Handling hardware device failures

Tutors: **Julia Lawall & Gilles Müller**

Student: **Louisa Bessad**

Contents

1 Introduction

Systems need to use drivers to be able to access devices. But when a device fails, this can cause driver to crash or hang. It has been demonstrated that most system failures are due to hardware device failures. Most of the time these failures are transient. By tolerating device failures drivers can prevent the system from crashing or hanging. In this way the reliability of the system can be improved. Most approaches for research on drivers failures fall into two categories. The first is the static bug finding ; it analyzes the interface between the driver and the kernel to find incoherences. The second is the runtime fault tolerance. Most of the time these approaches need new operating sytem or new driver models ; it is not the case for static analysis. Usually these approaches find the failure but do not fix it. While they try to reduce faults on the interface between driver and device, they need specific interfaces and languages.

This project fall in the first category. The goal is to find every unchecked inputs from drivers, it is possible to fix those failures. In this way different types of code need to be searched. First, we look for functions that are known to perform I/O and verify that the return value is checked. Second, we want to find variables used as pointers or indexies of an array and check that they do not reference a forbidden memory area. Third, we search structures dependant on input from the device that could be corrupted by a bad allocation. Fourth, calls to panic function are searched. Finally, we look for loops that wait for a specific hardware device state without timeout. We had studied this last point.

First we will talk about the loop without timeout and which solution can be proposed to this problem. In the second part we will present the results bring by our solution, and we will compare these to another results bring by a previous analysis with a different tools.

2 The choosen problem and its solution

The loops without timeout can be called infinite loop because the terminating condition can be not reached. We had considered as a timeout a variable that is incremented or decremented in the loop or another assignment (such as ≪= or ≫=). Moreover it is used as exit condition of the loop and it cannot be used as a pointer or index of an arrray. According to these definitions two kinds of infinite loops (can be / has been) found.

2.1 Infinite loops

The first type of infinite loops uses as an exit condition functions that call drivers or a pointer value that assigns the return of one of these functions. In this case if the function never returns a value that allows to exit of the loop, it could cause the crash or the hanging of the system.

Y a aussi les autres variables qu'on ne sait pas d'ou elles sortent.

Mettre un exemple de chaque

```
1         for (i = 0; i < nr_pages; i += pages_per_kpage)
2                 free_page((unsigned long)(queue->queue_pages)[i]);
```

Listing 1: An instance of a possible infinite for loop, the Sun Ethernet driver can hang if the pointer m->next has a bad assignment: net/ethernet/ibm/ehea/ehea_qmr.c

```
1  void omap_vrfb_restore_context(void)
2  {
3          int i;
4          unsigned long map = ctx_map;
5
6          for (i = ffs(map); i; i = ffs(map)) {
7                  /* i=1..32 */
8                  i--;
9                  map &= ~(1 << i);
10                 restore_hw_context(i);
11         }
12 }
```

Listing 2: video/omap2/vrfb.c

The second kind uses an infinite exit condition ";" and an alternative calling a driver function to exit of the loop without any break.

For instance for(;;), while(1)

```
1  static int uwb\_rc\_new\_index(void)
2  {
```

```
3           int index = 0;
4
5           for (;;) {
6                   if (!uwb\_rc\_find\_by\_index(index))
7                           return index;
8                   if (++index < 0)
9                           index = 0;
10          }
11  }
```

Listing 3: uwb/lr-cr.c

```
1   for (;;) {
2                   status = serial\_in(port, UART\_LSR);
3           %       if ((status & BOTH\_EMPTY) == BOTH\_EMPTY)
4                           return;
5                   cpu\_relax();
6           }
```

Listing 4: tty/serial/8250_early.c

These infinite loops could cause the same problems than the previous type of infinte loops.

To avoid these infinite loop we will insert a segment of code in the body of the loop. This segment will contain the addition of a timeout, its allocation and the exit condition of the loop will be modified.

2.2 Proposed solution

To find and fix these loops we used a bug finding tool called Coccinelle. This uses patches that contain rules, a rule will transform the code by the deletion and the addition of code. In this way, Coccinelle scans the code to find lines matching with the searching condition and applies the transformation. In our case there is two patches; one for the "while" infinite loops and another one for the "for" infinite loops. The condition rule will be an infinite loop and the transformation will be the addition and allocation of a timeout and the modification of the exit condition of the loop. Furthermore our patches will contain several rules, because there is many ways to write a same loop with C programming.

Mettre un exemple de deux boucles identique ++E1 ou E1++

Nevertheless Coccinelle does not allow to match the loop: do...while(...), so we chose not to handle this type of loop.

3 Results and Comments

3.1 With Coccinelle

The Linux3.2.59 Kernel had been used as testing platform. Firstly we wanted to know the pourcentage of loop and infinite loop in the kernel. However during the execution of patches some files have been skipped by Coccinelle to avoid a very long execution time. It appeared that whatever the computer used to execute the patches without the timeout option of Coccinelle, the execution could never end. For those files the analysis must be made by a programer. The following array shows the result of these executions, the "EXN" column correspond to the number of skipped files:

	Total loop	Infinite loop	EXN files
while	9342	4509	97
for	27616	468	87

We can see that 48% of the "while" loop does not have a timeout whereas only 2% of the "for" loop are infinite loop. / These infinite loops concern drivers. Nevertheless these drivers are old, in the recent kernel these bugs had been corrected. /

3.2 With another tool: Carburizer

Another study had been made by /blablablabla/, it uses an automatic code manipulation-tool :"Carburizer" created specificly for finding hardware device failures. In this way it takes a driver and scans its code. Then it lists bugs that can occur in it and suggests fixes for these bugs according to their type. In the case of infinite loop, Carburizer finds 2383 locations where drivers detect failures by timeouts on the Linux3.2.59 Kernel.

3.3 Comparison

We can that Carburizer finds 2594 infinite loops less. Given these results we can think, there is a problem with ours. But it is not the case, this is due to the fact that we have not the same definition of an infinite loop. Looking at Kadav's results we can see that only: blabla list des resultats ou tableau comparif.

4 Conclusion