THE UNIVERSITY of York

BLASTing Linux Code

— A Case Study on Software Model Checking —

Jan Tobias Mühlberg & Gerald Lüttgen
<muehlber|luettgen>@cs.york.ac.uk

University of York, UK

FMICS'06, Bonn, 26 August 2006

Motivation

- Operating systems are difficult to build and usually contain bugs; revealing, locating and correcting these bugs is costly
- Software Model Checking tools such as BLAST, MAGIC and SLAM are available, but relatively few case studies have been published
- Can BLAST aid a practitioner in order to find and avoid common programming errors?

1. BLAST Overview

- Generate instrumented program from C code by adding an ERROR label
- Compute abstraction; model check the abstraction in order to verify whether ERROR is reachable

  (Henzinger et al., 2002)

- Has been applied to Windows and Linux device drivers (Henzinger et al., 2002; Beyer et al., 2005)

1.1 Using BLAST

- Use a temporal safety specification
- Use `assert()` statements
- Directly instrument the source code with ERROR labels

```
global int lockstatus = 0;
event
 {
  pattern { lock($?); }
  guard   { lockstatus == 0 }
  action  { lockstatus = 1; }
 }
event
 {
  pattern { unlock($?); }
  guard   { lockstatus == 1 }
  action  { lockstatus = 0; }
 }
```

1.1 Using BLAST

- Use a temporal safety specification
- Use `assert()` statements
- Directly instrument the source code with ERROR labels

```
int lockstatus = 0;

void lock(void) {
  assert (lockstatus == 0);
  lockstatus = 1; return; }

void unlock(void) {
  assert (lockstatus == 1);
  lockstatus = 0; return; }
```

1.1 Using BLAST

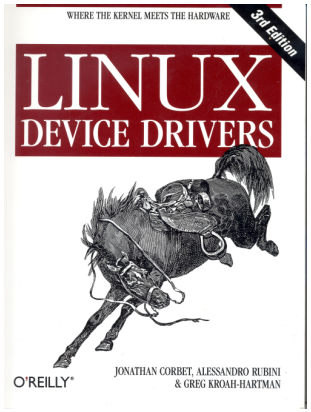
- Use a temporal safety specification
- Use `assert()` statements
- Directly instrument the source code with ERROR labels

```
int lockstatus = 0;

void error (void) {
ERROR: goto ERROR; }

void lock(void) {
  if (lockstatus == 1)
   { error(); }
  lockstatus = 1; }

void unlock(void) {
  if (lockstatus == 0)
   { error(); }
  lockstatus = 0; }
```

2. Case Study

- Interesting properties:
  - memory safety
  - correct locking
- The cases:
  - known bugs from Linux 2.6.13 and 2.6.14
  - 8 examples for each group of properties

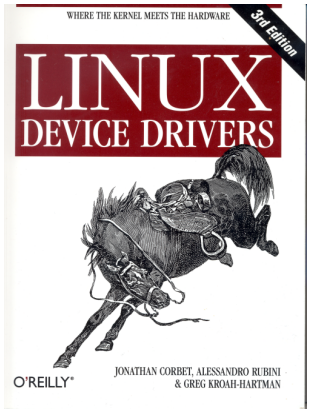
2.1 Memory Safety



"You should never pass anything to *kfree* that was not obtained from *kmalloc*." (Corbet et al., 2005)

- Several APIs for allocation and de-allocation
- General problems: Is a pointer valid at any time it is de-referenced?
  Is it valid for a specific operation?

2.2 Correct Locking



"Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time; should you attempt to do so, things simply hang." (Corbet et al., 2005)

- Are all locks initialised?
- Certain functions must not be called while a lock is held

2. Case Study

- Interesting properties:
  - memory safety
  - correct locking
- The cases:
  - known bugs from Linux 2.6.13 and 2.6.14
  - 8 examples for each group of properties

2.3 The Cases

Memory Safety	Example							
Error Type	1	2	3	4	5	6	7	8
NULL de-reference				x				
use-after-free	x	x	x		x			
double free								x
overrun error						x		
pointer arithmetic							x	
involves concurrency					x			

Locking Properties	Example							
Error Type	1	2	3	4	5	6	7	8
deadlock condition	x	x	x	x		x	x	
other API violation					x			x
involves concurrency					x			x

3. One Case in Detail

```
static int i2o_pci_probe()
 {
  struct i2o_controller *c;
  ...
  c = i2o_iop_alloc();
  ...
  i2o_iop_free(c);
  ...
  put_device
   (c->device.parent);
  ...
  return rc;
 }
```

1. Use a temporal safety specification
2. Manually add `ERROR` label; use modified `i2o_controller`
3. Manual slicing and abstraction

3.1 Temporal Safety Spec.

```
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 }
 }
```

- Preprocess source file
- Instrument code using `spec.opt`:

```
Fatal error: exception Failure
("Function declaration not
found")
```

3. One Case in Detail

```
static int i2o_pci_probe()
 {
  struct i2o_controller *c;
  ...
  c = i2o_iop_alloc();
  ...
  i2o_iop_free(c);
  ...
  put_device
   (c->device.parent);
  ...
  return rc;
 }
```

1. Use a temporal safety specification
2. Manually add `ERROR` label; use modified `i2o_controller`

```
pblast.opt:
Error found!
The system is unsafe :-(
```

3.2 Problems

#1 Pointers in General

```
void i2o_iop_free
 (struct i2o_controller *c)
{
 /* was: kfree(c); */

 if (c->alloc_status == 0)
  { goto ERROR; }
 else
  { c->alloc_status = 0;
    return; }

 ERROR: return;
}
```

#2 Function Pointers

```
struct i2o_controller *
 i2o_iop_alloc (void)
{ struct i2o_controller *c;
  ...
  c = kmalloc(sizeof(*c), ...);
  ...
  c->device.release =
   &i2o_iop_release;
  ...
  return c; }
```

3.2 Problems

#1 Pointers in General

```
void i2o_iop_free
 (struct i2o_controller *c)
{
 /* was: kfree(c); */

 if (c->alloc_status == 0)
  { goto ERROR; }
 else
  { c->alloc_status = 0;
    return; }

 ERROR: return;
}
```

#2 Function Pointers

```
struct i2o_controller *
 i2o_iop_alloc (void)
{ struct i2o_controller *c;
  ...
  c = kmalloc(sizeof(*c), ...);
  ...
  c->device.release =
   &i2o_iop_release;
  ...
  return c; }
```

3. One Case in Detail

```
static int i2o_pci_probe()
 {
  struct i2o_controller *c;
  ...
  c = i2o_iop_alloc();
  ...
  i2o_iop_free(c);
  ...
  put_device
   (c->device.parent);
  ...
  return rc;
 }
```

1. Use a temporal safety specification
2. Manually add `ERROR` label; use modified `i2o_controller`
3. Manual slicing and abstraction

4. Results

Memory Safety Error Type	Example 1	2	3	4	5	6	7	8
NULL de-reference				x				
use-after-free	x	x	x		x			
double free								x
overrun error						x		
pointer arithmetic							x	
involves concurrency					x			
Solved by	M	M	f	M	f	m	f	m

d = directly	m = minor modifications	M = manual abstraction	f = failed

4. Results

Locking Properties	Example							
Error Type	1	2	3	4	5	6	7	8
deadlock condition	x	x	x	x		x	x	
other API violation					x			x
involves concurrency					x			x
Solved by	M	M	f	f	f	m	m	f

d = directly	m = minor modifications	M = manual abstraction	f = failed

5. Conclusions

+ Good overall performance on modified source code
+ BLAST works much better for locking problems than for memory safety
- Modification of source code always required
- Non-support of pointers
- Lack of documentation
- Usability issues

6. Future Work

- Repeat the study once the demonstrated shortcomings of BLAST are resolved
- Develop metrics for a fair evaluation of software model checkers
- comparison with other tools such as MAGIC
- Conduct a study that allows one to draw quantitative conclusions

Thank you!

Further information:

http://www.cs.york.ac.uk/~muehlber/blast/

References

Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R.: 2005, Checking memory safety with BLAST, in *FASE 2005*, Vol. 3442 of *LNCS*, pp 2–18

Corbet, J., Rubini, A., and Kroah-Hartmann, G.: 2005, *Linux Device Drivers*, O'Reilly, 3rd edition

Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W.: 2002, Temporal-safety proofs for systems code, in *CAV 2002*, Vol. 2404 of *LNCS*, pp 526–538