Software Based Glitching Detection

Jakob Löw
Technische Hochschule Ingolstadt
Ingolstadt, Germany
jakob@löw.com

Dominik Bayerl
Technische Hochschule Ingolstadt
Ingolstadt, Germany
dominik.bayerl@carissma.eu

Prof. Dr. Hans-Joachim Hof
Technische Hochschule Ingolstadt
Ingolstadt, Germany
hof@thi.de

***Abstract*—Clock glitching is an attack surface of many microprocessors. While fault resistant processors exist, they usually come with a higher price tag resulting in their cheaper alternatives being used for small embedded devices. After describing the effects of fault attacks and their application to modern microprocessors, this paper presents a novel software based approach at protecting programs from fault attacks. Even though the protection mechanism is automatically added to a given program in a special compiler step, its use case is not to protect the full program. The approach comes with heavy performance implications, making it only useful for protecting important parts of programs, such as initialization, key exchanges or other cryptographic implementations.**

***Index Terms*—computer security, clocks, microcontrollers, program compilers, program control structures**

I. Introduction

Hardening software against glitching attacks manually is a tedious task and requires a trained developer. Hardware based glitch detection on the other hand increases cost of production. Thus the most efficient approach in order to protect against glitch attacks is with generalized and automated software mechanisms. The goal of this paper is to introduce a novel software based approach protecting a program from clock glitching attacks.
In order to introduce this approach, first, the nature and effects of glitching attacks in general and clock glitching attacks in particular are described in Section II. Section III discusses state of the art software based protection mechanisms. Then a novel approach detecting glitch attacks is introduced in Section IV. Finally in Section IV-D the performance impact of the novel approach is rated given its impact on common compiler optimizations.

II. Glitching Attack Models

In embedded IT Security, glitching attacks are a special kind of side channel attacks. Their target is to trigger misbehaviours of the target processor in order to alter execution or data flow. A typical goal of a glitch attack is changing the execution flow such that one instruction is skipped. For example, when glitching the conditional branch instruction of a signature check, the check is skipped and the program continues even if the signatures did not match. Triggering a glitch while the processor is loading a value from memory can cause the memory load to not finish correctly and often results in a zero value being loaded instead. Thus, glitching the data flow is often used to attack cryptographic algorithms by glitching the load of keys from memory or by glitching arithmetic operations [1].
The next Subsection will first describe clock glitching attacks, which this paper focuses on, in detail. Afterwards Section II-B will cover the exact effects of clock glitches targeting AVR Microprocessors.

A. Clock Glitching

Clock glitching is a specific form of glitching attacks. A glitch in the target processor is triggered by altering the provided clock signal. Normally a clock signal is generated by an oscillator with a constant frequency; Rising that frequency is called overclocking. Each processor has a maximum operating frequency, if the clock frequency rises above this threshold the processor starts to behave abnormally.
In a classical clock glitching attack, only a single targeted glitch is inserted into the clock signal, i.e., a second high signal is inserted causing the current instruction to not complete before the next one starts its execution. The effects depend on various parameters as well as on the processors architecture and design.
Figure 1 shows the electrical potential of a clock line during a clock glitch attack. The first Section, labeled as cycle A, shows a regular clock cycle, while cycle B shows a clock cycle with a glitch inserted [4].

B. Effects of Clock Glitches on AVR Microprocessors

The research by Balasch et. al [4] goes into detail about what exactly happens when a microprocessor is attacked by a glitching attack. They used a Field Programmable Gate Array (FPGA) to generate a clock signal for ATMega163 based smart cards. The FPGA allows clock signal modifications, such as inserting a glitch at a specific location. The ATMega runs a special firmware, which places all registers in a known state, executes the instruction targeted by the glitch and then examines the state of all registers of the microprocessors. From the transformations between the start state and the result state the executed instruction can be derived. This, however, is a non trivial task. For example when before the instruction the value `0x0f` was in register `r18` which changed to `0xf0` afterwards the executed instruction could either be a 4-bit left shift or an addition with `0x51`. Multiple runs with the same glitch period, the same instruction but different input states have to be performed in order to be able to identify the actual executed instruction.

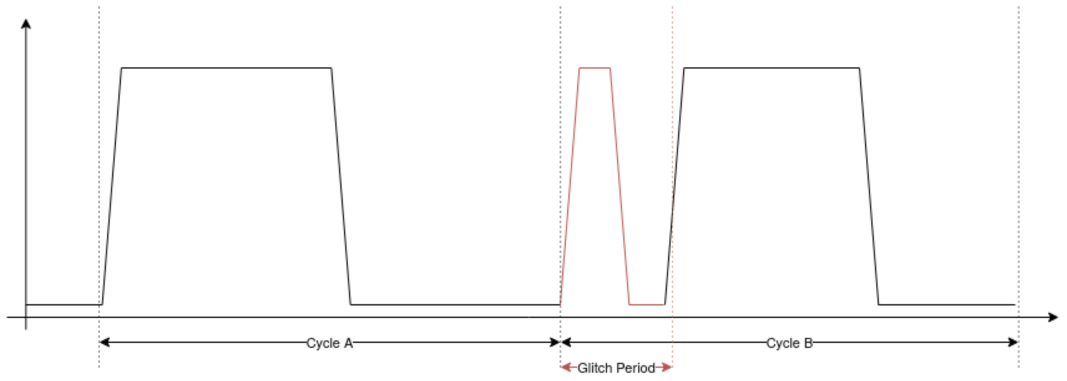


Fig. 1: Injection of a Clock Glitch

With these methods [4] shows the actual effect of clock glitches with different glitch periods on a target instruction. During instruction fetching the value of the instruction to execute next changes from the previous instruction to zero and then to the value of the following instruction. By injecting a glitch into this transition, depending on the length of the glitch period, either a decayed version of the previous instruction or a decayed, i.e. not yet fully loaded, version of the current instruction can be executed. Figure 2 shows this behaviour for a *Set all Bits in Register* (`SER`(instruction followed by a *Branch if Equal* (`BREQ`) instruction. In this specific case, for a glitch period up to 28 ns a decayed version of the `BREQ` instruction is executed. From 32ns and upwards an intermediate value of the transition from zero to `SER` is executed [4].

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`

Fig. 2: Instruction decay based on glitch period

III. Existing Software based Glitch Detection Techniques

With one of the first papers covering fault based attacks on cryptographic implementations dating back to 1997 [1], there are already multiple papers covering protection mechanisms against fault attacks using software or hardware based countermeasures. The software based countermeasures are usually based on either duplicating instructions or validating computations. The following sections describe some of the common approaches at glitch detection by example, before a novel approach is discussed in Section IV.

A. Instruction duplication mechanisms

A very common approach at protecting code from glitch attacks is instruction duplication or even triplication. It is usually implemented at a very late stage in the compilation process and works by simply duplicating memory load or even arithmetic instructions and checking their results for equality. A simple ARM64 assembly example is shown in Figure 3. Instead of only loading the value at `x0` once into register `w0` it is loaded a second time into `w1`. If a glitch occured in one of the two instructions, i.e. a wrong value was read from memory, the comparasion check fails and an error handler is called.

```
ldr     w1, [x0]
ldr     w0, [x0]
cmp     w1, w0
bne     glitch_error
```

Fig. 3: Validation using instruction duplication

While this approach is simple to implement it is flawed, especially when using modern microcontrollers with multi stage pipelines. As shown by Yuce et. al in [6] injecting a single glitch can affect multiple instructions. This is possible, because the two load instructions are not executed one after another, but rather go simultaneously through various stages in the processor pipeline.
In general placing the validation of an instruction too close to the instruction itself renders the validation vulnerable to single glitch attacks.

B. Loop count validation

In [8], Proy et. al describe an automated compiler based glitch detection mechanism. Instead of validating arbitrary expressions as shown later in this paper, the approach from [8] focuses on validating loop exit conditions and iteration counts. The goal is to prevent attacks which weaken the

security of cryptographic algorithms by reducing the number of encryption rounds.
A special compilation pass is added to LLVM, a very common compiler infrastructure. When encountering a loop with a iteration variable this optimization pass add a a second iteration variable which gets incremented or decremented the same as the original variable and thus allows to validate the loop exit condition after the loop exited. For example, the loop shown in 4a is modified to include a second variable and a condition check turning it into code for the loop shown in 4b.

```
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 4a with validation

Fig. 4: Basic loop validation example

This optimization works best for loops with simple iteration calculation, i.e. adding or subtracting a constant from the iteration variable each iteration. Loops which contain `break` statements or which use a complex iteration modification however increase complexity of correct validations. The code listings in Figure 5 demonstrate these special loop forms.
A glitch attack on the calculation of `x` in Figure 5b would affect not only the iteration variable, but also a possible validation variable. Thus for glitch robustness not only the iteration variable needs to be duplicated and recalculated, but also all variables used to modify it. In [8] this is achieved by tracing through the expressions used to modify the iteration variable and recalculating all these expressions.
The following section describes a similar, but broader approach, which not only validates loop conditions but rather all expressions calculated in a function.

```
int i = 0;
while(i < 10) {
        // ...
        int x = // ...
        if(x == 42)
                break;
        i++;
}
```

(a)

```
int i = 10;
while(i > 0) {
        // ...
        int x = // ...
        i -= x;
}
```

(b)

Fig. 5: Advanced loop validation examples

IV. Detecting Glitches using Expression Validations

Traditionally, glitch detection techniques use instruction duplication or even triplication. While this works for some architectures, as described in Subsection III-A, a duplicate instruction is still vulnerable to a single fault on processors featuring a multi stage pipeline. Thus in order to increase the robustness of glitching detection mechanism the validation has to be placed as far away from the original computation as possible. In compiler engineering functions a divided into multiple blocks through which execution flows linearly. Moving validations out of the basic block of the original computation, means the number of instruction executed between computation and validation can vary between just a few computations to multiple calls to other functions. Placing validations farther away from their original computations makes it harder for an attacker to glitch both computation and validation.
The following sections describe how to find the optimal locations for validations and how to validate both computations and conditional branches.

A. Identifying Locations for Validations

As described in Subsection III-A glitch detection mechanisms are still vulnerable to a single glitch fault when the duplicated instruction, in our case the second computation, is placed close to the original instruction. Placing the validation as far away from the original computation as possible ensures its robustness against single fault attacks.
The last possible location for a validation check is usually the end of the scope a value is defined in. For a value defined in a conditional or loop body this results in the check being placed at the end of the conditional or loop respectively. For a value defined in a function the last possible check is right before the function returns. Figure 6 shows an example with these two cases.

```
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;
}
```

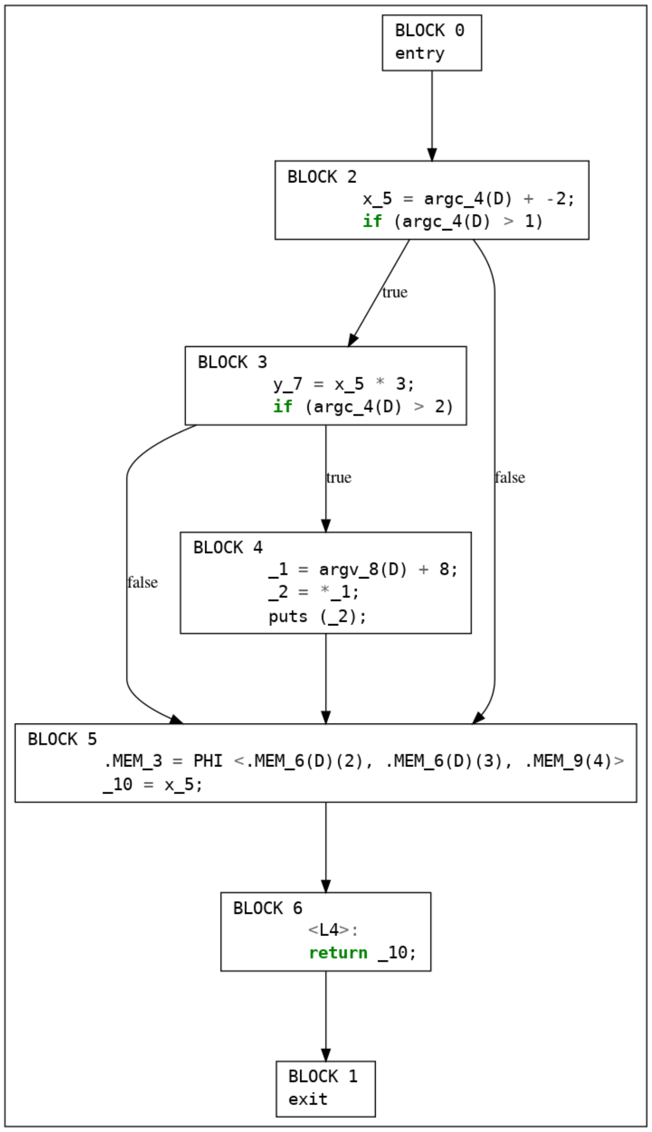
Fig. 6: Example Code

While it is trivial to find the optimal location for immutable variables in program code, a mutable variable might be changed between its first initialization and the end of the scope. In order to correctly validate all values of a mutable variable the location has to be determined during a later stage in the compilation process. The Static Single Assignment (SSA) form is a very common form of representing a program in compilers. In SSA form each variable is immutable and only assigned once, variables which are originally mutable and set multiple times are split up into seperate variables for each assignment. Additionally a function in SSA form is usually represented as basic blocks rather than loops and branches. Figure 7 shows how *gcc* represents the code listed in Figure 6 internally after SSA creation.

The validation of x, labeled x_5 in Figure 7, can be placed in block 5 (B_5). But there does not exist a block for the optimal location to validate y_7. It cannot be placed in B_5, as that block is also reachable from B_2 where y_7 does not exist. Thus a new block has to be created, with B_3 and B_4 as predecessors and B_5 as successor. The edges $B_3 \to B_5$ and $B_4 \to B_5$ have to be removed. The validation of y_7 can then be placed inside the newly created block.

In general a variable x created in block B_x can only be validated in B_x itself or in a block B_i where all predecessors $prec(B_i)$ are direct or indirect successors of B_x. The optimal location for the validation is by definition the block that is the farthest away from B_x while still meeting the required condition.

Figure 8 shows the SSA block graph of Figure 6 with validations. Block B_5 is the newly inserted block and B_6 the former block 5.


```
BLOCK 0
entry

BLOCK 2
        x_5 = argc_4(D) + -2;
        if (argc_4(D) > 1)

BLOCK 3
        y_7 = x_5 * 3;
        if (argc_4(D) > 2)

BLOCK 4
        _1 = argv_8(D) + 8;
        _2 = *_1;
        puts (_2);

BLOCK 5
        .MEM_3 = PHI <.MEM_6(D)(2), .MEM_6(D)(3), .MEM_9(4)>
        _10 = x_5;

BLOCK 6
        <L4>:
        return _10;

BLOCK 1
exit
```

Fig. 7: Basic Block graph in SSA form of 6 without validations

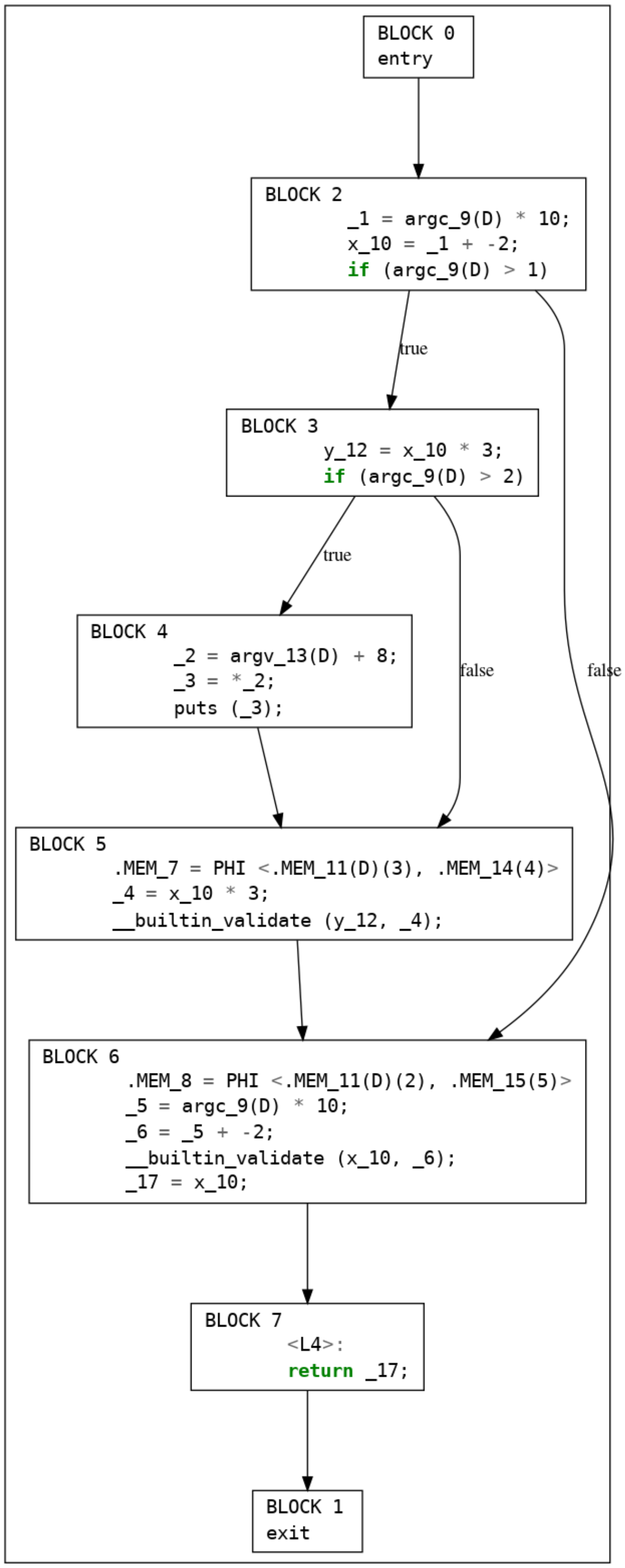


Fig. 8: Basic Block graph in SSA form of 6 with validations

B. Validating Calculations

Without deeper knowledge of the implemented algorithm validating calculations often boils down to simply recomputing all values and thus duplicating the entire calculation.
For example, the statement `int x = argc * 10 - 2;` from Figure 6, results in the SSA shown in the following listing:

```
_1 = argc_9(D) * 10;
x_10 = _1 + -2;
```

For a full validation both the SSA values `_1` and `x_10` have to be recalculated and validated:

```
_5 = argc_9(D) * 10;
__builtin_validate (_1, _5);
_6 = _5 + -2;
__builtin_validate (x_10, _6);
```

A simpler approach is to only validate the outermost result of one or more chained calculations. For the above example this is achieved simply by removing the first instance of `__builtin_validate` resulting in the code shown in 8. For larger entangled calculations removing redunant validations allows to greatly reduce the amount of validations required. For instance all variables in the following C code can be validated using a single validation of `z` instead of having to validate all variables or even all intermediate SSA values one by one.

```
int x = a * 10 + 3;
int y = x / 7;
int z = x * y * 13;
```

The `__builtin_validate` function acts similar to an assert equals function, it continues with execution if the two values are identical and cancels execution otherwise. In a production environment the function can be inlined producing an inequality check and a conditional jump to an error function, resulting in code similar to what gcc produces for calls to `assert`. Figure 9 shows the validation of `x` from Figure 6.

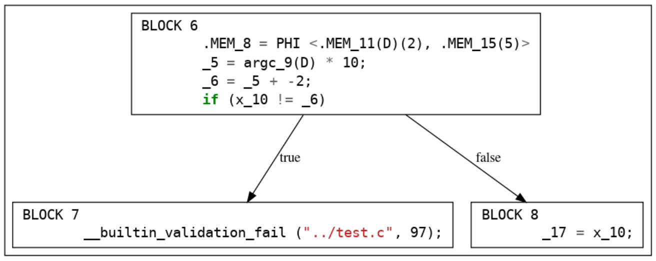


Fig. 9: Validation in production

C. Validating Comparasions and Conditional Jumps

In *gcc* the condition of a branch can not only be a single SSA value, but also a comparasion operation. An example is the `if (argc_4(D) > 1)` statement at the end of block 2 in Figure 7. This is because in most processor architectures a comparasion of two values used for a conditional jump is done without storing the result in a common register, i.e. the comparasion result is only stored in a flags register which is then immediately used by the following conditional jump instruction.

As there exists no SSA name for the result of such comparasions in *gcc* it cannot be validated as described in Subsection IV-B. A block with a conditional branch at the end always has two successors, one for when the condition is true, one for when its false. Therefore in order to validate the condition, two validations, one for each successor have to be created. Each validation follows the same rules as described in Subsection IV-A with their initial blocks being the targets of the conditional edges.

In general, for a block B_i with multiple successors, the branching condition can be validated using one validation placed as if a value j has been created in B_j for all edges $B_i \rightarrow B_j$.

If one of the successors B_j is also a direct or indirect successor of any of the other successors of B_i a new block between B_i and B_j has to be inserted. This is usually the case for loops and if statements without an else block. For example, in Figure 7 the validation for the condition of B_2 being false cannot be placed in B_5, as B_5 is also a successor of B_3.

D. Performance Considerations of Expression Validations

Simple instruction duplication mechanisms as described in III-A duplicate the runtime of the protected instructions. This holds true for simple microprocessors where each instruction takes a fixed amount of clock cycles. For advanced processors which incorperate memory caching a second load of a specific address will result in a cache hit, which is usually faster than a load from memory.

The novel glitch detection approach described in Section IV also duplicates instructions and thus has similar effect during runtime. The bigger impact, howver, is its prevention of possible compiler optimizations resulting in the generation of less performant instructions. Normally a compiler analyzes the lifetime of variables and the collisions between those lifetimes. The lifetime of a variable starts when the variable is first set and ends with its last usage. Two lifetimes collide when they are both alive at any given point in the function. When two lifetimes do not collide they can be placed in the same processor register. With too many lifetime collisions the compiler might run out of registers to assign and has to place variables in memory instead [5]. By definition, the optimal location for validation, as given in Subsection IV-A, extends the lifetime of variables to the maximum possible. Thus, with the novel detection approach, the register allocator of the compiler will have to place variables in memory more often, resulting in more memory accesses and decreased performance.

For example the SSA variable `y_12` of Figure 8 would normally live only for a short time in B_3. Its validation in B_5 extends its lifetime, making it collide with the SSA variables `_2`, `_3` and `_4`.

In order to decrease the performance impact expression validation can only be enabled for security relevant functions such as cryptographic implementations or credential checks by disabling validations for all functions and adding a special compiler attribute to relevant ones.

V. Conclusion

After giving an introduction to glitching attacks and clock glitches in particular, we discussed various software based approaches at hardening against glitching attacks. While the common protection mechanism discussed in Subsection III-A can easily be applied to a program via an additional compilation pass, it is also shown to be ineffective [6]. The protection mechanism discussed in Subsection III-B by Proy et. al [8] can easily be applied to existing codebases, but only validates loop conditions and loop iterators.

The novel approach described in Section IV tries to combine the best traits of the three described previous mechanisms. It is similar to the mechanism by Proy et. al [8] as it also comes in the form of a compiler pass and it also adds validations of existing computations to the program. However, it not only validates loop conditions, but rather generalizes validation of arbitrary computations and branch conditions. This allows it to also protect the program from glitch attacks targeting value computations or substitutions, instead of only protecting against attacks aimed at modifying loop execution counts.

As discussed in Subsection IV-D this novel approach comes with a big performance impact, doubling the execution time in the best case scenario, but usually having an even worse impact. Thus the approach is best applied only selectively to specific parts of a program, keeping performance impact low while still providing protection to curcial code parts.

References

[1] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, ”On the Importance of Checking Cryptographic Protocols for Faults”, Advances in Cryptology — EUROCRYPT ’97, pp. 37-51, 1997.

[2] D.I. Crecraft and S. Gergely ”Analog Electronics - Circuits, Systems and Signal Processing” Elsevier Science, 2002.

[3] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, ”Fault Attacks on RSA with CRT: ConcreteResults and Practical Countermeasures”, Cryptographic Hardware and Embedded Systems - CHES 2002 pp. 260-275, 2002.

[4] 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.

[5] Keith D. Cooper and Linda Torczon, ”Engineering a Compiler”, 2nd edition, Elsevier Science, 2012.

[6] 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.

[7] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, ”Fault Tolerant Infective Countermeasure for AES”, J Hardw Syst Secur 1, pp. 3-17, 2017.

[8] 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.

[9] B. Selmke, F. Hauschild, and J. Obermaier, ”Fault Injection into PLL-Based Systems via Clock Manipulation”, Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, pp. 85-94, 2019.