Improving Fault Attacks on Embedded Software using RISC Pipeline Characterization

Bilgiday Yuce, Nahid Farhady Ghalaty, Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
{bilgiday,farhady,schaum}@vt.edu

Abstract—A fault attack becomes more efficient when the fault behavior, the response of a device to a fault injection, is precisely understood. In this paper, we present a methodology for fault attacks and their analysis on pipelined RISC processors. For complex hardware structures such as microprocessor pipelines, modeling the fault behavior can become challenging. By analyzing the structure of the RISC pipeline, we obtain insight into the most likely faults, and we are able to pinpoint the most sensitive points during execution of a cryptographic software program. We use this result to apply a recent class of fault injection attacks, so-called biased fault injection attacks, to two different software implementations of AES. Our target microprocessor is a 7-stage pipeline LEON3, mapped into a Spartan6 FPGA. The paper explains the methodology, the fault injection setup, and the fault analysis on the embedded software design of AES. Our results are useful for embedded software designers who have a need to understand the fault attack sensitivity of their implementation, as well as for security engineers who are in charge of improving countermeasures, in hardware or in software, against fault attacks.

Index Terms—Fault Analysis; AES; RISC; Pipeline, Biased Fault.

I. INTRODUCTION

It has been known for several years that fault attacks are equally applicable to software as well as to hardware. In the case of a fault attack on software, the fault injection is aimed at the microprocessor hardware that executes the software, although the logical target of the fault attack is the software. The fault effects, such as faulty ciphertext or the input/output behavior, are observed indirectly as faulty instructions, i.e. instruction executions for which the microprocessor is experiencing a fault. For example, several authors have demonstrated the use of under-powering microprocessor hardware in combination with clock glitch injection, in order to temporarily alter the execution of an instruction. The typical microprocessor fault responses observed from such attacks generally fall in one of two categories: data errors or instruction-fetch errors. Data errors alter the values computed by the embedded software, but they do not change the program flow. Instruction-fetch errors, on the other hand, change the meaning of the program through changes in the instruction opcodes, as well as through instruction-skip, or instruction re-execution. The fault analysis of embedded software is cumbersome, because the fault behavior is defined by the microprocessor hardware, rather than by the software application. Indeed, data errors as well as instruction-fetch errors resulting from fault injection depend on the detailed architecture of the microprocessor.

A fault attack relies on a fault model, a high-level assumption on the actual fault. Traditional fault attack literature extensively relies on such generic fault models, such as bit-flip, random-byte, etc. Fault models for microprocessor instructions are relatively recent. In their fault attack overview, Karaklajik et al. mention attacks based on C-safe errors, M-safe errors, and faults that alter the program control flow. C-safe and M-safe error attacks rely on the detection of a fault during computation (C-error) or memory (M-error) operation, while program control flow attacks rely on a generic instruction-skip or -duplication technique. All of these attacks have in common that they only require a very limited understanding of the internal data-flow of the cryptographic software as it executes on the processor.

Several authors have proposed a black-box fault analysis of microprocessor hardware. Balasch et al. discuss the case of an AVR microcontroller in combination with clock glitch injection, in order to temporarily alter the execution of an instruction. The typical microprocessor fault responses observed from such attacks generally fall in one of two categories: data errors or instruction-fetch errors. Data errors alter the values computed by the embedded software, but they do not change the program flow. Instruction-fetch errors, on the other hand, change the meaning of the program through changes in the instruction opcodes, as well as through instruction-skip, or instruction re-execution. The fault analysis of embedded software is cumbersome, because the fault behavior is defined by the microprocessor hardware, rather than by the software application. Indeed, data errors as well as instruction-fetch errors resulting from fault injection depend on the detailed architecture of the microprocessor.

A fault attack relies on a fault model, a high-level assumption on the actual fault. Traditional fault attack literature extensively relies on such generic fault models, such as bit-flip, random-byte, etc. Fault models for microprocessor instructions are relatively recent. In their fault attack overview, Karaklajik et al. mention attacks based on C-safe errors, M-safe errors, and faults that alter the program control flow. C-safe and M-safe error attacks rely on the detection of a fault during computation (C-error) or memory (M-error) operation, while program control flow attacks rely on a generic instruction-skip or -duplication technique. All of these attacks have in common that they only require a very limited understanding of the internal data-flow of the cryptographic software as it executes on the processor.

Several authors have proposed a black-box fault analysis of microprocessor hardware. Balasch et al. discuss the case of an AVR microcontroller in combination with clock glitch injection, in order to temporarily alter the execution of an instruction. The typical microprocessor fault responses observed from such attacks generally fall in one of two categories: data errors or instruction-fetch errors. Data errors alter the values computed by the embedded software, but they do not change the program flow. Instruction-fetch errors, on the other hand, change the meaning of the program through changes in the instruction opcodes, as well as through instruction-skip, or instruction re-execution. The fault analysis of embedded software is cumbersome, because the fault behavior is defined by the microprocessor hardware, rather than by the software application. Indeed, data errors as well as instruction-fetch errors resulting from fault injection depend on the detailed architecture of the microprocessor.

Fig. 1. The proposed fault attack methodology uses a model of a RISC pipeline to enable more advanced fault attacks on software such as DFIA and FSA.
they confirm that it is possible to obtain biased faults on microprocessor hardware. However, their analysis does not reveal a systematic attack methodology.

In this paper, we introduce a microprocessor architecture fault sensitivity model for a RISC pipeline, and we demonstrate its application for a biased-fault attack. Figure 1 illustrates the relation of the proposed fault model to fault attacks. Without the model, traditional fault attacks such as DFA, C-safe/Errors or Flow Errors are still possible. With the fault sensitivity model, however, a new class of attacks based on fault bias becomes possible. These attacks include Fault Sensitivity Analysis (FSA) [6], [7], Non-Uniform Fault Analysis [8], [9], and Differential Fault Intensity Analysis (DFIA) [10]. This RISC pipeline fault sensitivity model is created through timing simulation of a gate-level model of the RISC processor. When a cryptographic application executes on the RISC pipeline, the fault sensitivity model will show what instructions and what clock cycles are most suitable (or most sensitive) to fault injection, in order to mount a fault attack. We will illustrate the technique by applying DFIA on AES software, and we will validate the proposed model using a prototype implementation on FPGA.

The novelty of our work can be summarized as follows.

1) This paper presents a fault sensitivity model for a RISC pipeline. Our method requires timing analysis of a gate-level netlist of the RISC pipeline. Although an adversary does not necessarily have this gate-level model available, we point out that there are only a finite number of RISC processor designs commonly in use. Hence, it is a reasonable assumption that this data is either public knowledge, or else can be easily obtained, in particular for embedded processors (eg. ARM). In this paper, we assume a fault trigger mechanism that causes timing errors in a microprocessor. While we do not claim that our fault model covers every possible fault injection mechanism, we believe that the methodology of fault sensitivity characterization is generic and will apply to other cases as well.

2) We show how the microprocessor fault sensitivity model can be combined with a cryptographic software target in order to construct a biased fault attack. We show that typically, several different clock cycles can be targeted, and that the fault sensitivity model indicates what clock cycles are most suitable for an attack.

3) Compared to earlier research on fault analysis for microprocessors, we study a complete attack. We use the proposed technique for differential fault intensity analysis (DFIA [10] of AES on a 7-stage RISC pipeline. An important conclusion is that a given fault objective (eg. a biased fault in the AES state) can be obtained through multiple avenues. We also study the effects of pipeline stalls and data dependencies, and show that this leads to additional opportunities for fault injection.

4) We implement the attack on a LEON3 RISC processor [11] mapped on an FPGA. The fault attack environment enables fault injection as well as detailed analysis of the fault effects on the RISC pipeline. This enables us to simultaneously study fault effects at instruction-level as well as at micro-architecture level.

The paper is organized as follows. In Section II, we discuss the fault behavior in a RISC pipeline, and we discuss the main characteristics of the proposed fault model. In Section III, we briefly review the AES software target under consideration. Section IV discusses a DFIA fault attack on AES software. Section V explains how we construct the fault sensitivity model for a RISC pipeline. Section VI demonstrates how to construct a biased fault attack by combining the AES software analysis and the fault sensitivity model of the RISC pipeline. Section VII presents the experimental setup. Section VIII presents the measurement results we obtained from a hardware prototype. Finally, Section IX concludes the paper.

II. FAULT BEHAVIOR IN A RISC PIPELINE

The experiments in this paper are done on a 7-stage LEON3 RISC pipeline. We will first review the main characteristics of LEON-3. Next, we describe how injected faults are propagated in this pipeline.

A. The LEON-3 Processor

LEON3 is a 32-bit SPARCv8-compliant RISC processor with (a modified) Harvard architecture, extensible through an AMBA 2.0 bus system. The core is distributed as a synthesizable VHDL model by Aeroflex Gaisler, together with a large library of peripherals and debug support modules. The core is highly configurable in capabilities and performance.

Figure 2 shows the block diagram of the LEON-3 configuration used in our research. We use a configuration with a 7-stage integer pipeline and 64 KB of on-chip RAM memory. To evaluate cache effects, we also configured 4KB direct-mapped caches for Instructions and Data, each with a line size of 32-byte. We note that, although the LEON3 follows a Harvard architecture, both caches map into the same 64-KB on-chip memory.

For this paper, the seven stages of the LEON-3 integer pipeline are of particular interest. The purpose of each stage is as follows.

1) Fetch (F): An instruction is fetched from the memory or instruction cache, and copied into the instruction register (IR).

2) Decode (D): The instruction from the IR is decoded. For branch and CALL instructions, the target next-address is computed.
3) **Register Access (A):** The instruction operands are read from the register file or from internal forwarding paths.

4) **Executed (E):** ALU, logical, and shift operations are performed. For memory-load and store, the target data address is computed.

5) **Memory (M):** The memory-access part of an instruction is completed (data read or data write into memory).

6) **Exception (X):** This stage resolves traps and interrupts.

7) **Write-back (W):** The results of the instruction execution are written into the register file.

**B. Fault injection using clock glitches**

We inject faults into the microprocessor by clock glitches. Figure 3 shows the effect of a glitch on the clock signal. A clock glitch will temporarily shorten the clock cycle period from $T_{clk}$ to $T_{glitch}$, thereby causing a timing violation of the digital logic. A clock glitch can be defined with two parameters, namely, glitch offset ($T_o$) and glitch width ($T_w$). The glitch offset specifies the delay of the glitch pulse from the positive edge of the nominal clock. The glitch width specifies the duration of the glitch pulse. The period $T_{glitch}$ of a glitched clock is the sum of the glitch width and glitch offset.

**C. Fault propagation in the RISC pipeline**

Figure 4 demonstrates the effect of a clock glitch on the execution of a seven-stage RISC pipeline. Observe that there are no branch instructions in this example. Now, assume that a clock glitch is injected in the RISC clock during the fetch stage of the 7th instruction. Since a clock glitch affects the entire pipeline, it will potentially affect up to seven instructions, each at a different point of execution. Careless fault injection therefore could cause a catastrophic and complex behavior in the pipeline. On the other hand, different pipeline stages may have a different critical path (or fault sensitivity). A fault will only occur if the glitch period is shorter than the critical path of a particular pipeline stage. Hence, by carefully controlling the clock glitch period, the effective fault can be limited to selected, slower pipeline stages.

In Figure 4, we assume that the execute stage of the pipeline has the highest fault sensitivity, or in other words, the longest critical path. Therefore there exists a clock glitch period that will cause a fault to happen only in the execute stage of the 4-th instruction. From thereon, the fault propagates further through the pipeline. Any of the instructions with a data dependency on instruction 4 can be affected by this fault as well; this could be either through forwarded faulty data, or else through faulty data passed through a register.

In addition to this basic mechanism of fault propagation, we make two additional observations. First, the nature of the fault depends on its location: faults in the fetch and decode stages will cause instruction errors, while faults in the later pipeline stages will cause data errors. Second, the impact of fault-injection in a RISC pipeline will not only depend on the fault sensitivity of individual pipeline stages, but also on the dynamic effects of pipeline stalls. In Figure 4, we assumed that instruction 6 is a memory-load operation with a data dependency on instruction 4. This will cause the execution of instruction 7 to stall until the memory access of instruction 6. If we would inject a clock glitch in Cycle 10, then only instructions 6, 5 or 4 would be affected. We will use this effect to fine-tune the fault injection in the target embedded software.

The next section briefly describes the two AES software implementations that we analyzed.

**III. Advanced Encryption Standard Target on LEON-3**

In this research, we aim at demonstrating DFIA of the standard AES block cipher, executing as a software implementation on a LEON-3 pipeline. We studied two different AES implementations. The first one is the byte-oriented reference implementation, while the second one uses the well-known AES TBOX implementation [12]. We will use the terms ‘SBOX AES’ and ‘TBOX AES’ to distinguish these two implementations. The TBOX design is a word-oriented implementation. The TBOX design is very well suited for 32-bit processors, and collapses several steps of an AES round (comprising SBOX-lookup, Shiftrows and Mixcolumns) into a single lookup. A full AES round can then be computed with sixteen table-lookups, as well as sixteen XOR operations.

Our objective is to mount a biased fault injection attack on the output of AES round 9, as would be typically done in DFIA.
For the SBOX design, we consider the AddRoundKey function as the last step in the execution of round 9. The reference implementation in C shows 16 statements of the following form, with state the state variable, and Roundkey an array with roundkey values.

\[
(*\text{state})[i][j] \leftarrow \text{RoundKey}[K];
\]

The LEON-3 assembly code for this C includes 5 instructions. The target for biased fault injection is register %g1, as well as all instructions that contribute to this value. For example, a biased fault in %o5 would be equally useful in reaching this objective.

\[
\begin{align*}
\text{ld} & \ [ \%03 + 0xb0 \ ], \%o4 \ //^*\text{state} \\
\text{ldub} & \ [ \%00 + 0xb \ ], \%o5 \ //\text{RoundKey}[K] \\
\text{ldub} & \ [ \%04 + 0xb \ ], \%g1 \ //(*\text{state})[i][k] \\
\text{xor} & \ %g1, \%o5, \%g1 \\
\text{stb} & \ %g1, [ \%o4 + 0xb ]
\end{align*}
\]

For the attack on the TBOX design, we consider the expression that generates (one quarter of) the round-9 output state. It includes four TBOX-table lookups, which are all added together with a roundkey to produce the round-9 output t0.

\[
t0 = T_{E0} \ [ s0 >> 24 \ ] \ \\
Te1[(s1 >> 16) & 0xff] \ \\
Te2[(s2 >> 8) & 0xff] \ \\
Te3[s3 & 0xff] \ \\
rk[36];
\]

In this case, the assembly code is much more complex, especially if the code is compiled with optimization (We used -O2). The following is a snippet of instructions just as they produce the output t0.

\[
\begin{align*}
\text{ld} & \ [ \%07 + 0xb0 \ ], \%o5 \ //Te3 \\
\text{xor} & \ %g1, \%o2, \%g1 \ //^*\text{Te2} \\
\text{ld} & \ [ \%fp + 0x4c \ ], \%l2 \ //\text{roundkey \ ptr} \\
\text{xor} & \ %g1, \%o5, \%g1 \ //^*\text{Te3} \\
\text{ld} & \ [ \%12 + 0x98 \ ], \%o4 \ //\text{roundkey} \\
\text{xor} & \ %g1, \%o4, \%l6 \ //^*\text{roundkey}
\end{align*}
\]

Hence, by studying the software, we are able to identify a set of instructions that can contribute to a biased error in %g1. However, as discussed in the previous section, we cannot blindly inject faults in thehope of hitting an instruction that creates fault bias on %g1. We will use a more effective method, based on white-box analysis of the pipeline. This is the topic of the next Section.

IV. AES SOFTWARE ANALYSIS ON RISC PIPELINE FOR DFIA

To determine the potential attack points in the execution of the AES, we apply an architectural analysis methodology. Our methodology relies on analyzing the AES behavior in the LEON3 pipeline.

We will use DFIA, and we will first recall the steps of a DFIA. A key step to apply DFIA in a RISC pipeline is to identify the fault sensitivity of individual instructions. We will discuss these sensitivities for the different instruction types found in AES (Section IV.B). Armed with this insight, we can then study the AES program execution in the RISC pipeline and identify suitable attack points for DFIA (Section IV.C). As a result of this architectural analysis, we have a set of \text{(instruction, pipeline stage)} pairs that we can attack to obtain the required faults. In the following sections we will demonstrate this methodology for a DFIA attack on a SBOX-based AES software implementation.

A. Differential Fault Intensity Analysis (DFIA)

This section provides a brief introduction to the DFIA attack. DFIA relies on the biased fault model, which assumes a gradual change in fault behavior as a result of a gradual change in fault intensity. Fault intensity (FI) is the strength by which the adversary is pushing the circuit out of its normal operating conditions. In clock glitching, the fault intensity is controlled by increasing/decreasing the glitch period \( T_{\text{glitch}} \) (Fig. 3).

DFIA has two phases. In the first phase, DFIA applies a plaintext to a given block cipher. Then, the adversary applies a fault intensity to the input of the non-linear layer (i.e., input of SBOX) in the block cipher and observes the faulty ciphertext. The adversary, then increases the fault intensity and repeats the first step multiple times. The second phase is the post-processing of the captured faulty ciphertexts. In this phase, the adversary computes some hypothetical intermediate variables for every possible key guess under each fault intensity. The final step is to find the correct key. The correct key must reveal the biased fault behavior. Based on the DFIA's assumption, increasing the fault intensity will cause small changes in the intermediate value. Therefore, the correct key is the one that carries this behavior in the hypothetical intermediate values. As a result, the correct key is the one for which the Hamming Distance of the intermediate values is minimal.

Ghalaty et. al explained the DFIA attack on the hardware implementation of AES in [10]. In this attack, the target of fault injection is the output of round 9 of the AES algorithm. We will have a similar strategy for the DFIA attacks on the software implementations of AES. In order to inject fault in the output of round 9 in the AES algorithm, the adversary can target any operation in round 9 (ShiftRows, MixColumns and AddRoundKey). Since these operations contain many instructions, for simplicity, we choose to explain the fault injection just in the AddRoundKey function.

B. Analyzing RISC Pipeline for Biased Data Errors

To apply DFIA to a RISC pipeline, we need to induce biased data errors into selected instructions. We will therefore analyze each stage of the RISC pipeline for the possibility of creating a biased data error. We will do this for three different instruction types: logical instructions, memory-load and memory-store. These are the three instruction types that make up the critical operations for AES.
1) **Pipeline Stages F and D:** Affecting the pipeline stages $F$ and $D$ of an instruction changes the meaning of the instruction and causes instruction-errors. Therefore, these stages cannot contribute to DFIA.

2) **Pipeline Stage A:** Disturbing the stage $A$ might induce biased faults in the operands of the instruction, which are fetched from the register file. Using faulty operands in the following pipeline stages will cause faulty computations (i.e., data errors). However, the induced data error can be biased (i.e., small) or random (i.e., big) depending on the instruction type. For *logical* instructions, small change in the operands yields a small change in the result of the instruction. For *load* and *store* instructions, faulty operands will affect the address calculation. Therefore, the instruction will *load/store* a random data. As a result, the pipeline stage $A$ is a valid fault-injection stage for *logical* instruction, but is not valid for *load* and *store* instructions.

3) **Pipeline Stage E:** A fault in this stage will affect the data calculation for *logical* instructions while affecting address calculation for *load* and *store* instructions. Therefore, the same arguments made for the stage $A$ apply to the stage $E$.

4) **Pipeline Stage M:** In this stage, *logical* instructions do not make any computations. Load and *store* instructions do memory transfer in this stage. We can create data-errors by violating the timing paths between the pipeline registers and memory ports. Thus, the stage $M$ is a valid fault-injection stage for *load* and *store* instructions.

5) **Pipeline Stage X:** In this stage, exceptions are handled. None of the instructions does a computation. Therefore, this stage is not useful for DFIA.

6) **Pipeline Stage W:** Logical and *load* instructions write their results into the register file in this stage. We can violate the timing paths between the register file and the pipeline register to induce faults. Then, it is a valid stage for *load* and *logical* instructions. Store instructions are idle in this stage, and thus, they are not a valid target for DFIA.

C. **Analyzing RISC Pipeline for SBOX AES**

In this section, we analyze the instructions of AES and investigate how to mount a DFIA attack on them. Table I shows the instructions in the AddRoundKey function of the SBOX implementation of AES in round 9. The three instructions in range of $[a0001ac4, a0001acc]$ are the instructions that load the key byte, the state byte and finally XOR the key and the state bytes.

As shown in Table I from the output of the %g1 operand of STB6, there are multiple dependencies to earlier instructions. XOR5 has a dependency on STB6 through %g1; LDUB4 and LDUB3 have dependencies on XOR5 through %g1 and %o5; and LDI2 has a dependency on LDUB4 through the data memory.

Figure 5 shows the pipeline behavior for these seven instructions. This figure shows the sequence of instructions in the pipeline. Each column of the figure shows the status of a different pipeline stage. The rows show the clock cycle and the status of each instruction in pipeline.

The instructions with red circle show the fault propagation path. The arrows in the figure show the dependency between the instructions in the pipeline. For example, XOR5 requires to have access to both %o5 and %g1. %o5 will be ready in the LDUB3(M) and %g1 is ready in the LDUB4(M). The blue instructions are the stages of each instruction that would be the preferred target for biased fault injection. The blue instructions are obtained using the information in Section IV.B. Now we enumerate the possible cycles for fault injection that will result in a biased fault in %g1.

1) **Fault injection in Cycle 0 through 5:** These fault injection attempts will not result in an effective biased fault in %g1, so we refrain from fault injection in those cycles.

2) **Fault injection in cycle 6:** In case of injecting fault into cycle 6, the only valid fault can be injected into the LDUB3(M). The faulty results can be biased since it is in the memory stage of load instruction and has a direct
effect on the value of the output of round 9. However, the attacker will need to select a fault intensity that only affects this instruction.

3) **Fault injection in Cycle 7:** As shown in the pipeline behavior, due to the data dependency of LDUB4 and XOR5, there will be a stall in the pipeline. If the attacker triggers the glitch in cycle 7, he potentially affects the STB6(F), XOR5(A), LDUB4(M), LDUB3(X). An important observation in this analysis is the use of pipeline stalls as a source of injecting biased faults in the data.

4) **Fault injection in Cycle 8:** If the attacker triggers the glitch in cycle 8, he has the potential to affect the STB6(D), STB6(A), XOR5(E) and LDUB3(X). Due to the generated stall in the previous cycles, the STB6(D) is doing the same operation as the cycle 7. Therefore, the timing of STB6(D) is very close to 0. As a result, the attacker can only inject fault into STB6(A), XOR5(E) and LDUB3(X) which are all the potential targets for DFIA.

To summarize, in this section we demonstrated how to identify the clock cycles that can yield a biased error in the round-9 output of an AES program. In the next section, we discuss how to select the proper fault intensity.

V. **TIMING CHARACTERIZATION OF RISC PIPELINE**

A fault injection attempt into the operation of a microprocessor potentially affects all of the pipeline stages depending on the applied clock glitch period (i.e., fault intensity). This causes uncertainty in the injected faults. In this section, we determine the fault sensitivity model of the RISC pipeline.

We use gate-level simulation to perform this fault sensitivity analysis. Gate-level simulation has been employed by Sugawara et al. [13] and Barenghi et al. [14] to create better fault models of AES ASIC implementations against FSA. Our work is different in two aspects. First, we characterize the timing of a complex 7-stage RISC pipeline. Second, we combine the timing characterization results with the architectural information of the processor for a better understanding of the fault effects in the execution of a software.

In the next sections, we provide the details of our gate-level simulation approach and a RISC pipeline fault sensitivity model for the LEON3 processor.

A. **Gate-Level Simulation**

For a given target instruction, we characterize the timing of each pipeline stage by extracting its critical path delay via gate-level simulation. First, we simulate the microprocessor while it is running a test program. The program includes the target instruction surrounded by nop instructions. Then, we analyze the signal activity within the fan-in cone of pipeline registers (see Fig 6) to determine critical path values. Repeating this analysis for each target instruction, we obtain the microprocessor fault sensitivity model. The model consists of the critical path information of each pipeline stage for each target instruction. We can represent the characterization results in a table similar to Table II.

### B. Timing Characterization Results

**Table II** shows the microprocessor fault sensitivity model for our LEON3 implementation. We created this model by simulating the potential target instructions of DFIA on SBOX and TBOX AES programs. For SBOX AES, we can attack STB, LDUB, and XOR instructions (Sections III and IV). For TBOX AES, the potential target instructions are LD, LDI, and XOR instructions. In Table II, we also include the instructions that surround the target instructions in the AES programs (ST, SLL, SRL) as they can be affected by a fault injection attempt into the target instructions.

In the gate-level simulation, we simulated each instruction one time with arbitrarily selected operands. In other words, we assume that, for a given instruction, the data-dependency of a pipeline stage’s propagation delay is independent of the data values. In our case, this is a valid assumption because the data dependencies on the instructions (ST, LD, and XOR) have a bit-wise character. Thus, we do not have to deal with data-dependent effects such as carry propagation.

Furthermore, we also point out that DFIA does not rely on data-dependent fault-sensitivity, but rather on fault bias.

For other fault attack strategies or cryptographic software implementations, one might need to simulate the instructions for multiple data sets and to apply statistical processing to create the processor fault sensitivity model. For instance, FSA utilizes the data-dependency of the timing. Therefore, we need to apply multiple gate-level simulations with different operand values to include this information in our microprocessor fault sensitivity model. In the next Section, we combine the analysis of AES from Section IV with the pipeline fault sensitivity model developed in this section. This will result in a DFIA for AES software.
VI. DFIA ATTACKS ON AES SOFTWARE

In this section, we demonstrate how we construct a biased fault attack by combining the micro-architectural analysis (i.e., microprocessor fault sensitivity model) and architectural analysis (i.e., the AES software analysis) of the RISC pipeline.

A. A DFIA Attack on SBOX-based implementation of AES

To construct a DFIA attack on SBOX AES, we use the software analysis from Section IV.C and the timing characterization results from Section V.B. By combining them for a specific clock cycle, we define a valid $T_{\text{glitch}}$ range (i.e., fault intensity range), in which an adversary can only inject data errors. Figure 7 demonstrates a fault sensitivity analysis for Cycle 7 of the SBOX AES program (Table II) to define a valid $T_{\text{glitch}}$ range.

Figure 8 shows the instructions in the pipeline and the critical delay values of the pipeline stages during Cycle 7 of the AES SBOX program. The adversary can violate LDI7(F), SLL2(D), XOR5(A), Stall(E), LDUB4(M), and XOR8. Because of the existence of a stall in the pipeline, Execute (E) stage will not be affected. Among the remaining instructions, only LDUB4(M) and XOR5(A) can contribute to DFIA (Section IV.C). Therefore, the other instructions are invalid fault injection targets. The upper bound of the valid $T_{\text{glitch}}$ range is the maximum critical delay of the valid fault injection targets. For our example, this value is the critical path of LDUB4(M) and it is 7.2ns. $T_{\text{glitch}}$ values greater than this value do not induce any errors. The lower bound of the valid $T_{\text{glitch}}$ range is the maximum critical delay of invalid fault injection targets. This corresponds to the critical delay of LDI2(W), which is 4.45ns, for our example. As a result, the valid $T_{\text{glitch}}$ range is from 4.45ns to 7.2ns.

B. A DFIA Attack on TBOX-based implementation of AES

Table III shows some instructions in the AddRoundKey function of the TBOX AES in round 9. The instructions in range of [a0002b48, a0002b6c] are the instructions that are doing shift and XOR operations on the state word (%g1). Instruction a0002b60 load the key word, and instruction a0002b66 XORs the key and the state words. Figure 8 shows the behavior of these instructions in the pipeline. Similar to the SBOX AES, there are some data dependencies between the instructions for TBOX as well which are shown by red circles in the graph. All the data dependencies can be solved by forwarding technique, except the dependency from LDI7 to XOR8. Therefore, the pipeline will have a stall in cycle 26. Due to the cache miss processing for LDI5, the pipeline will be held still until the data is ready. the HOLD cycles extend the time of the instruction causing the miss by the corresponding number of cycles. Based on the analysis in Section IV.B, the potential points of observing biased fault is shown by blue instructions in the pipeline.

The fault attack targets can be defined by the same strategy for SBOX AES. The potential points of fault attack can be any column in the Figure 8. In order to perform an efficient

---

### Table III

<table>
<thead>
<tr>
<th>Inst. #</th>
<th>Inst. Address</th>
<th>Inst. Opcode</th>
<th>Inst. Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD1</td>
<td>a0002b48</td>
<td>d4 04 c0 0b</td>
<td>ld [ %l3 + %o3 ], %o2</td>
</tr>
<tr>
<td>SLL2</td>
<td>a0002b4c</td>
<td>99 2b 20 02</td>
<td>sll %o4, 2, %o4</td>
</tr>
<tr>
<td>LD3</td>
<td>a0002b50</td>
<td>03 03 0c 0c</td>
<td>ld [ %o7 + %o4 ], %o5</td>
</tr>
<tr>
<td>XOR4</td>
<td>a0002b54</td>
<td>82 18 40 0a</td>
<td>xor %g1, %o2, %g1</td>
</tr>
<tr>
<td>LD5</td>
<td>a0002b58</td>
<td>c4 07 0c 4d</td>
<td>ld [ %p + 0x4c ], %12</td>
</tr>
<tr>
<td>XOR6</td>
<td>a0002b5c</td>
<td>82 18 40 0d</td>
<td>xor %g1, %o5, %g1</td>
</tr>
<tr>
<td>LD7</td>
<td>a0002b60</td>
<td>d8 04 0a 98</td>
<td>ld [ %l2 + 0x98 ], %o4</td>
</tr>
<tr>
<td>XOR8</td>
<td>a0002b64</td>
<td>ac 18 40 0c</td>
<td>xor %g1, %o4, %16</td>
</tr>
<tr>
<td>SRL9</td>
<td>a0002b68</td>
<td>9b 37 60 0c</td>
<td>srl %i5, 0x5e, %o5</td>
</tr>
<tr>
<td>SRL10</td>
<td>a0002b6c</td>
<td>83 37 20 18</td>
<td>srl %i4, 0x18, %g1</td>
</tr>
</tbody>
</table>
attack, we need to inject the fault in the way that will only affect the circled pipeline stages only. In order to find the fault injection location and the valid fault intensity range, we need to look into the timing characterization of each of the potential targets.

There are 32 opportunities to inject fault into AddRoundKey of Tbox since we can inject fault in any cycles of this function. In Figures 9(a)-9(d), we show four of these targets. Based on analyzing the instructions that might be effected by each of these targets, we will choose the specific target and fault intensity to inject the fault.

1) **Fault Injection in Cycle 6** (Figure 9(a)): The only blue operations in cycle 6 are the XOR4(E) and LD3(M). Figure 9(a) shows the length of critical path for each operation in cycle 7. The largest critical path is for LD3(M), which is equal to 7.58ns. By gradually increasing the fault intensity, we will affect the LDI5(A) which is 5.45ns. Since LDI5(A) is invalid fault injection target, Figure 8, the maximum fault intensity can be 5.45ns.

2) **Fault Injection in Cycle 25** (Figure 9(b)): By injecting fault in this cycle, we are allowed to affect XOR6(E), XOR4(X) and LD3(W). The largest critical path belongs to LDI5(M). Since LDI5(M) is not blue, we cannot inject fault into this cycle.

3) **Fault Injection in Cycle 27** (Figure 9(c)): In this figure, we are allowed to affect XOR8(A) and LD17(M). We can only affect LD17(M) with the fault intensity of 7.58ns. The maximum fault intensity will be 5.91ns since we do not wish to affect the SRL10(F).

4) **Fault Injection in Cycle 28** (Figure 9(d)): In this figure, we are allowed to effect XOR8(E), XOR6(W). We can affect the XOR8(E) and the maximum intensity that we can apply is the delay of SRL9(A) (4.7ns), since SRL9(A) is invalid target of fault injection in Figure 8.

### VII. Fault Injection and Analysis Setup

In this section, we describe the measurement setup that enabled us to experimentally verify our method. Figure 10 shows the high-level diagram of our experimental setup. The setup consists of a PC, a Device Under Test (DUT), and a fault injection module. The PC manages the fault injection process by controlling and configuring both the fault injection module and DUT. The DUT executes a cryptographic algorithm and sends a trigger signal to the fault injection module. Trigger signal synchronizes the fault injection module and the DUT. It enables us to have a cycle-accurate control over the glitch timing. The fault injection module consists of two blocks. The clock glitch controller keeps the glitch parameters and controls the clock glitch injector. A predefined number of clock cycles after the trigger signal, the clock glitch injector injects the glitch specified by the glitch controller.

A typical fault injection experiment consists of the following steps:

1) The controlling PC configures the fault injection module. It writes the number of wait cycles after the trigger event, the glitch width, and glitch offset into the clock glitch controller registers.
2) The controlling PC configures and initializes the DUT.
3) The controlling PC arms the clock glitch injector.
4) The controlling PC starts the execution of the DUT.
5) The DUT sets the trigger signal when it reaches the predefined point in its execution flow.
6) The clock glitch injector injects the glitch with the parameters specified in the Step 1.
A. Implementation of Clock Glitch Injector

We implemented the fault injection module on the controller FPGA (Xilinx Spartan-6 XC6SLX9) of the SAKURA-G board [15]. To inject glitches into the nominal clock, we first generate two phase-shifted clocks from the nominal clock, and then, we combine these three clock phases (Figure 11 and 12) by applying logical operations [16], [17]. We use the Digital Clock Manager (DCM) blocks of the FPGA to generate the shifted clock phases. In this work, we use a 24-MHz nominal clock generated by an external pulse generator (Agilent 81110A). Using the dynamic phase-shifting feature of Xilinx DCM blocks and partial reconfiguration approach introduced by O’Flynn et al. [14], we can generate \( T_{\text{glitch}} \) 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. We can dynamically set all of the glitch parameters via commands from the PC.

B. Implementation of Data Acquisition

Figure 13 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, and an on-chip logic analyzer core (LOGAN) provided as a part of GRLIB IP library [18].

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 idle and the software-visible processor state can be accessed by DSU. DSU can read/write the architectural registers and memory locations, can load the program executable, can start the execution of a program, and can halt/continue the operation of the processor. It can also set/use breakpoints and watchpoints.

ITB is a circular buffer that stores the executed instructions. It is located in LEON3 processor and read out via DSU core. It traces the instruction address, instruction result, load/store data and address, and timing information for the instruction. We use an 64-entry ITB for our experimental setup.

LOGAN implements an on-chip logic analyzer core and enables us to trace arbitrary signals inside LEON3 processor. It consists of a circular buffer to store the traced sampled signals, and a trigger block to detect a user-defined pattern on the sampled signals. When it is armed, it continuously samples a set of signals until it detects a user-defined trigger condition. The trigger condition makes LOGAN stop tracing. Then the traced data can be read out. LOGAN core can store 4096 samples of 256 signals.

The controlling PC uses GRMON Debug Monitor [19] program to manage/configure the hardware data acquisition cores, to load the executable of the cryptographic software, to start the execution of the program, and read out the processor state. 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 LOGAN core can access any on-chip signal, and thus, it can provide information about the fault effects on the micro-architecture of the LEON3 processor. For instance, we can directly observe the fault effects on the pipeline registers through the LOGAN core.

In conclusion, we developed an experimental setup that enables high-precision fault injection and detailed analysis of the fault effects on the RISC pipeline.

VIII. RESULTS

In this section, we verify our claims with experimental results, which were collected via fault injection experiments on the LEON3 FPGA implementation. In Section VIII.A, we validate the proposed microprocessor fault sensitivity model by comparing the gate-level simulation results and on-board fault injection results. In Section VIII.B, we show feasibility and efficiency of the developed DFIA attacks under two different assumptions on the capabilities of the adversary.

A. Experimental Verification of Fault Sensitivity Model

Based on our previous analysis on the pipeline behavior of the AES (Section IV) and the timing characterization of the LEON3 pipeline (Section V), we defined targets for the fault injection on the SBOX and TBOX implementations of AES.

Table IV shows the results of biased fault injection for the SBOX and TBOX implementations of AES algorithm. We have two target cycles (i.e, Cycle 6 and 7) for the TBOX AES. For the SBOX AES, we chose Cycle 7 as the target of fault injection.

The third column of Table IV shows the valid $T_{\text{glitch}}$ range values calculated using the timing characterization and the analysis of software pipeline in Section VI. An adversary cannot inject any error if the $T_{\text{glitch}}$ value is greater than the upper bound. A $T_{\text{glitch}}$ value within the valid range causes only data errors. The $T_{\text{glitch}}$ values that are smaller than the lower bound can create both data and instruction errors.

We validated the $T_{\text{glitch}}$ range values shown in the third column via biased fault injection on the LEON3 FPGA implementation. For each software program and target cycle, we applied a $T_{\text{glitch}}$ range of 15.8ns to 3.0ns and observed the fault behavior using LOGAN. The fourth column of Table IV shows the $T_{\text{glitch}}$ ranges, in which we only observed biased data errors. As it is seen, the valid $T_{\text{glitch}}$ ranges obtained after gate-level simulation and after on-board fault injection are close to each other.

B. DFIA Attacks on the FPGA Implementation of LEON3

We evaluated the efficiency of the proposed methodology and DFIA attacks by investigating two case studies for SBOX and TBOX AES implementations:

1) White-box approach: In this case, the adversary has the netlist of the RISC processor and has the full knowledge of the software behavior in the pipeline. Therefore, the adversary can identify the most suitable clock cycles and fault intensity range for fault injection, aiming at creating biased data errors.

2) Black-box approach: In this case, the adversary does not have the netlist. In addition, the adversary’s knowledge about the pipeline behavior of the software is limited. The adversary can still do a limited timing characterization for the processor using the existing black-box characterization approaches [2], [4], [5]. However, the pipeline state is unknown to the adversary. Therefore, the adversary cannot combine this timing characterization with the software behavior in the pipeline.

In both cases, we applied the DFIA attack to obtain the secret key. We used one plaintext value and one key value for our experiments. We collected faulty ciphertexts for different fault intensity values. We controlled the fault intensity by increasing/decreasing the $T_{\text{glitch}}$ value. Here, we present our results for retrieving one byte of the AES key.

DFIA attacks need to obtain biased data errors. In white-box strategy, the adversary combines the AES software analysis result (Section IV) and the timing characterization of pipeline (Section V), to find the most suitable points for biased data fault injection. Then, he injects faults into these points in the execution of AES software and collects faulty ciphertexts.

In black-box case, the adversary first characterizes the timing of the processor with one of the previously mentioned black-box approaches. As the adversary does not have access to the internal status of the pipeline, he is not able to pinpoint a specific point in software execution. Therefore, in order to collect biased data errors, he has to exhaustively inject fault into all points in the execution of software and hope they will lead to biased data errors. However, in this approach, some fault injection attempts will cause random effects in the fault injection point. For example, a fault injection attempt might affect the address calculation of a memory-load instruction and causes fetching an irrelevant data from the memory. This will create a random effect, rather than a biased effect, in the fault injection point. Although this fault injection creates a faulty ciphertext, it will not be useful for DFIA. Next, we investigate the required effort for biased fault injection for both cases.

Table IV shows the $T_{\text{glitch}}$ range and the number of clock cycles that an adversary can exploit to create biased data errors. We demonstrate these results for white-box and black-box attack strategies on TBOX and SBOX AES implementations.
The first column of the table lists the number of clock cycles during which a fault injection attempt might yield a biased data error. The total number of these cycles for SBOX and TBOX AES is 13 and 16, respectively (Fig.5 and 8). For the black-box case, the adversary attempts to inject faults into all of these cycles as he is not aware of the pipeline behavior of the AES software. For the white-box case, these numbers reduce to 6 and 9, respectively, via proposed methodology (Section VI).

The second column of Table V shows the $T_{glitch}$ range in which the adversary attempts to inject biased data errors. The overall range for our LEON3 implementation and experimental setup is $[3.0, 15.8]$ns, as the critical path of the processor is 15.8ns and the minimum glitch period of our setup is 3.0ns. For the black-box strategy, the adversary tries all possible glitch period (i.e, fault intensity) values. For the white-box case, this range will be reduced due to the proposed methodology. For every target cycle, we will have a different valid $T_{glitch}$ range as it is previously shown. In our case, we have 6 and 9 valid $T_{glitch}$ ranges for SBOX and TBOX AES, respectively. In Table V we list the average lower and upper bound values for the white-box approach rather than showing bounds of each $T_{glitch}$ range.

The third column of Table V is the total number of fault injection attempts for 162ps step-size. We obtained this number by multiplying the number of attacked clock cycles and the number of $T_{glitch}$ levels within the corresponding $T_{glitch}$ range. As it can be seen, the number of total fault injections for the black-box case is greater than the white-box case. As a result, the proposed methodology significantly reduces the effort for biased fault injection. The reader should note that these numbers do not show the efficiency of the DFIA attack. They show the required fault injection attempts to explore all possible $T_{glitch}$ values that might yield biased data errors.

Now, we will demonstrate the efficiency of the DFIA attack for black-box and white-box approach. For this purpose, we will use the results that we obtained for SBOX and TBOX AES software on LEON3. Figures 14(b) and 14(a) show the number of faulty ciphertext pairs that DFIA uses for black-box and white-box strategies. The horizontal axis is the number of faulty ciphertext pairs and the vertical axis shows the number of remaining key guesses for the secret key byte. As it is shown, both DFIA attacks need more fault injections for the black-box case to retrieve the secret key byte.

In conclusion, we experimentally demonstrated that DFIA attacks are feasible on pipelined RISC processors for the black-box and white-box approaches. We also showed that the white-box strategy significantly reduces the fault injection effort required to create biased data errors.

**X. ACKNOWLEDGMENT**

This research was supported through the National Science Foundation Grant 1441710, Grant 1115839, and through the Semiconductor Research Corporation.

**REFERENCES**


