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 [1] 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 brought by our solution, and we will compare these to anothers from a previous analysis using a different tool.

2 The chosen problem and its solution

When the terminating condition of a loop is never reached, we stay in the loop indefinitely. 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 terminating condition can be a variable or metavariable defined in another file, a function that call drivers, a pointer value that is assigned by the return of one of these functions. 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 which 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 be satisfied, 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

We use a bug finding tool called Coccinelle [2] to find and fix the infinite loops of the kernel. Coccinelle uses patches which 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 are 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. Nevertheless Coccinelle does not allow to match the "*do*{} *while*" loop, so we choose to not handle this type of loop.

Listing 5: Modification of a "for" infinite loop

```
1   //Code to add
2   + unsigned long long delta = (cpu/khz/HZ)*2;
3   + unsigned long long _start = 0;
4   + unsigned long long _cur = 0;
5   + unsigned long long timeout;
6   + timeout = rdstcll(start) + delta;
7   //Code to match
8   // ... means everything
9   for(...;...;...){
10     ...
11  //Code to add
12  + if (_cur < timeout){
13  + rdstcll(_cur);
14  +}
15  + else {
16  + break;
17  +}
18  //Code to match
19  }
```

Listing 6: Modification of a "while" infinite loop

```
1   //Code to add
2   + unsigned long long delta = (cpu/khz/HZ)*2;
3   + unsigned long long _start = 0;
4   + unsigned long long _cur = 0;
5   + unsigned long long timeout;
6   + timeout = rdstcll(start) + delta;
7   //Code to match
8   while(...){
9     ...
10  //Code to add
11  + if (_cur < timeout){
12  + rdstcll(_cur);
13  +}
14  + else {
15  + break;
16  +}
17  //Code to match
18  }
```

Listing 7: The result of the patch's execution on the while infinite loop from Listing 1

```
1   unsigned long long delta = (cpu/khz/HZ)*2;
2   unsigned long long _start = 0;
3   unsigned long long _cur = 0;
4   unsigned long long timeout;
5   timeout = rdstcll(start) + delta;
6   while ((readl(keypad->reg_base + SKE_CR) & SKE_KPASON)
7   && --retries){
8     msleep(5);
9     if (_cur < timeout){
10      rdstcll(_cur);
11    }
12    else {
13      break;
14  }
```

The added timeout uses three predefined global variables of the kernel: "cpu", "khz" and "HZ". It also uses a maccro "rdstcll" which adds a timeout value to the parameter value. The "cur" variable is incremented at each turn of the loop, the "timeout" variable is fixed. In this way, the condition at the line 11 will not be true after some time, so the

line 15 will be executed which leads to the exit the loop. This is a real solution to the infinite loop problem.

Nevertheless our patches cannot conain only one rule (Listings 5 & 6), because every type of loop "while" or "for" infinite or not could match this rule. So before this code we have to add others rules, that we will catch the non infinite loops; the loops which contain a timeout. The remainding loops will match the last rule and be modified (Listings 1 to 4 for instance).

Listing 8: A rule to catch one type of "for" non infinite loop

```
1  //Example:
2  //for(i=x;i<y;i++)
3  //Instruction;
4
5  //Metavariables: 2 expressions E1 and E2
6  @@ expression E1, E2 @@
7  // <+... x ...> means at least one appearence of x
8  for(...;...;<+... E1++ ...+>)
9  E2;
```

Listing 9: A rule to catch one type of "while" non infinite loop

```
1  //Example:
2  //while( x && (i <= y) && z){
3  //Some instructions
4  //}
5
6  //Metavariables: 1 expression E1 and 1 constant
7  @@ expression E1; constant C @@
8  while(<+... E1 <= C ...+>){
9  ...
10 }
```

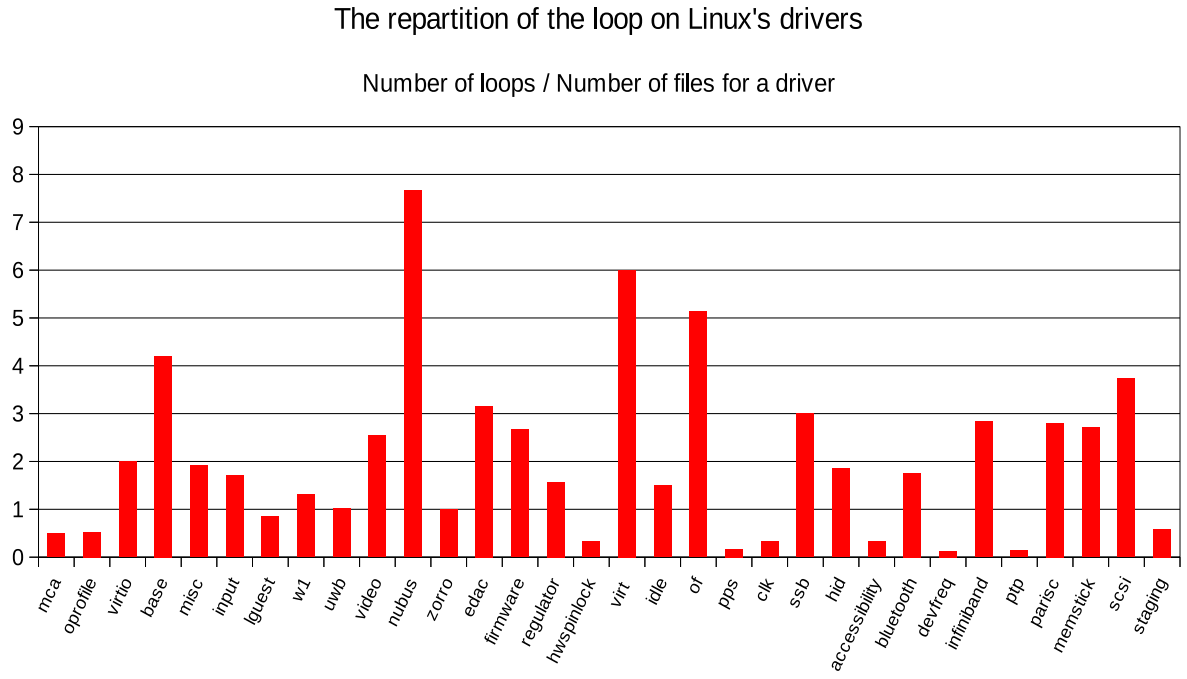
We have studied the issue of the loop without timeout and exposed one solution to this problem. Now we can test this solution and compare our results to anothers from a previous analysis using a different tool.

3 Results and Comments

In this section we present the results produced by our patches, these obtained with Carburizer and a comparison ofthese results. We both used the Linux3.2.59 Kernel as testing platform, infinite loops have not been searched on the entire kernel code, but only on drivers.

3.1 With Coccinelle

First, to have representative results we have to know the pourcentage of loop and infinite loop in the kernel. The following graph presents a repartition of the loop according to the type of the driver:



In this graph we represent the quotient of the number of loops on the number of files for a driver, because it is more representative. If we represent only the number of loops per driver we c, for instance, the “scsi” driver has almost 2500 loops whereas the folder which contains the base driver has arount 10 loops. We can say nothing about that. With this graph we can see that these two types of drivers have the same purcentage of loop.

Concerning the purcentage of infinite loops, during the execution of the 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 these files the analysis must be made by a programer. The following array shows the result of these executions, the “EXN” column

corresponds 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" loops does not have a timeout whereas only 2% of the "for" loops are infinite loop. Among the drivers that contain infinite loops 81% of them dated from the 2000s and the remainding from 90's. Nevertheless in the results produced by Coccinelle there are some false positives. In the drivers' code, there are loops that use as exit condition a call to a function that does not call the driver. For instance the functions that read or write on the registers. We cannot consider these as an I/O, but our patches do not consider the difference between a function which makes an I/O and another one. There also false negatives in our results, some loops have a timeout and thus a finite condition. But these loops can call a function which make an I/O in their body. This has to be consider as a possible infinite loop, because even if the loop's condition is finite, the function could never return. Yet in our patches we have consider that a loop with a finite condition cannot be infinite and thus we do not modify this loop. This is a problem. Even if there are false positives and false negatives in our results, the major part is in agreement with Carburizer's results.

3.2 With another tool: Carburizer

Another study of the drivers has been made. It uses an automatic code manipulation-tool called "Carburizer" [1]. It has been created specificly for finding hardware device failures. In this way it takes a driver and scans its code to find input from the drivers. Then it lists bugs that can occur on it and suggests fixes for these bugs according to their type before corruption appeared. It can be a recovery function to restore the driver to a functionning state, a validation of an unchecked value, the handling of interruptioin with a runtime service, or the addition of a timeout in the case of infinite loop. Carburizer has its own defintion of a timeout:

> A timeout counter is a variable that is either incremented or decremented in the loop, not used as an array index or in pointer arithmetic, and used in a terminating condition for the loop, such as a while clause or in an if before a break , goto , or return statement. If a loop contains a counter, Carburizer determines that it will not loop infinitely. We also detect the use of the kernel jiffies clock as a counter.

In the case of infinite loop, Carburizer finds 2383 locations where drivers detect failures by timeouts on the Linux3.2.59 Kernel. We can see that there is a difference between the

4977 infinite loops that we found and this result, Carburizer finds 2594 infinite loops less. In this way we will compare our results to understand this diffirence.

3.3 Comparison

Given the difference between our results and these of Carburizer, we can think taht there is a problem with ours. It is not the case, this is due to two factors. Firstly, the results found with Carburizer does not present the number of infinite loops like us. But it represents the number of file that contain these infinite loops. Moreover these files does not contain the source code, they are the ".o" files, this number of file is inferior to the number of ".c" files. Secondly, the definition of an infinite loop is not the same in the two studies.

The two definitions consider that loop with a counter cannot loop infinitely, so Carburizer should have the same false negatives that we have with Coccinelle. However Carbyrizer considers as a bug the loops with a counter that call a function which makes an I/O, this is the first difference. The second is that in the case of infinite loops we modify every loops, whereas Carburizer fixes only infinite loops that call functions which make I/O. Moreover Carburizer does not handle the condition with a comparison of a pointer's value, even when the body of the loop contains a function which calls the driver. This type of infinite loop is matched and modified by our patches. The same difference appears when the condition of the loop contains a comparison with a maccro, even when the maccro is not defined in the file that contains the infinite loop.

Nevertheless Carburizer finds loops that are missed by our patches. This occurs when the condition of the loop contains a comparison bit-a-bit of the return of a driver function and a constant or a macro or a pointer. It also happens when the condition is only a call to a function, in this case Carburizer only fixes the loops where the function makes an I/O. Moreover Carburizer can fix the "*do*{} *while*" loops that Coccinelle does not handle.

Despite of these differences, there are infinite loops that are found by both solutions. The infinite loop with calling to I/O functions are found by the two solutions. The infinite loops with an alternative which contains a break are also found. Carburizer extends this case of infinite loops to the loops without timeout with an alternative which contains "goto" and "return" code.

Finally we can say that the major difference in the count of infinite loops is due to the fact that in our patches we consider that each calling functions could not return whereas Carburizer only considers the I/O functions like this.

4 Conclusion

To conclude, according to our definition, this new solution to the handling of infinite loops and hardware device failures shows that there is a lot of problems with the management of the driver on the Linux kernel. Thanks to Coccinelle we can find and fix these bugs, that we did. Some of these problems have been solved in the new version of the kernel. As the major part of these failures are transient, the users does not notice /it/, the impact is critical and more visible in fields where drivers are the base of everything.

By studying the kernel at a law-level, that I did not knwo, I learnt that the consequences of the way that we write a code can be really serious. In this way I could noticed that there is a serious problem of security in the driver's code, an issue that I will keep on study.

References

[1] M. J. Renzelmann A. Kadav and M. M. Swift. Tolerating hardware device failures in software. *Symposium on Operating Systems Principles*, 2009.

[2] A program matching and transformation tool for system code. `http://coccinelle.lip6.fr`.