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 and 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 will study the loop without timeout and propose a solution to this problem. Then we will present the results bring by our solution, and we will compare these to anothers from a previous analysis using a different tool.

2 The choosen problem and its solution

When the terminating condition of a loop is never reached, we stay in the loop infinitely. This could cause the crash or hanging of the system, it is the behaviour of a loop without timeout. So the existence of a timeout in a loop is really important as its definition. We consider as a timeout a variable that is incremented or decremented in the loop or another assignment (such as $\ll=$ or $\gg=$). Moreover it has to be used as exit condition of the loop, or a part of this condition. Furthermore it cannot be used as a pointer or index of an arrray. In this way every loop that does not respect these conditions will be considered as a loop without timeout, also called infinite loop.

2.1 Types of infinite loops

By studiing the source of a kernel we can find two types of loop without timeout. The first type correponds to loops with an exit condition which is not a timeout variable. In this case the therminating condition can be a function that call drivers, a pointer value that assigns the return of one of these functions, a variable or metavariable defined in another file. But if the function never returns a value that allows to exit of the loop, the system will stay infinitely in the loop Listing 1. The problem is the same if the pointer or the variable from another file used as exit condition of a loop have always a bad allocation.

Listing 1: This loop calls the readl function which will ask a data, referenced as the argument of the function, to a driver which is a keyboard controller in this case.

```
1  while ((readl(keypad->reg_base + SKE_CR) & SKE_KPASON)
2  && --retries)
3  msleep(5);
```

The second kind of loops without timeout uses an infinite exit condition: “for(;;)”or “while(1)”. For instance:

Listing 2: This loop does not have an exit condition. Whereas it is a part of the code that handle the register of every driver during bootup.

```
1  while (1) {
2    spin_lock(&req_lock);
3    /.../
4    set_current_state(TASK_INTERRUPTIBLE);
5    spin_unlock(&req_lock);
6    schedule();
7    __set_current_state(TASK_RUNNING);
8  }
```

Sometimes to ensure a possible exit of the infinite loop, the body of the loop contains an alternative. Usually the alternative's condition is based on variables' value, but there are also alternatives whitch use a call to a driver function as condition. However this function could never return the right value to exit the loop, in this case the alternative is useless. For the alternative which the condition can besatisfied, there are alternatives with a break, Listing 3, or another way to exit the loop Listing 4.

Listing 3: This loop of an Intel driver has two possibilities to exit of this infinite loop.

```
1   while (1) {
2     variable_name_size = 1024;
3     status = efivars->ops->get_next_variable(
4     &variable_name_size,
5     variable_name,
6     &vendor);
7     if (status != EFI_SUCCESS) {
8       break;
9     } else {
10      if (!variable_is_present(variable_name,
11      &vendor)) {
12        found = true;
13        break;
14      }
15    }
16  }
```

Listing 4: In this loop we can notice the presence of an alternative without any break. If the condition never come true the program will never exit this loop. Moreover if alternative's condition is satifsied the code following the loop will never be executed. This loop is contained in a fonction of a console driver for a printer device

```
1   for (;;) {
2     status = serial_in(port, UART_LSR);
3     if ((status & BOTH_EMPTY) == BOTH_EMPTY)
4     return;
5     cpu_relax();
6 }
```

So even when the infinite loop has a possibility to exit the loop with an alternative containing a "break" or a "return", it stays very dangerous. Because the possibility could never occur or segment of code are skipped. So most of the time the alternatives are not a solution to solve the second type of infinite loops.

To avoid these two types of 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