Implementation of a Software Based Glitching Detection Mechanism

Jakob Löw
Technische Hochschule Ingolstadt
Ingolstadt, Germany
jakob.loew@thi.de

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

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 the concept and implementation of a novel software based approach at protecting programs from fault attacks. Even though the protection mechanism can be automatically added to a given program in a special compiler step, its use case is not to protect the full program. As shown by the performance analysis in this paper, 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 and rate a 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 an approach at detecting glitch attacks is introduced in Section IV. This paper is the extended version of Löw et al. [1], it introduces an implementation of the glitch detection approach in Section V. The performance impact of the approach is then rated using this implementation in Section VI.

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 [3].
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 microprocessors using Atmels AVR microarchitecture.

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 [6].

B. Effects of Clock Glitches on AVR Microprocessors

The research by Balasch et al. [6] 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

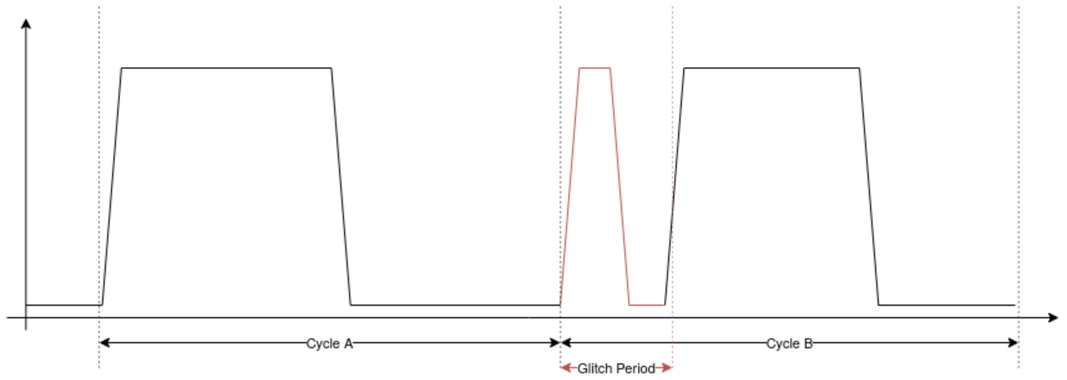


Fig. 1: Injection of a Clock Glitch

`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.

With these methods Balasch et al. [6] show 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 [6].

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

C. *Glock glitching with modern Microprocessors*

As microcontrollers such as the AVR tiny and mega series usually clock between 1 and 8 MHz they tend to have a direct clock input line, where a oscillator with the desired clock frequency has to be attached.

On higher clocking microcontrollers such as the *ARM Cortex M0* series operating on a clock frequency of usually around 50MHz to 100MHz a phase locked loop (PLL) is used to generate the clock signal [11]. A PLL uses an input signal of an oscillator and is able to multiply that signal allowing it to generate higher frequencies [4]. This is achieved by dividing the output of a voltage controlled oscillator (VCO) and synchronizing this divided voltage with an input voltage obtained from a quartz oscillator. This way the input frequency is effectively multiplied achieving a higher frequency.

A PLL output does not immediately respond to changes in the input frequency, which means a classical clock glitch has little to no effect on the actual clock frequency of the microprocessor. Increasing the input frequency for a longer period of time, does however make the PLL output frequency rise. This effect is used by B. Selmke et al. in [11] to induce glitches into an *ARM Cortex M0* processor even though it utilizes a PLL based clock multiplier. Figure 3 shows the principle of this approach: By increasingt he input frequency for a specific duration the processor frequency generated by the PLL increases. While the processor might be able to be slightly overclocked over the nominal frequency, at a specific point the frequency becomes too high for the processor to work correctly, at which point the processor shows faulty behaviour. The frequency required for entering the CPU fault zone depends on the chip kind, quality and even on the instructions executed. This makes glitching processors with PLLs hard, but [11] proves it is still achievable by attacking an AES implementation running on an *ARM Cortex M0*.

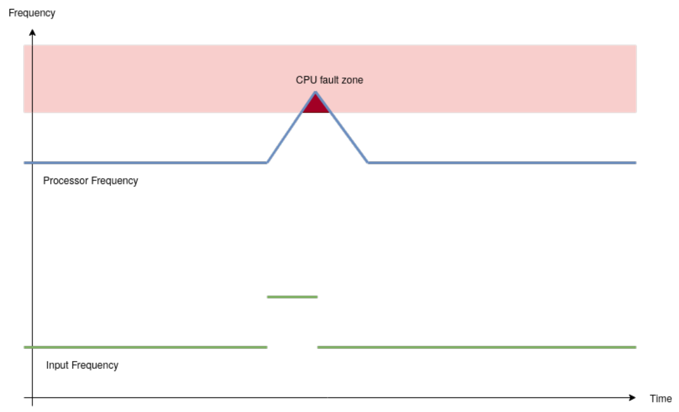


Fig. 3: Glitching a processor with a PLL

III. Existing Software based Glitch Detection Techniques

Papers covering fault based attacks on cryptographic implementations date back to 1997, and 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 4. 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 comparison check fails and an error handler is called.

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

Fig. 4: 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 [8] 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 [10], 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 [10] 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 5a is modified to include a second variable and a condition check turning it into code for the loop shown in 5b.

```
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 5a with validation

Fig. 5: 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 6 demonstrate these special loop forms.
A glitch attack on the calculation of `x` in Figure 6b 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 [10] 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. 6: Advanced loop validation examples

C. *Mathematical validation of computed values*

The paper [5] by Aumüller et al. focuses on protecting the Rivest-Shamir-Adleman (RSA) cryptosystem. Instead of classical instruction duplication or expression validation techniques as described in Sections III-A and III-B it makes modifications and validations on a higher level. Rather than looking at algorithm implementations or even machine code, the computation is validated by modifying the RSA CRT algorithm, a common optimization to RSA using the Chinese Reminder Theorem (CRT).
Given a message m, two primes p and q, and the exponent d, the regular RSA CRT algorithm consists of the following steps to calculate the signature S of a message m:

$$d_p = d \mod (p-1) \tag{1}$$

$$d_q = d \mod (q-1) \tag{2}$$

$$S_p = m^{d_p} \mod p \tag{3}$$

$$S_q = m^{d_q} \mod q \tag{4}$$

$$S = S_q + q \cdot ((S_p - S_q) \cdot q^{-1} \mod p) \tag{5}$$

Aumüller et al. [5] extend the calculation of S_p and S_q shown in (3) and (4) with a third, smaller prime t and two random numbers r_1 and r_2:

$$p' = p \cdot t \tag{6}$$

$$d'_p = d_p + r_1 \cdot (p-1) \tag{7}$$

$$S'_p = m^{d'_p} \mod p' \tag{8}$$

$$q' = q \cdot t \tag{9}$$

$$d'_q = d_q + r_2 \cdot (q-1) \tag{10}$$

$$S'_q = m^{d'_q} \mod q' \tag{11}$$

The two values S'_p and S'_q can then be used to calculate S_p and S_q and thus S using Equation (5).

$$S_p = S'_p \mod p \tag{12}$$

$$S_q = S'_q \mod p \tag{13}$$

This modified algorithm allows validation of the result using the conditions shown in equations (14) to (19). [5]:

$$0 \equiv p' \mod p \tag{14}$$

$$0 \equiv q' \mod q \tag{15}$$

$$d_p \equiv d'_p \mod (p-1) \tag{16}$$

$$d_q \equiv d'_q \mod (q-1) \tag{17}$$

$$0 \equiv S - S'_p \mod p \tag{18}$$

$$0 \equiv S - S'_q \mod q \tag{19}$$

While this approach allows to implement a glitch hardened RSA implementation it only applies to RSA. It is not a general approach at hardening algorithms against glitch attacks. While its principles could be applied to other algorithms each requires manual work by a developer, rather than e.g., automatically applying protection using a compiler pass as shown in the following section.

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 are 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, based on the short version of this paper [1], 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 7 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. 7: 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 8 shows how *gcc* represents the code listed in Figure 7 internally after SSA creation.
The validation of x, labeled x_5 in Figure 8, 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 $B_3 \rightarrow B_5$ and $B_4 \rightarrow B_5$ 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 9 shows the SSA block graph of Figure 7 with validations. Block B_5 is the newly inserted block and B_6 the former block 5.

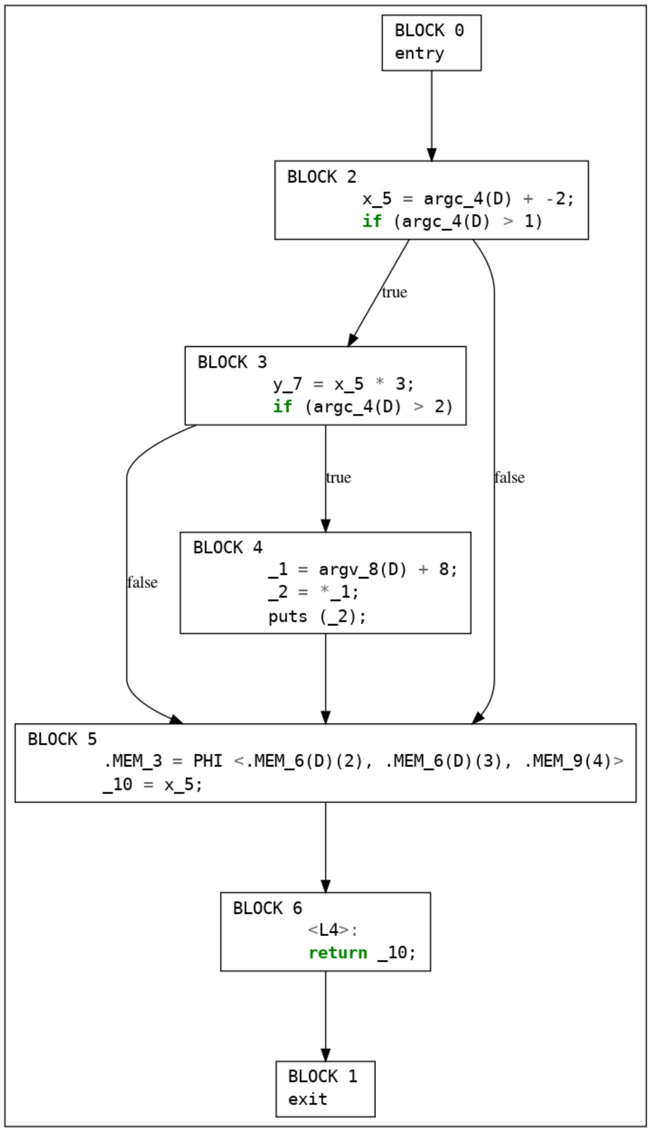
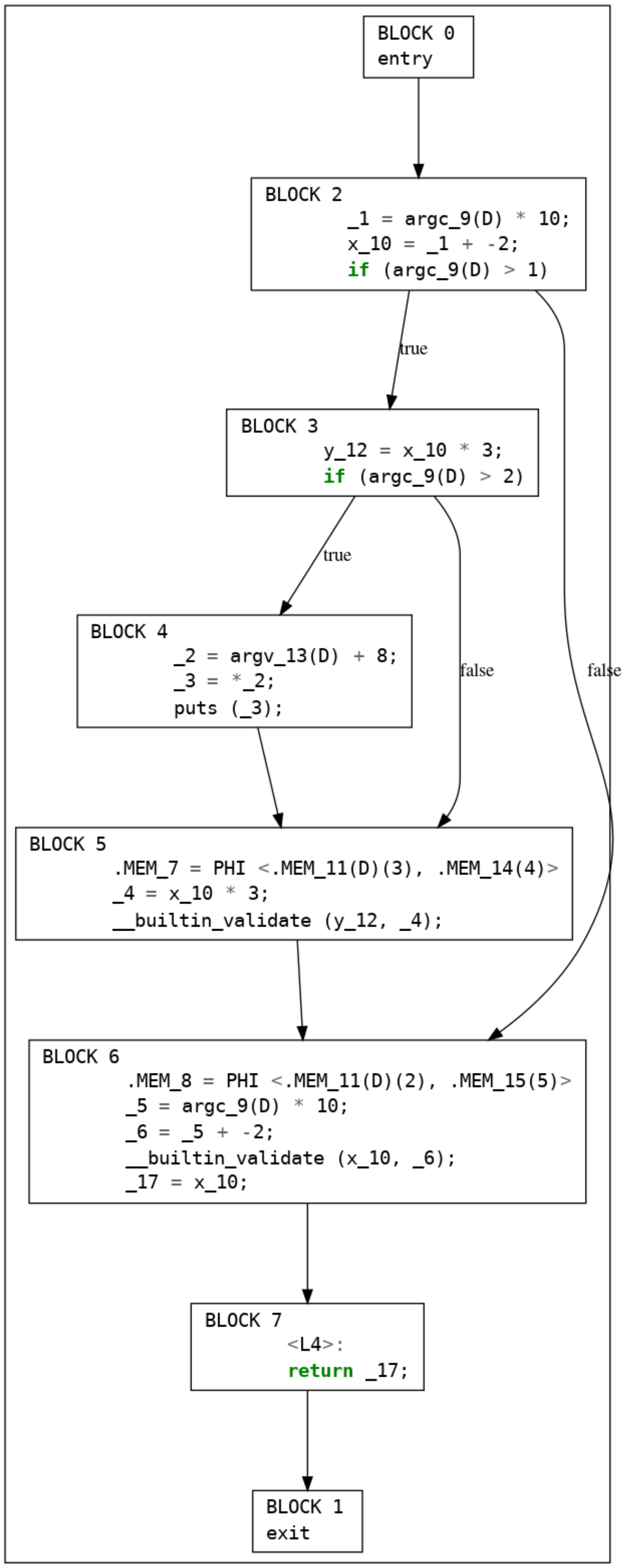

```
BLOCK 0
entry

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



```
BLOCK 0
entry

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

BLOCK 3
        y_12 = x_10 * 3;
        if (argc_9(D) > 2)

BLOCK 4
        _2 = argv_13(D) + 8;
        _3 = *_2;
        puts (_3);

BLOCK 5
        .MEM_7 = PHI <.MEM_11(D)(3), .MEM_14(4)>
        _4 = x_10 * 3;
        __builtin_validate (y_12, _4);

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;

BLOCK 7
        <L4>:
        return _17;

BLOCK 1
exit
```

Fig. 9: Basic Block graph in SSA form of 7 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 7, results in the SSA shown in the following listing:

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

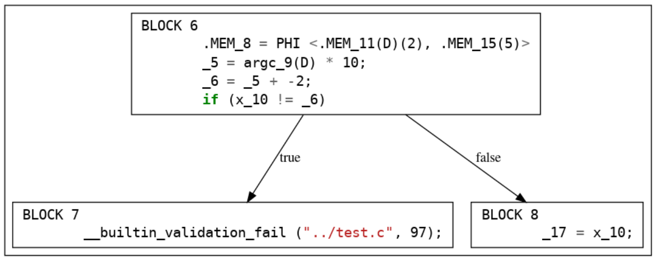
For a full validation both the SSA values `_1` and `x_10` have to be recalculated and validated:

```
_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 9. 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 10 shows the validation of `x` from Figure 7.



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

BLOCK 7
        __builtin_validation_fail ("../test.c", 97);

BLOCK 8
        _17 = x_10;
```

Fig. 10: Validation in production

C. Validating Comparisons and Conditional Jumps

In *gcc* the condition of a branch can not only be a single SSA value, but also a comparison operation. An example is the `if (argc_4(D) > 1)` statement at the end of block 2 in Figure 8. This is because in most processor architectures a comparison of two values used for a conditional jump is done without storing the result in a common register, i.e., the comparison 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 comparisons 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 $B_i \to B_j$.

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 8 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 [7]. 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 9 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. Proof of Concept Implementation of Expression Validations

In order to test the feasability and actual performance impact of expression validations, the validations as described in Section IV were implemented into an existing compiler backend. The compilation library *GNU libjit* was picked as a framework for the implementation, as its internal immediate representation (IR) is simple and it already includes the liveness algorithms described in [7]. The liveness information will also be used in the following sections in order to implement the validation placement as described in Section IV-A.

A. libjit Immediate Representation

A compilation unit in libjit is a function. Each function consists of one or more basic blocks. Each basic block contains instructions, which are executed linearly. Having a call, jump or conditional jumps ends the current basic block and starts a new one. This way execution always flows linearly through a basic block. A block a, which may jump to another block b, is called bs *predecessor* and a is called a *successor* of b. Each block can have multiple *successors* as well as multiple *predecessors*. The *control flow graph (CFG)* of a function is a representation of the functions control flow using a directed graph G with a node v_i for each basic block i and edges v_{ij} for each successor v_j of v_i.

An instruction in libjits immediate representation consists of an operator `op` and up to three values. These values are named `dest`, `value1` and `value2`. Thus the usual form of an instruction is `dest = value1 op value2`. Complex mathematical expressions have to be broken down to a series of such instructions with the use of temporary values. For example the expression `y = 2 * x + 7` in libjit IR uses a temporary value `i0` as shown in the following listing:

```
i0 = 2 * x
y  = i0 + 7
```

There however exist many instructions where neither `dest` nor `value2` exist or just one of them. One example for such instructions are the conversion instructions, such as converting a 64-bit integer to a 64-bit floating point value denoted as `dest = long_to_float64(value1)`.

Each instruction includes flags what kind of operation is performed and which of the three values are used. For the purpose of expression validations only arithmetic and comparison instructions are relevant. These instructions can simply be cloned and given a new `dest` value, which is then compared to the original `dest`.

Libjit does not represent instructions in SSA form. This means a value used as a destination for an instruction might be used as a destination for other instructions too and thus its value might change. This makes determining locations for expression validations harder, as a cloned instruction might depend on a value changed between the original computation and the duplicated one.

B. Liveness Flags in libjit

Libjit implements liveness and dataflow analysis as described by Cooper and Torczon in [7]. Each basic block is fitted with flags about *killed*, *upwards exposed* and *living out* values. A value k is *killed* in block v_i if its value is changed inside v_i. A value is *upwards exposed* when it is used without being or before being *killed*. A value k *lives out* a block v_i when one of v_i successors has k *upwards exposed*.
In libjit these three attributes are stored in bitsets for each block, with one bit for each value that exists in the function. As libjit expressions are not in SSA form, for expression validation mainly the *kill* set of a block is relevant, that is when trying to validate an instruction in the form of `c = a + b` the validation cannot be placed after a modification of `a`, `b` or `c`.

C. Basic Block Domination

In Graph theory a node v_i in control flow graph G is said to be dominated by another node v_j when all possible paths from the entry point to v_i go through v_d. Thus in program execution when v_i is executed the instructions in v_d have been executed at least once.
Block domination is easily described in a recursive manner: A block v_j is dominated by itself and the intersection of all dominators of its preceeding nodes v_i:

$$Dom(v_j) = \{v_j\} \cup (\cap_{v_i \in preds(v_j)} Dom(v_i)) \qquad (20)$$

This recursive definition can be implemented using a fix point algorithm as shown in Figure 11. This direct implementation has a complexity of $\mathcal{O}(n^2)$ with n being the number of blocks in the control flow graph. In 1979 Lengauer et Tarjan described a faster algorithm for finding block dominators with a runtime of $\mathcal{O}(m \cdot log(n))$ with m being the number of edges and n the number of nodes [2].
With m and n staying below 1000 in all test cases and because the latter algorithm comes with a higher implementation complexity, the direct implementation, shown in Figure 11, was chosen for libjit.

```
dominators[$v_0$] = {};

foreach $v_i$ in $G$:
  dominators[$v_i$] = {$v_0$, $v_1$, ..., $v_n$};

while dominators changed:
  foreach $v_j$ in $G$:
    dominators[$v_j$] = \
      $\{v_j\} \cup (\cap_{v_i \in \mathrm{preds}(v_j)}$dominators[$v_i$]);
```

Fig. 11: CFG Dominator Algorithm

D. Finding Expression Validation Locations in libjit

Section IV-A described the optimal validation location, based on program code with the concept of scopes. In libjit IR the concept of scopes does not exist directly, as it operates on a lower level than regular program code. In program code a scope ends when the execution flow of multiple possible branches join without all branches being created within the given scope. In the example code in Figure 7 the scope where variable `y` was created ends with the end of the outer `if` statement. However the scope where variable `x` was created does not end at that location, as both the `if` and `else` branch were created within the scope of `x`.
When mapping the scope lifetime to basic blocks a scope of variable `k` is exactly alive in all blocks, which are dominated by the block creating `k`. This means a variable `k` can be validated in all blocks v_i where the creating block v_k of `k` is dominating v_i: $v_k \in Dom(v_i)$.
Additionally, as described in Section V-B, in libjit a block v_j is not a possible validation candidate when k is killed by one of its direct or indirect predecessors v_i, without v_i being the block with the original instruction that is to be validated. While applying these rules to all blocks gives us the list of *possible* validation locations what we are actually looking for is the *last possible* location for a validation. A block v_i is the last possible location for a validation of a value created in v_k when not all of its successors v_j are possible validation candidates for `k`, i.e., when not all successors are dominated by v_k.
When an instruction cannot be validated by any other block than the creating block itself the validation has to be placed at the end of the block. When the destination value or one of the operands is killed later inside the same block the validation has to be placed inside of the block, before the kill occurs.

E. Placing Expression Validations in libjit

In order to validate an instruction, first the original computation has to be recomputed and afterwards its result compared to the original computation result. In case of inequality a branch to a specific expression validation failure label is performed. Because of this conditional branch placing a validation actually starts a new basic block at the validation location. Adding new blocks invalidates the computed dominating block information as well as the control flow graph itself. Therefore, all expression validation locations are first computed and collected in a list before actually placing them into the libjit IR code.
Validations of instructions can either occur at the start or at the end of a basic block. Placing them at the start is easier, as a new basic block can simply be prepended before the first instruction. Placing validations at the end of a block requires moving the last instruction to another new block.

VI. Performance Impact of Expression Validations

As described in Section IV-D the expected runtime increase is at least 100%. This section measures the actual performance impact of the reference expression validation implementation in libjit. As libjit is simply a library used for compiling low level instructions to machine code a higher level language called *PointerScript* is used for implementing the test algorithms. *PointerScript* is a language with JavaScript like syntax, but direct access to C functions through its built-in foreign function interface. For the purpose of performance measurements several common algorithms such as array sorting, simple addition loops as well as special worst case scenarios were implemented and ran both with expression validations enabled and disabled.

A. Performance Test Setup and Programs

As libjit is a JIT compilation library the total program execution time normally also includes the compilation steps of the program code. For the purpose of this paper only the actual code runtime after compilation was measured. Even though for programs running multiple seconds compilation time only takes a small amount of the runtime, this way the additional compilation time caused by expression validation placement is ignored.
The POSIX gettimeofday function is called before and after calling the compiled program code and the difference is used for measuring the runtime of the programs. Each test program is run ten times and their runtimes are averaged. As runtime differs based on processor model and speed, for comparison not the actual runtime, but rather the runtime increase in percent is used as a metric.
The first test program is a simple loop calculating the sum of numbers between 1 and n. Its code is shown in Figure 12
The second test program partly listed in Figure 13 is a textbook implementation of the bubble sort sorting algorithm applied to an array of n random integers generated using the `rand` function.
The last code snippet shown in Figure 14 is an example crafted especially to visualize the performance overhead introduced by expression validations. It includes six variables all initialized with a random value and used in a tight loop with a high iteration count. Between the initialization of the variables and the actual loop is a call to an external function. Because of this function call all values created before the call have to be placed either in callee saved registers or in memory. Without expression validations there are exactly six values used after the call, which is identical to the amount of callee saved registers in the System V ABI on the x86-64 architecture. With expression validations enabled the register allocator has to spill some of the values, i.e., place them in memory instead. Figures 15 and 16 show this effect in the disassembly of Figure 14 with validations disabled and enabled respectively.

```
var sum = 0;
for(var i = 0; i < 10000000000; i++)
{
  sum += i;
}
```

Fig. 12: Non-Gauß Addition

```
for(var i = 0; i < len; i++)
{
  for(var j = 0; j < len - 1; j++)
  {
    if(parts[j] > parts[j + 1])
    {
      var tmp = parts[j + 1];
      parts[j + 1] = parts[j];
      parts[j] = tmp;
    }
  }
}
```

Fig. 13: Bubblesort

```
var x = rand();
var a = x + 1;
var b = x + 2;
var c = x + 3;
var d = x + 4;
var e = x + 5;

printf("");

for(var i = 0; i < 100000000; i++)
{
  a++; b++; c++; d++; e++;
}
```

Fig. 14: Register Allocation Spill Triggering Code

```
48 ff c3  inc %rbx
49 ff c4  inc %r12
49 ff c5  inc %r13
49 ff c6  inc %r14
49 ff c7  inc %r15
```

Fig. 15: Disassembly of the loop body from Figure 14 with validations disabled

```
4c 8b 4d f8     mov -0x8(%rbp),%r9
4d 8d 51 01     lea 0x1(%r9),%r10
49 8b c2        mov %r10,%rax
48 89 45 f8     mov %rax,-0x8(%rbp)
4c 8b 5d f0     mov -0x10(%rbp),%r11
4d 8d 73 01     lea 0x1(%r11),%r14
49 8b c6        mov %r14,%rax
48 89 45 f0     mov %rax,-0x10(%rbp)
```

Fig. 16: Disassembly of just two of the increments from Figure 14 with validations enabled

B. Performance Impact of Expression Validations

Figure 17 shows the relative runtimes of the three code samples. Each runtime is normalized to the runtime of the program with glitch detection disabled.
As expected the minimum runtime increase is around 100%, i.e., the runtime is doubled compared to running the same code without expression validations. This is especially true for the code samples listed in figures 12 and 13, where the only impact of expression validations is the added second computation of expressions within the loop.
For the code sample listed in Figure 14 however the expression validations do not only duplicate computations, but also have an impact on the register allocator and thus result in a significantly worse performance decrease. Running the program with expression validations enabled results in an approximatily 4.5 times as long execution time. The forth pair of measurements in Figure 17 describes the runtime increase of 14 with libjits graph coloring register allocator disabled. The execution time in this case is given relativly to the code running with the advanced allocator enabled. The overhead factor is reduced to around 3, which is mainly caused by the non-validated program also storing and loading values from memory and becoming nearly two times slower, while the runtime with expression validations only increases from 4.5 to 5.5 times slower than the runtime with advanced register allocation and without expression validations.

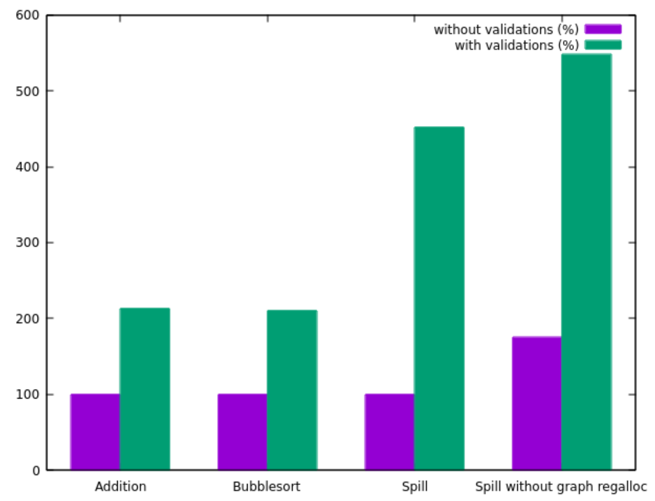


Fig. 17: Relative runtimes of 12, 13, 14 and 14 using only a basic register allocator

VII. 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 [8]. The protection mechanism discussed in Subsection III-B by Proy et al. [10] can easily be applied to existing codebases, but only validates loop conditions and loop iterators.
The 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. [10] 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.
Section V discusses the steps of implementing the glitch detection technique, first described by Löw et al. in [1], into existing compiler architectures and gives details about the implementation into the GNU libjit compiler backend. Section VI shows the performance impact of this proof of concept implementation based on three representative code samples. It shows the assumptions about runtime usually doubles, as assumed in Section IV-D, but also shows the performance impact can be way worse in specific scenarios. 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] J. Löw, D. Bayerl, and H.J. Hof, ”Software Based Glitching Detection”, The Fifteenth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE), Athen, pp. 41-46, 2022.
[2] T. Lengauer and R. E. Tarjan, ”A fast algorithm for finding dominators in a flowgraph”, ACM Transactions on Programming Languages and Systems Volume 1 Issue 1, pp. 121-141, 1979.
[3] 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.
[4] D.I. Crecraft and S. Gergely ”Analog Electronics - Circuits, Systems and Signal Processing” Elsevier Science, 2002.
[5] 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.
[6] J. Balasch, B. Gierlichs, and I. Verbauwhede, ”An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs”, 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, 2011, pp. 105-114, 2011.
[7] K. D. Cooper and L. Torczon, ”Engineering a Compiler”, 2nd edition, Elsevier Science, 2012.
[8] 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.
[9] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, ”Fault Tolerant Infective Countermeasure for AES”, J Hardw Syst Secur 1, pp. 3-17, 2017.
[10] 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.
[11] 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.