Software Fault Resistance is Futile:
Effective Single-glitch Attacks

Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
{bilgiday,farhady,harika90,chinmay,conorpp,schaum}@vt.edu

Abstract—Fault attacks are a serious threat for the secure embedded software running on a wide spectrum of embedded devices. Fault attacks can be thwarted using countermeasures in software. Among them, instruction-level countermeasures provide a fine-grained protection by executing redundant copies of an assembly instruction, and verifying their results for fault detection. It is assumed that this fine-grained security can only be broken by injecting multiple faults with expensive tools. In this work, we break the security of state-of-the-art instruction-level countermeasures by injecting single clock glitches with a low-cost fault injection setup. We first analyze their vulnerabilities by considering micro-architectural aspects such as pipelining effects. Second, we experimentally demonstrate the feasibility of exploiting these vulnerabilities on a SAKURA-G board. Finally, as a case study, we apply a recent biased fault attack on a fault-resistant software implementation of LED block cipher, and retrieve its secret key.

Index Terms—Fault Analysis; Software Countermeasures; Fault-attack Resistant Microprocessor Extensions.

I. INTRODUCTION

In our daily life, we are increasingly putting our trust in embedded software applications, which run on a range of processor-based embedded systems from smartcards to pay-TV units. This trend expands the threat model of embedded applications from software into hardware. Over the last 20 years, fault attacks have emerged as an important class of hardware attacks against embedded software security. In fault attacks, an adversary breaks the security by injecting well-chosen, targeted faults during the execution of embedded software, and systematically analyzing software’s fault response.

In case of a fault attack on embedded software, the faults are injected into the underlying processor hardware and their effects are observed in the executed software’s output. First, an adversary temporarily alters the execution of instructions by running the processor hardware beyond its nominal operating conditions. Second, the adversary observes the effects of faulty instructions during the execution of the software. Finally, the adversary breaks the security by performing a systematic fault analysis method such as Differential Fault Analysis (DFA) on the observed output. As a result, the success of a fault attack is affected by the executed software, instruction set architecture (ISA) of the processor, and the processor hardware (Fig. 1).

Embedded software is commonly protected with software countermeasures. The motivations behind software countermeasures are the following. First, they do not need any modification to the underlying hardware. Second, they are easy-to-deploy and portable. Third, they protect the behavior of the cryptosystem rather than the hardware implementation of it, and therefore, they are agnostic to the fault injection mechanism. The initially proposed software countermeasures rely on algorithm-level fault detection mechanisms. The most straightforward method for the fault detection is running an algorithm twice, and comparing the outputs of both executions. Another approach is using error detection codes or parity bits for the critical data of an algorithm and checking them at the end of the algorithm. However, it has been shown that these algorithm-level countermeasures are insecure against multiple fault injections and adaptive adversaries.

To address this vulnerability, instruction-level software countermeasures have been proposed. An example instruction-level scheme is duplicating an instruction, and comparing the results of original and duplicated copies using another instruction. The instruction-level countermeasures are assumed to be secure from single fault injections because of the redundancy. They are also believed to be practically secure against multiple fault injections based on the following assumption:
Injecting faults into two consecutive instructions requires expensive fault injection tools and very high synchronization capabilities. Therefore, it is not considered as a practical threat.

In this paper, we demonstrate that we can break instruction-level software countermeasures using a single fault injection, revealing that they are much easier to break than thought. We achieve this by exploiting a common limitation of all software countermeasures that the programmer’s view of a processor is limited to processor’s ISA (Fig. 1). The micro-architectural aspects of the processor such as pipeline effects, cache effects, and physical implementation are invisible to the software countermeasures. Because of this limitation, the software countermeasures themselves are also vulnerable to fault injection as much as the protected software is. Therefore, we are able to find weak points during their execution on the processor hardware and break the countermeasures with a single fault injection. The contributions of our work can be summarized as follows:

1) We analyze the state-of-the-art instruction-level countermeasures by considering the micro-architectural aspects and identify their vulnerabilities. The analyzed countermeasures include instruction duplication, instruction triplication, instruction-level parity checking, and fault-tolerant instruction sequences. This analysis shows that all of these countermeasures can be broken with a single fault injection. To our knowledge, this is the first work that provides such an analysis on the software countermeasures.

2) We experimentally demonstrate the feasibility of the aforementioned findings. We break all of the analyzed instruction-level countermeasures using a single clock glitch injection, despite previous tests and formal verification of some countermeasures. This is the first work to demonstrate a practical single-fault attack on instruction-level countermeasures with low-cost clock glitch injection.

3) As a case study, we applied a complete Differential Fault Intensity Analysis (DFIA) attack on a fault-protected software implementation of the LED block cipher. The double-protected LED implementation uses algorithm-level parity codes in addition to instruction-level duplication. We mounted the attack on a 7-stage-pipeline LEON3 processor mapped on an FPGA.

The paper is organized as follows. In Section II, we provide a summary of existing software countermeasures. In Section III, we explain the fault injection and effects on a RISC pipeline. In Section IV, we analyze the execution of several instruction-level countermeasures on a RISC pipeline, and reveal their vulnerabilities. Section V explains the experimental setup. In Section VI, we experimentally demonstrate the feasibility of breaking instruction-level countermeasures. Section VII provides a case study, in which we break the security of a software-protected LED block cipher implementation. Section VIII concludes the paper.
Fault effects can be classified into two classes. An adversary will want to leverage these two classes to produce meaningful outputs for fault analysis attacks [13], [14].

1) **Instruction Faults (IF):** These faults change the control flow of a program. A fault could change an instruction or even skip it. These are created by faulting instructions in the F or D stages as shown in Figure 2. An adversary will leverage IF to bypass instructions that check integrity or branch when an error is detected. If a fault is detected or handled, then the software output will not be useful for fault analysis.

2) **Computational Faults (CF):** These faults cause errors in data used by a program. They are created by any error in the stages A, E, M, and W as indicated in Figure 2. An error in any of these stages can contribute a faulty value to the register file or memory. It is important for an adversary to target these stages to be able to propagate errors to the output of the software. Once there are observable errors in the output, fault analysis attacks can be conducted.

In the case of clock glitching, it is important to understand fault sensitivity [15]. Fault sensitivity is a measure of how vulnerable a circuit is to a particular fault. When multiple instructions are in the pipeline, it is possible to affect some, but not all of the instructions. This is because each pipeline stage has varying sensitivities for each instruction. For example, arithmetic instructions have a long critical path in the E stage and will be more likely to get an error from a clock glitch. A branch calculates the next address in the D stage so its critical path will be long and sensitive to clock glitches. These pipeline characteristics can be taken advantage of by an adversary.

A branch in the E stage does not use the ALU and will have a very short critical path. If an adversary was targeting a different stage, he could inject a very intense clock glitch and not have to worry about introducing an error in the E stage.

As for other fault injection methods, we believe they will have similar microarchitectural characteristics they can take advantage of.

### III. Fault Behavior in LEON3

We implemented our work in a 32 bit LEON3 processor on a Xilinx FPGA to gain insight into fault behavior. The LEON3 implements a modified Harvard architecture. It contains a 7 stage RISC pipeline. It is distributed as a modular VHDL model by Aeroflex Gaisler. It has different modules and debugging support that can be configured. It is an ideal softcore for gaining a lower level and deeper understanding of pipeline response to fault injection.

Since the architecture is pipelined, multiple instructions are being executed in parallel. It is likely that more than one instruction will be executing when a fault is injected. The instruction could be in any of the following standard stages: Fetch (F), Decode (D), Register Access (A), Execute (E), Memory (M), Exception (X), Write-back (W). Depending on the instruction stream, particular stages can be targeted for fault injection. For instance, if one load instruction is in the E stage and another one is in the M stage, then an injected fault could easily cause errors in both instructions. Instruction stalls are typically unaffected by faults.

Figure 2 demonstrates how multiple instructions can be exploited with a single fault injection. A fault is injected in cycle 4 when the following instructions in their respective stages are executing: F6, D5, A4, E3, stall, X2, W1. The brown instructions are potentially affected by the injected fault. The X stage is not useful for causing fault injection.
A. Instruction Duplication Countermeasure

We first examine the Instruction Duplication techniques proposed by Barenghi et. al in [2]. In this technique, certain instructions in the program are duplicated and the results are stored in different registers. The two copies of the instruction results are compared and an alarm is raised in case of a mismatch.

The following code sample illustrates the Instruction Duplication countermeasure on a Memory Load (LD1) instruction. This code protects the LD1 instruction by storing the load value in both %g2 and %g3 registers. The values of these two registers are compared by CMP instruction and an error policy is called if a mismatch is detected.

```
LD [%fp−12], %g2 : LD1
LD [%fp−12], %g3 : LD2
CMP %g2, %g3 : CMP
BNE .error : BNE
```

In order to break this countermeasure, we first analyze its behavior in the pipeline in Figure 3. This figure shows the sequence of instructions in the pipeline.

The gray instructions show stalls in the pipeline. Stall 1 in Figure 3 is due to the data dependency between the LD2 and CMP instructions. As shown, the results of LD2 will be ready in the Stage E at Cycle 4. The CMP instruction in Stage A waits for the result of LD2 until Cycle 5 and then continues its execution. Stall 2 on the BNE instruction is due to the branch interlock. The branch interlock happens in the case of a conditional branch. When a conditional branch is performed in 1-2 cycles after an instruction which modifies the condition codes, 2 cycles of delay is added to allow the condition to be computed.

The target of fault injection to thwart this countermeasure can have two forms. First, injecting two identical faults in each of the LD instructions to bypass the equality check. Second, a single fault injection into the LD1 instruction and skipping the CMP or BNE. In this section, we use the pipeline analysis of the Instruction Duplication code to find vulnerable points of fault injection to create different scenarios explained above. The first step is to define the valid targets of fault injection in each cycle. Based on the analysis in Section III the valid stages to inject faults are defined as follows. The red circled instructions show the valid instructions to target computation faults. These instructions can be targeted to generate different faulty values in registers. Afflicting the black squared pairs of (instruction, pipeline stages) will cause instruction faults. For example, targeting the LD instructions in Stage M or W stage will generate different faulty values and targeting the CMP or BNE instruction in Stage F and Stage D will cause instruction faults. Then, an adversary can define different scenarios for fault injection in each cycle.

Table I summarizes the two different scenarios. In this table, columns 2 and 3 show the potential targets of fault injection in the pipeline for achieving each scenario. The last column shows the type of fault that is injected into each instruction of the pipeline. The two scenarios are explained in detail as follows.

1) Scenario A.1. Double Computation Fault: The purpose of this fault injection is to inject exactly the same faults into the original and redundant copies of the LD1 instruction. These fault injections must not affect the CMP or BNE instructions. This scenario can be achieved by injecting a fault into Cycle 3 of the pipeline. Injecting a fault into this cycle does not have any effect on the CMP or BNE instructions, due to Stall 1 in the pipeline.

2) Scenario A.2. Single Computation Fault-Single Instruction Fault: Another way to bypass this countermeasure is to create faulty values in register %g2 by a computational fault in LD1 and skip the CMP or BNE instructions. To achieve this type of fault, we can trigger the fault injection in different cycles.

- **Single fault injection**: Scenario A.2. can be obtained by a single glitch injection. The single glitch injection must accurately target the cycle that is performing both computational operations on LD1 and instructional operations on CMP or BNE. As shown in the pipeline, these cycles can be Cycle 2 that affects CMP (F) or Cycle 4 that can affect CMP (D) or BNE (F).

- **Multiple fault injection**: Scenario A.2. can be obtained by multiple fault injections as well. To create a data fault, the adversary can target Cycle 3 or 6. Injecting fault into Cycle 3 or 6 will create a faulty value in LD1 (E) and LD1 (W) respectively. Cycle...
### TABLE I
Fault Attack Scenarios to Thwart Instruction Duplication Countermeasure

<table>
<thead>
<tr>
<th>Scenarios</th>
<th># of Glitch Injections</th>
<th>Targeted Cycles</th>
<th>Instruction, Fault Type</th>
<th>LD1</th>
<th>LD2</th>
<th>CMP</th>
<th>BNE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1.</td>
<td>1</td>
<td>3</td>
<td>CF</td>
<td>CF</td>
<td>CF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3,4</td>
<td>CF</td>
<td>CF</td>
<td>IF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5,7</td>
<td>CF</td>
<td>CF</td>
<td>IF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A.2.</td>
<td>2</td>
<td>3</td>
<td>CF</td>
<td>CF</td>
<td>IF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>CF</td>
<td>CF</td>
<td>IF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B.1.</td>
<td>1</td>
<td>10</td>
<td>CF</td>
<td>CF</td>
<td>IF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>B XOR</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>B XOR</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>B XOR</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
</tr>
</tbody>
</table>

6 is a suitable target for fault injection since Stages F, D, A and M are busy waiting due to Stall 2. After achieving the computation fault in register %g2, we can inject an instruction fault into Cycle 7 to skip the BNE (D) instruction or into Cycle 4 to cause an instruction fault (opcode change) into CMP (D) or BNE (F).

### B. Instruction Triplication Countermeasure

This software countermeasure is proposed by Barenghi et al in [2]. This technique repeats the instruction three times and stores the result in two unused registers. The countermeasure then checks the equality of each pair of instructions and raises a single bit flag in case of a mismatch. The value of the flag will later used to determine whether a single error correction or a double error detection is required. A LEON3 assembly code for the countermeasure is given below.

```assembly
; Instruction Triplication on LD
; clear g5
MOV 0, %g5
; three copies of the instruction
LD [%fp-12], %g2 ;LD1
LD [%fp-12], %g3 ;LD2
LD [%fp-12], %g4 ;LD3
: if g2 ! = g4, set flag 0 (g5 = g5 XOR 1)
CMP %g2, %g4
BNE .setFlag0 ;BNE
NOP .returnFromsetFlag0:
: if g3 ! = g4, set flag 1 (g5 = g5 XOR 2)
CMP %g3, %g4
BNE .setFlag1 ;BE
NOP .returnFromsetFlag1:
: if g2 ! = g3, set flag 2 (g5 = g5 XOR 4)
CMP %g2, %g3
BNE .setFlag2 ;BE
NOP .returnFromsetFlag2:
: if all flags are set, raise an alarm
CMP %g5, 7
BE .raiseErrorSignal ;BE
NOP
: if g4 is faulty but g2 and g3 are correct
write the correct value into g4 (g4 = g2)
CMP %g5, 3
BE .correctResult ;BE
NOP

In the above assembly code, the original instruction to protect is a Memory Load instruction (LD) that loads a value from [%fp-12] into register %g4. The results are stored in the target register (%g4) and two dummy registers (%g2 and %g3). The result of each pairwise comparison is stored in the register %g5 as a 1-bit flag. At the beginning of the code, the value of the flags are initialized to zero. If a comparison operation results in inequality, its flag in %g5 is set. Finally, the value of register %g5 is checked to decide whether to correct the value of %g4 by majority voting, raise an error signal in case of uncorrectable error, or keep the result of %g4 unchanged.

An adversary can come up with many different attack scenarios for this countermeasure because the execution of the countermeasure takes several clock cycles. In this work, we only focus on a part of its execution as shown in Figure 4.

![Fig. 4. Pipeline Behavior for Instruction Triplication Countermeasure for LD Instruction](image)

### TABLE II
Fault Attack Scenarios to Thwart Instruction Triplication Countermeasure

<table>
<thead>
<tr>
<th>Scenarios</th>
<th># of Glitch Injections</th>
<th>Targeted Cycles</th>
<th>Instruction, Fault Type</th>
<th>LD1</th>
<th>LD2</th>
<th>LD3</th>
<th>CMP</th>
<th>BNE</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1.</td>
<td>1</td>
<td>5</td>
<td>CF</td>
<td>CF</td>
<td>CF</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
<td>LD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>NOP</td>
<td>BNE</td>
<td>CMP</td>
<td>LD1</td>
<td>LD1</td>
<td>LD1</td>
</tr>
</tbody>
</table>

In the above assembly code, the original instruction to protect is a Memory Load instruction (LD) that loads a value from [%fp-12] into register %g4. The results are stored in the target register (%g4) and two dummy registers (%g2 and %g3). The result of each pairwise comparison is stored in the register %g5 as a 1-bit flag. At the beginning of the code, the value of the flags are initialized to zero. If a comparison operation results in inequality, its flag in %g5 is set. Finally, the value of register %g5 is checked to decide whether to correct the value of %g4 by majority voting, raise an error signal in case of uncorrectable error, or keep the result of %g4 unchanged.

An adversary can come up with many different attack scenarios for this countermeasure because the execution of the countermeasure takes several clock cycles. In this work, we only focus on a part of its execution as shown in Figure 4. To illustrate the single-glitch attacks on this focused part, we will define three example attack scenarios. Using the same approach, further attack scenarios can also be derived for both single and multiple fault injections. The purpose of our attack scenarios is to save a faulty value in %g4 without raising an alarm or without activating the error correction function. These scenarios are also summarized in Table II.
1) Scenario B.1. Triple Computation Fault: The purpose of this fault injection is to inject exactly the same faults into the original and redundant copies of the LD instruction. These fault injections must not affect the CMP or BNE instructions. This scenario can be achieved by injecting a fault into Cycle 8 of the pipeline. Injecting fault into this cycle does not have any effect on the CMP or BNE instructions, due to the Stall 1 in the pipeline.

2) Scenario B.2. Double Computation Fault: The purpose of this fault injection is to inject identical faults into two redundant copies of the LD instruction. In this case, the equality check will detect the occurred fault and call the correction function. However, the correction function saves the wrong value into the register %g4 due to the majority voting. To inject this fault, the target of fault injection can be Cycle 3 or 10. Cycle 10 is a suitable target for this fault injection as the only stages that can be affected by the fault injection are Stage M and Stage W that hold LD3 and LD1.

3) Scenario B.3. Single Computation Fault-Single Instruction Fault: Another way to bypass this countermeasure is to create faulty values in register %g4 by a computational fault in LD3, and create an instruction fault in the CMP or BNE instructions. The single glitch injection must accurately target the cycle that is performing both computational operations on LD3 and instructional operations on CMP or BNE. As shown in the pipeline, one of these cycles can be Cycle 9, in which BNE (F), CMP (D), and LD3 (E) can be affected. Similarly, BNE (D) and LD3 (W) can be affected in the Cycle 12.

C. Parity Countermeasure

This software countermeasure is proposed by Barenghi et al in [2]. In this technique, we first save the precomputed value for the parity bit in a register. Then, the parity is computed on the fly for the protected register’s value. The computed parity value is compared to the precomputed value and an alarm is raised if a mismatch happens.

In the following assembly example, the instruction that is protected is a Memory Load instruction that loads a value from [%fp−12] into register %g3. The precomputed parity value is stored in %g2. The computed parity value is obtained using some Shift-Right (SRL) and XOR instructions and stored in %g4. The value of the precomputed parity bit will be compared to the value of %g4 and the countermeasure will raise an alarm in case of mismatch.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parity on LD</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>[%fp−12], %g3 :LD1</td>
</tr>
<tr>
<td>MOV</td>
<td>%g3, %g4 :MOV</td>
</tr>
<tr>
<td>SRL</td>
<td>%g3, 4, %g5 :SRL1</td>
</tr>
<tr>
<td>XOR</td>
<td>%g4, %g5, %g4 :XOR1</td>
</tr>
<tr>
<td>SRL</td>
<td>%g4, 2, %g5 :SRL2</td>
</tr>
<tr>
<td>XOR</td>
<td>%g4, %g5, %g4 :XOR2</td>
</tr>
<tr>
<td>SRL</td>
<td>%g4, 1, %g5 :SRL3</td>
</tr>
<tr>
<td>XOR</td>
<td>%g4, %g5, %g4 :XOR3</td>
</tr>
<tr>
<td>AND</td>
<td>%g4, 1, %g4 :AND</td>
</tr>
</tbody>
</table>

As shown in Figure 5, there are several opportunities to inject useful faults into this countermeasure. The parity countermeasure is vulnerable to many types of fault injection scenarios as it contains many instructions for computing the parity. Some of these opportunities are explained below, that is summarized in Table III.

1) Scenario C.1. Single Computation Fault: The purpose of this fault injection is to inject computational faults into the original LD instruction, so that the effects of fault can change an even number of bits in register %g3. Therefore, the computed value for parity bit will still be the same as the correct execution. These fault injections must not affect the CMP or BNE instructions. This scenario can be achieved by injecting a fault into the Cycle 2, 3, 4 or 13.

2) Scenario C.2. Single Computation Fault-Multiple Instruction Fault: The purpose of this fault injection
is to inject computation fault into the LD instruction and inject instruction faults in the computation of the parity bit. This scenario can be obtained by a single glitch injection. The single glitch injection must accurately target the cycle that is performing both computational operations on LD3 and instruction operations on SRL1(F-D-E), MOV(D-M), XOR1(F-A), XOR2(F), SRL2(D), XOR1(F). The first fault injection can be targeted in Cycle 3, 4 and 13. The instruction faults can be targeted in Cycle 13 to 17.

3) Scenario C.3. Single Computation Fault-Single Instruction Fault: Another way to bypass this countermeasure is to create faulty values in register %g3 by a computational fault in LD1 and skip the CMP or BNE instructions. To achieve this type of fault, we should trigger the computation fault in Cycle 2 and the instruction fault in Cycle 20 to target the BNE(F).

**D. Instruction Skip Countermeasure**

This countermeasure is proposed by [3]. This redundancy technique is used to avoid the instruction skip faults. Instruction skip fault is defined as a fault that can change an instruction to an effective NOP instruction. Therefore, the fault model is not defined as computational fault model, but it can only change the opcode of the protected instruction. Following shows the assembly code for the instruction skip countermeasure on LD instruction.

```
; Instruction Skip on LD
 LD [%fp−12], %g2 :LD1
 LD [%fp−12], %g2 :LD2
```

The pipeline behavior for the instruction skip countermeasure is shown in Figure 6. As shown, since the adversary does not target for computational faults, he can only target the first three cycles. Therefore, there is only one scenario that can achieve such a fault.

1) Scenario D.1. Double Instruction Fault: There are two ways to achieve instruction faults in both LD instructions.

- **Single Fault Injection:** By injecting a fault into Cycle 1, the adversary can change the opcode of the instruction in the LD2 (F) and LD1 (D) (as shown in Table IV).
- **Multiple Fault Injection:** By targeting the first fault injection in Cycle 0, the adversary can affect the LD1 (F). The second fault can target either Cycle 1 or 2 to modify the LD2 (F) or LD2 (D).

**V. EXPERIMENTAL SETUP**

In this section, we provide an outline of our fault injection and measurement setup that enables us to verify our method and to establish the results. Figure 7 gives an overview of the setup. It consists of a Control PC, a Device Under Test (DUT), a pulse generator, and a clock glitcher module. We implement the glitcher module and DUT on a SAKURA-G board [16]. The Control PC manages the fault injection process by controlling both the clock glitcher module and DUT. The clock glitcher module takes a glitch-free clock signal from the pulse generator, and produces a glitchy clock signal. It consists of clock glitch controller and clock glitch injector. Initially the PC configures the clock glitch controller by setting the required glitch parameters. The clock glitch controller records these specified parameters and waits for trigger from DUT. The PC also configures the DUT which runs a cryptographic program. The DUT sends a trigger signal to the clock glitcher module when it reaches the predefined point in the program. After receiving the trigger signal, the clock glitch controller arms the injector. Then after a predefined number of cycles, the injector injects the specified glitch. It also generates capture

![](image_url)
TABLE V
Fault Injection Results on Instruction Duplication Countermeasure

<table>
<thead>
<tr>
<th>Glitching Scenario</th>
<th>Target of Fault Injection</th>
<th>Glitch Width (ns)</th>
<th>Impacted Instruction in Pipeline</th>
<th>Observed Fault Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1. Single Glitch</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A.2. Single Glitch</td>
<td>Cycle 2</td>
<td>9.0 - 12.6</td>
<td>LD1, A</td>
<td>Fault in %g2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CMP, F</td>
<td>CMP to SRL</td>
</tr>
<tr>
<td>A.2. Single Glitch</td>
<td>Cycle 4</td>
<td>12.0 - 14.6</td>
<td>LD1, M</td>
<td>Fault in %g2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BNE, F</td>
<td>BNE to NOP</td>
</tr>
<tr>
<td>A.2. Multiple Glitch</td>
<td>Cycle 3</td>
<td>14.7 - 14.9</td>
<td>LD1, E</td>
<td>Fault in %g2</td>
</tr>
<tr>
<td></td>
<td>Cycle 7</td>
<td>12.6 - 13.5</td>
<td>BNE, D</td>
<td>BNE to NOP</td>
</tr>
</tbody>
</table>

signal for latching values of pipeline registers into Pipeline Trace Register (PTR) after glitch injection. These signals are later analyzed to understand the effect of fault injection on pipeline registers.

**A. Implementation of Clock Glitcher**

We implemented the clock glitcher module on the controller FPGA (Xilinx Spartan-6 XC6SLX9) of the SAKURA-G board. The top part of Figure 7 shows the block diagram for the clock glitcher module of our setup. The glitcher module takes a glitch-free clock signal as an input from the pulse generator and generates a glitchy clock signal as an output. In this setup, we use 24 MHz clock input generated by pulse generator (Agilent 81110A). The generation of glitches is done with two variable phase shift modules [17].

The Control PC communicates with glitcher via USB communication link. We dynamically set the glitch parameters using Python scripts from the Control PC. The glitch injector then injects the glitch with these specified parameters when it receives the trigger signal. Using our setup, we can generate $T_{glitch}$ values between 3ns and 20ns with 100ps step-size. We also have a control on the time between the trigger event and the glitch injection, which allows us to target a specific pipeline stage of a given instruction.

**B. Implementation of Data Acquisition**

The bottom part of Figure 7 shows the block diagram for the data acquisition part of our setup. Our DUT is a LEON3 processor, which is implemented on the main FPGA (Xilinx Spartan-6 XC6SLX75) of the SAKURA-G board. For data acquisition, we utilize three hardware blocks: Debug Support Unit (DSU) of LEON3, Instruction Trace Buffer (ITB) of LEON3 provided as a part of GRLIB IP library [18] and a Pipeline Trace Register (PTR).

DSU is a non-intrusive on-chip debug core which controls the operation of the processor in the debug mode. In the debug mode, the processor pipeline is held idle and the software-visible processor state can be accessed by DSU. DSU can read/write the architectural registers and memory locations, load the program executable, start the execution of a program, and halt/continue the operation of the processor. It can also set/use breakpoints and watchpoints.

ITB is an on-chip memory buffer that stores the executed instructions. The buffer operates as a circular buffer, continuously capturing trace information until it is halted. It is located in LEON3 processor and is read out via DSU core. It traces the instruction address, instruction result, load/store data and address, and instruction timing information. We use typical 64-entry ITB for our experimental setup.

PTR stores the values of pipeline registers just after the faulty (glitchy) cycle. The operation of PTR is controlled by capture signal generated by the clock glitcher. When the capture signal is asserted, the contents of Pipeline Registers are copied into PTR. When the capture goes low, it will freeze the contents of the register and stored data can be read out via the DSU interface.

The Control PC runs the GRMON Debug Monitor [19] program. GRMON is used to manage/configure the hardware data acquisition cores, to load the executable of the cryptographic software and to start the execution of the program. It also supports reading out the processor state, managing breakpoint and watchpoint, reading the ITB and PTR. GRMON connects to the on-chip components via a JTAG Debug Link.

The DSU and ITB cores can only access the software-visible architectural registers and memory locations. Therefore, they can only show the fault effects on the software-visible architecture of the LEON3 processor. On the other hand, the PTR can access any signal in the processor pipeline, and thus, it can provide information about the fault effects on the micro-architecture of the LEON3 processor.

In conclusion, our experimental setup enables high-precision fault injection and detailed analysis of the fault effects on the pipeline register.

**VI. Experimental Evaluation of Instruction Level Software Countermeasures**

In this section, we verify our claims in Section IV with experimental results. For fault injection analysis, we use the experimental setup explained in the previous section. For each attack scenario and countermeasure investigated in Section IV, we launched a clock glitch injection campaign using our experimental setup. For each countermeasure, we show two different types of fault injection with glitch parameters and the observed faulty behavior. The following results are specific to the experimental setup that we used. The exact values of glitch parameters might change if another implementation of the processor is used, however the fault behavior will remain same.
Table VI shows the results of our experiments on the Instruction Duplication countermeasure. The first column of this table, shows the selected strategy of fault injection. Column 2 and 3 show the target of fault injection in the pipeline and the glitch width parameters, respectively. The last two columns list the faulty instructions and the observed fault effect.

In our experiments, we were not able to achieve the Scenario A.1., which requires injecting the same fault into both copies of the instruction with a single glitch injection. We observed that the effect of fault on the two LD instructions is different because they are in different stages of the pipeline. The table shows that we have successfully injected faults that result in other scenarios. We successfully created Scenario A.2. by injecting a single glitch fault into Cycle 2 or Cycle 4. By injecting glitches in Cycle 2, we were able to affect the operands fetched in the Stage A of LD1 instruction, and change the value stored in %g2. This fault injection also affected the Stage F of CMP, and changed this CMP instruction into a shift-right instruction (SRL). By injecting a fault in Cycle 4, we affected the Stage M of LD1 and Stage F. This fault injection caused a faulty value in the result of LD1 (%g2), and replaced the branch-on-not-equal (BNE) instruction into a NOP instruction. Therefore, the code did not jump to the error handling procedure although there was a mismatch between the results of LD1 and LD2. We also managed to observe Scenario A.2. by injecting multiple glitches in Cycle 3 and Cycle 7. We selected these cycles because stalls are occurring in some of the pipeline during these cycles.

Table VII demonstrates the experimental results on the Instruction Triplication countermeasure. We were not able to create Scenario B.1. and B.2.. Because all copies of the LD as the instruction are in different pipeline stages, we could not inject the identical fault value into the all copies. Using single fault injections in Cycle 9, we were able to create a faulty value in the protected register (%g4) and to convert BNE instruction to an OR instruction.

Table VIII shows the result of fault injection into the Instruction Parity countermeasure. As shown, since the parity countermeasure has many instructions for computation of the parity bit, there are several points in the program that are vulnerable to the fault injection. In this table, we show that the adversary can exploit single glitch injection to either corrupt multiple bits in the protected register’s value or the computation of the parity bit. For example, by injecting the glitch in Cycle 13, the condition codes and the opcode of the XOR instruction changes to 0000000100 which is a NOP instruction. By using multiple glitch injections, the attacker is also able to create fault in register %g3 and skip the comparison check of precomputed and computed parity bits. In case of injecting multiple glitches, the adversary can inject a fault in Cycle 2 to affect the value loaded in %g3 (0x0000 instead of the correct value if 0x000F). Then, another fault can change the opcode of the BNE to NOP in order to skip the fault detection.

Table VIII shows the result for the Instruction Skip coun-
We use a lookup-table-based approach to implement LED block cipher. Similar to the approach used by Guo et al. [20], we combined two consecutive nibbles into a unit for each lookup table. Each lookup table applies SC, SR, and MC operations on each unit. In total, we build 8 lookup tables each with 256 64-bit entries. We also precompute all round constants and store them in a lookup table of 32 64-bit entries, one entry for each round. On top of this unprotected implementation, we added an algorithm-level parity checking and an instruction-level duplication to make it fault-resistant.

We implement the **parity checking countermeasure** using the approach proposed by Karri et al. [7]. The basic idea is to compute the input state parity, predict the output parity, compute the actual output state parity and then finally compare the predicted parity to the actual output parity. A mismatch in these parity bits detects a possible fault.

Figure 5 explains the parity check countermeasure for LED. The input state parity is first computed. This input parity bit is modified according to the operations being performed in each step of the cipher. For the AddRoundConstant operation, the parity of the state is XOR’d with the parity of the corresponding RoundConstant (PRC). All the other transformations of the round (SubCells, ShiftRows and MixColumnSerial) are implemented by a lookup table (LUT), so the parity bit is modified according to the computed parity bits of the corresponding entry of lookup table (P(LUT)). This procedure is repeated for all the four rounds of each step. At the end of each step as the state is XOR’d with the shared key (SK), parity of the state is XOR’d with the parity of the shared key (P(SK)). This will be the predicted parity of the state at end of each step of the cipher. Also, at the end of each step (4 rounds), the parity of the state output from the step is computed. This computed parity is compared with the predicted parity bit at this stage. If there is a mismatch, a fault is detected. For implementing the parity check countermeasure, we add two more parity lookup-tables to store the parity bits. For each of the entries of the original lookup-table (which performs SC, SR and MC transformations), parity is computed and this is stored as one entry in the corresponding parity lookup-table. Similarly parity bits for the round-constants table are also computed and stored in another lookup-table. As the lookup-table entries are precomputed, the parity bits for the corresponding entries are also precomputed.

The second countermeasure that we implement is **Instruction Duplication** as proposed by Baregghi et. al in [2]. This countermeasure is implemented as explained in Section IV. Duplicating all the instructions would be very expensive, so we choose to duplicate only some part of the code. We duplicate the instructions in AddRoundConstant operation of the last round as this is the target of DFIA attacks for retrieving the secret key. Four instructions are duplicated and a compare instruction is added for each one of them. The instructions which were duplicated are shown below. First, the **LDD1** instruction loads the 64-bit LED state into the registers %g2 and %g3. Then **XOR** instructions are applied on the 64-bit state (%g2, %g3) and the round constant (%o4, %o5). Finally, the **STD** instruction writes the updated state to the memory.
C. DFIA on LED

In this section, we use the proposed pipeline analysis in order to break the software countermeasures applied on the LED algorithm. Our objective is to obtain the secret key of the LED-64. We used one plaintext value and one key value for our experiment. To obtain the key, first, we injected faults into the target program while it is running on the LEON3 processor. After collecting the faulty ciphertexts, we mount a biased fault attack, Differential Fault Intensity Analysis (DFIA) to retrieve a nibble of the key.

The objective of this attack is to retrieve the first nibble of the key. The target of fault injection for this attack is the output of AddRoundConstant function in the last round of the LED algorithm. To generate biased faulty values, we target the \texttt{lde} \[%fp + -56\], \%g2 instruction. In the protected version, this instruction is converted to the following code.

```c
; Instruction Duplication on LDD1
LDD \ [%fp\ -\ 56\], \%g2 ; LDD1
LDD \ [%fp\ -\ 56\], \%g4 ; LDD2
CMP \ %g2, \%g4 ;LDD2
BNE \ error
```

To bypass the Instruction Duplication countermeasure, we first observed the pipeline behavior while executing this part of the code (shown in Figure 9). Then, we identified the potential targets for affecting the result of LDD1 instruction and skipping the equality check using Scenario A.2 (Single Computation Fault-Single Instruction Fault) in Section IV. The target of fault injection is Cycle 6. Injecting fault into this cycle will result in injecting computation fault into LDD1 (W) and instruction fault into BNE (F).

Table IX shows the selected glitch parameters and the affected instructions. To obtain different faulty values, we changed the width of the glitch from 31.2ns to 33.6ns with the step size of 0.1ns. Using these fault injections, we obtained 5 faulty ciphertexts. Then, we applied the DFIA attack and successfully retrieved the two nibbles of the key. To retrieve the whole key, we need to apply the same strategy for other nibbles of the key.

VIII. CONCLUSION

In this work, we show that the existing countermeasures are not completely secure from fault attacks because their view of the microprocessor is limited to Instruction Set Architecture (ISA). We analyze the security of several state-of-the-art software countermeasures considering the micro-architectural aspects of the microprocessor such as pipeline effects. Then, using the result of our analysis, we experimentally demonstrate that even formally-verified software countermeasures can be broken by low-cost, single clock glitch injections. Therefore, the assumptions of the existing software countermeasures on the capabilities of the adversary need to be revised to include the micro-architectural aspects. We also conclude that software countermeasures alone are ineffective against fault attacks by an adversary with knowledge of the hardware.

IX. ACKNOWLEDGMENT

This research was supported in part through the National Science Foundation Grant 1441710 and 1314598, and in part through the Semiconductor Research Corporation.

REFERENCES


