Compilation of a Countermeasure Against Instruction-Skip Fault Attacks
Thierno Barry, Damien Couroussé, Bruno Robisson

To cite this version:

HAL Id: cea-01296572
https://cea.hal.science/cea-01296572
Submitted on 5 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Compilation of a Countermeasure Against Instruction-Skip Fault Attacks

Thierno Barry†∗ Damien Couroussé† Bruno Robisson∗
† Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LIST, MINATEC Campus, F-38054 Grenoble, France
∗ CEA/EMSE, Secure Architectures and Systems Laboratory
CMP, 880 Route de Mimet, 13541 Gardanne, France
fistname.lastname@cea.fr

ABSTRACT
Physical attacks especially fault attacks represent one the major threats against embedded systems. In the state of the art, software countermeasures against fault attacks are either applied at the source code level where it will very likely be removed at compilation time, or at assembly level where several transformations need to be performed on the assembly code and lead to significant overheads both in terms of code size and execution time. This paper presents the use of compiler techniques to efficiently automate the application of software countermeasures against instruction-skip fault attacks. We propose a modified LLVM compiler that considers our security objectives throughout the compilation process. Experimental results illustrate the effectiveness of this approach on AES implementations running on an ARM-based microcontroller in terms of security overhead compared to existing solutions.

Categories and Subject Descriptors
C.3 [Special-Purpose And Application-Based Systems]: Microprocessor/microcomputer applications; K.6.5 [Security and Protection]: Physical security; D.3.4 [Processors]: Compilers—LLVM

Keywords
Fault Attacks, Countermeasures, Compiler, LLVM, AES

1. INTRODUCTION
In the last decade, embedded systems have increasingly become a critical part of our daily life, they represent the largest consumer electronics market segment. Despite the crisis affecting the global economy, this sector still continues to grow and generating profits. Only in 2014 there have been more than 8 billion of smart cards sold worldwide [7]. Most of these systems such as payment cards, access cards, SIM cards, smartphones, store and manipulate data of different kind: personal, confidential and sometimes critical. Hence, the security of these systems reveals itself as a major concern both for Industrial companies and also for states organizations. Physical attacks represent one of the most fearsome threats against embedded systems. Unlike classical cryptanalysis which basically rely on the mathematical robustness of cryptosystems, these attacks in turn exploit weaknesses of their implementations. Among them, Fault Attacks introduced by Boneh et al. [6] aims to analyze the effect of a deliberate disturbance of a circuit during its operation. The disruption can be induced through several ways, the most common techniques are: Variation of the supply voltage, variations in the clock signal, extreme variation of the temperature, focused white light, electromagnetic injection, X-rays and ion beams [3]. Since an electronic device is composed of several levels of hardware and software abstractions, talking about fault attack requires precising which abstraction layer we are dealing with and what kind of effects are expected, known as Fault Model. Verbauwhede et al. [12] present a pyramidal classification of different fault models, from algorithm level to logic level.

In this paper we consider the fault model presented by Moro et al. in [10], which assumes that the effect of the injected fault on a 32-bit microcontroller leads to an instruction skip. Moro et al. [11] and Barenghi et al. [4] have proposed implementations of the Instruction Redundancy technique as a countermeasure against this fault model, and formally proven by Moro et al. [11]. However the general approach of their propositions consists of duplicating instructions at the assembly code level. The downside of this approach is that most of assembly instruction cannot be duplicated trivially, as an example: the ARM assembly instruction add r0, r0, r1 which performs an addition of r1 and r0 and then stores the result in r0 cannot be executed more than once due to the fact that r0 is both a source and destination register. Therefore several transformations need to be performed on assembly instructions in order to adapt them to the instruction redundancy countermeasure, and lead to significant overhead.

To date, existing solutions are often ad hoc [5, 4, 11] and manually inserted by experts in the field. As a consequence, the application of such countermeasures is still challenging and costly task in the industry area. To reduce design cost and increase the confidence in security designs (because the manual insertion of protections is error prone), industries are strongly in demand of automatized tools.

DOI: http://dx.doi.org/10.1145/0000000.0000000
To address these concerns, we propose an LLVM-based compiler to efficiently automate the application the instruction redundancy countermeasure during the compilation. To achieve this goal, some existing passes of the original compiler are modified and new ones are introduced. Our compiler takes as input the source code to protect and produces a protected binary code ready to run. To the best of our knowledge this work presents the first use of compiler techniques to implement a countermeasure against instruction-skip fault model. Experimental results show the effectiveness of our solution in terms of overhead mitigation compared to existing approaches.

This paper is organized as follows: The section 2 discusses about limitations of the existing countermeasure approaches. Then we describe our proposed compilation approach in section 3. The section 4 is dedicated to experimental results and evaluations, followed by concluding remarks in section 5.

2. EXISTING APPROACHES

Since the emergence of fault attacks in 1997 [6], several hardware and software countermeasures have already been proposed, and essentially aim to avoid, detect and/or correct occurred faults. The advantage of software countermeasures is that no hardware modifications are needed, and in the case where a new generation of attacks appear, they are easy to update.

Lalande et al. in [9] propose a methodology to detect harmful intra-procedural jump attacks by inserting routine at source code level that checks and increments a counter before each instruction. The downside of applying countermeasures at source code level is that at least all the compiler code optimizers have to be disabled at compilation time. Otherwise we have no guarantee that the countermeasure will remain intact inside the binary code. This phenomenon is known as WYSINWYX (What You See Is Not What You eXecute) [2], which refers to the mismatch between what the source code description seems to indicate and what actually will be executed by the processor.

Bareghi et al. in [5] proposed an implementation of instruction duplication and instruction triplication at assembly level. However their solution only covers a small number of assembly instructions and designed to fit their AES implementation. The second problem of this approach is how to find free registers at the assembly code level? Authors suggest to use the r12 register which is considered as a scratch register in the ARM Application Binary Interface (ABI) [1]. But what if several instructions need to be transformed? In [5] authors propose an ad-hoc solution, they assume that their AES implementation uses only 9 registers and therefore, there still 4 available registers on their ARM-based Microcontroller, which prevents their solution to work on other implementations.

The first problem of this approach is the overhead introduced by these transformations. In the previous example to protect one single instruction we need to replace it by four instructions. [11] showed that to protect the instruction: \texttt{umlal rl0, rhi, rm} we need 14 instructions. It severely impacts both the execution performance and the code size.

This approach suffers from the fact that most of assembly instructions need to be transformed before duplication, because the compiler generates assembly code regardless of our requirements. Our approach consists of modifying the compiler in order to consider security objectives throughout the code generation process, so that instructions like \texttt{umlal} will no longer be generated, they will automatically be replaced by suitable ones. The register allocator and other components of the compiler are modified to maximise idempotent instructions.

3. COMPILATION APPROACH

3.1 Overview

In this section, we describe how we modify the LLVM compiler in order to apply the countermeasure formally verified by Moro et al. [11]. The countermeasure is activated by a specific compilation flag so that the developer can select the source files that will be protected against fault attacks. Future works will allow the developer to select the program

Figure 1: Duplication of idempotent instructions

```plaintext
1  add r0, r1, r2
2  add r0, r1, r2
```

(a) Original code

```plaintext
1  add r0, r1, r2
2  add r0, r1, r2
```

(b) Duplicated Code

Figure 2: Transformation and duplication of a non idempotent instruction

```plaintext
1  add r0, r0, r1
2  mov r0, r0
3  mov r0
4  add r0, r0, r0
```

(a) Original code

```plaintext
1  add r0, r0, r1
2  mov r0, r0
3  mov r0
4  add r0, r0, r0
```

(b) Duplicated Code

Figure 2. Transformation and duplication of a non idempotent instruction

3.1 Overview

In this section, we describe how we modify the LLVM compiler in order to apply the countermeasure formally verified by Moro et al. [11]. The countermeasure is activated by a specific compilation flag so that the developer can select the source files that will be protected against fault attacks. Future works will allow the developer to select the program
section to protect with source code annotations (using pragmas) in order to target specific functions or even code sections. This practical issue does however not reduce the general interest or effectiveness of our approach.

Our protection scheme [11] consists in (1) transforming all the machine instructions into a semantically equivalent sequence of idempotent instructions, and then (2) duplicating all the idempotent instructions. An idempotent instruction is an instruction that can been freely re-executed without changing the resulting state of the program. Hence, even if one of the duplicated instructions is faulted, resulting in an instruction skip according to the fault model, the other instruction is still executed so that the program execution produces a correct result.

In fact, an instruction is idempotent when its inputs are not modified during its execution. Input values of an instruction are modified in two cases:

1. When the characteristic of the instruction is that it implicitly updates one of its source registers, for example: push (r0) which writes the value of r0 at the address contained in the sp register and then decrements sp by 4, that makes sp as an implicit modified source register.

2. When registers are allocated in such a way that one of the source registers of the instruction is also a destination register, for example: add r0, r0, r1 which accumulates the result of r0+r1 in r0.

Hence, the proposed scheme consists of (1) modifying the Register Allocator pass (section 3.4) in order to guarantee that input and output values will never share the same registers; (2) transforming the instructions into idempotent ones (sections 3.3 and 3.5). Once all the instructions have been transformed into a semantically equivalent idempotent form, they can be safely duplicated (section 3.5).

3.2 Instruction Duplication

In the LLVM compiler, a program goes through several representations before arriving to the target-dependent representation. Performing the duplication inside the compiler raises the question of which representation is the most suitable for instruction duplication. Ideally one would like it to be on the intermediate representation, because in this case the countermeasure could be generalized for all languages and architectures supported by the compiler. But actually it depends on the countermeasure model, it may require to be more or less close to the hardware. In our case, the instruction duplication cannot be carried out at the IR representation due to the so called SSA (Static Single Assignment) form, which prevents affecting a virtual register more than once inside a delimited code region (Basic Block).

Figure 3 illustrates the limitations caused by the SSA form. The sub-figure 3b is the representation in LLVM byte-code of 3a, names of virtual registers are preceded by: %. The sub-figure 3c represents an attempt to duplicate instruction at the IR, where we can notice that duplicated instructions are no longer identical to actual ones. Actually virtual registers of duplicated instructions have been renamed to comply with the SSA form (only one assignment for each virtual register). And since these renamed virtual registers are never used elsewhere, they are considered as dead, and corresponding instructions will trivially be removed by the DCE (Dead Code Elimination) pass. This is the reason why the instruction duplication must happen only after the SSA form (after the physical registers are allocated). And because the idempotence is the necessary and sufficient condition to duplicate an instruction, we have modified two LLVM passes to maximize the number of idempotent instructions, and implemented six new passes to transform the rest of instructions into idempotent forms. The internal structure of our modified compiler is illustrated by figure 4, where grey boxes represent modified LLVM passes and black boxes depict the new passes we have implemented.

```c
int foo(int a, int b, int c) {
    return a * b * c;
}
```

(a) Source Code

```llvm
%mul1 = mul %a, %b
%mul11 = mul %mul1, %c
ret %mul12
```

(b) LLVM Byte-Code

```llvm
%mul1 = mul %a, %b
%mul11 = mul %a, %b
%mul12 = mul %mul1, %c
%mul122 = mul %mul1, %c
ret %mul12
ret %mul12
```

(c) Attempted duplication

Figure 3: Illustration of instruction duplication on the IR

3.3 Modification of the Instruction Selector (IS)

The Instruction Selection pass is composed of a set of LLVM passes responsible for transforming the program from a tree-based representation into a low-level representation very close to the target language, and interfacing with the target ABI (Application Binary Interface) in order to select appropriated target instructions for each operation described by the program developer. For example, in the case of a multiplication by a power of two values, this pass has to find whether it is profitable for the target hardware in terms of execution speed to select a mul or shift instruction. We have modified this pass to take into account both the ABI specifications and also the needs of the implemented countermeasure, so that idempotent instructions are the ones privileged during the selection. For example: the figure 5 shows the tree-based representation of this operation: (a * b) + c. When transforming this tree into the target architecture representation (ARM microcontroller in our case), the Multiply an Accumulate (mla) instruction is matched by default,
which is a non-idempotent instruction. Instead, we force the selection of a `mul` followed by an `add`, and they are both idempotent. However, this first step of modification concerns only the selection of the suitable instruction opcode, not the number of operands they will take. In fact, if the underlying architecture supports 16-bit instructions encoding, this pass attempts to reduce the number of operands underpinning architecture supports 16-bit instructions encoding, this pass attempts to reduce the number of operands to two if possible. In the case of the `mul` instruction:

```plaintext
add reg1, reg2
```

is equivalent to

```plaintext
add reg1, reg1, reg2
```

meaning

```plaintext
reg1 = reg1 + reg2
```

The first one is definitely non idempotent and the second one can be idempotent if registers are allocated in this way:

```plaintext
add reg0, reg1, reg2
```

This is the core motivation of modifications introduced in the Register Allocator (presented below).

### 3.4 Modification of the Register Allocator (RA)

The role of the register allocator is to map the endless number of SSA virtual registers to a limited number of physical registers. Thereby the RA attempts to reuse as much as possible each register in order to avoid assigning variables into memory locations (spill slots). For that, the RA implements an analysis pass called liveness analysis. This pass computes for each variable an interval in which the variable is considered alive. The liveness interval is a pair of program points, start and end, which starts when the value of the variable is defined and ends at the last point where the value is used (read). Thus, if two variables have disjoint liveness intervals, they means they can be assigned to the same physical register.

Let’s see how the following C-like instruction is transformed:

```plaintext
x3 = x2 + x1;
```

Just before the RA, the instruction looks like:

```plaintext
add x3, x2, x1.
```

The RA computes the liveness interval for each virtual variable. Let

```plaintext
L_{x3}, L_{x2}, L_{x1}
```

be the liveness interval of `x3`, `x2` and `x1`. The result of the liveness analysis is as follows:

1. `L_{x1} \cap L_{x2} \neq \emptyset`
   because both `x1` and `x2` are needed by the Arithmetic Logic Unit (ALU) of the processor to compute the addition operation, therefore they are alive at the same time.

2. `L_{x1} \cap L_{x3} = \emptyset` and `L_{x2} \cap L_{x3} = \emptyset`
   `x3` is the destination variable, it’s where the result of the addition will be stored. So the liveness of `x3` starts once the result of the operation is available at the output of the ALU, while the liveness of `x1` and `x2` if not used elsewhere ends at the moment their values are taken into account at the input of the ALU

Because of the relation (1) `x1` and `x2` will be assigned to different physical registers. As thanks to the relation (2), the RA decides to assign `x3` and `x2`, or `x3` and `x1` to the same physical register. Finally, the resulting instruction will looks like:

```plaintext
add r0, r0, r1
```

and is not idempotent.

Our modification occurs at this level. And it concerns all instruction of this form:

```plaintext
opcode dst, src1, [src2]
```

where `opcode` can be an opcode of any instruction that stores its result in a register, such as arithmetic, bitwise or load instructions. `dst` is the destination register, `src1` is a source register and `[src2]` is an optional source register. We have modified this pass in such a way that the physical register to allocate for the destination register is always different to source registers even if the relation (2) is true:

```plaintext
(dst \neq src1) \land (dst \neq src2)
```

Thus in the previous example, instead of generating `add r0, r0, r1` we generate `add r2, r0, r1` which in turn is idempotent.

### 3.5 Transformations Passes

The modified LLVM passes presented above, allow to increase the number of idempotent instructions, but there are some instructions that need special treatments. For these special instructions, we have implemented the following LLVM passes (black boxes in figure 4) to process each of them.

#### 3.5.1 Store Sanitizer Pass

The ARM thumb2 instruction set provides different variants of the store instruction [1]. Some of them are idempotent, such as:

```plaintext
str r0, [r1, #4]
```

that stores the value of `r0` at the address in `r1` added of 4.

But for instance this other variant of store:

```plaintext
str r0, [r1], #4
```

does a slightly different operation, it stores the value of `r0` at the address in `r1` and increments `r1` of 4. This variant is not idempotent because the source register `r1` is modified during the operation.

The role of this pass is to transform store instructions into a one of its idempotent variant, illustrated in figure 6.

```
1. str r0, [r1], #4  
2. add r0, r1, #4  
3. mov r1, r0
```

(a) Before  
(b) After

Figure 6: Store Sanitization

#### 3.5.2 Push/Pop Elimination Pass

`push` and `pop` are two pseudo instructions that respectively write and read a value from memory, decrements and increments the stack pointer (SP) register [1], and both are
Table 1: Overhead in terms of code size (in bytes) and execution speed (in clock cycles) for each AES implementation. The last two columns report corresponding results presented by Moro et al. [11] with the same AES implementations.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cycles</td>
<td>size</td>
<td>cycles</td>
<td>size</td>
</tr>
<tr>
<td>-O0</td>
<td>19698</td>
<td>11808</td>
<td>13808</td>
<td>×1.70 ×1.16</td>
</tr>
<tr>
<td>-O1</td>
<td>14688</td>
<td>11552</td>
<td>13248</td>
<td>×1.69 ×1.14</td>
</tr>
<tr>
<td>-O2</td>
<td>68000</td>
<td>12528</td>
<td>15264</td>
<td>×1.90 ×1.22</td>
</tr>
<tr>
<td>-O3</td>
<td>5168</td>
<td>12688</td>
<td>15824</td>
<td>×2.09 ×1.25</td>
</tr>
</tbody>
</table>

Moro et al.’s AES

MiBench AES

-00 1908 66924 3555 79260 ×1.76 ×1.18
-01 1142 64604 2188 68988 ×1.92 ×1.08
-02 1142 60092 2188 69452 ×1.92 ×1.16
-03 1140 59628 2185 68956 ×1.92 ×1.16

×2.86 ×2.90

The role of the scheduler (illustrated in figure 10) is to rearrange the execution order of instructions in order to improve the execution time while preserving the original behavior of the program. The rearrangement is based on data dependencies, instruction latencies [1] and instruction-level parallelism of the target architecture. The instruction duplication can be performed before or after the scheduling according to the user choices. The advantage of duplicating instructions before the scheduling is that duplicated instructions will be scheduled together with original ones. The code represented in figure 10b runs faster than the one in figure 10a due to the better exploitation of processor pipeline, illustrated by the number of clock cycles required to execute each code block.

4. EXPERIMENTAL RESULTS

We have experimented our approach with two different AES implementations, the same versions as the ones used by Moro et al. [11] including the MiBench [8]. Compiled with our modified LLVM compiler (based on the stable version 3.6) to generate binary codes where the entire instructions are duplicated. Our experimental platform is an ARM-based microcontroller embedding a cortex M3 core and supporting the thumb2 instruction set. Table 1 reports

3.5.5 Instruction Duplication Pass

The role of this pass is to duplicate all the instructions. But right before, the Pre-Instruction Duplication Pass is run to ensure the idempotence of all instructions. In fact several other LLVM passes that are not represented in figure 4 (for the sake of brevity) perform multiple transformations beside ours, and may remove or introduce new instructions. This pass actually checks whether newly introduced instructions (if there are ones) are idempotent or if one of our transformation passes need to run again.

3.6 Instruction Scheduling

The Thumb2 instruction set provides conditional execution by the use of If-Then (IT) blocks, which means up to 4 instructions can be executed conditionally. Conditions can be all the same, or some of them can be the logical inverse of the others [1]. Neither the it instruction nor conditional instructions are idempotent. An it block is a compact form of classical If-Then-Else blocks. The role of this pass as illustrated in figure 9 is to transform the it block into the corresponding If-The-Else block, regardless to whether instructions inside the blocks are idempotent or not.
for each AES implementation the size of its binary code, the number of clock cycles it takes to execute and the corresponding overhead introduced by the countermeasure. The last two columns highlight the results reported by Moro et al [11] on the same implementations. We can notice the overhead mitigation of our approach compared to the assembly one.

Intuitively one might expect that the instructions duplication at least doubles the execution speed and the code size. And yet, with -O0 optimization level we obtained $\times 1.76$ and $\times 1.18$ of overheads respectively for the execution speed and code size, while Moro et al. for the same implementation have $\times 2.86$ and $\times 2.90$. The reduced overhead regarding the execution speed is due to the fact that more than 95% of generated instructions are idempotent (only less than 5% of them need to be transformed) and also because duplicated instructions are efficiently scheduled by the compiler (the role of the scheduler is presented in 3.6). And the reduced overhead in the code size is due to the fact that our processor supports both 16-bit (thumb 1) and 32-bit (thumb2) instruction set, and for the same operation the selected instruction may be different depending on whether the countermeasure is being applied or not. To sum up, applying the countermeasure during compilation allows the compiler to take better decisions based on our security objectives.

5. CONCLUDING REMARKS

We presented the use of compilation techniques to efficiently automate the application of a software countermeasure against fault injection attacks based on the instruction skip model. We have shown that existing solutions are penalized by the fact that several transformations on the assembly code need to be performed before applying the countermeasure. We have presented different modifications introduced inside our compiler to generate idempotent instructions, and new passes to duplicate instructions before scheduling. Experimental results showed the efficiency of our approach, leading to a strongly reduced overhead in code size and execution speed compared to existing solutions.

Acknowledgements

This work was partially funded by the French National Research Agency (ANR) as part of the project COGITO funded by the program Digital Engineering and Security (INS-2013) under grant agreement ANR-13-INSE-0006-01, and as part of the project PROSECCO funded by the program AAP-2015 under grant agreement ANR-15-CE39. Great thanks to Nicola Moro for his valuable assistance.

References