Analyzing the Fault Injection Sensitivity of Secure Embedded Software

BILGIDAY YUCE, NAHID FARHADY GHALATY, CHINMAY DESHPANDE, HARIKA SANTAPURI, CONOR PATRICK, LEYLA NAZHANDALI, and PATRICK SCHAUMONT, Virginia Tech

Fault attacks on cryptographic software use faulty ciphertext to reverse engineer the secret encryption key. Although modern fault analysis algorithms are quite efficient, their practical implementation is complicated because of the uncertainty that comes with the fault injection process. First, the intended fault effect may not match the actual fault obtained after fault injection. Second, the logic target of the fault attack, the cryptographic software, is above the abstraction level of physical faults. The resulting uncertainty with respect to the fault effects in the software may degrade the efficiency of the fault attack, resulting in many more trial fault injections than the amount predicted by the theoretical fault attack. In this contribution, we highlight the important role played by the processor microarchitecture in the development of a fault attack. We introduce the microprocessor fault sensitivity model to systematically capture the fault response of a microprocessor pipeline. We also propose Microarchitecture-Aware Fault Injection Attack (MAFIA). MAFIA uses the fault sensitivity model to guide the fault injection and to predict the fault response. We describe two applications for MAFIA. First, we demonstrate a biased fault attack on an unprotected Advanced Encryption Standard (AES) software program executing on a seven-stage pipelined Reduced Instruction Set Computer (RISC) processor. The use of the microprocessor fault sensitivity model to guide the attack leads to an order of magnitude fewer fault injections compared to a traditional, blind fault injection method. Second, MAFIA can be used to break known software countermeasures against fault injection. We demonstrate this by systematically breaking a collection of state-of-the-art software fault countermeasures. These two examples lead to the key conclusion of this work, namely that software fault attacks become much more harmful and effective when an appropriate microprocessor fault sensitivity model is used. This, in turn, highlights the need for better fault countermeasures for software.

CCS Concepts: • **Security and privacy → Embedded systems security**; **Side-channel analysis and countermeasures**; Software and application security;

Additional Key Words and Phrases: Fault attack, embedded system security, microarchitecture-aware fault analysis, AES, RISC

ACM Reference Format:
Bilgiday Yuce, Nahid Farhady Ghalaty, Chinmay Deshpande, Harika Santapuri, Conor Patrick, Leyla Nazhandali, and Patrick Schaumont. 2017. Analyzing the fault injection sensitivity of secure embedded software. ACM Trans. Embed. Comput. Syst. 16, 4, Article 95 (July 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3063311

This research was supported in part through the National Science Foundation Grant 1441710, and in part through the Semiconductor Research Corporation.
Authors' addresses: B. Yuce, N. F. Ghalaty, C. Deshpande, H. Santapuri, C. Patrick, L. Nazhandali, and P. Schaumont, Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061; emails: {bilgiday, farhady, chinmay, harika90, conorpp, leyla, schaumont}@vt.edu.

DOI: http://dx.doi.org/10.1145/3063311

1. INTRODUCTION

Today, a range of secure embedded systems from low-end smartcards to high-end network equipments are extensively used to transfer, store, access, and manipulate sensitive data such as personal passwords and copyrighted content [Kocher et al. 2004]. The secure embedded systems employ confidentiality, integrity, and authentication mechanisms to ensure the security of the sensitive data. Implementation attacks, exploiting the vulnerabilities in the implementation of the security mechanisms, pose a serious threat to the security of these systems. Implementation attacks on secure embedded systems are relevant when the adversary can observe or influence the physical implementation of security mechanisms. Implementation attacks are applicable to secure embedded systems in the Internet of Things, as well as to server computing applications that coexist with processes from different users on the same machine [Genkin et al. 2015; Piscitelli et al. 2015; Gruss et al. 2015; Inci et al. 2015]. The main principle of an implementation attack is to correlate observable effects of the physical implementation with the hypothesized (key-dependent) operation of the secure system internals. A positive correlation will then reveal a small part of the secret key, thereby eliminating the exponential complexity of a brute-force key search.

The two main categories of implementation attacks include side-channel attacks [Mangard et al. 2007] and fault attacks [Joye and Tunstall 2012]. In a side-channel attack, the adversary passively observes physical variables of the security operations and then correlates these with a model of the side-channel leakage of the secure embedded system. In a fault attack, the adversary actively disturbs the execution of the secure system, extracts a faulty result, and then correlates it with a fault model of the system. Both side-channel attacks as well as fault injection attacks rely on models to approximate the physical behavior of the secure embedded system [Mangard et al. 2007; Otto 2005].

There exists a rich collection of side-channel leakage models, which originate from the insight into the exact source of side-channel leakage. For example, execution timing and power consumption of a digital system can be modeled and predicted accurately using the synchronous (single-clock) design abstraction [Mangard et al. 2007]. Similarly, cache side-channel attacks, which rely on the complex timing effects of software execution and cache organization, are possible because the timing effects are well understood and predictable [Liu et al. 2015].

However, the same observation does not hold for fault attacks. The bulk of the literature on fault attacks, including several seminal overviews, treat the modeling of fault behavior only at the most basic level [Bar-El et al. 2006; Karaklajic et al. 2013; Barenghi et al. 2012; Otto 2005; Joye and Tunstall 2012]. Fault injection methods (under-powering [Barenghi et al. 2009], voltage and clock glitches [Korak and Hoefler 2014], EM pulses [Moro et al. 2013a], laser pulses [Van Woudenberg et al. 2011]) cause a fault effect, which is then captured in a fault model. Typical fault models only capture basic parameters, such as the fault duration (transient or permanent), the fault type (bit-flip, random, stuck-at), and the fault location [Otto 2005]. In recent years this model was further refined into fault-sensitivity, which acknowledges a link between fault injection intensity and fault effect [Li et al. 2010]. Yet, we argue that this hardware-oriented model is still insufficient in many practical situations, including the case of the embedded software fault attacks examined in this article. Indeed, a fault attack on software requires a fault model that captures fault effects in software, and not fault effects in the microprocessor that executes software.

A second challenge for fault modeling relates to model accuracy. Fault analysis techniques commonly assume a perfect match between the fault model, and the actual fault obtained in the implementation as a result of the fault injection [Sakiyama et al. 2012; Ali and Mukhopadhyay 2011]. However, the efficiency of the fault analysis decreases

quickly if this is not the case, and even a mismatch of 1 in 100 fault injections can render the fault analysis ineffective [Ferretti et al. 2014].

An important insight contributed by this article is the need to expand the fault modeling mechanism beyond the basic parameters of fault duration, fault type, and fault location. A fault model for embedded software should capture the fault effects experienced by microprocessor instructions in order to analyze the effects of a fault injection on software. Thus, we need to model fault effects in the instruction-set architecture rather than faults purely at the hardware level. There are compelling reasons for raising the abstraction level.

—First, the instruction set architecture (ISA) provides an execution model, and with it comes the ability to model the impact of faults in the execution of instructions. A typical instruction-execution cycle includes at least instruction-fetch, decode, and execute. The effect of a fault can change according to each phase of the instruction-execution cycle. For example, a fault on the instruction-fetch could affect the instruction opcode, while a fault on the execution phase could change the instruction operands. Each of these faults has a different effect on the software program.
—Second, the instruction set architecture makes a clear distinction between data processing (e.g., arithmetic instructions), control (e.g., jump instructions), storage (e.g., load/store of data), and input/output operations [Karaklajic et al. 2013]. Faults have a different effect for each different instruction, and hence, software fault models should be instruction-dependent.
—Finally, the data dependencies, which are crucial to understand the propagation of faults in software, only become visible in the software. It is impossible to analyze fault propagation through the microprocessor hardware alone.

In this article, we will introduce a fault model for a pipelined microprocessor and demonstrate how the model can be used for efficient fault attacks on secure microprocessor software, and even for breaking fault attack countermeasures. We claim the following novel contributions.

(1) We demonstrate a method to construct a fault sensitivity model of a microprocessor, for the specific purpose of fault attacks and fault analysis. We also provide Microarchitecture-Aware Fault Injection Attack (MAFIA) to demonstrate the use of the model. MAFIA allows an adversary to gain insight into the probable faults, and to pinpoint the most suitable moments for fault injection during the execution of a software program. The proposed method determines the required fault injection intensity for the attack, such that the mismatch between the injected fault and the modeled fault becomes minimal.
(2) We show how the fault sensitivity model is used in a fault attack on secure software. We demonstrate differential fault intensity analysis (DFIA) on Advanced Encryption Standard (AES) software executing on a seven-stage Reduced Instruction Set Computer (RISC) pipeline. We demonstrate that fault modeling at the instruction-architecture level reveals the effects of pipeline stalls and data dependencies, which results in additional opportunities for fault injection.
(3) We show how the microprocessor fault sensitivity model helps to pinpoint the weaknesses of a class of fault attack countermeasures in software, which rely on instruction-level redundancy. In particular, we demonstrate successful attacks on Instruction Duplication, instruction-level parity checking, and fault-tolerant instruction sequences.

This article is based upon two earlier conference publications, published at Fault Diagnosis and Tolerance in Cryptography (FDTC) 2015 [Yuce et al. 2015a] and FDTC 2016 [Yuce et al. 2016b], respectively. Compared to these publications, this article

generalizes the development of a fault sensitivity model for microprocessors and then demonstrates how this concept unifies the attacks discussed in both papers.

The rest of the article is organized as follows. In Section 2, we introduce important preliminaries and differentiate fault attacks on hardware secure systems from fault attacks on software secure systems. We discuss fault effects in a typical stored-program computer built around a RISC processor. In Section 3, we show how these effects can be captured in a microprocessor fault sensitivity model, and how such a model can be used to develop a fault attack (i.e., MAFIA) on embedded software. In Section 4, we use the microprocessor fault sensitivity model to mount a biased-fault attack on the well-known TBOX implementation of AES. In Section 5, we use the microprocessor fault sensitivity model to break a collection of software-based fault countermeasures. In Section 6, we provide details on the practical implementation setup. In Section 7, we summarize the key results of our experiments. In Section 8, we conclude the article.

2. PRELIMINARIES

This section explains the fundamental concepts that are required to follow the rest of the article.

2.1. Using Faults as a Hacking Tool

In a fault attack, an adversary first alters the physical operating conditions of a secure system to inject well-defined, targeted faults into the system operation. The estimated fault effects are captured with a fault model. Next, the adversary breaks the security of the secure system by systematically correlating the actual fault response of the system with the fault model. A well known technique that follows this strategy is Differential Fault Analysis [Giraud 2005], and many variations have been demonstrated that follow this attack scheme. In a typical fault attack framework, an adversary is not capable of directly modifying/monitoring the internals of a chip, or changing the binary of the target program [Lemke-Rust and Paar 2006]. The adversary can alter the execution of a program/system via fault injection, and can observe the fault effects at output of the program/system.

To alter the execution in a desired way, the adversary varies the *fault intensity*, which is the strength of the physical stress by which the system is pushed outside of its nominal operating conditions. The adversary controls the fault intensity via fault injection parameters. For clock glitching, shortening the length of the glitch increases the fault intensity. It is controlled by glitch/pulse voltage and length for voltage glitching, electromagnetic pulse injection, and laser pulse injection. The laser and electromagnetic pulse injections also enable the adversary to localize the fault intensity by controlling the shape, size, and position of the injection probe.

The previous works have revealed a correlation between the fault intensity and the fault response: A gradual increase in fault intensity yields a proportional change in the fault response of the system. We call this relation *biased fault behavior*. This behavior is valid independent of the used fault injection method, and it enables the adversary to control the induced fault effects [Korak and Hoefler 2014; Moro et al. 2013a; Courbon et al. 2014; Yuce et al. 2015a].

Because of the biased fault behavior, the adversary is able to find a critical fault intensity point, at which the target system begins to reflect faulty behavior. This critical point is the *fault sensitivity* of the target system. To find the fault sensitivity, the adversary starts with a low fault intensity value and gradually increases the intensity until seeing a faulty output.

Next, we will compare and contrast the fault manifestation and propagation mechanisms in the hardware and software secure systems.

2.2. Comparison of Fault Attacks on Hardware and Software Secure Systems

Fault attacks on both hardware and software systems require the knowledge of the target system's implementation. Having more knowledge of the target system increases the adversary's control on the induced fault effects, and thus, yields more efficient fault attacks. In practice, the fault manifestation and the propagation mechanisms in hardware and software systems are different. Therefore, mounting an efficient fault attack on a software system requires the knowledge of different abstraction layers than attacking a hardware system does.

In a hardware secure system, a security algorithm is mapped into a netlist of logic gates and registers. The execution of the mapped algorithm is embodied as a sequence of register-transfer operations scheduled over multiple clock cycles. Every bit-level operation of the algorithm thus maps into a particular clock cycle and register transfer. During a fault attack on the hardware system, an adversary will target a sensitive bit-level operation by selecting specific clock cycles and register-transfers, and by applying a suitable fault injection method. Next, the effects of the fault injection will be propagated into a faulty system output through register transfers with a dependency on the faulty register transfer. In order to mount a fault attack on such a hardware model, it is sufficient for the adversary to understand the hardware-level operation, and to apply generic, gate-level fault models.

In a software secure system, a security algorithm is implemented as a sequence of instructions executed by a microprocessor. Each instruction execution, in turn, is composed of several steps. The processor loads each instruction from program memory (*instruction-fetch*), determines the meaning of the current instruction through its opcode (*instruction-decode*), executes the current instruction (*instruction-execution*), and then determines the next instruction to fetch (*program counter update*). The number of steps in the instruction-execution cycle is architecture dependent, and it can vary considerably from one microprocessor to the next. However, we observe that the smallest execution step in software, the instruction, still corresponds to multiple register-transfers at the hardware level.

A fault injection in a microprocessor will affect the correctness of instruction execution. The effect of the fault will be propagated into a faulty system output by instructions that have data-dependencies or control-dependencies on the faulty instruction. Hence, knowledge of the microprocessor architecture is insufficient to mount a successful fault attack; the adversary also needs to understand the dependencies defined by the software. Furthermore, the precise effect of a fault on a microprocessor instruction depends on the type of the instruction. For instance, a bit-flip fault injected during the execution step of an addition instruction may yield a single-bit fault in the result of this instruction. However, the same bit-flip fault injected during a memory-load instruction would cause a single-bit fault in the effective address calculation, and thus, data is loaded from a wrong memory location. In the former case, only a single bit of the destination register is faulty, while in the latter case, the destination register has a random number of faulty bits.

We conclude that a fault model for a software secure system must capture both the hardware and software aspects of the fault behavior. For a given processor architecture, it should show the potential fault effects for each instruction during each step of the instruction cycle. The fault effects may depend on the type of the instruction as well as on the step of the instruction cycle affected by the fault. Furthermore, in complex processor architectures such as the RISC processor architectures considered in this article, the execution of multiple instructions is overlapping. A single fault injection during a single clock cycle may affect sequential instructions. To analyze these complex effects, the adversary needs to understand the fault sensitivity of instructions with overlapping execution. In the microprocessor fault model proposed in this

article, we provide support for overlapping instruction execution. We also introduce a microarchitecture-aware fault attack methodology based on the proposed fault model.

In the following sections, we elaborate these concepts for the case of the RISC pipeline.

2.3. Fault Behavior in a RISC Pipeline

The RISC architecture is the dominant processor architecture in modern embedded systems, and, therefore, the analysis of this architecture is a meaningful starting point to develop a systematic fault model for software. In the following sections, we provide a brief review of a typical RISC pipeline. Next, we define the generic fault injection mechanism that we will assume for the rest of the article. We then describe the expected fault effects in a standard RISC-based stored-program architecture consisting of a pipeline and a memory. This leads to the classification of fault types utilized in our work. Using these preliminaries, we will then introduce our fault attack methodology in Section 3.

2.3.1. RISC Pipeline. We describe the example of the LEON3 pipeline, an open-source, 32-bit, The Scalable Processor Architecture (SPARC)-compilant RISC processor. LEON3 partitions the instruction cycle into seven steps, which leads to the following seven-stage pipeline.

(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 to determine the operation and operands. For branch and CALL instructions, the target address is computed.
(3) *Register Access (A):* The operands are read from the data register file or from internal forwarding paths.
(4) *Execution (E):* Arithmetic, logical, and shift operations are computed. For memory-load and store, the effective data address is computed.
(5) *Memory (M):* Memory instructions access the data memory using the effective address.
(6) *Exception (X):* Traps and interrupts (such as divide-by-zero exceptions) are resolved.
(7) *Write-back (W):* The results of the instruction are written into the register file.

2.3.2. Cycle Accurate Fault Injection. An adversary can choose among many different forms of fault injection, from very coarse and generic to very precise and specific. However, there is a tradeoff between the accuracy of a fault injection and the complexity to perform it. Therefore, we make the following assumption, which covers a large portion of practical fault-injection cases.

We assume that the adversary can select a precise clock cycle for fault injection (precise timing control), but not necessarily a precise fault location (imprecise location control). We also assume that the adversary can vary the intensity of the fault injection, and that it's possible to achieve different fault effects by varying the fault intensity.

Some examples of fault injection mechanisms that are feasible in the cycle-accurate fault injection scheme are clock-glitch injection [Korak and Hoefler 2014], injection of synchronized power-line voltage-glitches [Barenghi et al. 2009], and injection of synchronized magnetic pulses [Moro et al. 2013a]. These injection methods can be tuned with voltage starvation, temperature hikes, or over-clocking.

2.3.3. Fault Injection in the RISC Pipeline. Figure 1 demonstrates the effect of a cycle-accurate fault injection on the execution of the seven-stage LEON3 pipeline. In any given clock cycle, multiple instructions execute simultaneously, each of them at a different step of the instruction cycle. Therefore, a fault injection can potentially affect as many instructions as the number of pipelines stages. However, the actual number

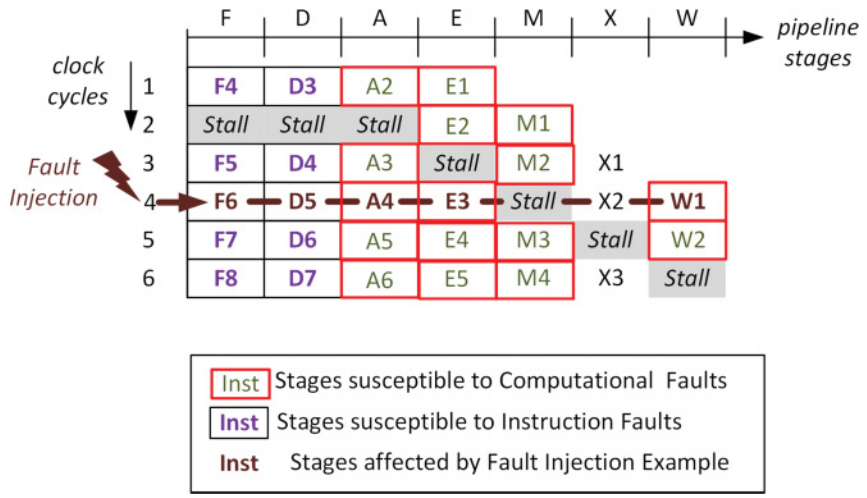

Fig. 1. Fault propagation in a seven-stage RISC pipeline. A glitch can affect as many instructions as there are RISC pipeline stages. Careful control of the fault intensity limits the fault to the slowest pipeline stage. Furthermore, pipeline stalls will temporarily *blind* stalled pipeline stages from glitches.

of affected pipeline stages is determined by fault injection intensity, and by pipelining effects.

Pipelining effects such as data dependencies between the concurrently executed instructions, cache misses, and branch interlocks cause stalls in the pipeline. The stalled pipeline stages are not affected by the fault injection. For example, the fault injection in cycle 4 in Figure 1 does not affect the Memory Stage.

Under a constant fault injection intensity, the fault effect will be different for different stages of the pipeline. This is because of variations in the detailed architecture of pipeline. Therefore, it is also possible that a fault injection at a specific fault intensity can affect some pipeline stages while leaving others untouched. A pipeline stage is affected by the fault injection only if the applied fault intensity is greater than the fault sensitivity for that pipeline stage. For example, consider again cycle 4 of the pipeline in Figure 1. By reducing the fault intensity, the adversary will eventually end up with a single faulty pipeline stage, namely the most sensitive pipeline stage. Assume for example that the execution stage would have the longest critical delay. Then, a fault injection with low intensity will affect only E3, the execution of instruction 3 in cycle 4. After such an isolated fault is injected, it will further propagate through the pipeline stages and affect instructions with dependencies on the faulty instruction 3.

Thus, we observe two important effects in a running RISC pipeline that provide an adversary with *additional* opportunities for controlled fault injection. The first effect is the pipeline hazard, which may depend on software as well as on the processor architecture. The second effect is the variation of fault sensitivity with RISC pipeline stage, which may provide the ability to target a single pipeline stage through a carefully chosen fault injection intensity. In our proposed attack methodology, we utilize both of these effects to our advantage.

2.3.4. Instruction Faults and Computation Faults. As every instruction moves through the instruction-execution cycle, its fault behavior changes as a function of the step within the instruction-execution cycle. We partition the fault effects into two main categories.

(1) *Instruction Faults (IF)* are faults that affect the control flow or instruction sequence of a program. Instruction Faults are created by fault injection in the fetch or decode

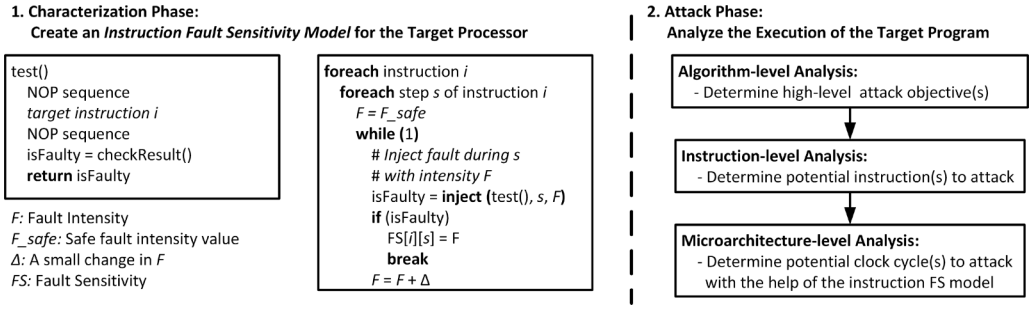

Fig. 2. Overview of MAFIA: (1) An instruction fault sensitivity model is built for the target processor. (2) The created model is combined with the analysis of the software program to determine the best fault injection parameters.

pipeline stages (see Figure 1). For example, a fault could skip an instruction. An adversary can leverage this to break software fault countermeasures by bypassing instructions that check integrity or branch when an error is detected. A fault could also change the meaning of an instruction by modifying the opcode. Balasch describes the effect of faults on the instruction fetch of an AVR microcontroller [Balasch et al. 2011]. By gradually increasing the fault intensity, the opcodes of instructions are modified (as a result of timing violations) until they eventually turn into a `nop` instruction.

(2) *Computational Faults (CF)* are faults that cause errors in the data used by a program. They are created by any error in the stages A, E, M, and W as indicated in Figure 1. An error in any of these stages can contribute a faulty value to the register file or memory. Faults in these stages will eventually propagate to the output, and they are suitable for traditional fault analysis mechanisms such as Differential Fault Analysis (DFA) or DFIA.

2.3.5. Fault Injection in the Memory. An alternate target for fault injection is the instruction-memory or data-memory used by a RISC processor. We can distinguish two types of memory faults: those that happen as a result of the direct execution of a RISC instruction, and those that happen independently of software execution. Our instruction fault model concentrates on the first case, and it can capture faults that occur as a result of instruction-fetch, operand-fetch, and result-writeback. On the other hand, in this article, we will not further develop fault attacks based on independent memory faults. We observe that error-check mechanisms are quite common in contemporary memory architectures—and this provides additional justification for our focus on processor faults.

3. MAFIA: MICROARCHITECTURE-AWARE FAULT INJECTION ATTACK

This section proposes an *instruction fault sensitivity model* and a fault attack strategy *Microarchitecture-Aware Fault Injection Attack (MAFIA)* to satisfy the requirements examined in the previous section. As it is shown in Figure 2, the proposed methodology consists of two phases: (i) Characterization Phase and (ii) Attack Phase.

The purpose of the *characterization phase* is to create an instruction fault sensitivity model for the target processor. The instruction fault sensitivity model shows the fault sensitivity for the steps of each instruction. A processor potentially carries out each step of an instruction on a different microarchitectural block. Therefore, one can also think of the instruction fault sensitivity model as the characterization of main microarchitectural blocks for each instruction. The key point is that the fault sensitivity is instruction-dependent because each instruction uses a specific subset of the

available microarchitectural blocks. The instruction fault sensitivity model can be generated once for a specific fault injection method and a target processor, and it can be used multiple times for different attacks on the target processor. Section 3.1 explains the details of the characterization phase for an example architecture and fault injection method.

The purpose of the *attack phase* is to design and implement a fault attack on a target program running on the characterized processor. The adversary analyzes the target program at different abstraction levels, and then combines this analysis with the instruction fault sensitivity model to determine the best clock cycle(s) and processor part(s) to attack. The attack phase consists of three steps. An adversary first analyzes the target program at the *algorithm-level* to determine the application-specific attack objectives. Then the adversary applies an *instruction-level* analysis to determine potential instructions to attack in the program flow. In the last step, the adversary analyzes the execution of the software program on the target microarchitecture to identify potential clock cycles to attack. Finally, the adversary uses an instruction fault sensitivity model to determine fault injection parameters for each potential clock cycle to attack. As the result, the adversary has a set of (*clock cycles*, *fault parameters*) pairs to attack. Using this information, the adversary can carry out the actual fault injection experiments. Section 3.2 explains these steps on an example.

The proposed method requires ISA-level, microarchitecture-level, and hardware-level knowledge of the target system. The ISA-level knowledge is required to understand specifications and semantics of the instructions. The adversary can use open-source architecture manuals to acquire this knowledge. The microarchitectural-level knowledge enables the adversary to understand how a given instruction is executed on the target platform. The adversary may have an instruction-accurate model (e.g., a software-level emulator such as QEMU [Bellard 2005]), a cycle-accurate model (e.g., a microarachitecture simulator such as gem5 [Binkert et al. 2011]), a register-transfer level model, or a commercial-of-the-shelf (COTS) implementation of the processor. The hardware-level knowledge is necessary to determine the sensitivity of the processor hardware to the physical fault injection. The adversary can gather hardware-level information from the gate-level netlist, transistor-level netlist, layout, register-transfer level definition, or a COTS implementation of the processor hardware. As a result, an adversary's knowledge level may vary from zero-knowledge to full-knowledge in practice. The key point is that a fault attack would be more efficient if an adversary has more knowledge of the implementation. The proposed method provides a methodology to systematically combine the knowledge of different abstraction layers.

Next, we explain the details of the proposed methodology by means of an example. This illustrates how the proposed instruction fault model is used in practice. We follow the assumption of a conventional, pipelined RISC microprocessor. Instructions flow through seven stages of a pipeline and can be stalled because of hazards. In addition, the effects of hazards can be minimized through conventional techniques such as forwarding or careful instruction scheduling.

3.1. Characterization Phase: The Instruction Fault Sensitivity Model

In this section, we provide an example instruction fault sensitivity model for a subset of SPARC instructions. We characterized the fault sensitivity of instructions on a LEON3 processor for setup-time violation based faults. We implemented LEON3 processor on a 45nm Spartan 6 FPGA (Field Programmable Gate Array), and we characterized the fault sensitivity by applying gate-level simulation on the post place-and-route model of the implementation. We used combinational path delays as the fault sensitivity metric because it has been shown that there is a significant correlation between the path delays of an implementation and its sensitivity to setup-time violation attacks [Yuce

Table I. LEON3 Pipeline Fault Sensitivity Model for DFIA (Critical Delay in ns)

Instruction	**F**	**D**	**A**	**E**	**M**	**X**	**W**	Meaning
`xor reg, reg, reg`	5.54	3.72	6.52	5.12	0	3.26	4.92	Bit-wise xor
`ld mem, reg`	5.72	3.36	5.4	5.2	7.58	2.82	5.6	Load from memory
`ldi mem, reg`	3.53	3.26	5.62	5.2	7.58	3.17	4.45	Load Indexed
`st reg, mem`	5.91	3.5	5.78	5.37	5.35	3.77	0	Store Word
`sll reg, imm, reg`	5.53	3.26	5.2	5.09	0	2.16	5.6	Shift Left Logical
`srl reg, imm, reg`	5.91	2.83	4.7	5.67	0	3.25	5.61	Shift Right Logical
`cmp reg, reg`	6.86	5.53	4.98	5.23	0	4.78	5.6	Compare
`bne add`	6.40	5.00	5.85	3.92	2.020	0	5.6	Branch on Not Equal
Fault Type	**Instr Fault**	**Computation Fault**						

et al. 2015b; Zussa et al. 2012]. Gate-level simulation has been employed earlier to create a better fault sensitivity models of AES ASIC (Application Specific Integrated Circuit) implementations [Sugawara et al. 2012; Barenghi et al. 2015]. However, we used the gate-level simulation for characterization of individual processor instructions.

Fault sensitivity of an instruction is not constant. Rather, it varies as a function of the pipeline stage. We verified this by extracting the combinational delay of individual pipeline stages for each instruction. For each target instruction, we wrote a test program and applied a fault sensitivity characterization algorithm as shown in Figure 2. The test program includes the target instruction surrounded by two sequences of `nop` instructions. For a test program, we characterized the sensitivity of a pipeline stage by injecting faults with varying fault intensities during the execution of this specific stage: Starting from a safe fault intensity value, we gradually increased the fault intensity until we observe a faulty output. Repeating this analysis for each target instruction, we obtained the instruction fault sensitivity model.

In our simulations, we simulated each instruction one time with arbitrarily selected operands to accelerate the characterization process. We assumed that the impact of the operand values on the variation of instruction timing is smaller than the impact of the instruction type and the pipeline stage. We made this assumption based on the previous work [Korak and Hoefler 2014; Balasch et al. 2011]. For example, Korak and Hoefler [2014] analyzed two different microcontrollers (an ATMega256 and an ARM Cortex-M0) against voltage and clock glitching. They observed different fault intensity intervals for different instructions and pipeline stages. They showed that the decode stage of their target was always fault-free while they were able to create faults in fetch and execute stages. Their results also demonstrated that execution stages of different instructions have different timing values. In Sections 6 and 7, we experimentally demonstrated that such an approach is sufficient to mount efficient fault attacks. One can also run each instruction with multiple data sets and apply statistical processing of the results to create the instruction fault sensitivity model. Such an approach would capture the variation effects in a more accurate way while increasing the characterization time.

Table I illustrates the obtained instruction fault sensitivity model. The figures in the table cells indicate the critical delay (in ns) of the instruction as it flows through each pipeline stage. One can clearly distinguish a variation in timing horizontally, across a single instruction. This confirms that the critical delay varies as a function of the pipeline stage. In addition, one can see a timing variation vertically, across a column for a particular pipeline stage. This means that different instructions have a different timing requirement. Now, imagine a running software program; the pipeline is filled with instructions as illustrated in Figure 1. Using the instruction fault sensitivity model, it is clear that we can predict which pipeline stages will be faulty when we

inject a fault with a given fault intensity. This will be the basis of the fault attack methodology, explained in the next section.

The obtained model captures several fault injection methods because setup-time violation faults can be achieved by various techniques including clock glitching, voltage glitching, voltage underfeeding, overheating, and electromagnetic pulses [Zussa et al. 2012; Hayashi et al. 2015]. For other fault injection methods such as laser pulses, the same approach can be used to obtain a similar instruction sensitivity model. In addition, the same algorithm can also be used to build an instruction fault sensitivity model with physical fault injection. If an adversary has a COTS implementation of the target and a fault injection setup, the adversary can use the same test program and algorithm to create an instruction fault sensitivity model.

3.2. Attack Phase: Analyzing the Target Embedded Software

This section demonstrates how the instruction fault-sensitivity model can be used to design an attack on a software secure system. We will describe a generic methodology, which will be applied on specific cases in later sections. The methodology has three phases, and we will briefly describe each of them through an example.

3.2.1. Algorithm-level Analysis. Initially, the adversary investigates the software implementation at the highest level of abstraction available. This could be C source code, or a binary of the software secure system. During algorithm analysis, the adversary defines the application-dependent attack objectives. For example, the differential fault analysis attack of Piret and Quisquater [2003] requires a fault differential on the ciphertext, obtained after injecting a fault into the last-round state of block cipher. In the following example, we assume a last-round computation of the following form. The fault attack target, in this case, is the `state`, and the objective is to capture a faulty `ciphertext`.

```
ciphertext = key xor sbox[state]
```

3.2.2. Instruction-level Analysis. Next, the adversary studies the software implementation at instruction level, in order to identify the candidate instructions for the fault attack. Consider the following example, which is a simplified implementation of the last-round of a block cipher, implemented using LEON3 instructions.

```
LD      [%fp - 12], %g1      ;LD1 (state)
LD      [%fp - 16], %g2      ;LD2 (key)
LD      [0x100 + %g1],%g3    ;LD3 (lookup[state])
XOR     %g3, %g2, %g4        ;XOR1 (key, state)
STD     %g4, [%fp - 20]      ;STD1 (output)
```

This program snippet loads a state variable and a key from data memory, substitutes the state variable using a lookup table, and applies exclusive-OR (XOR) on the result and key. The output is then transferred to a memory location. Based on the DFA technique of Piret, the candidate instruction for fault injection is the first load instruction, which transfers the last-round state from data memory into processor register `%g1`. Additional study of the control-flow and data-flow of the instructions may be required in order to identify dependent instructions, which are also suitable for the fault attack. In this case, only the first load `LD1` is a suitable fault-injection candidate. The second load, `LD2` does not contribute to the desired fault analysis, and faults in the third load `LD3` would cause faults at the output of the lookup table rather than at the input.

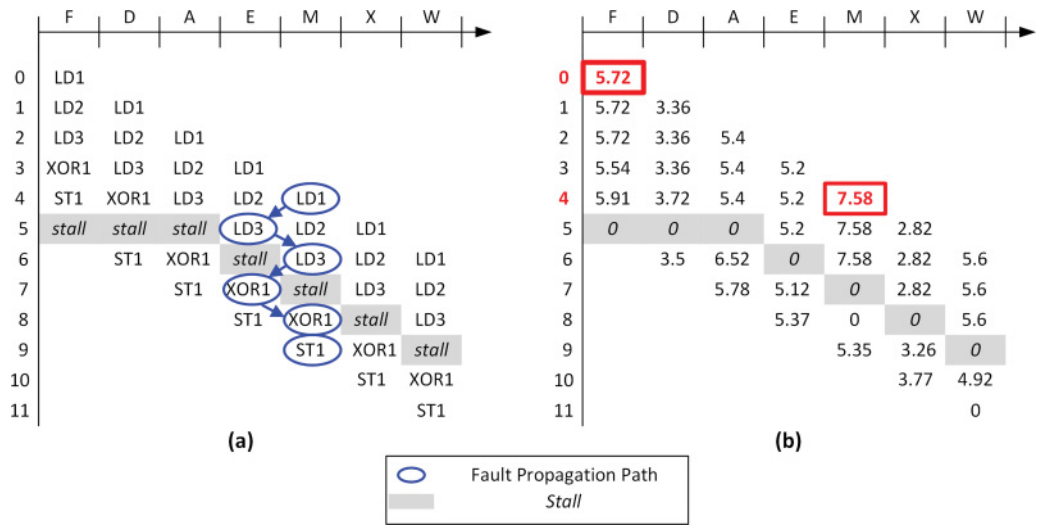

Fig. 3. (a) Pipeline analysis and fault propagation path for the lookup table target. (b) Pipeline sensitivity analysis for the lookup table target.

3.2.3. Microarchitecture-level Analysis. The third step is to examine the software behavior closely in order to select the exact clock cycle that would lead to the desired error. One would need to do this analysis at the cycle-accurate level, in order to account for pipeline effects. Figure 3(a) illustrates the result of this analysis. The instruction sequence requires 12 clock cycles to complete and experiences one data hazard to accommodate the table lookup. According to the pipeline analysis, `LD1` occupies seven clock cycles and a fault in any of these seven cycles can potentially create a fault of the desired kind.

However, not all seven cycles can lead to useful `LD1` errors. Some of the fault injections may also affect other instructions. We can use the instruction fault-sensitivity model to identify the cycle most suited for fault injection. Figure 3(b) shows an overlay of the instruction fault sensitivity data (Table I) on the instructions in the pipeline. Looking across rows, one can now identify the most sensitive pipeline stage in every clock cycle. For the `LD1` instruction, the instruction-fetch and memory-access step are most sensitive, which implies that clock cycle 0 and clock cycle 4 are suitable candidates for fault injection. The fault intensity would have to be chosen such that a fault is induced in `LD1` but not in other instructions. Assuming that we would select cycle 4 for fault injection, Figure 3(a) shows the fault propagation to the memory stage of the final instruction `STD`. As we will demonstrate in the next sections, the analysis of the fault propagation path may sometimes lead to additional candidates for fault injection.

4. CASE STUDY: FAULT ATTACKS ON SECURE EMBEDDED SOFTWARE

In this section, we will apply the principles of the proposed fault attack methodology on two case studies - one involving a DFIA analysis on AES and the other involving an attack on software-implemented fault countermeasures.

4.1. CASE STUDY I: DFIA on TBOX AES

This section builds a DFIA on the software implementation of the AES algorithm on LEON3. We implemented the AES algorithm as a TBOX design. TBOX design is a word-oriented implementation suited for 32-bit microprocessors. The details of the DFIA attack on the AES algorithm can be found in Ghalaty et al. [2014].

4.1.1. Algorithm-level Analysis. The objective of the DFIA attack on the AES algorithm is to mount a biased fault injection attack on the output of AES round-9. 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 = Te0[ s0 >> 24    ] ^
Te1[(s1 >> 16) & 0xff] ^
Te2[(s2 >>  8) & 0xff] ^
Te3[s3 & 0xff        ] ^
rk[36];      //Target for biased fault injection: Output of Round 9: t0
```

4.1.2. Instruction-level Analysis. Now, we must define the set of instructions that would lead to the required fault model. The following code shows some instructions in the `AddRoundKey` function of the TBOX AES in round-9. The instructions in range of [48, 5*c*] are the instructions that are doing shift and XOR operations on the state word (%g1). Instruction *LDI7* loads the key word, and instruction *XOR8* applies a bit-wise XOR operation on the key and the state words.

```
48: LD  [%l3 + %o3], %o2     ;LD1
4C: SLL %o4, 2, %o4          ;SLL2
50: LD  [%o7 + %o4], %o5     ;LD3
54: XOR %g1, %o2, %g1        ;XOR4
58: LD  [%fp + 0x4c], %l2    ;LD5
```

```
5C: XOR %g1, %o5, %g1        ;XOR6
60: LD  [%l2 + 0x98], %o4    ;LD7
64: XOR %g1, %o4, %l6        ;XOR8
68: SRL %i5, 0xe, %o5        ;SRL9
6C: SRL %i4, 0x18, %g1       ;SRL10
```

4.1.3. Microarchitecture-level Analysis. To apply DFIA to a RISC pipeline, we need to induce biased data errors into selected instructions. Biased data errors mean that the effects of the injected fault must be revealed on the data computation rather than the control flow of the program or the instruction opcodes. Therefore, we will consider a (*pipeline stage, instruction*) pair as a valid fault injection target if attacking that pair yields a biased fault.

Figure 4 shows the behavior of the previously analyzed instructions in the pipeline. There are some data dependencies between the instructions for TBOX as well which are shown by blue 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. The potential points of observing biased fault is shown by red circled instructions in the pipeline in the Figure 4.

In order to perform an efficient attack and obtain useful faulty values, the adversary must only affect the circled pipeline stages only and avoid affecting other pipeline stages in a cycle. 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 cycle of this function. In Figures 5(a)–5(d), we show four of these targets. Based on analyzing the instructions that might be affected 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 5(a)): The only blue operations in cycle 6 are the XOR4(E) and LD3(M). Figure 5(a) shows the length of critical path for each operation in cycle 6. 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),

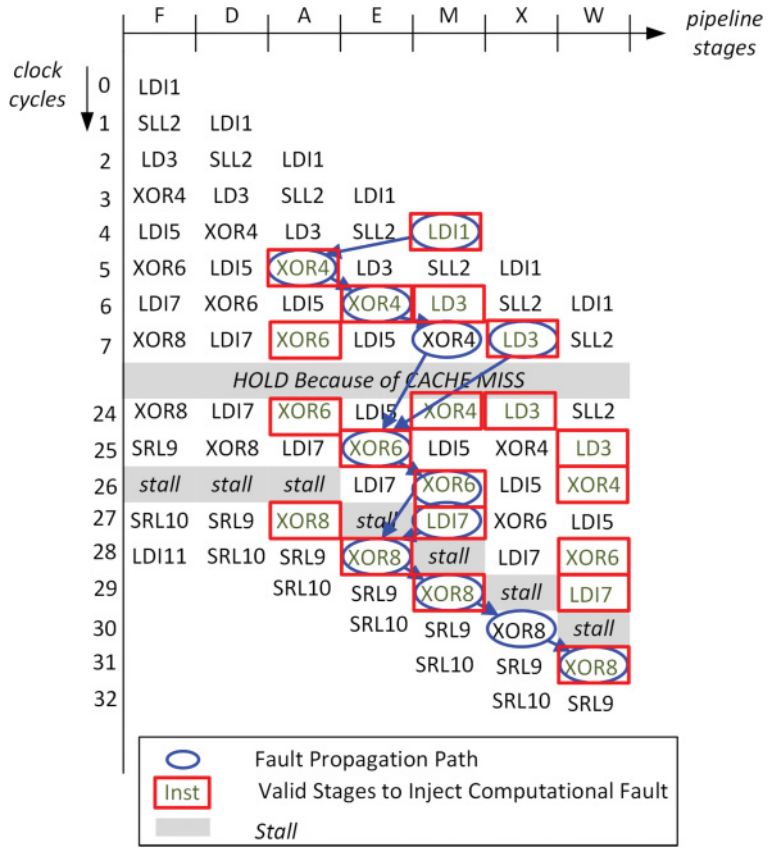

Fig. 4. Pipeline behavior for AddRoundKey of TBOX AES.

which is 5.45ns. Since LDI5(A) is an invalid fault injection target, Figure 4, the maximum fault intensity can be 5.45ns.

(2) *Fault Injection in Cycle 25* (Figure 5(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 5(c)): In this figure, we are allowed to affect XOR8(A) and LDI7(M). We can only affect LDI7(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 5(d)): In this figure, we are allowed to effect XOR8(E) and 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 an invalid target of fault injection in Figure 4.

Using this analysis, we launched the DFIA attack on the TBOX implementation of the AES algorithm. The number of required fault injections and the results of this attack will be discussed in Section 7.

4.2. CASE STUDY II: Analysis of Instruction-level Countermeasures on LEON3 Pipeline

The redundancy-based, instruction-level countermeasures against fault attacks have been trusted for years. Traditionally, it is assumed that breaking these

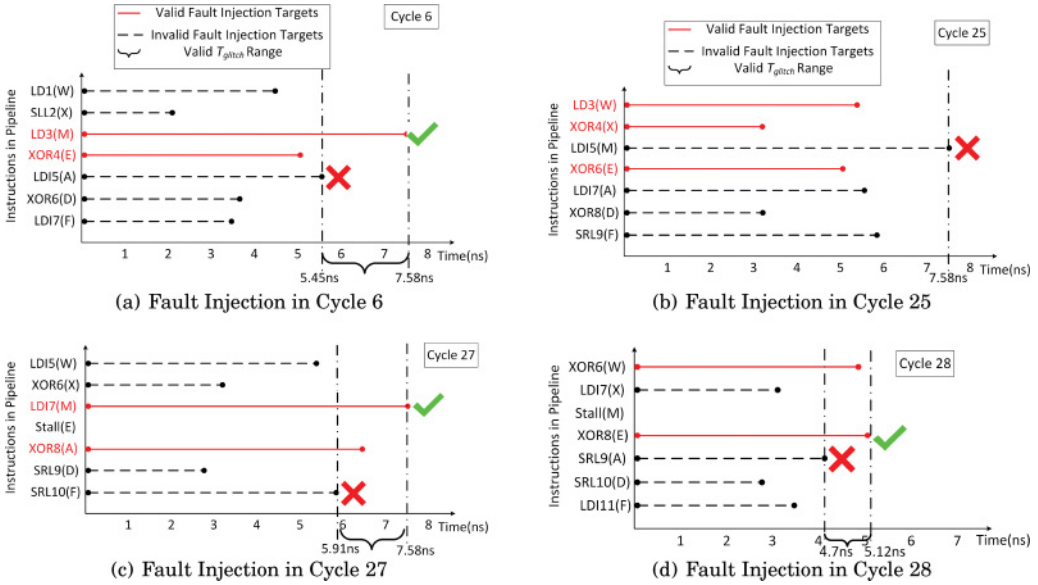

Fig. 5. Fault sensitivity analysis for different target cycles of the TBOX AES.

countermeasures requires multiple identical faults, which can be achieved by expensive fault injection setups with back-to-back injection capabilities [Barenghi et al. 2010; Moro et al. 2013b]. In this section, to break common instruction-level countermeasures, we describe different attack scenarios that can be potentially achieved by single glitches without requiring back-to-back injections. We will use a microarchitecture-level analysis to identify the attack scenarios. In Section 7, we experimentally verify the defined scenarios with single clock glitch injections.

4.2.1. Instruction Duplication Countermeasure. In the Instruction Duplication countermeasure [Barenghi et al. 2010], sensitive instructions are duplicated selectively and their computation results are stored in different destinations. Then, these results are compared to detect faults.

Figure 6 illustrates the Instruction Duplication countermeasure on a memory-load (`LD1`) instruction and its behavior on the pipeline. 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.

The gray instructions show stalls in the pipeline. *Stall 1* in Figure 6 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 that modifies the condition codes, 2 cycles of delay are added to allow the condition to be computed.

The target of fault injection to thwart this countermeasure can have two forms. The first involves injecting two identical faults in each of the `LD` instructions to bypass the equality check. The second is 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 earlier. The first step is to define the valid targets of fault injection in each

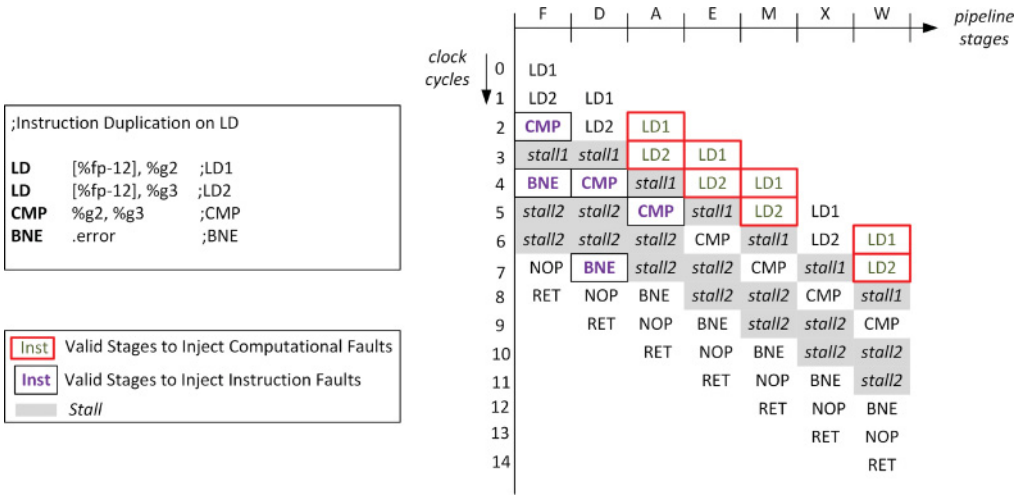

Fig. 6. Pipeline behavior for Instruction Duplication countermeasure for LD instruction.

Table II. Attack Scenarios to Thwart Instruction Duplication Countermeasure

Scenarios	# of Glitch Injections	Targeted Cycles	Instruction, Fault Type			
			LD1	LD2	CMP	BNE
A.1.	1	3	CF	CF	-	-
A.2.	1	2	CF	-	IF	-
		4	CF	-	IF	-
			CF	-	-	IF

cycle. Based on the previous analysis, 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. Affecting the black squared pairs of (*instruction, pipeline stages*) will cause instruction faults. For example, targeting the `LD` instructions in *Stage M* or *W* 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 II summarizes the two potential 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. 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

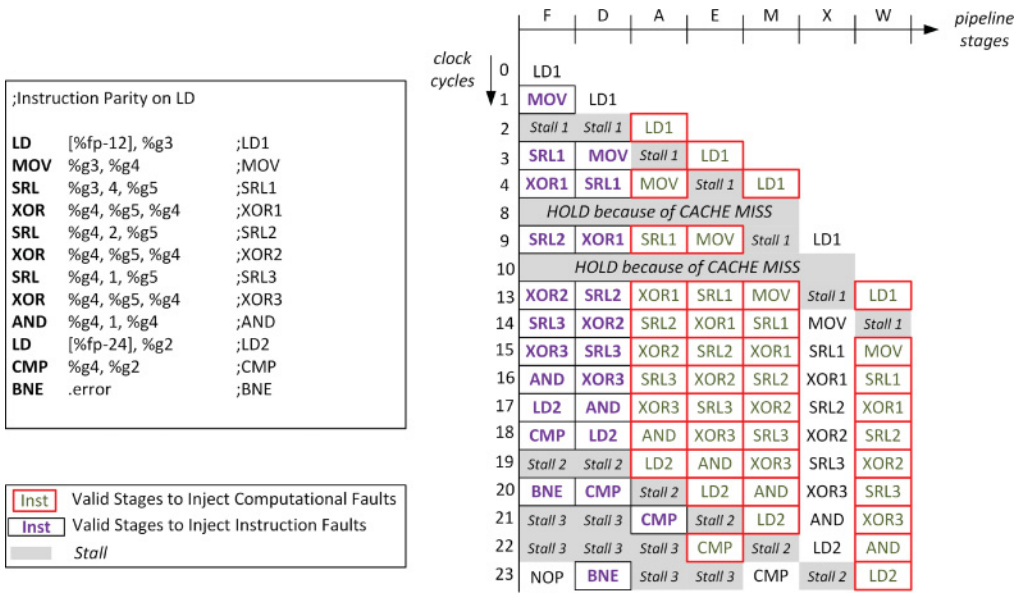

Fig. 7. Pipeline behavior for instruction parity countermeasure for LD instruction.

Table III. Fault Attack Scenarios to Thwart Instruction Parity Countermeasure

Scenarios	# of Glitch Injections	Targeted Cycles	Instruction, Fault Type				
			LD1	SRL1-AND	LD2	CMP	BNE
C.1.	1	2,3,4,13	CF	-	-	-	-
C.2.	1	3,4,13	CF	IF	-	-	-
C.3.	1	2,20	CF	-	-	-	IF
			CF	-	-	IF	-

pipeline, these cycles can be *Cycle 2* that affects `CMP(F)` or *Cycle 4* that can affect `CMP(D)` or `BNE(F)`.

4.2.2. Instruction Parity Countermeasure. This software countermeasure is proposed by Barenghi et al. [2010]. 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.

Figure 7 shows an example of the parity countermeasure. In this example, the protected instruction 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.

As shown in Figure 7, 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 in the following text, which 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

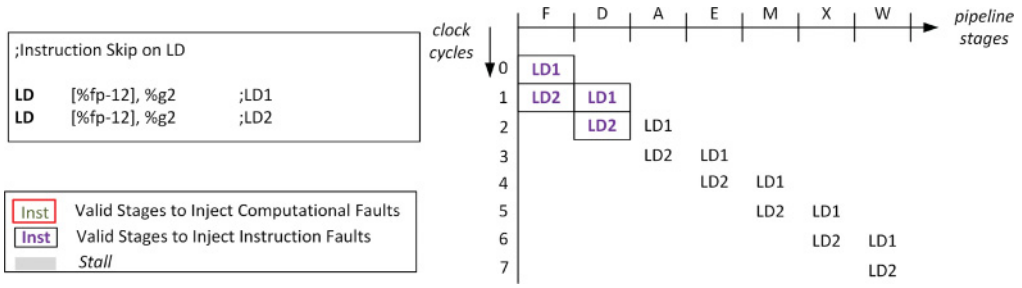

Fig. 8. Pipeline behavior for instruction skip countermeasure for LD instruction.

Table IV. Fault Attack Scenarios to Thwart Instruction Skip Countermeasure

Scenarios	# of Glitch Injections	Targeted Cycles	Instruction, Fault Type	
			LD1	LD2
D.1.	1	1	IF	IF

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 instructional operations on `SRL1(F-D-E)`, `MOV(D-M)`, `XOR1(F-A)`, `XOR2(F)`, `SRL2(D)`, `XOR1(F)`.

(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)`.

4.2.3. Instruction Skip Countermeasure. This countermeasure is proposed by Moro et al. [2013b]. 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 a computational fault model, but it can only change the opcode of the protected instruction. The following shows the assembly code for the instruction skip countermeasure on `LD` instruction.

The pipeline behavior for the instruction skip countermeasure is shown in Figure 8. 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).

5. EXPERIMENTAL SETUP

The example fault model in Section 3.2 characterizes the sensitivity of the target processor to setup-time violation faults. Our fault injection and measurement setup injects

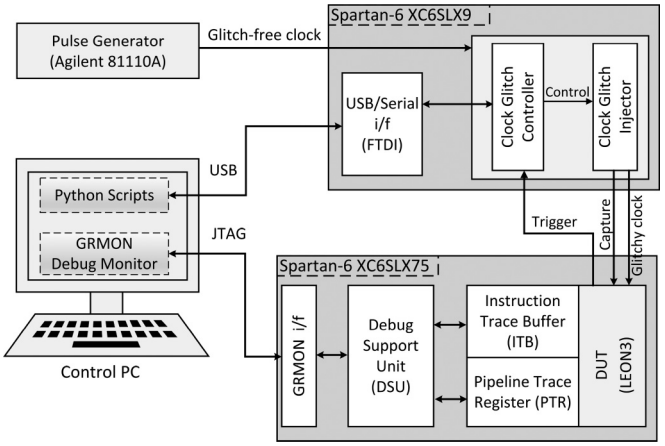

Fig. 9. Block diagram of experimental setup.

setup-time violation faults via clock glitching: A clock glitch causes timing violations by temporarily shortening the clock period of the target processor. Our setup can generate clock glitches up to a few hundreds of MHz, which is sufficient to demonstrate our method on our target. For more advanced targets, other methods such as voltage glitching can also be used to generate timing violations [Timmers et al. 2016; Giller 2015].

Figure 9 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 [Satoh 2013]. 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 a 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 the *capture* signal for latching values of pipeline registers into the Pipeline Trace Register (PTR) after glitch injection. These signals are later analyzed to understand the effect of fault injection on pipeline registers.

5.1. 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 9 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 a 24MHz clock input generated by a pulse generator (Agilent 81110A). The generation of glitches is done with two variable phase shift modules [Endo et al. 2011].

The Control PC communicates with the glitcher via a 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

20*ns* with 100*ps* 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.

5.2. Implementation of Data Acquisition

The bottom part of Figure 9 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 [Gaisler 2016a], and a 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 a 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 the *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 [Gaisler 2016b] 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 breakpoints and watchpoints, and reading the ITB and PTR. GRMON connects to the on-chip components via a Joint Test Action Group 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 microarchitecture of the LEON3 processor.

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

6. EXPERIMENTAL EVALUATION OF CASE STUDY I

In this section, we provide experimental results to demonstrate the impact of the proposed method on the efficiency of fault injection and fault analysis parts of the DFIA attack. In the injection part, an adversary applies the physical stress (e.g., clock glitches) and collects faulty outputs. In the analysis part, the collected faults are analyzed to retrieve the secret key. We attacked the TBOX AES program running on FPGA implementation of LEON3 for two cases:

(1) **Grey-box approach:** In this case, the adversary uses the information provided by the instruction fault sensitivity model. The adversary also has the knowledge of the software behavior in the pipeline. Therefore, the adversary can identify the

Table V. Comparison of Fault Injection Cost for Black-box and Grey-box Attack Strategies on TBOX AES

	Number of Attacked Cycles	Tglitch Range	Number of Fault Injections (Step size = 162ps)
Black-box Approach	16	[3.0, 15.8]	1,280
Grey-box Approach	9	[5.43, 6.59]	81

most suitable clock cycles and fault intensity range for fault injection, aiming at creating biased data faults.

(2) **Black-box approach:** In this case, the adversary does not have the instruction fault sensitivity model. 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 approaches [Korak and Hoefler 2014; Balasch et al. 2011; Moro et al. 2013a]. 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, our purpose was retrieving the AES secret key with the DFIA attack. 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 glitch width T_{glitch}. Here, we present our results for retrieving 1 byte of the AES key.

The adversary starts with the fault injection part to collect biased data faults, which are required for the fault analysis part of the DFIA attack. In the grey-box strategy, the adversary uses MAFIA (Section 3) to find the most suitable points and injection parameters for biased data fault injection. Then, he injects faults into these points in the execution of AES software and collects faulty ciphertexts.

In the black-box case, the adversary has a limited knowledge of the target system. Therefore, to collect biased data faults, he has to exhaustively inject faults into all points in the execution of software and hope they will lead to biased data faults. However, in this approach, a large number of 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 cause the fetching of 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 V demonstrates the effect of the proposed method on the efficiency of the fault injection part of the attack. The table shows T_{glitch} range and the number of clock cycles that an adversary can exploit to create biased data faults. The first column of the table lists the number of clock cycles during which a fault injection attempt might yield a biased data fault. The total number of these cycles for the TBOX AES is 16 (Figure 4). 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 grey-box case, this number reduces to nine with the microarchitecture-aware fault attack strategy.

The second column of Table V shows the T_{glitch} range in which the adversary attempts to inject biased data faults. 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 width (i.e., fault intensity) values. For the grey-box case, this range will be reduced due to the proposed methodology. For every target cycle, we will

Table VI. Fault Injection Results on Instruction Duplication Countermeasure

Glitching Scenario	Target of Fault Injection	Glitch Width (ns)	Impacted Instruction(s)	Observed Fault Effect
A.1. Single Glitch	-	-	-	-
A.2. Single Glitch	*Cycle 2*	9.0 - 12.6	LD1, A CMP, F	Fault in %g2 CMP to SRL
A.2. Single Glitch	*Cycle 4*	12.0 - 14.6	LD1, M BNE, F	Fault in %g2 BNE to NOP

have a different valid T_{glitch} range as it is previously shown. In our case, there are nine valid T_{glitch} ranges. In Table V, we list the average lower and upper bound values for the grey-box approach rather than showing bounds of each *Tglitch* range.

The third column of Table V is the total number of fault injection attempts for *162*ps step-size. These numbers show the required fault injection attempts to explore all possible T_{glitch} values that might yield biased data faults. We obtained these numbers 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 (1,280) is greater than the grey-box case (81). As a result, the proposed methodology significantly increases the efficiency of the fault injection part by reducing the effort for biased fault injection.

In the fault analysis part, the adversary uses the collected faults to retrieve the secret key byte. In this experiment, the DFIA attack was able to retrieve the key byte with 31 faults in the black-box case, and with 17 fault injections in the grey-box case. In other words, the faults in the grey-box case provide more information on the secret key then the faults in the black-box case do. This is expected as the proposed method increases the control of the adversary on the induced fault effects.

In conclusion, these experimental results demonstrate that DFIA attacks are feasible on pipelined RISC processors for the black-box and grey-box approaches. They also show that the grey-box strategy significantly reduces the fault injection effort required to create biased data faults, and it enables more efficient DFIA attacks.

7. EXPERIMENTAL EVALUATION OF CASE STUDY II

For each attack scenario and countermeasure investigated in Section 5, we launched a clock glitch injection campaign using our experimental setup. In the following paragraphs, we will list the observed faulty behavior for each countermeasure.

Table VI shows the results of our experiments on the **Instruction Duplication** countermeasure. The first column of this table shows the fault injection scenario. Column 2 and 3 show the clock cycle for the fault injection and the glitch width, 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 *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`.

Table VII shows the result of fault injection into the **Instruction Parity** coun-

Table VII. Fault Injection Results on Instruction Parity Countermeasure

Glitching Scenario	Target of Fault Injection	Glitch Width (ns)	Impacted Instruction(s)	Observed Fault Effect
C.1. Single Glitch	*Cycle 4*	13.2	LD1, M	Fault in %g3
C.2. Single Glitch	*Cycle 13*	12.1- 13.3	LD1,W XOR2, E	Fault in %g3 XOR to NOP

Table VIII. Fault Injection Results on Instruction Skip Countermeasure

Glitch Scenario	Target of Fault Injection	Glitch Width (ns)	Impacted Instruction(s)	Observed Fault Effect
D.1. Single Glitch	*Cycle 1*	9.0 - 9.2	LD1, D LD2, F	LD1 to SETHI LD2 to NOP

termeasure. 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 XOR instruction changes to a NOP instruction.

Table VIII shows the result for the **Instruction Skip** countermeasure. As shown, with single, we are able to change the opcode of the LD instructions to other opcodes and skip two consecutive instructions. For example, by injecting fault in the *Cycle 1*, we converted LD1 instruction into a SETHI instruction and LD2 instruction into a TADCC instruction. The SETHI instruction uses the least significant 22 bits of the instruction value, and writes this value into the destination register (%g2 in this case). TADCC is a special addition instruction and it does not update any register; it is effectively a NOP. As a result, we update the destination register with a random value. Other scenarios of fault injection are explained in Yuce et al. [2016b].

8. CONCLUSIONS

This work was motivated by the need for an efficient fault model, which can explain the fault effects experienced by a software program in the instruction-set-architecture level, and can guide the fault injection process to induce the desired fault effects in the program. To fulfill these requirements, we developed a fault sensitivity model for the underlying processor on which the embedded program is executed. This model allows an adversary to identify the potential fault effects on every (*instruction*, *pipeline stage*) pair. It also enables the adversary to tune the fault injection parameters to induce the desired effects in each (*instruction*, *pipeline stage*) pair.

Relying on this model, we also built a systematic fault attack methodology, so-called MAFIA. MAFIA enables an adversary to launch an efficient fault attack on an embedded software by exploiting different layers of abstraction. The adversary starts with algorithm-level analysis to determine application-specific fault injection and analysis objectives. Then, the adversary studies the software implementation of the algorithm in the instruction level and finds the candidate (*instruction*, *pipeline stage*) pairs for fault injection. Finally, the adversary examines the execution of the instructions on the pipeline to determine clock cycles to inject faults as well as the fault injection parameters to create the desired effects.

We showed the efficiency of MAFIA with two case studies. First, we developed a DFIA attack on an unprotected AES software program running, and we showed that the use of the proposed methodology reduced the number of fault injections an order of magnitude in comparison to the traditional attack methodology. Second, we studied

instruction-level software countermeasures. We identified their vulnerabilities using our method, and demonstrated that all of the studied countermeasures can be broken with single fault injections with low-cost injection setups such as clock glitching. We also experimentally demonstrated the attacks on an FPGA implementation of LEON3 processor. Finally, we conclude that efficient countermeasures to protect embeddedsoftware must consider multiple abstraction layers [Yuce et al. 2016a].

REFERENCES

Subidh Ali and Debdeep Mukhopadhyay. 2011. An improved differential fault analysis on AES-256. In *Proc. of AFRICACRYPT'11*. 332–347.

Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. 2011. An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In *Proc. of FDTC'11*. 105–114.

Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2006. The sorcerer's apprentice guide to fault attacks. *Proc. IEEE* 94, 2 (2006), 370–382.

Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. 2009. Low voltage fault attacks on the RSA cryptosystem. In *Proc. of FDTC'09*. 23–31.

Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. 2012. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proc. IEEE* 100, 11 (Nov 2012), 3056–3076.

Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. 2010. Countermeasures against fault attacks on software implemented AES: Effectiveness and cost. In *Proc. of WESS'10*. 7:1–7:10.

Alessandro Barenghi, Luca Breveglieri, Andrea Palomba, and Gerardo Pelosi. 2015. Fault sensitivity analysis at design time. In *Trusted Computing for Embedded Systems*. Springer, 175–186.

Fabrice Bellard. 2005. QEMU, A fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference, FREENIX Track*. 41–46.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and others. 2011. The gem5 simulator. *ACM SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

Franck Courbon, Philippe Loubet-Moundi, Jacques J. A. Fournier, and Assia Tria. 2014. Adjusting laser injections for fully controlled faults. In *Proc. of COSADE'14*. 229–242.

Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. 2011. An on-chip glitchy-clock generator for testing fault injection attacks. *J. Cryptographic Eng.* 1, 4 (2011), 265–270.

Claudio Ferretti, Silvia Mella, and Filippo Melzani. 2014. The role of the fault model in DFA against AES. In *Proc. of HASP'14*. 4:1–4:8.

Jiri Gaisler. 2016a. GRLIB IP library. Retrieved June 20, 2016 from http://www.gaisler.com/index.php/products/ipcores/soclibrary.

Jiri Gaisler. 2016b. GRMON2 Debug Monitor. Retrieved June 20, 2016 from http://www.gaisler.com/index.php/products/debug-tools/grmon2.

Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2015. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. Cryptology ePrint Archive, Report 2016/230. (2016). http://eprint.iacr.org/.

Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. 2014. Differential fault intensity analysis. In *Proc. of FDTC'14*. 49–58.

Brett Giller. 2015. Implementing Practical Electrical Glitching Attacks. Retrieved from https://www.blackhat.com/docs/eu-15/materials/eu-15-Giller-Implementing-Electrical-Glitching-Attacks.pdf.

Christophe Giraud. 2005. DFA on AES. In *Advanced Encryption Standard–AES*. Springer, 27–41.

Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2015. Rowhammer.js: A Remote Software-induced Fault Attack in Javascript. arXiv preprint arXiv:1507.06955.

Yu-ichi Hayashi, Naofumi Homma, Takaaki Mizuki, Takafumi Aoki, and Hideaki Sone. 2015. Fundamental study on fault occurrence mechanisms by intentional electromagnetic interference using impulses. In *Proc. of APEMC'15*. 585–588.

Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, Get Off My Cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898. (2015). http://eprint.iacr.org/.

Marc Joye and Michael Tunstall. 2012. *Fault Analysis in Cryptography*. Springer.

Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. 2013. Hardware designer's guide to fault attacks. *IEEE Trans. VLSI Syst.* 21, 12 (2013), 2295–2306.

Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Moderator-Ravi. 2004. Security as a new dimension in embedded system design. In *Proc. of the DAC'04*. 753–760.

Thomas Korak and Michael Hoefler. 2014. On the effects of clock and power supply tampering on two microcontroller platforms. In *Proc. of FDTC'14*. 8–17.

Kerstin Lemke-Rust and Christof Paar. 2006. An adversarial model for fault analysis against low-cost cryptographic devices. In *Proc. of FDTC'06*. 131–143.

Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. 2010. Fault sensitivity analysis. In *Proc. of CHES'10*. 320–334.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proc. of the IEEE Symposium on Security and Privacy*. 605–622.

Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power analysis attacks - revealing the secrets of smart cards*. Springer.

Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013a. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *Proc. of FDTC'13*. 77–88.

Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. 2013b. Formal verification of a software countermeasure against instruction skip attacks. Cryptology ePrint Archive, Report 2013/679. Retrieved from http://eprint.iacr.org/.

Martin Otto. 2005. *Fault Attacks and Countermeasures*. Ph.D. Dissertation. University of Paderborn.

Gilles Piret and Jean-Jacques Quisquater. 2003. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In *Proc. of CHES'03*. 77–88.

Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. 2015. Fault attacks, injection techniques and tools for simulation. In *Proc. of DTIS'15*. 1–6.

Kazuo Sakiyama, Yang Li, Mitsugu Iwamoto, and Kazuo Ohta. 2012. Information-theoretic approach to optimal differential fault analysis. *IEEE Trans. on Inf. Forensics Security* 7, 1 (2012), 109–120.

Akashi Satoh. 2013. SAKURA specifications. (2013). Retrieved June 20, 2016 from http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G_Spec_Ver1.0_English.pdf.

Takeshi Sugawara, Daisuke Suzuki, and Toshihiro Katashita. 2012. Circuit simulation for fault sensitivity analysis and its application to cryptographic LSI. In *Proc. of FDTC'12*. 16–23.

Niek Timmers, Albert Spruyt, and Marc Witteman. 2016. Controlling PC on ARM using fault injection. In *Proc. of FDTC'16*. 25–35.

Jasper G. J. Van Woudenberg, Marc F Witteman, and Federico Menarini. 2011. Practical optical fault injection on secure microcontrollers. In *Proc. of FDTC'11*. IEEE, 91–99.

Bilgiday Yuce, Nahid F. Ghalaty, Chinmay Deshpande, Conor Patrick, Leyla Nazhandali, and Patrick Schaumont. 2016a. FAME: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In *Proc. of HASP'16*. 8.

Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. 2016b. Software fault resistance is futile: Effective single-glitch attacks. In *Proc. of FDTC'16*. 47–58.

Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. 2015a. Improving fault attacks on embedded software using RISC pipeline characterization. In *Proc. of FDTC'15*. 97–108.

Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. 2015b. TVVF: Estimating the vulnerability of hardware cryptosystems against timing violation attacks. In *Proc. of HOST'15*. 72–77.

Loic Zussa, Jean-Max Dutertre, Jessy Clédiere, Bruno Robisson, Assia Tria, and others. 2012. Investigation of timing constraints violation as a fault injection means. In *Proc. of DCIS'12*.

Received June 2016; revised November 2016; accepted February 2017