc. This file is read 2 times.
# rmmod helloproc.ko
```

Helloproc contains four global variables: `counter`, `mutex`, `dir` and `file`. `counter` stores the number of times the read callback has been called. `mutex` holds a reference to a mutex that protects the variable `counter`. Finally, the global variables `dir` and `file` respectively point to a `procfs` file and directory.

```
static int counter = 0; static struct mutex *mutex = 0;
static struct procfs_dir *dir = 0; static struct procfs_file *file = 0;

/*@ predicate module_state() =
  pointer(&dir, ?dir) &*& pointer(&file, ?file) &*& procfs_dir(dir, 1)
  &*& procfs_file(file, dir, read_proc_callback) &*& cleanup_debt(1);

predicate lock_invariant() = integer(&counter, _)

predicate read_callback_data(void *callback)() =
  pointer(&mutex, ?mutex) &*& mutex(mutex, lock_invariant); @*/

int read_proc_callback(struct procfs_callback_handle *handle)
  //@: procfs_read_callback
{
  int counter_backup;
  procfs_put_string(handle, "This is helloproc. This file is read ");
  mutex_lock(mutex);
  counter_backup = counter;
  if (counter < INT_MAX)
    counter++;
  else
    procfs_put_string(handle, "more than ");
  procfs_put_int(handle, counter);
  procfs_put_string(handle, " times.\n");
  if (procfs_is_buffer_full(handle)) counter = counter_backup;
  mutex_unlock(mutex);
  return 0;
}

int helloproc_main_module_init() //@: module_setup(helloproc_main)
{
  dir = procfs_mkdir("helloproc"); if (dir == 0) goto error_mkdir;
  counter = 0;
  mutex = mutex_create/*@(lock_invariant)@*/();
  if (mutex == 0) goto error_mutex;
  file = procfs_create_file("myfile", dir, read_proc_callback);
  if (file == 0) goto error_file;
  return 0;

  error_file: mutex_dispose(mutex);
  error_mutex: procfs_rmdir(dir);
  error_mkdir: return -1;
}

void helloproc_main_module_exit() //@: module_cleanup(helloproc_main)
{
  procfs_remove_file(file, dir); mutex_dispose(mutex); procfs_rmdir(dir);
}
```

Helloproc contains three functions: `read_proc_callback`, an initialization function, and a cleanup function. `read_proc_callback` is the function that is registered with a procfs file during initialization by `helloproc_main_module_init`, and which is called by the kernel whenever a user-space process performs a read operation on that file. The *`//@:procfs_read_callback`* annotation indicates that the function is a valid instance of the type definition `procfs_read_callback`. The predicate family *`read_callback_data`* mentioned in the pre- and postcondition is interpreted in the context of `read_proc_callback` as the permission to access the global variable `mutex` and the permission to use that mutex. The mutex's lock invariant indicates that the mutex protects the global variable `counter`. Note that the read callback only receives a fraction of *`read_callback_data`*, meaning that the callback can only read the global variable `mutex` and that it can only try to acquire but not dispose the mutex.

`helloproc_main_module_init` and `helloproc_main_module_exit` are respectively the module's initialization and cleanup function. The former function creates a procfs directory and file, registers `read_proc_callback` as a callback with the file and initializes the global variables, while the latter function disposes the resources created by the former. The predicate `module_state` contains *(1)* the permissions to access the global variables `dir` and `file`, *(2)* the predicates representing the validity of the file and directory, and *(3)* the obligation to dispose the directory.

Helloproc was proven to be correct using the VeriFast program verifier. Correctness implies memory safety, data race freedom, the absence of API usage violations and the absence of integer under- and overflow. The verifier is available online at `http://www.cs.kuleuven.be/~bartj/verifast`. Helloproc and the kernel wrapper API are included in the VeriFast distribution. Note that the specifications and code shown in the paper differ slightly from the ones available online in order to save space and to streamline the presentation.

We conclude the discussion of Helloproc by explaining how VeriFast would detect various bugs in the implementation, or mistakes in the specification:

- Omitting the `mutex_lock` call can lead to data races. Our tool detects this bug because it checks at each read and write operation that the necessary permissions are available. The permission required to access the global variable `counter` is not available unless the mutex is acquired (permission to access `counter` is the lock invariant).
- Omitting the `mutex_unlock` call prevents other threads from acquiring the mutex. Our tool detects this particular liveness issue because the read callback will be unable to establish its postcondition (which demands *`mutex`* instead of *`mutex_held`*)[4].
- Omitting one of the dispose operations in the cleanup function leads to a leak. VeriFast detects such leaks because the predicate `cleanup_debt` will not hold the value zero required by the function's postcondition.

[4] VeriFast does not detect all liveness issues. In particular, verified programs can contain infinite loops or deadlocks.

– The module could pass the permission to access a global variable to the kernel, and the kernel could use this permission to access the global variable after unloading the module. We prevent this error by including *module* (which groups the permissions to access the global variables) in the cleanup function's postcondition.
– Illegal memory accesses are prevented by the separation logic permission system, which enforces that a memory location can only be accessed if the corresponding permission is available. Note that having a permission to access a memory location implies that the memory location is allocated.
– The module could pass wrong parameters to API function, thereby causing memory errors in the kernel API itself. Calling `procfs_put_int(0, 0)` is an example of such an illegal call. Our tool checks before each API call that the callee's precondition (which includes the necessary permissions) holds.
– The lock invariant for the mutex could be wrong. If we change the lock invariant to *false*, then we will be unable to prove the precondition of `mutex_create`. If we change the lock invariant to *true*, then we will be unable to prove we have the permissions necessary to access the `counter` in the read callback.

## 5 Related Work

In this section we discuss related case studies on verifying operating system (OS) components, and tools that have been applied in the context of OS verification. The reader is referred to [12] for a discussion of the related work on VeriFast.

During the last decade several automated tools for verifying C programs emerged. Most notably, the CEGAR-based [7] model checkers BLAST [11] and SLAM/SDV [3] have been applied to check the conformance of device drivers with a set of API usage rules. In contrast with our work, these tools do not provide support for identifying errors with respect to the inherently concurrent execution environment device drivers are operating in. The tools also assume either that the program under consideration "does not have wild pointers" [2] or, as shown in [14], perform poorly when checking OS components for memory safety.

Aiming to support a richer subset of C, including pointers and bit-vector operations, the SatAbs [8] model checker has been proposed. In [20] SatAbs is extended to support the verification of concurrent programs, and evaluated in a case study on Linux device drivers. In difference to our work, SatAbs requires a test harness with a fixed number of threads to be generated for each driver. The tool further checks for the violation of assertions in the code, while VeriFast uses generic API contracts. Therefore, VeriFast can be used to check each function of a driver in isolation, which contributes to the scalability of our approach.

Recently, bounded model checking and symbolic execution techniques have been successfully applied to the source code [17,13] as well as to the compiled object code [15] of kernel modules. In contrast to the VeriFast approach, these techniques suffer from severe limitations with respect to reasoning about concurrently executing kernel threads.

A competing toolkit to VeriFast is the Verifying C Compiler (VCC) [9]. VCC verifies a program annotated with contracts in Boogie. The tool generates verification conditions from the annotated program, which are then discharged by SMT an solver. VCC can be expected to require fewer annotations than VeriFast, however, at the expense of a less predictable search times. The toolkit has been employed in a case study on verifying the MS Hypervisor.

## 6 Conclusion & Future Work

In this paper we introduce a verification approach for Linux kernel modules, which relies heavily on the formal specifications, in terms of separation logic, of a wrapper around the Linux API. We also present a small kernel module, Helloproc, as an extensible case study for verification efforts. While Helloproc consists of only about 30 lines of C code, it involves a number of challenges to specification and verification. Namely, those are potential memory leaks, use-after-free bugs, and data races due to concurrent callbacks. We apply the general-purpose software verifier VeriFast to Helloproc, proving that the module satisfies our API specifications and, thus, is free of the above errors.

The total effort invested in this project amounts to about three man-months. Notably, the specifications were developed by somebody *not* previously familiar with software verification and separation logic. Hence, our most important lesson learned is that the verification of a simple Linux kernel module with VeriFast is absolutely feasible, and can be conducted by a skilled developer after only a few months of practical experience with VeriFast.

We are currently extending the supported API in order to verify a USB keyboard driver. In the future, we plan to integrate additional language constructs into our verifier so that we no longer require an API wrapper, but can instead verify modules with respect to the real kernel API. In addition, we want to improve VeriFast such that it detects liveness issues, e.g., infinite loops and deadlocks.

## References

1. Ball, T. The verified software challenge: A call for a holistic approach to reliability. In *VSTTE '08*, vol. 4171 of *LNCS*, pp. 42–48, Heidelberg, 2005. Springer.
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. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.
3. Ball, T. and Rajamani, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN '01*, vol. 2057 of *LNCS*, pp. 102–122, Heidelberg, 2001. Springer.
4. Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. Permission accouting in separation logic. In *POPL*, 2005.
5. Bovet, D. and Cesati, M. *Understanding the Linux Kernel.* O'Reilly, 3rd edition, 2005.

6. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. An empirical study of operating system errors. In *SOSP '01*, pp. 73–88, New York, 2001. ACM.
7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
8. Clarke, E. M., Kroening, D., Sharygina, N., and Yorav, K. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS '05*, vol. 3440 of *LNCS*, pp. 570–574, Heidelberg, 2005. Springer.
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *TPHOLs '09*, vol. 5674 of *LNCS*, pp. 23–42, Heidelberg, 2009. Springer.
10. Corbet, J., Rubini, A., and Kroah-Hartmann, G. *Linux Device Drivers.* O'Reilly, 3rd edition, 2005.
11. Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV '02*, vol. 2402 of *LNCS*, pp. 382–399, Heidelberg, 2002. Springer.
12. Jacobs, B., Smans, J., and Piessens, F. VeriFast: Imperative programs as proofs. In *VSTTE 2010 workshop proceedings*, pp. 63–72, 2010.
13. Kim, M. and Kim, Y. Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 251–265, Heidelberg, 2009. Springer.
14. Mühlberg, J. T. and Lüttgen, G. BLASTing Linux code. In *FMICS '06*, vol. 4346 of *LNCS*, pp. 211–226, Heidelberg, 2007. Springer.
15. Mühlberg, J. T. and Lüttgen, G. Verifying compiled file system code. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 306–320, Heidelberg, 2009. Springer.
16. O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *CSL*, 2001.
17. Post, H., Sinz, C., and Küchlin, W. Towards automatic software model checking of thousands of Linux modules – a case study with Avinux. *Softw. Test. Verif. Reliab.*, 19:155–172, 2009.
18. Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pp. 55–74, Washington, 2002. IEEE.
19. Swift, M., Bershad, B., and Levy, H. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.
20. Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. Model checking concurrent Linux device drivers. In *ASE '07*, pp. 501–504, New York, 2007. ACM.