Software Based Glitching Detection

Jakob Löw

15.11.2021
• The schematic shows the clock line of a targeted microprocessor
• Normally a clock signal consists of a rectangle wave with fixed period like shown in „Cycle A“
• „Cycle B“ shows a cycle with a glitch
• The clock signal is modified to include a second high-signal within the duration of „Cycle B“
• The goal is to alter the execution of the processor, i.e. skipping an instruction
Balasch et. al described the effects of clock glitches in detail for AVR microprocessors.

The table shows the actually executed instruction depending on the period of the induced glitch.

According to the decayed states with different glitch periods the actually executed instruction first decays to a zero state before the new instruction opcode is loaded.

<table>
<thead>
<tr>
<th>Glitch period</th>
<th>Instruction</th>
<th>Opcode (base 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TST R12</td>
<td>0010 0000 1100 1100</td>
</tr>
<tr>
<td>-</td>
<td>BREQ PC+0x02</td>
<td>1111 0000 0000 1001</td>
</tr>
<tr>
<td></td>
<td>SER R26</td>
<td>1110 1111 1010 1111</td>
</tr>
<tr>
<td>≤ 57ns</td>
<td>LDI R26,0xEF</td>
<td>1110 1110 1010 1111</td>
</tr>
<tr>
<td>≤ 56ns</td>
<td>LDI R26,0xCF</td>
<td>1110 1100 1010 1111</td>
</tr>
<tr>
<td>≤ 52ns</td>
<td>LDI R26,0xF</td>
<td>1110 0000 1010 1111</td>
</tr>
<tr>
<td>≤ 45ns</td>
<td>LDI R16,0x09</td>
<td>1110 0000 0000 1001</td>
</tr>
<tr>
<td>≤ 32ns</td>
<td>LD R0,Y+0x01</td>
<td>1000 0000 0000 1001</td>
</tr>
<tr>
<td>≤ 28ns</td>
<td>LD R0,Y</td>
<td>1000 0000 0000 1000</td>
</tr>
<tr>
<td>≤ 27ns</td>
<td>LDI R16,0x09</td>
<td>1110 0000 0000 1001</td>
</tr>
<tr>
<td>≤ 15ns</td>
<td>BREQ PC+0x02</td>
<td>1111 0000 0000 1001</td>
</tr>
</tbody>
</table>
**Existing Glitch Detection Techniques**

*Instruction Duplication*

- The code shows the duplication and check of a single memory load (ldr) instruction
- Instead of simply loading the value at x0 into w1 it is loaded twice
- Afterwards the two loaded values are compared (cmp), if they do not match a glitch error is generated
- Simple instruction duplication is still vulnerable to single clock glitches as shown by Yuce et. al

```
  ldr  w1, [x0]
  ldr  w0, [x0]
  cmp  w1, w0
  bne  glitch_error
```
Existing Glitch Detection Techniques

Loop Count Validation

- Proy et. al describe glitch detection via loop count validation
- The two code examples on the right show the concept of this glitch detection mechanism
- For each loop variable a second variable is added, which is modified the same way the original loop variable is
- After the loop the second variable is used to validate the loop condition

```c
int i = 0;
while (i < 10) {
    // ...
    i++;
}
```

(a) Loop with iteration variable

```c
int i = 0;
int j = 0;
while (i < 10) {
    // ...
    i++;
    j++;
}

assert (j >= 10);
```

(b) Loop from (a) with validation

Fig. 4: Basic loop validation example
Expression validation is similar to instruction duplication

Rather than placing the duplication right next to the original instruction it is placed at the last location inside the function where the computed value is still in scope

The code on the right shows the validation location for two variables \( x \) and \( y \)

```c
int main(int argc, char **argv)
{
    int x = argc * 10 - 2;
    if(argc > 1)
    {
        int y = x * 3;
        if(argc > 2)
            puts(argv[1]);

        // <-- validate 'y' here
    }

    // <-- validate 'x' here
    return x;
}
```
In a compiler a function is represented as a control flow graph (basic blocks + edges)

Expressions are converted to SSA form, where each value is only assigned once

The best possible spot for a validation is the end of the last block where all preceding blocks are successors of the creation of the variable

Block 6 shows the validation of variable ‘x_10’ which was created in Block 2, corresponding to variable ‘x’ shown on the previous slide
Novel Glitch Detection Approach: Expression Validations

Performance Impact

- Best possible performance decrease is the same as with instruction duplication
- The validations are placed at the last possible position and thus extend the life range of variables to the maximum possible
- Longer life ranges make register allocation harder resulting in the Compiler generating less performant code
- => The novel glitch detection approach is best applied selectively instead of on the full program
References

