BLASTing Linux Code*

Jan Tobias Mühlberg and Gerald Lüttgen

Department of Computer Science, University of York, York YO10 5DD, U.K.
<muehlber|luettgen>@cs.york.ac.uk

Abstract. Computer programs can only run reliably if the underlying operating system is free of errors. In this paper we evaluate, from a practitioner's point of view, the utility of the popular software model checker BLAST for revealing errors in Linux kernel code. The emphasis is on important errors related to memory safety in and locking behaviour of device drivers. Our conducted case studies show that, while BLAST's abstraction and refinement techniques are efficient and powerful, the tool has deficiencies regarding usability and support for analysing pointers, which are likely to prevent kernel developers from using it.

1 Introduction

Today's application software critically depends on the reliability, safety and security of the underlying operating system (OS). However, due to their complicated task of managing a system's physical resources, OSs are difficult to develop and even more difficult to debug. Quite frequently major errors stay undiscovered until they are exploited in security attacks or are found "by accident".

In recent years, automatic approaches to discover OS bugs via runtime checks or source code analysis have been explored. Despite the fact that many of these approaches do not focus on an exhaustive analysis, they still helped developers to detect hundreds of safety problems in the Linux and BSD OS kernels. Most of the programming errors found were either related to *memory safety* or incorrect *locking behaviour* [6]. Here, "memory safety" typically is interpreted as the property that an OS component never de-references an invalid pointer, since this would cause the program to end up in an undefined state. "Correct locking behaviour" means that functions that ensure mutual exclusion on the physical resources of a system are called in a way that is free of deadlocks and starvation. Both classes of problems are traceable by checking whether an OS component complies with basic usage rules of the program interface provided by the kernel.

Software model checking. By having the potential of being exhaustive and fully automatic, *model checking*, in combination with *abstraction* and *refinement*, is a successful technique used in software verification [7]. Intensive research in this area has resulted in software model checkers like Bandera [9] for Java programs or SLAM/SDV [1], MAGIC [5] and BLAST [16] (*Berkeley Lazy Abstraction*

* Research funding was provided by the EPSRC under grant GR/S86211/01.

Software verification Tool) for analysing C source code. The major advantage of these tools over model-based model checkers such as Spin [17] is their ability to automatically abstract a model from the source code of a given program. User interaction should then only be necessary in order to provide the model checker with a specification, against which the program can be checked. Since complete formal specifications are not available for most programs, verification will usually be relative to a partial specification that covers the usage rules of the *Application Program Interface* (API) used by the program. However, up to now all releases of SLAM are restricted to verifying properties for Microsoft Windows device drivers and do not cover memory safety problems [19], while BLAST and MAGIC are able to verify a program against a user defined temporal safety specification and thus allows checking of arbitrary C source code.

The* BLAST *toolkit . This popular toolkit implements an advanced abstraction algorithm, called "lazy abstraction" [15], for building a model of some C source code, and model-checking algorithm for checking whether some specified label placed in the source code is reachable. This label can either be automatically introduced by instrumenting the source with an explicit temporal safety specification, be added via `assert()` statements, or be manually introduced into the source. In any case, the input source file needs to be preprocessed using a standard C preprocessor like `gcc`. In this step, all header and source files included by the input file under consideration are merged into one file. It is this preprocessed source code that is passed to BLAST to construct and verify a model using *predicate abstraction*.

This paper. In this paper we investigate to which extent software model checking as implemented in BLAST can aid a practitioner during OS software development. To do so, we analyse whether BLAST is able to detect errors that have been reported for recent releases of the Linux kernel. We consider programming errors related to *memory safety* (cf. Sec. 3) and *locking behaviour* (cf. Sec. 4). The code examples utilised in this paper are taken from releases 2.6.13 and 2.6.14 of the Linux kernel. They have been carefully chosen by searching the kernel's change log for fixed memory problems and fixed deadlock conditions, in a way that the underlying problems are representative for memory safety and locking behaviour as well as easily explainable without referring to long source code listings.[1] Our studies use version 2.0 of BLAST, which was released in October 2005.

The focus of our work is on showing at what scale a give problem statement and a program's source code need to be adapted in order to detect an error. We discuss how much work is required to find a certain usage rule violation in a given snippet of a Linux driver, and how difficult this work is to perform in BLAST. Due to space constraints, we cannot present all of our case studies in full here; however, all files necessary to reproduce our results can be downloaded from `www.cs.york.ac.uk/~muehlber/blast/`.

[1] All source code used is either included or referenced by a *commit key* as provided by the source code management system *git* which is used in the Linux kernel community; see `www.kernel.org` for further information on *git* and Linux.

***Related studies with* BLAST.** BLAST has been applied for the verification of memory safety as well as locking properties before [3,13,16,14]. In [3], the use of CCURED [21] in combination with BLAST for verifying memory safety of C source code is explained. This is done by inserting additional runtime checks at all places in the code where pointers are de-referenced. BLAST is then employed to check whether the introduced code is reachable or can be removed again. The approach focuses on ensuring that only valid pointers are de-referenced along the execution of a program, which is taken to mean that pointers must not equal `NULL` at any point at which they are de-referenced. However, invalid pointers in C do not necessarily equal `NULL` in practise. In contrast to [3], we will interpret pointer invalidity in a more general way and conduct our studies on real-world examples rather than constructed examples.

A methodology for verifying and certifying systems code on a simple locking problem is explained in [16], which deals with the spinlock interface provided by the Linux kernel. *Spinlocks* ensure that a kernel process can spin on a CPU without being preempted by another process. The framework studied in [16] is used to prove that calls of `spin_lock()` and `spin_unlock()` in Linux device drivers always alternate. In contrast to this work, our case studies will be more detailed and thereby will be providing further insights into the usability of BLAST.

2 Programming Errors in OS Code

There is quite a long list of commonly found OS errors. While most of them mainly affect a system's safety, others have a security-related background. An insightful study of OS errors has been published in [6]; see Table 1 for a summary of its results. The study shows that the majority of programming errors in OS code can be found in device drivers. Its authors highlight that most errors are related to problems causing either deadlock conditions or driving the system into undefined states by de-referencing invalid pointers.

Although memory safety problems have a direct impact on an OS's reliability, API rules for OS kernels are usually described in an informal way. For example, in the Linux device driver handbook [8, p. 61] it is stated that one "should never pass anything to *kfree* that was not obtained from *kmalloc*" since, otherwise, the system may behave in an undefined way. The functions `kmalloc()` and `kfree()` are kernel-space functions which are used to dynamically allocate and de-allocate memory, respectively. Another common example are buffer overrun errors, where data is written beyond the size of an allocated area of memory, thus overwriting unrelated data.

Correct locking of resources is another major issue causing problems in OS code. As shown in [6], deficiencies resulting in deadlocks in the Linux and BSD kernels make up a large amount of the overall number of errors found. In the documentation explaining the API of the Linux kernel, quite strict rules about the proper use of functions to lock various resources are stated. For example, in [8, p. 121], one of the most basic rules is given as follows: "Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time; should

Table 1. Results of an empirical study of OS errors [6]

% of Bugs	Rule checked
63.1%	**Bugs related to memory safety**
38.1%	Check potentially NULL pointers returned from routines.
9.9%	Do not allocate large stack variables (> 1K) on the fixed-size kernel stack.
6.7%	Do not make inconsistent assumptions about whether a pointer is NULL.
5.3%	Always check bounds of array indices and loop bounds derived from user data.
1.7%	Do not use freed memory.
1.1%	Do not leak memory by updating pointers with potentially NULL realloc return values.
0.3%	Allocate enough memory to hold the type for which you are allocating.
33.7%	**Bugs related to locking behaviour**
28.6%	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.
2.6%	Restore disabled interrupts.
2.5%	Release acquired locks; do not double-acquire locks.
3.1%	**Miscellaneous bugs**
2.4%	Do not use floating point in the kernel.
0.7%	Do not de-reference user pointers.

you attempt to do so, things simply hang." The rational for this lies in the functionality provided by spinlocks: a kernel thread holding a lock is spinning on one CPU and cannot be preempted until the lock is released. Another important rule is that any code holding a spinlock cannot relinquish the processor for anything except for serving interrupts; especially, the thread must never sleep because the lock might never be released in this case [8, p. 118].

3 Checking Memory Safety

This section focuses on using Blast for checking usage rules related to memory safety, for which we have analysed several errors in different device drivers. The examples studied by us include use-after-free errors in the kernel's SCSI[2] and InfiniBand[3] subsystems. The former is the *small computer system interface* standard for attaching peripheral devices to computers, while the latter is an industry standard designed to connect processor nodes and I/O nodes to form a system area network. In each of these examples, an invalid pointer that is not `NULL` is de-referenced, which causes the system to behave in an undefined way. This type of bug is not covered by the work on memory safety of Beyer et al. in [3] and cannot easily be detected by runtime checks.

[2] Commit `2d6eac6c4fdaa69656d66c80754d267be233cc3f`.
[3] Commit `d0743a5b7b837334cb414b773529d51de3de0471`.

The example we will study here in detail is a use-after-free error spotted by the Coverity source code analyser (`www.coverity.com`) in the I2O subsystem of the Linux kernel (cf. Sec. 3.1). To check for this bug in Blast we first specify a temporal safety specification in the Blast specification language. Taking this specification, Blast is supposed to automatically generate an instrumented version of the C source code for analysis (cf. Sec. 3.2). However, due to an apparent bug in Blast, this step fails for our example, and we are therefore forced to manually instrument our code by inserting `ERROR` labels at appropriate positions (cf. Sec. 3.3). However, it will turn out that Blast does not track important operations on pointers, which is not mentioned in Blast's user manual and without which our example cannot be checked (cf. Sec. 3.4).

3.1 The I2O Use-After-Free Error

The I2O subsystem bug of interest to us resided in lines 423–425 of the source code file `drivers/message/i2o/pci.c`. The listing in Fig. 1 is an abbreviated version of the file `pci.c` before the bug was fixed. One can see that function `i2o_iop_alloc()` is called at line 330 of the code extract. This function is defined in `drivers/message/i2o/iop.c` and basically allocates memory for an `i2o_controller` structure using `kmalloc()`. At the end of the listing, this memory is freed by `i2o_iop_free(c)`. The bug in this piece of code arises from the call of `put_device()` in line 425, since its parameter `c->device.parent` causes an already freed pointer to be de-referenced. The bug has been fixed in commit `d2b0e84d195a341c1cc5b45ec2098ee23bc1fe9d`, by simply swapping lines 424 and 425 in the source file.

```
drivers/message/i2o/pci.c:
 300  static int __devinit
       i2o_pci_probe(
       struct pci_dev *pdev,
 301   const struct pci_device_id
       *id)
 302  {
 303   struct i2o_controller *c;
```

```
330   c = i2o_iop_alloc();

423   free_controller:
424    i2o_iop_free(c);
425    put_device(
         c->device.parent);

432  }
```

Fig. 1. Extract of `drivers/message/i2o/pci.c`.

This bug offers various different ways to utilise Blast. A generic temporal safety property for identifying bugs like this would state that *any pointer that has been an argument to* ***`kfree()`*** *is never used again* unless it has been reallocated. A probably easier way would be to check whether *the pointer* `c` *in* ***`i2o_pci_probe()`*** *is never used again after* ***`i2o_iop_free()`*** *has been called* with `c` as its argument. Checking the first, more generic property would require us to put function definitions from other source files into `pci.c`, since Blast considers only functions that are available in its input file. Therefore, we focus on verifying the latter property.

Checking for violations even of the latter, more restricted property will lead to a serious problem. A close look at the struct `i2o_controller` and its initialisation in the function `i2o_iop_alloc()` reveals that `i2o_controller` contains a function pointer which can be used as a "destructor". As is explained in BLAST's user manual, the "current release does not support function pointers"; they are ignored completely. Further, the manual states that "correctness of the analysis is then modulo the assumption that function pointer calls are irrelevant to the property being checked." This assumption is however not always satisfied in practise, as we will see later in our example.

3.2 Verification With a Temporal Safety Specification

Ignoring the function pointer limitation, we developed the temporal safety specification presented in Fig. 2. The specification language used by BLAST is easy to understand and allows the assignment of status variables and events. In our specification we use a global status variable `allocstatus_c` to cover the possible states of the struct `c` of our example, which can be set to 0 meaning "not allocated" and 1 meaning "allocated". Furthermore, we define three events, one for each of the functions `i2o_iop_alloc()`, `i2o_iop_free()` and `put_device()`. All functions have special preconditions and calling them may modify the status of `c`. The special token `$?` matches anything. Intuitively, the specification given in Fig. 2 states that `i2o_iop_alloc()` and `i2o_iop_free()` must be called alternately, and `put_device()` must only be called when `c` has not yet been freed. Note that this temporal safety specification does not cover the usage rule for `i2o_iop_free()` and `put_device()` in general. We are using one status variable to guard calls of `i2o_iop_free()` and `put_device()` regardless of its arguments. Hence, the specification will work only as long as there is only one pointer to an `i2o_controller` structure involved.

```
global int allocstatus_c = 0;

event
 {
  pattern { $? = i2o_iop_alloc(); }
  guard   { allocstatus_c == 0 }
  action  { allocstatus_c = 1; }
 }
```

```
event
 {
  pattern { i2o_iop_free($?); }
  guard   { allocstatus_c == 1 }
  action  { allocstatus_c = 0; }
 }

event
 {
  pattern { put_device($?); }
  guard   { allocstatus_c == 1 }
 }
```

Fig. 2. A temporal safety specification for `pci.c`.

Using the specification of Fig. 2, BLAST should instrument a given C input file by adding a global status variable and error labels for all violations of the

preconditions. The instrumentation is done by the program `spec.opt` which is part of the BLAST distribution. For our example taken from the Linux kernel, we first obtained the command used by the kernel's build system to compile `pci.c` with `gcc`. We appended the option `-E` to force the compilation to stop after preprocessing, resulting in a C source file containing all required parts of the kernel headers. This step is necessary since BLAST cannot know of all the additional definitions and include paths used to compile the file. Unfortunately, it expands `pci.c` from 484 lines of code to approximately 16k lines, making it difficult to find syntactical problems which BLAST cannot deal with. Despite spending a lot of effort in trying to use `spec.opt`, we never managed to get this work. The program mostly failed with unspecific errors such as `Fatal error: exception Failure("Function declaration not found")`. Finding such an error in a huge source without having a line number or other hint is almost impossible, especially since `gcc` compiles the file without any warning. We constructed several simplifications of the preprocessed file in order to trace the limitations of `spec.opt`, but did not get a clear indication of what the source is. We suspect it might be a problem with parsing complex data structures and inline assembly imported from the Linux headers.

Given the bug in BLAST and in order to demonstrate that our specification indeed covers the programming error in `pci.c`, we developed a rather abstract version of `pci.c` which is shown in Fig. 3. Using this version and the specification of Fig. 2, we were able to obtain an instrumented version of our source code without encountering the bug in `spec.opt`. Running BLAST on the instrumented version then produced the following output:

```
$ spec.opt test2.spc test2.c
[...]
$ pblast.opt instrumented.c
[...]
Error found! The system is unsafe :-(
```

In summary, the example studied here shows that the specification used in this section is sufficient to find the bug. However, the approach required by BLAST has several disadvantages. Firstly, it is not automatic at all. Although we ended up with only a few lines of code, it took quite a lot of time to produce this code by hand and to figure out what parts of the original `pci.c` are accepted by BLAST. Secondly, the methodology only works if the bug is known beforehand, hence we did not learn anything new about unwanted behaviour of this driver's code. We needed to simplify the code to an extent where the relation to the original source code may be considered as questionable. The third problem lies in the specification used. Since it treats the allocation and de-allocation as something similar to a locking problem, we would not be able to use it in a piece of code that refers to more than one dynamically allocated object. A more generic specification must be able to deal with multiple pointers. According to [2], such a generic specification should be possible to write by applying a few minor modifications such as defining a "shadow" control state and replacing `$?`

```
test2.h:
#include <stdio.h>
#include <stdlib.h>

typedef struct device
{
 int parent;
} device;

typedef struct i2o_controller
{
 struct device device;
} i2o_controller;

i2o_controller *i2o_iop_alloc
 (void);
void i2o_iop_free
 (i2o_controller *c);
void put_device (int i);
```

```
test2.c:
#include "test2.h"

i2o_controller *i2o_iop_alloc
 (void)
{ i2o_controller *c;
  c = malloc(
   sizeof(struct i2o_controller));
  return (c); }

void i2o_iop_free
 (i2o_controller *c)
{ free (c); }

void put_device (int i) { }

int main (void)
{ i2o_controller *c;
  c = i2o_iop_alloc ();
  i2o_iop_free (c);
  put_device (c->device.parent);
  return (0); }
```

Fig. 3. Manual simplification of `pci.c`.

with `$1`. However, in practise the program generating the instrumented C source file failed with obscure error messages.

3.3 Verification Without a Temporal Safety Specification

Since Blast could not deal with verifying the original `pci.c` using an explicit specification of the use-after-free property, we will now try and manually instrument the source file so that our bug can be detected whenever an `ERROR` label is reachable.

When conducting our instrumentation, the following modifications were applied by hand to `pci.c` and related files:

1. A variable `unsigned int alloc_status` was added to the definition of `struct i2o_controller` in `include/linux/i2o.h`.
2. The prototypes of `i2o_iop_alloc()` and `i2o_iop_free()` were removed from `drivers/message/i2o/core.h`.
3. The prototype of `put_device()` was deleted from `include/linux/device.h`.
4. C source code for the functions `put_device()`, `i2o_iop_free()`, `i2o_iop_release()` and `i2o_iop_alloc()` was copied from `iop.c` and `drivers/base/core.c` into `pci.c`. The functions were modified such that the new field `alloc_status` of a freshly allocated `struct i2o_controller` is set to 1 by `i2o_iop_alloc()`. `i2o_iop_free()` no longer de-allocates the structure but checks whether `alloc_status` equals 1 and sets it to 0; otherwise, it jumps

to the ERROR label. put_device() was modified to operate on the whole struct i2o_controller and jumps to ERROR if alloc_status equals 0.

By feeding these changes into the model checker it is possible to detect duplicate calls of i2o_iop_free() on a pointer to a struct i2o_controller, as well as calls of put_device() on a pointer that has already been freed. Even calls of i2o_iop_free() and put_device() on a pointer that has not been allocated with i2o_iop_alloc(), should result in an error report since nothing can be said about the status of alloc_status in such a case.

After preprocessing the modified source files and running BLAST, we get the output "Error found! The system is unsafe :-(". Even after we reduced the content of i2o_pci_probe() to something quite similar to the main() function shown in Fig. 3 and after putting the erroneous calls of put_device() and i2o_iop_free() in the right order, the system was still unsafe from BLAST's point of view. It took us some time to figure out that BLAST does not appear to consider the content of pointers at all.

3.4 The Problem with BLAST and Pointers

We demonstrate this apparent shortcoming of BLAST regarding handling pointers by means of another simple example, for which BLAST fails in tracing values behind pointers over function calls.

```
test5.c:
 1  #include <stdlib.h>
 2
 3  typedef struct example_struct
 4   {
 5    void   *data;
 6    size_t size;
 7   } example_struct;
 8
 9
10  void init (example_struct *p)
11   {
12    p->data = NULL;
13    p->size = 0;
14
15    return;
16   }
17
18  int main (void)
19   {
20    example_struct p1;
21
22    init (&p1);
23    if (p1.data != NULL ||
          p1.size != 0)
24     { goto ERROR; }
25     else
26     { goto END; };
27
28  ERROR:
29    return (1);
30
31  END:
32    return (0);
33   }
```

Fig. 4. An example for pointer passing.

As can be seen in the code listing of Fig 4, label ERROR can never be reached in this program since the values of the components of our struct are explicitly set by function init(). However, BLAST produces the following output:

```
$ gcc -E -o test5.i test5.c
$ pblast.opt test5.i
[...]
Error found! The system is unsafe :-(
Error trace:
23 :: 23: Pred((p1@main).data!=0) :: 29
-1 :: -1: Skip :: 23
10 :: 10: Block(Return(0);) :: -1
12 :: 12: Block(* (p@init ).data = 0;* (p@init ).size = 0;) :: 10
22 :: 22: FunctionCall(init(&(p1@main))) :: -1
-1 :: -1: Skip :: 22
 0 ::  0: Block(Return(0);) :: -1
 0 ::  0: FunctionCall (__BLAST_initialize_test5.i()) :: -1
```

This counterexample shows that BLAST does not correlate the pointer `p` used in `init()` and the struct `p1` used in `main()`, and assumes that the `if` statement in line 23 evaluates to true. After adding a line "`p1.data = NULL; p1.size = 0;`" before the call of `init()`, BLAST claims the system to be safe, even if we modify `init()` to reset the values so that they differ from `NULL` (and `0`).

We were able to reproduce this behaviour in similar examples with pointers to integer values and arrays. Switching on the BDD-based alias analysis implemented in BLAST also did not solve the problem. The example shows that BLAST does not only ignore function pointer calls as stated in its user manual, but appears to assume that all pointer operations have no effect. This limitation is not documented in the BLAST manual and renders BLAST almost unusable for the verification of properties related to our understanding of memory safety.

3.5 Results

Our experiments on memory safety show that BLAST is able to find the programming error discovered by the Coverity checker. Out of eight examples, we were able to detect two problems after minor modifications to the source code, and three after applying manual abstraction. Three further programming errors could not be traced by using BLAST. Indeed, BLAST has some major restrictions. The main problem is that BLAST ignores variables addressed by a pointer. As stated in its user manual, BLAST assumes that only variables of the same type are aliased. Since this is the case in our examples, we initially assumed that our examples could be verified with BLAST, which is not the case. Moreover, we encountered bugs and deficiencies in `spec.opt` which forced us to apply substantial and time consuming modifications to source code. Most of these modifications and simplifications would require a developer to know about the error in advance. Thus, from a practitioner's point of view, BLAST is not of much help in finding unknown errors related to memory safety. However, it needs to be mentioned that BLAST was designed for verifying API usage rules of a different type than those required for memory safety. More precisely, BLAST is intended for proving the adherence of pre- and post-conditions denoted by integer values and for ensuring API usage rules concerning the order in which certain functions are called, regardless of pointer arguments, return values and the effects of aliasing.

4 Checking Locking Properties

Verifying correct locking behaviour is something used in almost all examples provided by the developers of BLAST [2,16]. In [16], the authors checked parts of the Linux kernel for correct locking behaviour while using the *spinlock* API and stated that BLAST showed a decent level of performance during these tests. Spinlocks provide a very simple but quite efficient locking mechanism to ensure, e.g., that a kernel thread may not be preempted while serving interrupts. The kernel thread acquires a certain lock by calling `spin_lock(l)`, where `l` is a previously initialised pointer to a struct `spinlock_t` identifying the lock. A lock is released by calling `spin_unlock()` with the same parameter. The kernel provides a few additional functions that control the interrupt behaviour while the lock is held. By their nature, spinlocks are intended for use on multiprocessor systems where each resource may be associated with a special spinlock, and where several kernel threads need to operate independently on these resources. However, as far as concurrency is concerned, uniprocessor systems running a preemptive kernel behave like multiprocessor systems.

```
global int lockstatus = 2;

event
 {
  pattern { spin_lock_init($?); }
  guard   { lockstatus == 2 }
  action  { lockstatus = 0; }
 }

event
 {
  pattern { spin_lock($?); }
  guard   { lockstatus == 0 }
  action  { lockstatus = 1; }
 }
```

```
event
 {
  pattern { spin_unlock($?); }
  guard   { lockstatus == 1 }
  action  { lockstatus = 0; }
 }

event
 {
  pattern { $? = sleep($?); }
  guard   { lockstatus == 0 }
 }
```

Fig. 5. A temporal safety specification for spinlocks.

Finding examples for the use of spinlocks is not difficult since they are widely deployed. While experimenting with BLAST and the spinlock functions on several small components of the Linux kernel we experienced that it performs well with functions using only one lock. We focused on functions taken from the USB subsystem in *drivers/usb/core*. Due to further unspecific parse errors with the program `spec.opt` we could not use a temporal safety specification directly on the kernel source. However, in this case we were able to generate the instrumented source file and to verify properties by separating the functions under consideration from the remaining driver source and by providing simplified header files.

In Fig. 5 we provide our basic temporal safety specification for verifying locking behaviour. Variable `lockstatus` encodes the possible states of a spinlock; the initial value 2 represents the state in which the lock has not been initialised, while 1 and 0 denote that the lock is held or has been released, respectively. The pattern within the specification varies for the different spinlock functions used within the driver source under consideration, and the specification can easily be extended to cover forbidden functions that may sleep. An example for a function `sleep()` is provided in the specification of Fig. 5.

Difficulties arise with functions that acquire more than one lock. Since all spinlock functions use a pointer to a struct `spinlock_t` in order to identify a certain lock, and since the values behind pointers are not sufficiently tracked in Blast, we were forced to rewrite parts of the driver's source and the kernel's spinlock interface. Instead of the pointers to `spinlock_t` structs we utilise global integer variables representing the state of a certain lock. We have used this methodology to verify an example of a recently fixed deadlock[4] in the Linux kernel's SCSI subsystem. In Fig. 6 we provide an extract of one of the functions modified in the fix. We see that the spinlocks in this example are integrated in more complex data structures referenced via pointers. Even worse, this function calls a function pointer passed in the argument `done` in line 1581, which was the source of the deadlock before the bug was fixed. To verify this special case, removing the function pointer and providing a dummy function `done()` with a precondition assuring that the lock on `shost->host_lock` is not held is needed. However, we were able to verify both the deadlock condition before the fix had been applied, as well as deadlock freedom for the fixed version of the source.

```
1564 int ata_scsi_queuecmd(struct
      scsi_cmnd *cmd, void
      (*done)(struct scsi_cmnd *))
1565 {
1566 struct ata_port *ap;
1567 struct ata_device *dev;
1568 struct scsi_device
      *scsidev = cmd->device;
1569 struct Scsi_Host
      *shost = scsidev->host;
```

```
1571 ap = (struct ata_port *)
      &shost->hostdata[0];
1573 spin_unlock(shost->host_lock);
1574 spin_lock(&ap->host_set->lock);

1581 done(cmd);

1597 spin_unlock(&ap->host_set->lock);
1598 spin_lock(shost->host_lock);
1600 }
```

Fig. 6. Extract of `drivers/scsi/libata-scsi.c`.

During our experiments we analysed several other examples of deadlock conditions. The more interesting examples are the spinlock problem explained above, and another one in the SCSI subsystem,[5] as well as a bug in a IEEE1394 driver[6]. We were able to detect the locking problems in all of these examples and proved the fixed source files to be free of these bugs.

[4] Commit `d7283d61302798c0c57118e53d7732bec94f8d42`.
[5] Commit `fe2e17a405a58ec8a7138fee4ebe101858b636e0`.
[6] Commit `910573c7c4aced8fd5f45c334cc67862e3424d92`.

Results. Out of eight examples for locking problems we were able to detect only five. However, when comparing our results with the conclusions of the previous section, BLAST worked much better for the locking properties because it required fewer modifications to the source code. From a practitioner's point of view, BLAST performed acceptable as long as only one lock was involved. After considerable efforts in simplifying the spinlock API — mainly removing the use of pointers and manually adding error labels to the spinlock functions — we also managed to deal with multiple locks. However, we consider it as fairly difficult to preserve the behaviour of functions that may sleep and therefore must not be called under a spinlock. Even for large portions of source code, BLAST returned its results within a few seconds or minutes, on a PC equipped with an AMD Athlon 64 processor running at 2200 MHz and 1 GB of RAM. Hence, BLAST's internal slicing and abstraction techniques work very well.

We have to point out that the code listing in Fig. 6 represents one of the easily understandable programming errors. Many problems in kernel source code are more subtle. For example, calling functions that may sleep is something that needs to be avoided. However, if a driver calls a function not available in source code in the same file as the driver under consideration, BLAST will only be able to detect the problem if there is an event explicitly defined for this function.

5 Issues with BLAST

This section highlights various shortcomings of the BLAST toolkit which we experienced during our studies. We also present ideas on how BLAST could be improved in order to be more useful for OS software verification.

Lack of documentation. Many problems while experimenting with BLAST were caused by the lack of consistent documentation. For example, a significant amount of time could have been saved in our experiments with memory safety, if the BLAST manual would state that almost all pointer operations are ignored. An in-depth discussion of the features and limitations of the alias analysis implemented in BLAST would also be very helpful to have.

Non-support of pointers. The fact that BLAST does not properly support the use of pointers, in the sense of Sec. 3.4, must be considered as a major restriction, and made our experiments with the spinlock API rather difficult. The restriction forces one to carry out substantial and time consuming modifications to source code. Furthermore, it raises the question whether all important predicates of a given program can be preserved in a manual step of simplification. In some of our experiments we simply replaced the pointers used by the spinlock functions with integers representing the state of the lock. This is obviously a pragmatic approach which does not reflect all possible behaviour of pointer programs. However, it turned out that it is expressive enough to cover the usage rules of the spinlock API. As such modifications could be introduced into the source code automatically, we consider them as an interesting extension for BLAST.

The missing support of function pointers has already been mentioned in Sec. 3. It is true that function pointers are often used in both application space and OS development. In most cases their effect on the program execution can only be determined at run-time, not statically at compile-time. Therefore, we assume that simply skipping all calls of function pointers is acceptable for now.

Usability. There are several issues regarding BLAST's usability which are probably easy to fix, but right now they complicate the work with this tool. Basically, if a piece of C source is accepted by an ANSI C compiler, it should be accepted by BLAST rather than raising uninformative error messages.

A nice improvement would be to provide wrapper scripts that automate preprocessing and verification in a way that BLAST can be used with the same arguments as the compiler. It could be even more useful if functions that are of interest but from other parts of a given source tree, would be copied in automatically. Since we obviously do not want to analyse the whole kernel source in a single file, this should be integrated into BLAST's abstraction/model checking/refinement loop.

6 Related Work

Much work on techniques and tools for automatically finding bugs in software systems has been published in recent years.

Runtime analysis. A popular runtime analysis tool which targets memory safety problems is Purify (`www-306.ibm.com/software/awdtools/purify/`). It mainly focuses on detecting and preventing memory corruption and memory leakage. However, Purify and other such tools, including Electric Fence (`perens.com/FreeSoftware/ElectricFence/`) and Valgrind (`valgrind.org`), are meant for testing purposes and thereby only cover the set of program runs specified by the underlying test cases. An exhaustive search of a programs state space, as is done in model checking, is out of the scope of these tools.

Static analysis and abstract interpretation. Static analysis is another powerful technique for inspecting source code for bugs. Indeed, most of the memory safety problems within the examples of this paper had been detected earlier via an approach based on system-specific compiler extensions, known as *meta-level compilation* [11]. This approach is implemented in the tool Coverity (`www.coverity.com`) and was used in [6]. A further recent attempt to find bugs in OS code is based on abstract interpretation [10] and presented in [4]. The authors checked about 700k lines of code taken from recent versions of the Linux kernel for correct locking behaviour. The paper focuses on the kernel's spinlock interface and problems related to sleep under a spinlock. Several new bugs in the Linux kernel were found during the experiments. However, the authors suggest that their approach could be improved by adopting model checking techniques. An overview of the advantages and disadvantages of static analysis versus model checking can be found in [12].

***Case studies with* Blast.** We have already referred to some such case studies in the introduction. Two project reports of graduate students give further details on Blast's practical use. In [20], Mong applies Blast to a doubly linked list implementation with dynamic allocation of its elements and verifies correct allocation and de-allocation. The paper explains that Blast was not powerful enough to keep track of the state of the list, i.e., the number of its elements. Jie and Shivkumar report in [18] on their experience in applying Blast to a user level implementation of a virtual file system. They focus on verifying correct locking behaviour for data structures of the implementation and were able to successfully verify several test cases and to find one new error. However, in the majority of test cases Blast failed due to documented limitations, e.g., by not being able to deal with function pointers, or terminated with obscure error messages. Both studies were conducted in 2004 and thus based on version 1.0 of Blast. As shown in this paper, Blast's current version has similar limitations.

7 Conclusions and Future Work

We exposed Blast to analysing 16 different OS code examples of programming errors related to memory safety and locking behaviour. Details of the examples which we could not show here due to a lack of space, can be found at `www.cs.york.ac.uk/~muehlber/blast/`. In our experience, Blast is rather difficult to apply by a practitioner during OS software development. This is because of (i) its limitations with respect to reasoning about pointers, (ii) several issues regarding usability, including bugs in `spec.opt`, and (iii) a lack of consistent documentation. Especially in the case of memory safety properties, massive changes to the source code were necessary which essentially requires one to know about a bug beforehand. However, it must be mentioned that Blast was not designed as a memory debugger. Indeed, Blast performed considerably better during our tests with locking properties; however, modifications on the source code were still necessary in most cases.

Blast performed nicely on the modified source code in our examples for locking properties. Even large portions of C code — up to 10k lines with several locks, status variables and a relatively complex program structure — were parsed and model checked within a few minutes on a modern PC. Hence, the techniques for abstraction and refinement as implemented in Blast are quite able to deal with most of the problems analysed in this paper. If its limitations are ironed out, Blast is likely to become a very usable and popular tool with OS software developers in the future.

Regarding future work we propose that our case study is repeated once the most problematic errors and restrictions in Blast are fixed. An analysis allowing one to draw *quantitative* conclusions concerning Blast's ability of finding certain programming problems could then give results that are more interesting to kernel developers. To this end, metrics for the evaluation of Blast are required, as is a more precise classification of the chosen examples.

Acknowledgements. We thank Radu Siminiceanu for his constructive comments and suggestions on a draft of this paper.

References

1. Ball, T. and Rajamani, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN 2001*, vol. 2057 of *LNCS*, pp. 103–122.
2. Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R. The BLAST query language for software verification. In *PEPM 2004*, pp. 201–202. ACM Press.
3. Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. Checking memory safety with BLAST. In *FASE 2005*, vol. 3442 of *LNCS*, pp. 2–18.
4. Breuer, P. T. and Pickin, S. Abstract interpretation meets model checking near the 10^6 LOC mark. In *AVIS 2006*. To appear in ENTCS.
5. Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., and Yorav, K. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2–3):129–166, 2004.
6. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. An empirical study of operating system errors. In *SOSP 2001*, pp. 73–88. ACM Press.
7. Clarke, E. M., Grumberg, O., and Peled, D. A. *Model checking*. MIT Press, 2000.
8. Corbet, J., Rubini, A., and Kroah-Hartmann, G. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.
9. Corbett et al, J. C. Bandera: Extracting finite-state models from Java source code. In *ICST 2000*, pp. 439–448. SQS Publishing.
10. Cousot, P. and Cousot, R. On abstraction in software verification. In *CAV 2002*, vol. 2404 of *LNCS*, pp. 37–56.
11. Engler, D. R., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 2000*. USENIX.
12. Engler, D. R. and Musuvathi, M. Static analysis versus software model checking for bug finding. In *VMCAI 2004*, vol. 2937 of *LNCS*, pp. 191–210.
13. Henzinger, T. A., Jhala, R., and Majumdar, R. Race checking by context inference. In *PLDI 2004*, pp. 1–13. ACM Press.
14. Henzinger, T. A., Jhala, R., Majumdar, R., and Sanvido, M. A. A. Extreme model cecking. In *Verification: Theory & practice*, vol. 2772 of *LNCS*, pp. 232–358, 2003.
15. Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *POPL 2002*, pp. 58–70. ACM Press.
16. Henzinger et al, T. A. Temporal-safety proofs for systems code. In *CAV 2002*, vol. 2404 of *LNCS*, pp. 526–538.
17. Holzmann, G. J. *The SPIN model checker*. Addison-Wesley, 2003.
18. Jie, H. and Shivaji, S. Temporal safety verification of AVFS using BLAST. Project report, Univ. California at Santa Cruz, 2004.
19. Microsoft Corporation. Static driver verifier: Finding bugs in device drivers at compile-time. `www.microsoft.com/whdc/devtools/tools/SDV.mspx`.
20. Mong, W. S. Lazy abstraction on software model checking. Project report, Toronto Univ., Canada., 2004.
21. Necula, G. C., McPeaki, S., and Weimer, W. CCured: Type-safe retrofitting of legacy code. In *POPL 2002*, pp. 128–139. ACM Press.