Verification of Linux Device Driver Code

The `unsigned int` Case

Martin Rathgeber, Christoph Zengler, and Wolfgang Küchlin

Symbolic Computation Group, W. Schickard Institute of Informatics
University of Tübingen, D-72076 Tübingen, Germany
www-sr.informatik.uni-tuebingen.de

Abstract. We examine a Linux device driver used on IBM mainframes and focus on the misuse of unsigned integers. First, we develop a special-purpose tool which analyzes the parse-tree of the source code. It reports easy cases of misuses immediately, and otherwise annotates the source code with suitable assertions. In a second step, we try to solve the hard cases by proving or disproving these assertions with CBMC. We found 27 errors in this device driver which were not known before.

1 Introduction

Since a large number of operating system crashes are due to device driver errors [1], Microsoft initiated the SLAM project and produced the tool set "Static Driver Verifier" (SDV) [2]. To remain competitive, the Linux community must come up with similar tools to support their developers. One effort in this direction was Avinux, developed by Post et al. [3, 4]. Avinux checks thousands of Linux device drivers for the absence of certain kinds of errors. Another recent effort is LDV [6] by the Russian Academy of Sciences.

Here, we analyze the (mis-)use of unsigned integers in the EHCA Linux device driver. EHCA is an IBM specific implementation of the Infiniband switched fabric communications link used on IBM's Z-series mainframes. Similar to [3], our verification builds on CBMC [5], a bounded model checker for the C language, but alternatives such as LLBMC by Sinz et al. [7] could be used instead.

For any assignment `u=f(x)` to an unsigned integer variable `u`, one must assure that `f(x)` only returns non-negative values. But in our analysis we found situations in which this was not always true in the EHCA code (and other code surrounding this module). Several errors of this kind were already fixed in the months prior to our work in 2010 (cf. the EHCA patches), but apparently not all occurrences were found. Our tool systematically detects errors of this kind at compile time and thus helps developers produce more reliable code.

2 Unsigned Integers

The C language provides several unsigned integer types such as `unsigned`

`char`, `unsigned short`, `unsigned int` and `unsigned long`. Since exact value ranges of those types are architecture dependent they are usually not used in Linux kernel code. Instead the Linux kernel defines its own unsigned integer types `u8`, `u16`, `u32` and `u64` with architecture independent sizes.

The type of misuse we consider in this paper boils down to type violations, as in `u32 u = -1;`. At first this seems an obvious error which should occur rarely. But as recent patches of the Linux kernel and ou results show this happens quite frequently, and in some special cases nothing goes wrong. Consider the following program:

```
int main() {
   u32 u = -1;
   if (u == -1) printf("yes");
   else printf("no");}
```

The output will be `yes` and this is exactly what should happen. There will be no warnings from the compiler (at least without special compiler options) and also the execution of the program will work absolutely fine. Now consider a slight variation with type `u8` for `u`:

```
int main() {
   u8 u = -1;
   if (u == -1) printf("yes");
   else printf("no");}
```

Now the output will be `no`.

Hence even simple cases like this are tricky because the semantics of the program may depend on compiler optimizations or it may depend on the bitsize of the processor (in our case causing extensions by leading zeroes).

So our conclusion is: It is always an error to assign a negative value to a variable of an unsigned integer type, despite the fact that it may work in some cases.

3 Annotation

We have to prove that no variable of unsigned integer type will ever be assigned a negative value. This means we have to look at every relevant assignment and insert a correctness assertion. In practice, this needs automation and is best done in a style that resembles Aspect Oriented Programming (AOP). From the AOP point of view we represent our crosscutting concern of verification by adding assertions at join points in the program.

If the value assigned is a constant `n`, we have an easy case which is an error if `n<0`. In general, however, we need to prove an assertion. We add the assertions as follows, carefully avoiding any side-effects of a second call to `f(x)`:

```
{typeof(f(x)) t; u=t=f(x);\\
 assert(t>=0);}
```

To make sure the line numbers remain unchanged we place all the annotations in the same line as the assignment we annotate. Later we will try to prove these assertions with CBMC.

Unfortunately, we did not find a suitable reliable AOP tool for our purpose. We developed our own tool ANNOTATOR which walks the parse-tree and inserts the assertions automatically. ANNOTATOR also immediately reports the easy cases of misuses such as `u = n`, `u < 0`, `u == n` which any compiler might detect. For example `u < 0` only makes sense if it is possible that `u` has a negative value (or else it may be a dirty short-cut to check for the most significant bit). So something looks suspicious and ANNOTATOR will report this.

4 Results

Our verification process consisted of the following steps:

1. Annotate the source code with assertions and search for obvious errors (done by ANNOTATOR).
2. Do some necessary steps of preprocessing to expand makros and `include` directives (done by some special scripts, the kernel makefile and the GCC)
3. Try to prove or disprove the assertions with CBMC.

We applied those steps on the EHCA device driver in Linux kernel version 2.6.35.2. During Step 1 ANNOTATOR found 16 easy problem cases. Three errors were of type `u = n` in line 1279 of file `include/linux/mm.h`, line 145 of `ehca_irq.c`, and line 783 of `hcp_if.c`. In line 235 of `ehca_cq.c` it found `u < 0`. Twelve errors of type `u == n` were found in line 358 of `ehca_cq.c`, lines 155 and 721 of `ehca_irq.c`, and in `hcp_if.c` in lines 251, 289, 364, 559, 595, 603, 638, 660, and 669.

More interesting errors were found in Step 3 with CBMC. To (dis-)prove the assertions in a normal program, CBMC traverses the whole program, beginning with the main function. However, a device driver is not one contiguous piece of software with one main entry function, but a collection of functions, each providing a special functionality needed by the kernel to control a hardware device (e.g. triggered by an interrupt). Therefore one has to call CBMC separately for each function provided by the device driver to the Linux kernel. In the EHCA driver we identified 47 such functions.

In 3 cases CBMC aborted due to an internal error. In 5 cases we aborted the execution of CBMC because it took more than three hours. In the remaining 39 cases we got a result. In 31 cases CBMC proved all the assertions. In 8 cases CBMC disproved an assertion.

CBMC disproves at most one assertion per run. So to find all the assertions that can fail we repeatedly removed the assertions disproved so far and called CBMC again. All in all we found eleven errors with CBMC. These were in file `hcp_if.c` (on lines 386, 534, 557, 594, 601, 697, 866, 886), in `ehca_uverbs.c` (on lines 272, 294), and on line 808 of `ehca_mrmw.c`.

In many cases the verification process with CBMC is very quick. In 37 out of 44 cases CBMC needed less than 20 seconds to prove all the assertions or to disprove one assertion.

References

1. A. Chou, J. Yang, B. Chelf. S. Hallem and D. Engler: An Empirical Study of Operating Systems Errors. In *Proc. SOSP 2001.*
2. T. Ball, E. Bounimova, V. Levin, R. Kumar and J. Lichtenberg. The Static Driver Verifier Research Platform. In *Proc. CAV 2010.*
3. H. Post and W. Küchlin. Integrated Static Analysis for Linux Device Driver Verification. In *Proc. IFM 2007*
4. H. Post, C. Sinz and W. Küchlin. Towards Automatic Software Model Checking of Thousands of Linux Modules—A Case Study with Avinux. *Software Testing, Verification and Reliability, 19(2)*, 2009.
5. E. Clarke, D. Kroening and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. TACAS 2004, LNCS 2988.*
6. `http://linuxtesting.org/project/ldv` (2011-08-01).
7. C. Sinz, S. Falke and F. Merz. A Precise Memory Model for Low-Level Bounded Model Checking. In *Proc. 5th Intl. Workshop on Systems Software Verification (SSV 2010)*, 2010.