CARISSMA

Institute of Electric, Connected and Secure Mobility

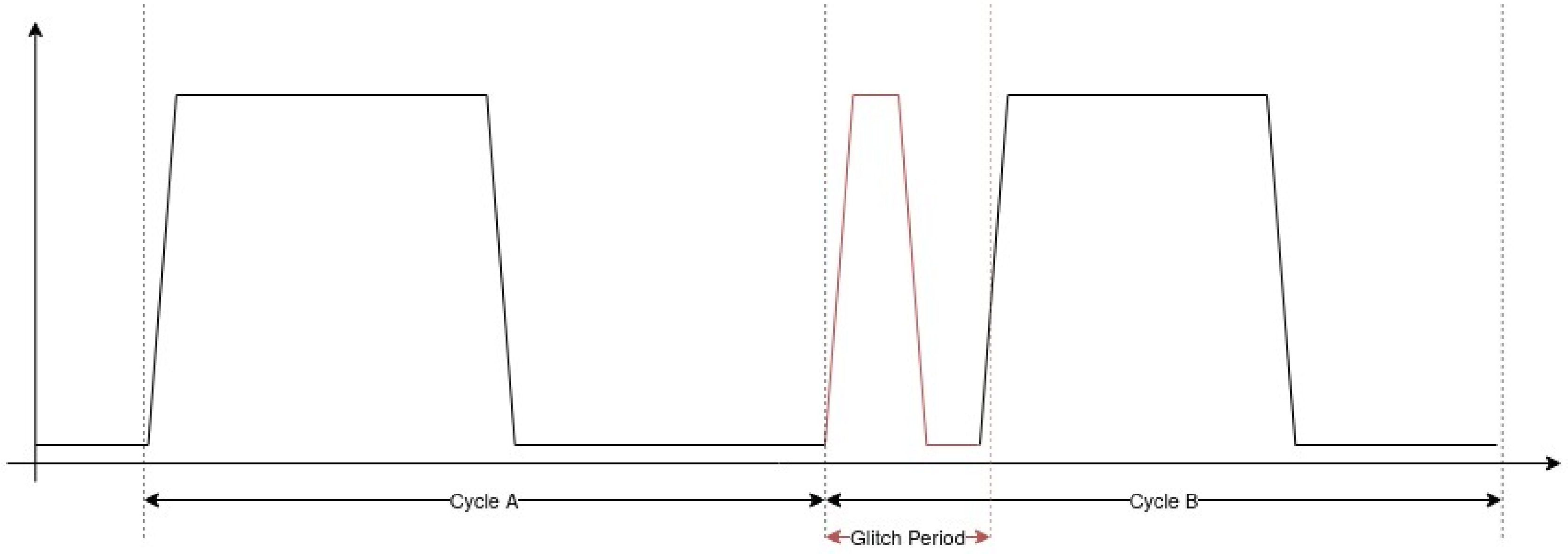
Software Based Glitching Detection

Jakob Löw

15.11.2021

Technische Hochschule Ingolstadt

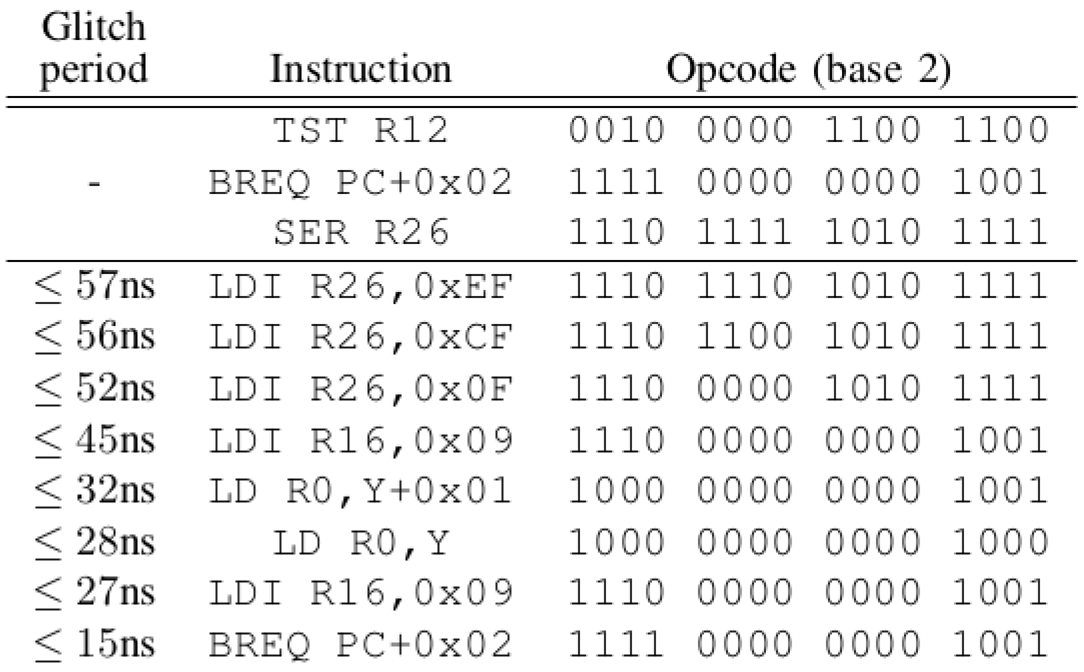
Clock Glitching



- 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

Exact Effects of Clock Glitches

- **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 dacays to a zero state before the new instruction opcode is loaded.**



Glitch period	Instruction	Opcode (base 2)
	TST R12	0010 0000 1100 1100
-	BREQ PC+0x02	1111 0000 0000 1001
	SER R26	1110 1111 1010 1111
$\leq$ 57ns	LDI R26,0xEF	1110 1110 1010 1111
$\leq$ 56ns	LDI R26,0xCF	1110 1100 1010 1111
$\leq$ 52ns	LDI R26,0x0F	1110 0000 1010 1111
$\leq$ 45ns	LDI R16,0x09	1110 0000 0000 1001
$\leq$ 32ns	LD R0,Y+0x01	1000 0000 0000 1001
$\leq$ 28ns	LD R0,Y	1000 0000 0000 1000
$\leq$ 27ns	LDI R16,0x09	1110 0000 0000 1001
$\leq$ 15ns	BREQ PC+0x02	1111 0000 0000 1001

Existing Glitch Detection Techniques

Instruction Duplication

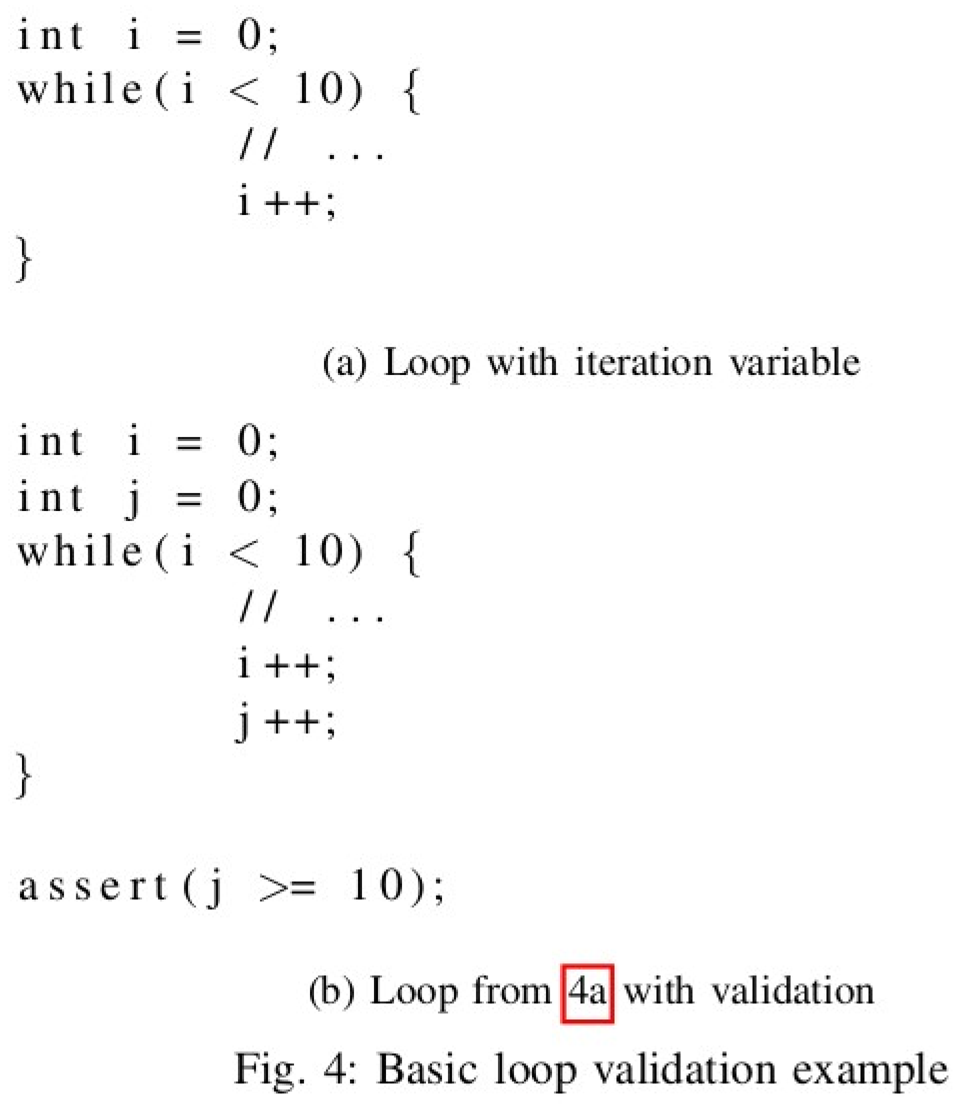
- **The code shows the duplication and check of a single memory load (ldr) instruction**
- **Instead of simply loading the value at x0 into w1 it is loaded twice**
- **Afterwards the two loaded values are compared (cmp), if they do not match a glitch error is generated**
- **Simple instruction duplication 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**



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

(a) Loop with iteration variable

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

assert(j >= 10);
```

(b) Loop from $\boxed{4a}$ with validation

Fig. 4: Basic loop validation example

Novel Glitch Detection Approach: Expression Validations

Identifying Locations for Validations (1/2)

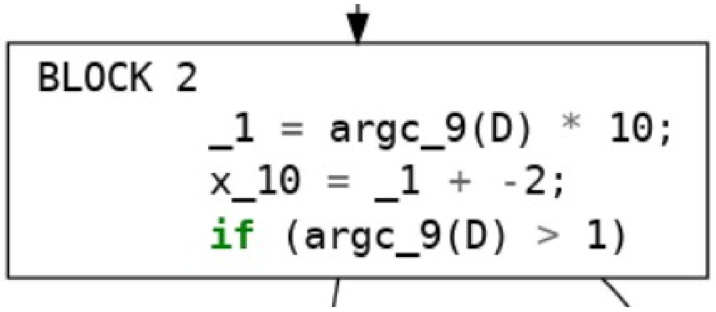
- **Expression validation is similar to instruction duplication**
- **Rather than placing the duplication right next to the original instruction it is placed at the last location inside the function where the computed value is still in scope**
- **The code on the right shows the validation location for two variables ‚x‘ and ‚y‘**

```
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;
}
```

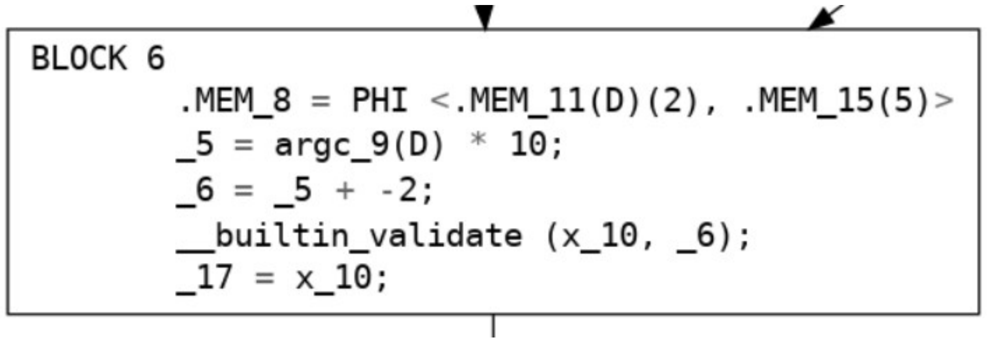
Novel Glitch Detection Approach: Expression Validations

Identifying Locations for Validations (2/2)

- **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 preceeding 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**



```
BLOCK 2
          _1 = argc_9(D) * 10;
          x_10 = _1 + -2;
          if (argc_9(D) > 1)
```



```
BLOCK 6
          .MEM_8 = PHI <.MEM_11(D)(2), .MEM_15(5)>
          _5 = argc_9(D) * 10;
          _6 = _5 + -2;
          __builtin_validate (x_10, _6);
          _17 = x_10;
```

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

- J. Balasch, B. Gierlichs, and I. Verbauwhede, ”An In-depth and Black-box Characterizationof the Effects of Clock Glitches on 8-bit MCUs”, 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, 2011, pp. 105-114, 2011.
- B. Yuce et. al, ”Software Fault Resistance is Futile: Effective Single-Glitch Attacks”, 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 47-58, 2016.
- J. Proy, K. Heydemann, A. Berzati, and A. Cohen, ”Compiler-Assisted Loop Hardening Against Fault Attacks”, ACM Trans. Archit. Code Optim. 14, 4, pp. 1-25, 2017.