Verifying the Use of Unsigned Integers in Linux Device Driver Code

Martin Rathgeber
Christoph Zengler
Wolfgang Küchlin

Symbolic Computation Group
(Prof. Küchlin – Wolfgang@Kuechlin.info)
www-sr.informatik.uni-tuebingen.de
W.-Schickard Institute for Informatics
U. Tübingen

3 March 2011
Verification of Linux Device Drivers

Software in safety/mission critical systems must be correct!

Problems with FLOSS:
- Many different (external) programmers
- No single source that guarantees quality

Opportunity with FLOSS:
- Open Source permits Open Correctness checks
  - by user/reseller of the FLOSS
  - by verification provider
  - by the community

Why device drivers?
- Device drivers are part of the operating system
- Device drivers have error rates up to three to seven times higher than the rest of the kernel
Infiniband Device Driver

Goal: Verification of the Infiniband/EHCA Device Driver (in loose cooperation with IBM Böblingen Labs.):

Infiniband is
- a switched fabric communications link
- used in high performance computing

EHCA is
- an IBM specific implementation of the Infiniband standard
- used on IBM’s Z series („Zero downtime“) mainframes

→ EHCA driver is critical for performance and reliability
Verification by Bounded Model Checking (BMC)

1. Manually add assertion(s) to program: `assert(property)`
2. Automatically prove/disprove the assertion by BMC
   - CBMC compiles program + (negated) assertion to a Boolean formula and solves it with a SAT-Solver without human help

```
program + assertions -> Boolean formula B -> SAT-Solver
```

- B unsatisfiable → VERIFICATION SUCCESSFUL
- B satisfiable → VERIFICATION FAILED

Rathgeber, Zengler, Küchlin (U. Tübingen)
Research Problems

Some problems

- What is the power of Bounded Model Checking and CBMC?
- „Assertion Engineering:“ How to write the assertions?
- How to add global assertions to the program in all the right places?
- How to precondition the program?
  - Remove constructs not accepted by CBMC
  - Abstract from CBMC
- How to catch compiler errors?

In general, a tool-chain is needed (cf. Avinux [PostSinKuechlin 2008])

- Write assertions
- Weave assertions into the program
- Precondition the program
- Call the verifier back-end
**CBMC: C Bounded Model Checker**

CBMC: verification tool for C programs by Clarke, Kroening and Lerda  
**Bounded**: Cut loops to bounded length

### Example (Prove/Disprove using CBMC)

<table>
<thead>
<tr>
<th>CBMC-Input</th>
<th>CBMC-Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int main() {</code></td>
<td><code>int main() {</code></td>
</tr>
<tr>
<td><code>  int x = 3;</code></td>
<td><code>  int x = 3;</code></td>
</tr>
<tr>
<td><code>  int y = 4;</code></td>
<td><code>  int y = 5;</code></td>
</tr>
<tr>
<td><code>  assert(x * y == 12);</code></td>
<td><code>  assert(x * y == 12);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

**CBMC-output:**  
VERIFICATION SUCCESSFUL

<table>
<thead>
<tr>
<th>CBMC-Input</th>
<th>CBMC-Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int main() {</code></td>
<td><code>int main() {</code></td>
</tr>
<tr>
<td><code>  int x = 3;</code></td>
<td><code>  int x = 3;</code></td>
</tr>
<tr>
<td><code>  int y = 5;</code></td>
<td><code>  int y = 5;</code></td>
</tr>
<tr>
<td><code>  assert(x * y == 12);</code></td>
<td><code>  assert(x * y == 12);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

**CBMC-output:**  
Violated property:  
line 4 function main  
assertion  
\[ x \times y = 12 \]  
VERIFICATION FAILED
CBMC Advanced Example 1

Function: a << 4. Property: (a << 4) == a*16.

```c
unsigned int multiply16(unsigned int a) {
    unsigned int p = a << 4;
    assert(p == a * 16);
    return p;
}
```

cmd line output:

cbmc mult.c --function multiply16
file mult.c: Parsing
Converting
...
Solving with MiniSAT2 without simplifier
78 variables, 107 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 0.001s
VERIFICATION SUCCESSFUL
CBMC Advanced Example 2

Function: abs(x). Property: abs(x) >= 0.

```c
short myabs(short a) {
    short result;
    if (a < 0)
        result = -a;
    else
        result = a;
    assert(result >= 0);
    return result;
}
```

CBMC output:

```
abs::myabs::1::result=-32768 (1000000000000000000000000)
```

Violated property:

```
file test.c line 8 function myabs
assertion
result >= 0
```

VERIFICATION FAILED
## Integer data types in C

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>(signed) char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>(signed) short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>(signed) int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>(signed) long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>(signed) long long</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>s8</td>
<td>u8</td>
</tr>
<tr>
<td>s16</td>
<td>u16</td>
</tr>
<tr>
<td>s32</td>
<td>u32</td>
</tr>
<tr>
<td>s64</td>
<td>u64</td>
</tr>
</tbody>
</table>

### Problem

- Unsigned integers represent non-negative values only.
- If negative integer values are assigned, the program continues.
- The bit-pattern is re-interpreted as a positive integer.
Example of Undetected Misuse

Example

```c
s32 some_function() {
    return -5;
}

int main() {
    u32 u;
    s32 s = -13;

    u = s;
    u = some_function();
    u = -1;
}
```

The code compiles and runs without error!
More Examples

The misuse may work just fine *sometimes* …

<table>
<thead>
<tr>
<th>u32 u = -1; if (u == -1) printf(&quot;yes&quot;); else printf(&quot;no&quot;);</th>
<th>u8 u = -1; if (u == -1) printf(&quot;yes&quot;); else printf(&quot;no&quot;);</th>
<th>u32 u = -1; if (u &lt; 0) printf(&quot;yes&quot;); else printf(&quot;no&quot;);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: yes</td>
<td>Output: no</td>
<td>Output: no</td>
</tr>
</tbody>
</table>
A Possible Explanation

- `u32 u = -1;
  if (u == -1) ...`

  Bit pattern `u` (32 bits) 1111 1111 1111 1111 1111 1111 1111
  Bit pattern `-1` (32 bits) 1111 1111 1111 1111 1111 1111 1111

  ⇒ Bit patterns are equal, `(u == -1)` yields `true`.

- `u8 u = -1;
  if (u == -1) ...`

  Bit pattern `u` (8 bits) 1111 1111
  Bit pattern `u` (32 bits) 0000 0000 0000 0000 0000 0000 1111 1111
  Bit pattern `-1` (32 bits) 1111 1111 1111 1111 1111 1111 1111 1111

  ⇒ `u` is extended to 32 bits.
  ⇒ Since `u` is unsigned extension is by zeroes.
  ⇒ Bit patterns are different, `(u == -1)` yields `false`. 
Error Patterns Checked

• Assigning constant to Unsigned Integer

Example

```c
u = 4;
u = -3;
```

• Comparing constant to Unsigned Integer

Example

```c
u == -3;
u < 0;
u == 4;
```

• Assigning variable value to Unsigned Integer

Example

```c
u = s;
u = some_function();
```
How to write the assertions

Problem situation

u = <exp>;

1st Ansatz

assert(<exp> >= 0); u = <exp>;

→ Wrong! <exp> may have side-effects!

2nd Ansatz

u = t = <exp>; assert(t >= 0);

→ Better, but no universal type for t.

3rd Ansatz

typeof(<exp>) t; u = t = <exp>; assert(t >= 0);

→ More complicated, but universal solution.
Weaving Assertions into the Program Automatically

- CBMC can add assertions for some types of problems automatically:
  - Array Bounds
  - Division by Zero
  - Arithmetic Overflow
- How can we add an assertion for every assignment to an unsigned type?
  $\Rightarrow$ need a tool which weaves our assertions into the source code.
- Similar to Aspect Oriented Programming
  $\Rightarrow$ weave statements for „verification aspect“ into source code.
Annotator: Source Code Annotation

Our (Flex/Bison-based) tool for source-code annotation with Unsigned-Assertions.

- Catches easy errors by itself
  - unsigned = negative_constant
  - unsigned == negative_constant
  - and similar ...
- Annotates hard case unsigned = variable_value with assertion.

\[\downarrow\]

(Unsigned-)Annotator

Handles easy cases by itself, annotates hard cases

\[\downarrow\]

CBMC

Proves / disproves assertions for hard cases.
Example Annotation

```c
u16 a;
....
a = f(x) + c;
...

⇓

u16 a;
....
typeof(f(x) + c) t; a = t = f(x) + c; assert(t >= 0);
...
```
Reminder: Verification with CBMC

program + assertions → boolean formula $B$ → SAT-Solver

- $B$ unsatisfiable → VERIFICATION SUCCESSFUL
- $B$ satisfiable → VERIFICATION FAILED
Why Preconditioning?

Preconditioning: prepare the annotated code for processing by CBMC

• At the time, CBMC could not handle Gnu-C (only ANSI C)
• At the time, CBMC could not handle embedded Assembler
• Now Gnu-C is accepted, as well as embedded Assembler

General reasons for preconditioning:

• Abstract from verifier back-end
  • Remove statements or modify assertions which back-end cannot process
• Catch compiler errors/optimizations
  • Pass code through compiler front-end first
  • Generate fresh C code from compiler intermediate code
• Abstract from source language
  • Many compilers can emit C code
  • Different back-ends may accept different language flavors
A Toolchain for Pre-Processing LINUX code

\[\text{remove-asm.pl}
\]
removes embedded assembler statements

\[\text{llvm-gcc -c -emit-llvm}
\]
Translates C-Code into LLVM-Bytecode

\[\text{llc -march=c}
\]
Translates LLVM-Bytecode back to (ANSI) C-Code

\[\text{CBMC}
\]
Proves or disproves assertions
Uses of Unsigned Ints in the Infiniband Driver

1 Using Annotator:
   - Annotator catches 16 easy errors by itself
   - Weaves assertions into the source-code

2 Applying CBMC:
   - Device drivers don’t have a `main` function
   - CBMC must be called repeatedly on all 47 EHCA entry functions
     - 3 × CBMC aborts due to internal error
     - 5 × verification takes more than 3 hours
     - 31 × CBMC proves all assertions
     - 8 × CBMC disproves one or more assertions
   - CBMC catches a total of 11 errors.
   - CBMC takes about 10sec on most entry functions
   - CBMC takes more than 20sec on 7 out of 44 entry functions
(Easy) Errors Found by Annotatorator

<table>
<thead>
<tr>
<th>File</th>
<th>Line</th>
<th>Error type</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/linux/mm.h</td>
<td>1279</td>
<td>unsigned = negative number</td>
</tr>
<tr>
<td>ehca_irq.c</td>
<td>145</td>
<td>unsigned = negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>783</td>
<td>unsigned = negative number</td>
</tr>
<tr>
<td>ehca_cq.c</td>
<td>235</td>
<td>unsigned &lt; 0</td>
</tr>
<tr>
<td>ehca_cq.c</td>
<td>358</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>ehca_irq.c</td>
<td>155</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>251</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>289</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>364</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>559</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>595</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>603</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>638</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>660</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>hcp_if.c</td>
<td>699</td>
<td>unsigned == negative number</td>
</tr>
<tr>
<td>ehca_irq.c</td>
<td>721</td>
<td>unsigned == negative number</td>
</tr>
</tbody>
</table>
(Hard) Errors found by CBMC

<table>
<thead>
<tr>
<th>File</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>hcp_if.c</td>
<td>386, 534, 557, 594, 601, 697, 866</td>
</tr>
<tr>
<td>ehca_uverbs.c</td>
<td>272, 294</td>
</tr>
<tr>
<td>ehca_mrmw.c</td>
<td>808</td>
</tr>
</tbody>
</table>
Summary / Lessons

Summary
- Found 27 bugs in real Linux code in real industrial setting
- 11 bugs found by formal verification tool
- Built source-code annotation tool
  - specialized for Unsigned errors
  - generalizable approach

Lessons
- Formal software verification is here
- Tools like CBMC find real errors in real code at compile-time
- General specification language is needed
- Tool-chain is needed
  - Pre-process „foreign“ languages (Gnu-C, C++, ...)
  - Abstract from verifier back-end (CBMC, ...)
  - Weave global assertions/specifications into programs
References


Thank you for your attention!

Questions?