Journal of Hardware and Systems Security (2018) 2:111–130
https://doi.org/10.1007/s41635-018-0038-1

Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation

Bilgiday Yuce[1] · Patrick Schaumont[1] · Marc Witteman[2]

Received: 2 February 2018 / Accepted: 26 April 2018 / Published online: 10 May 2018

Abstract

Embedded software is developed under the assumption that hardware execution is always correct. Fault attacks break and exploit that assumption. Through the careful introduction of targeted faults, an adversary modifies the control flow or data flow integrity of software. The modified program execution is then analyzed and used as a source of information leakage, or as a mechanism for privilege escalation. Due to the increasing complexity of modern embedded systems, and due to the difficulty of guaranteeing correct hardware execution even under a weak adversary, fault attacks are a growing threat. For example, the assumption *that an adversary has to be close to the physical execution of software, in order to inject* an exploitable fault into hardware, has repeatedly been shown to be incorrect. This article is a review on hardware-based fault attacks on software, with emphasis on the context of embedded systems. We present a detailed discussion of the anatomy of a fault attack, and we make a review of fault attack evaluation techniques. The paper emphasizes the perspective from the attacker, rather than the perspective of countermeasure development. However, we emphasize that improvements to countermeasures often build on insight into the attacks.

Keywords Fault attacks · Secure embedded software · Embedded systems

1 Introduction

In this paper, we consider the fault attack threat against secure embedded software. Software plays a crucial role in the functionality of embedded computers. For example, in a System on Chip, software provides flexibility and it provides the logical integration of specialized hardware components. *Secure* embedded software is any software that employs security mechanisms (e.g., confidentiality, integrity, authentication, and access control mechanisms) to ensure the security of sensitive data and functionality. Secure embedded software therefore not only is limited to cryptographic software but also covers access control and permission-rights management.

Embedded computers that run secure embedded software are all around us. A large portion of the information ecosystem consists of embedded connected computers that participate in the physical control and the measurement of critical infrastructures and utilities, such as smart grid and automotive and industrial controls. In addition, information technology is pervasive in the immediate vicinity of people such as in cell phones, activity trackers, medical devices, or biometric tokens. The data handled by these embedded computers is sensitive. Computing devices in critical infrastructure execute safety-critical commands and collect sensitive measurement data protected by cryptographic keys and authentication codes. Human-centric information systems work with private end-user data, passwords, PIN codes, biometric data, location history, and usage patterns. Furthermore, the firmware and configuration data of embedded computing systems may represent valuable intellectual property.

The data and the unauthorized access of these embedded devices is an obvious target for attackers. At high level, the purpose of an attacker is to obtain control over the execution of the embedded software, or to extract internal data

✉ Bilgiday Yuce
bilgiday@vt.edu

Patrick Schaumont
schaum@vt.edu

Marc Witteman
witteman@riscure.com

1 The Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

2 Riscure – Security Lab, Delft, Netherlands

values processed by the embedded software. We identify three different attackers, distinguished by the abstraction level they operate on. The *input-output* attacker manipulates the data inputs of an embedded software application to trigger internal buffer overflows or internal software bugs in the application. The *memory* attacker co-exists with the embedded software application, for example as a malicious software task, and snoops the memory space in order to directly manipulate or observe a secure embedded application [1, 2]. Both of these attackers succeed because they break an implicit assumption made by the secure embedded software application. The input-output attacker exploits the assumption that there are no malformed inputs to the program. Using malformed inputs, the attacker exploits bugs in secure embedded software such as missing memory bounds checks. The memory attacker exploits the assumption that memory space is private to the secure embedded application. This privacy gets lost when the architecture cannot provide memory region isolation to the application [3]

The third type of attacker, the *hardware fault* attacker, is the focus of this paper. Like the previous two, the hardware fault attacker breaks an implicit assumption made by the secure embedded software application. In this case, the assumption is that the embedded hardware guarantees correct execution of the software. Table 1 illustrates that the correct execution of software builds on many interdependent assumptions at different levels of abstraction in the hardware. Yet, *any* of the abstraction levels is a potential target for the hardware fault attacker.

– At the *instruction level*, a programmer assumes that the opcodes executed by the embedded hardware (the microprocessor) are correct. A hardware fault attacker who can manipulate opcodes can change the meaning of a program.
– At the *micro-architecture level*, a programmer assumes that correct opcodes imply correct execution of the instruction. A hardware fault attacker who can manipulate the micro-architecture can break this assumption and still change the meaning of a program.
– At the *circuit level*, the correct execution of software requires that digital logic in the processor will operate with the proper timing, and using the proper voltage levels to capture digital-0 and digital-1. A hardware fault attacker who can influence circuit timing or logic threshold levels can change the correctness of digital execution and hence still change the meaning of a program.
– At the lowest abstraction level, the correct execution of software requires that the physical environment of a digital circuit has nominal operating conditions, that it is using the proper temperature, the proper circuit voltage, and the proper electromagnetic environment. A hardware fault attacker who can influence any of these parameters can change the correctness of the architecture and hence still change the meaning of a program.

This paper is only concerned with the hardware *fault* attacker, that is, an attacker who builds on the manipulation of execution correctness at the architecture level or below. We do not ignore side-channel attacks that build on the observation of the physical effects of computing. In fact, some of the recent fault attacks successfully combine ideas of side-channel attacks with fault analysis. However, for this paper, we consistently develop the viewpoint of the hardware fault attacker as a threat to secure embedded software. An interested reader may refer to the existing works [4–8] for the viewpoint of the fault-resistant system designers.

It may appear as if a hardware attacker must be physically close to the digital architecture to break execution correctness (at any abstraction level). However, this is not always the case. A hardware fault attacker can be a physical entity (in hardware) or a virtual entity (in software). In the latter case, the attacker is logically present as somebody who runs attack software in conjunction with a victim program. Recent work has shown that execution correctness of the hardware can be manipulated by such a software attacker.

This paper addresses the following questions in detail.

– What are the common techniques for fault injection in a digital architecture, and how do faults appear as a result of fault injection?
– How do faults propagate through the micro-architecture and across the architecture level into the secure embedded software?

Table 1 Embedded software targets for the hardware attacker

Abstraction level	Cause of security failure	Attacker
Input/output	Software bugs	Input/output attacker
Application level	Lack of memory region isolation	Memory attacker
Instruction level	Opcode modification	Hardware attacker
Micro-architecture level	Instruction execution is wrong	
Circuit level	Timing, threshold levels are not met	
Environment	Operating conditions are abnormal	

– How does the attacker exploit these faults towards a fault attack?
– What testing equipment can be used to study the fault injection, propagation and exploitation of secure embedded software?

This paper is organized as follows. In the next section, we develop a systematic threat model against embedded software from the viewpoint of fault attacks, and we break down a fault attack into smaller steps. In Section 3, we describe commonly used fault injection techniques. In Section 4, we investigate the impact of fault injection on microprocessor execution. In Section 5, we analyze how faults propagate from the micro-architecture into the embedded software functionality. Section 6 describes commonly used fault exploitation techniques. Section 7 describes fault attack evaluation technologies, and certification of embedded software against fault attacks. We then conclude the paper.

2 Background

2.1 Threat Model

The aim of a fault attack is breaching the security of a software program by forcing a security-sensitive asset into unintended behavior. For this purpose, the adversary injects well-crafted, targeted hardware faults by deliberately altering the operating conditions of the microprocessor that runs the target software. Then, the adversary exploits the effects of the faults on the target software's execution and breaks the security. Consequently, the target of exploitation is the software layer while the origin of vulnerability (i.e., faults) is the hardware layer.

In a typical fault attack, the adversary is not capable of directly modifying or monitoring the internals of a chip, or changing the binary of a program. The adversary is able to alter the execution of a target program by controlling the physical operating conditions (e.g., timing, supply voltage, temperature) of the processor hardware executing the program. The adversary can also provide input to the target program and observe the effects of abnormal operating conditions on the software execution through system output or a related side-channel such as power consumption, cache-activity-related timing, and performance counters.

2.2 Using Faults as a Hacking Tool

Figure 1 illustrates the steps and mechanisms involved in a typical fault attack on embedded software. A fault attack consists of two main phases, fault attack design and fault attack implementation (steps 1–5 in Fig. 1).

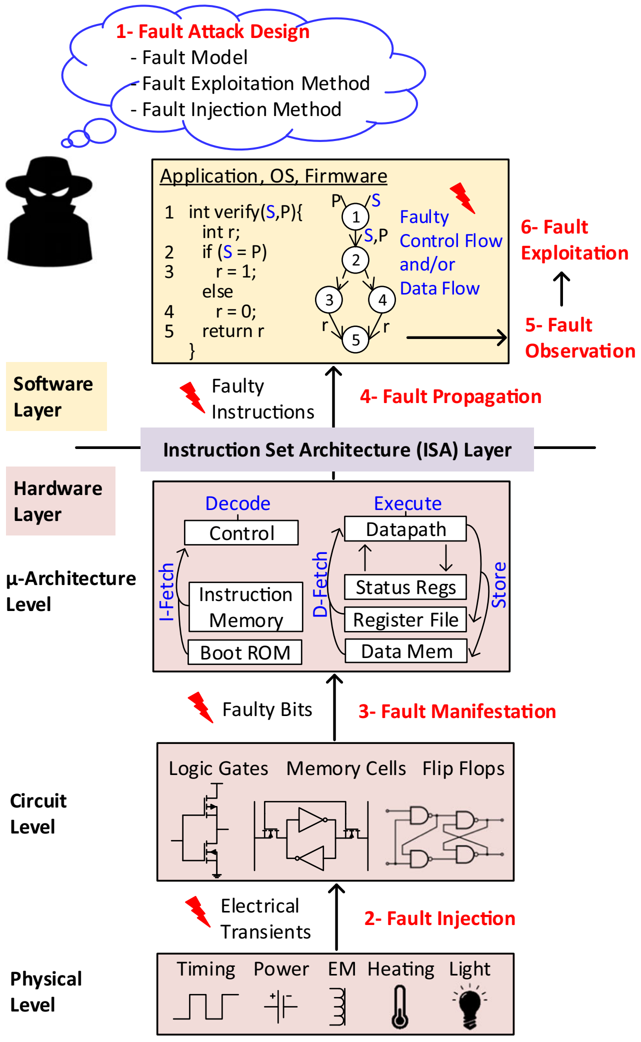

Fig. 1 Anatomy of a typical fault attack on embedded software: The target of fault injection is the hardware while the target of exploitation is the software

In the design step, the adversary analyzes the target to determine fault model (i.e., an assumption on the faults to be injected), fault exploitation method, and fault injection technique. For instance, an adversary may intend to inject faults into several assets such as an encryption program, a security-related verification code, a memory transfer function, the processor state register, a system call, the firmware, or configuration information of the target device. The adversary may then exploit the fault effects on the target asset for various attack objectives such as weakening the security, bypassing security checks, intellectual property theft, extracting the confidential data, privilege escalation, activating debug modes, and disabling secure boot of the device.

The implementation phase is a combination of five steps:

1. **Fault Injection:** In this step, the adversary applies a physical stress on the microprocessor to alter its

physical operating conditions and to induce hardware faults. The applied physical stress can be in various forms such as clock glitches, supply voltage glitches, electromagnetic (EM) pulses, and laser shots.

To induce the desired faults, the adversary varies *fault timing* and *fault intensity*. Fault timing specifies when the physical stress is applied on the target processor. Fault intensity is the degree of the physical stress by which the microprocessor hardware is pushed beyond 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.

2. **Fault Manifestation**: The circuit-level effect of fault injection is creating electrical transients on the nets, combinational gates, flip-flops, or memory cells. A fault manifests at the micro-architecture level when the electrical transients are captured into a memory cell or flip-flop, and change its value.

   The number of manifested faulty bits in the micro-architecture level is correlated to the applied fault intensity: A gradual change in the fault intensity causes a gradual change in the manifested faults. 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 size (e.g., single byte) of the induced faults [9–11]. However, tuning the fault intensity alone is not sufficient to control the type (e.g., bit-set) and location (e.g., decode logic) of the induced faults. The adversary's control on the fault type and location is also affected by the type and precision of the fault injection equipment.

   The biased fault behavior also allows the adversary to find a critical fault intensity value, at which the electrical transients become strong enough to cause fault manifestation. That critical fault intensity value is called *fault sensitivity* of the target hardware [12].

3. **Fault Propagation**: In this step, the effects of the manifested faults are propagated to the software layer through execution of faulty instructions. The next two paragraphs briefly explain the mechanism behind fault propagation.

   Software security mechanisms are implemented as a sequence of instructions executed by the microprocessor hardware. In addition, each instruction goes through the *instruction-execution cycle* that consists of multiple steps carried out by a certain subset of available micro-architecture-level hardware blocks. The processor loads each instruction from program memory (*instruction-fetch*), then determines the meaning of the current instruction through its opcode (*instruction-decode*), then executes the current instruction (*instruction-execution*), and then updates the state of the processor based on the instruction's result (*instruction-store*). The number of steps in the instruction-execution cycle is architecture dependent, and it can vary considerably from one microprocessor to the next.

   The manifested faults may cause faulty bits in any micro-architectural hardware block such as instruction memory, controller, datapath and register file. The effects of the manifested faults are propagated to the software layer when an instruction uses the affected micro-architectural block. As each instruction uses a specific subset of the micro-architectural blocks, the precise effect of a hardware fault 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.

4. **Fault Observation:** An adversary needs to observe the effects faulty instructions in order to exploit them. An observable fault effect can be a faulty system output such as a faulty ciphertext, a side-channel information such as a sudden change in the power consumption, a single-bit information showing if fault injection was successful, or micro-architectural effects observed through performance counters [7, 13]. These effects become observable to the adversary when they are subsequent instructions that have data-dependencies or control-dependencies on the faulty instruction are executed.

5. **Fault Exploitation:** In the final step, the adversary exploits the observable fault effects and breaks the security. For example, the adversary can analyze the differential of the correct and faulty ciphertexts from a cipher to retrieve the secret key used for the encryption. For the same purpose, an adversary may also use a single-bit side-channel information of whether fault injection was successful. Similarly, the adversary may use the faults to trigger traditional logical attacks such as buffer overflows and privilege escalation.

3 Fault Injection Techniques

In a fault attack, it is essential to induce well-controlled faults during execution of the target software. An adversary achieves fault injection by deliberately applying physical stress to push the operating conditions of the underlying microprocessor hardware beyond their allowed margins. The adversary controls the induced faults through timing, location, and intensity of fault injection. The *timing of fault injection* is defined as the moment at which physical stress is applied to the processor. The *location of the fault injection* is the spatial portion of the processor that is exposed to physical stress. The *intensity of the fault injection* is the amount of physical stress applied to the processor.

This section discusses common techniques used for fault injection. We briefly describe main characteristics of each fault injection technique. We partition the fault injection techniques into two main categories (Fig. 2): Hardware-controlled fault injection and software-controlled fault injection.

Hardware-controlled fault injection techniques employ a separate external fault injection hardware to apply physical stress to the target hardware and induce faults in the victim software. Typically, the fault injection process is controlled by another software program (i.e., fault control software) running on the fault injection hardware. *In software-controlled fault injection techniques*, fault injection is controlled with a malicious software (i.e., fault injection and control software), which runs on the same hardware platform as the target software does. This malicious software alters the physical operating conditions of the target hardware to induce faults. While hardware-controlled techniques require physical proximity to the target system, software-controlled fault injection techniques enable remote fault attacks.

3.1 Hardware-Controlled Fault Injection Techniques

Several hardware-controlled fault injection techniques have been successfully demonstrated in the literature [7, 14]. The following sections provide an example list of commonly used techniques.

3.1.1 Tampering with Clock Pin

An adversary may inject faults by tampering with the external clock signal of the target device.

One way of exploiting the clock signal for fault injection is *overclocking* [15], in which the adversary persistently applies a higher-frequency clock signal than the nominal clock frequency of the device. This violates setup time constraints of the device and causes premature latching of the faulty values in flip-flops of the device [16]. The spatial precision of this method is low because the modifications in the external clock signal are distributed across the whole chip surface through a clock network. Similarly, the temporal precision of overclocking is also low because all of the clock cycles are affected by fault injection; the adversary cannot select the clock cycles to be affected by the fault injection. On the other hand, the adversary has a fine control on the fault intensity through clock frequency.

Another way of tampering with the clock signal is *clock glitching* [17], in which the adversary temporarily shortens the length of a single clock cycle. This causes setup time violations during the affected clock cycle. In comparison to overclocking, the adversary has a precise control on the temporal location (i.e., timing) of the fault injection. The intensity of the fault injection is controlled through the length of the glitched clock cycle. Similar to the overclocking, the spatial precision of clock glitching is low.

For the clock glitching and overclocking techniques, the state-of-the-art fault injection setups [18, 19] provide nanosecond-level temporal precision. The disadvantage of tampering with the clock signal is that this method requires physical access to an external clock pin. If a device uses an internally generated clock signal, using this method is infeasible.

3.1.2 Tampering with Supply Voltage Pin

An adversary can also inject faults by altering the external supply voltage of the target device. The adversary may use *underfeeding* [20], in which a lower voltage than the nominal voltage is supplied to the device. Lower supply voltage increases the delay of combinational paths. This

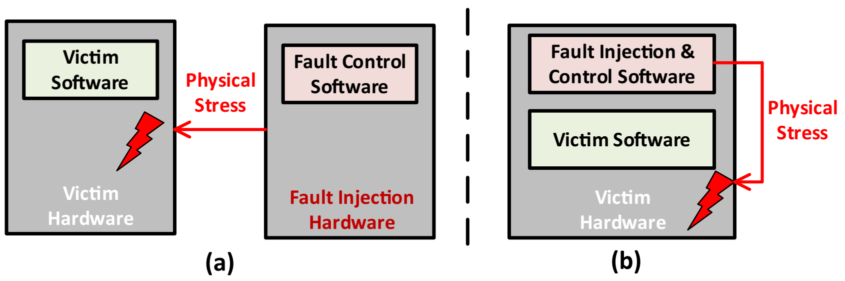

Fig. 2 Fault injection categories: **a** Hardware-controlled fault injection; **b** software-controlled fault injection

causes setup time violation when the voltage drop is large enough to make a path delay larger than the applied clock period. This method has low spatial precision as the supply voltage is distributed all over the chip through a power network. Similarly, the temporal precision of the fault injection is low because all of the clock cycles are exposed to the lower supply voltage. The adversary controls the fault intensity through the value of the external supply voltage.

The adversary can also use *voltage glitching* [21], which injects temporary voltage drops and provides the capability to control the temporal location of the fault injection. In this case, the adversary controls the intensity with the glitch offset from the sampling edge of the clock signal, glitch voltage, and glitch width similar to the clock glitching.

These methods require physical access to the supply voltage pin. Removing the external coupling capacitance on the supply voltage line improves the efficiency [21]. The drawback of tampering with external voltage pin is that the adversary does not have precise control on the timing and location of the fault injection.

3.1.3 Tampering with Operating Temperature

An adversary may also use *overheating* to trigger setup-time violations [16, 22] for fault injection. In this method, the adversary does not have precise control on the spatial and temporal location of the fault injection. The intensity of fault injection is controlled via operating temperature of the target device.

In addition to the setup time violation on the datapath, overheating also causes modification in memory cells in EEPROM [23], Flash [23], and DRAM [24] memories. While Govindavajhala et al. [24] use a low-spatial-precision light bulb as the heating source, Skorobogatov [23] employs a 650-nm wavelength laser to increase the spatial precision of heating.

3.1.4 Combination of Voltage, Frequency, and Temperature Fault Injection

Zussa et al. [16] demonstrated that overclocking, clock glitching, voltage glitching, underfeeding, and overheating exploit the same fault injection mechanism, which is the violation of a device's setup time constraints. In addition, Korak et al. [17, 25] showed overheating and voltage glitching improve the efficiency of clock glitching.

3.1.5 Optical Fault Injection

In optical fault injection, the adversary decapsulates the target integrated circuit (IC) and exposes the silicon die to a light pulse. The applied light pulse induces a photo-electric current in the exposed area of the IC, which then cause faulty computations [26]. The spatial location is controlled by the position and the size of the light source, and the temporal location is controlled by the offset of the pulse from a trigger signal. The intensity of the fault injection is determined by the energy and duration of the light pulse. It has been demonstrated that optical fault injection can be achieved with a low-cost camera flash light [26, 27]. The state-of-the-art optical fault injection setups [28] use laser beams for fault injection to achieve micrometer-level spatial and nanosecond-level temporal precision. They also provide precise control on the fault intensity. This enables an adversary to target a single transistor. Laser fault injection can be done from front side and back side of an IC. Front side attacks typically use light with shorter wavelengths. These beams have more energy and can easily penetrate between metal layers. Back side attacks use infrared light that penetrates the silicon substrate without being blocked by metal layers.

A disadvantage of the optical fault injection is that it requires decapsulation of the target IC. In addition, it can permanently damage the target IC. Despite these disadvantages, laser fault injection is popular because it provides the most precise and effective fault injection means.

3.1.6 Electromagnetic Fault Injection

In electromagnetic fault injection (EMFI), the adversary applies transient or harmonic EM pulses on the target integrated circuit (IC) through a fault injection probe, which is designed as an electromagnetic coil. The adversary places the probe above the target IC and applies a voltage pulse to the coil, which induces eddy currents inside the target IC. Then, the effects of the induced eddy currents are captured as faults. The adversary controls the temporal location of fault injection through offset of the EM pulse from a trigger signal. The spatial location of the fault injection is controlled via position and size of the injection probe. The fault intensity is determined by the voltage and duration of the applied EM pulse. The feasibility of EMFI on off-the-shelf microprocessor ICs has been demonstrated using both low-cost and high-cost injection setups. For instance, Schmidt et al. [27] use a simple gas lighter to induce EM pulses onto an 8-bit microcontroller with low spatial and temporal precision. The state-of-the-art EMFI setups [29–31] provide millimeter-level precision in spatial location and nanosecond-level precision in the temporal location of the EM pulse. Furthermore, these setups also provide precise control on the voltage and duration of the applied EM pulse. The advantages of EMFI are that it does not require decapsulation of the target IC and it can inject local faults. However, its spatial precision is lower than the spatial precision of the laser fault injection.

3.2 Software-Controlled Fault Injection Techniques

Software-controlled fault injection is a recently discovered research area. The following two sections briefly explain the existing two software-controlled fault injection techniques.

3.2.1 Tampering with DVFS Interface

In the modern embedded systems, Dynamic Voltage Frequency Scaling is a commonly used energy management technique, which regulates the operating voltage and frequency of a microprocessor based on its dynamic workload. In a typical DVFS scheme, kernel-level drivers control the frequency and voltage of a processor through on-chip regulators.

Tang et al. [32] demonstrated that an adversary can exploit the interface between the software drivers and hardware regulators to induce faults in a multi-core processor. In this technique, the adversary uses a malicious kernel-level driver running on a processor core to set the operating voltage and frequency of another core that executes the victim software. This method allows an adversary to violate setup time constraints of the victim core via overclocking and underfeeding it for a specific period of time. The adversary controls the temporal location with the endpoints of the overclocking or underfeeding period. As both the clock and voltage signals are chip-level global signals, the adversary does not have a direct control on the spatial location. The intensity of fault is determined by the overclocking frequency and the underfeeding voltage value. This method requires neither additional fault injection hardware nor physical access to the target device.

3.2.2 Triggering Memory Disturbance Errors

In this fault injection method, the adversary injects faults into memory cells by exploiting the reliability issues of modern memory hardware such as DRAM and Flash memory chips. The continuous scaling down in the process technology has enabled memory manufacturers to significantly reduce cost-per-bit by placing smaller memory cells closer to each other. However, this also increases electrical interference between memory cells: Accessing a memory cell electrically disturbs nearby memory cells. A disturbed memory cell loses its value and experiences a memory disturbance error when the amount of electrical disturbance is beyond noise margins of that disturbed cell [33, 34].

An adversary may trigger memory disturbance errors through a non-privileged fault injection program. This program repeatedly accesses a set of memory cells (i.e., aggressor memory cells) to induce disturbance errors in a set of victim memory cells storing security-sensitive data. This method allows an adversary to corrupt memory space of a security-sensitive program from memory space of the adversary-controlled fault injector program. Memory disturbance errors have been demonstrated on commodity DRAM and NAND Flash memory chips [33].

In DRAM memories, the memory disturbance errors are induced through Rowhammer mechanism [34]. Thus, it is called Rowhammering. A DRAM memory is internally organized as a two-dimensional array of DRAM cells, where each cell consists of an access transistor and a capacitor storing charge to represent a binary value. As capacitors lose their charges because of the leakage current, the DRAM cells are periodically refreshed to restore their charges. Each row of the array has a separate wordline, which is a wire connecting all memory cells on the corresponding row. To access a DRAM cell within the two-dimensional array, the corresponding row of the array is activated by raising the voltage of its wordline. Persistent access to the same row causes repeated voltage fluctuations on its wordline, which electrically disturbs nearby rows. This disturbance increases the charge leakage rate in the nearby DRAM rows [34]. As a result, a memory cell within a nearby row experiences a memory disturbance error (a bit-flip error) if it loses a significant amount of charge before it is refreshed. An adversary may take advantage of that physical phenomenon to inject faults. For this purpose, the adversary runs a malicious fault injection program on the target processor, which aims at altering a security-sensitive state of a victim program running on the same processor. The fault injection program continuously accesses an aggressor DRAM row in its own memory space and induces faults into a victim DRAM row within the victim program's memory space [35–37].

Similar disturbance mechanisms have been also demonstrated on multi-level cell (MLC) NAND Flash memories. Similar to DRAM memory, a Flash memory is also internally organized as an array of Flash memory cells, each of which is a floating-gate transistor. The amount of charge stored in the floating gate determines the threshold voltage of the transistor, which is used to represent the stored data, In MLC Flash memories, each cell stores two bits of data. Unlike DRAM memories, Flash memories do not require periodic refreshing. Cai et al. [33] demonstrated that the capacitive coupling between neighboring Flash cells enables two memory disturbance error mechanisms. The first mechanism, Cell-to-Cell Program Interference (CCI), introduces faults into a Flash cell when a nearby cell is programmed (i.e., written). The amount of interference is high when a specific data pattern for programming is used. Cai et al. [33] and Kurmus et al. [38] showed how a malicious fault injection program may trigger CCI mechanism to cause a security breach. The second mechanism is *Read-Disturb*, in which the content of a Flash cell is disturbed when a nearby cell is read. Cai et al. [33] demonstrated the use of read-disturb to cause security problems.

The advantage of fault injection by triggering memory disturbance errors is that it can induce single-bit to multi-bit faults into a certain memory location [37]. This enables an adversary to break several security mechanisms.

Table 2 summarizes the previously described fault injection techniques and the main characteristics of the physical stress applied to the target device. For each fault injection technique, the table provides spatial precision, temporal precision, and hardware cost of the applied physical stress. The table also provides a list of fault injection parameters to control the intensity of the applied physical stress.

4 Fault Manifestation in Processor Micro-Architecture

This section explains the effects of physical fault injection on the micro-architecture of the target processor. First, we will distinguish micro-architecture (i.e., internal architecture) of a processor from its architecture (i.e., external architecture). Then, we will briefly explain main characteristics of the induced faults into micro-architecture.

Any processor can be described from two distinct architectural perspectives. The architecture of the processor describes it as seen by programmers in terms of its instruction set and facilities. The architecture defines semantics and syntax of available instructions, program-visible processor registers, memory model, and how interrupts are handled. It is the boundary between hardware and software as well as a contract between programmers and hardware designers. The micro-architecture describes the physical organization and implementation of the architecture. This includes the memory hierarchy, pipeline structure, available functional units, employed mechanisms (e.g., out-of-order execution) for instruction-level parallelism, and so forth. The micro-architecture is optimized to satisfy cost and performance requirements.

Faults manifest as incorrect bits in the flip-flops or memory cells employed in the micro-architecture if the applied physical stress is beyond noise margins of target processor hardware. The parameters and type of physical fault injection technique determine characteristics of the manifested faults. In this work, we use four parameters to describe any manifested fault in micro-architecture level:

– **Location of the Manifested Fault:** This parameter specifies the micro-architectural blocks that contain faulty bits because of physical fault injection.

  Faults may manifest in any micro-architectural block in the control or datapath part of the processor such as instruction memory, instruction fetch block, instruction decode block, operand fetch block, execution block, data memory, register file, processor status register, and conditional flags. An adversary's control on the location of the manifested faults depends on the spatial precision of the used fault injection method, which is characterized as precise control, loose control, and no control [39].

  All of the previously described fault injection techniques, except software-based memory disturbance, can target the datapath, control, or memory of a processor.

Table 2 Fault injection techniques and the characteristics of the corresponding physical stress applied to a target device

Fault injection technique	Characteristics of the applied physical stress			
	Spatial precision	Temporal precision	Cost	Controlling the intensity
Overclocking	Low (global)	Low (global)	Low	Clock frequency
Clock glitching	Low (global)	High (local)	Low	Glitch width
Underfeeding	Low (global)	Low (global)	Low	Voltage level
Voltage glitching	Low (global)	High (local)	Low	Glitch voltage
				Glitch width
Overheating	Low (global)	Low (global)	Low	Ambient temperature
Light pulse	Medium (local)	Medium (local)	Low	Pulse width
				Pulse energy
				Pulse offset
Laser pulse	High (local)	High (local)	High	Pulse width
EM pulse	Medium (local)			Pulse energy
				Pulse offset
				Probe size
DVFS interface	Low (global)	Medium (local)	Zero	Supply voltage
				Clock frequency
Memory disturbance	High (local)	Medium (local)	Zero	Disturbance frequency

Memory disturbance can only inject faults into the memory.

- **Size of the Manifested Fault:** This parameter specifies the number of faulty micro-architecture bits induced by physical fault injection. An adversary can control the size of the manifested faults by adjusting the fault intensity. In the literature, manifested faults are commonly classified as single-bit faults, byte-size faults, word-size faults, and arbitrary-size faults [40]. The adversary influences the size of the manifested faults by tuning the fault injection intensity.
- **Effect of the Manifested Fault:** This parameter specifies the logical effect of the manifested fault on the fault location. Common fault effects are stuck-at fault, bit-flip fault, bit-set fault, bit-reset fault, and random fault [40].

  All of the aforementioned fault injection techniques are able to induce bit-flip effects. In addition, laser and EM pulses are also able to induce bit-reset, bit-set, and stuck-at faults.
- **Duration of the Manifested Fault:** Fault attacks typically exploit transient faults, which last as long as the physical stress is applied. Such faults are recovered when a new value is written into the faulty flip-flop or the memory cell. However, if a register is refreshed only infrequently, these faults can still last a long time. Some fault attack injection techniques are able to create long-lasting faults, such as recently demonstrated using focused X-ray injection [41]. Some fault injection techniques can even inject permanent faults, such as when a laser pulse causes permanent damage to a memory or register cell (stuck-at).

The next section explains how the manifested faults propagate to the software layer.

5 Fault Propagation to the Software Layer

The manifested faults propagate to the software layer as faulty instructions when the micro-architectural blocks containing the faulty bits are used by the instructions of the target program. Propagated fault effects are determined by the type of affected instruction, type of faulty micro-architectural block, and the characteristics (size, effect) of the manifested faults. As each processor implementation has its own micro-architecture, it is not possible to list all of the potential fault effects propagated to software. Instead, we provide an example to demonstrate a list of potential fault effects for a subset of SPARC instructions running on a hypothetical generic micro-architecture. Using the same approach, similar lists can be built for specific instruction sets and processor implementations.

As an example, we chose four SPARCv8 instructions: a memory-load (`ld`), a logic (`xor`), a comparison (`cmp`), and a conditional branch (`be`) instruction. Table 3 lists the instructions and their definitions.

The assumed generic micro-architecture contains the following blocks to carry out instruction-execution cycle for each instruction:

- **I-Mem Block** is the instruction memory that stores the instructions.
- **I-Fetch Block** prepares the address for the instruction memory, program counter (PC). Then, using the prepared PC, it fetches an instruction into the instruction register (IR).
- **I-Decode Block** takes the fetched instruction from IR and decodes it to determine the location of the source operands, the location of destination operands, and the operation to be applied. The source operands are fetched from the register file. The destination may be the register file, data memory (D-Mem), or conditional flags.
- **O-Fetch Block** uses the decoded information to fetch the input operands from the register file and to feed them to the execution block. The `be` instruction does not use this block because it does not fetch any operand from the register file.
- **Execute Block** applies the required operation on the fetched source operands and generates a result. For `ld`, it calculates the D-Mem address from `r1` and `r2`. For `xor`, it applies bitwise XOR operation on `r1` and `r2`. For `cmp`, it subtracts `r2` from `r1`. For `be`, it calculates the destination address from the current PC and `offset`. It also checks the conditional flags to determine if the branch will be taken.
- **Store Block** updates the destination location (D-Mem, register file, or flags) with the result computed by the execution block. For the `ld` and `xor`, it is the register `r3`. For the `cmp` instruction, the destination is conditional flags. For the `be` instruction, the destination

Table 3 An example set of SPARCv8 instructions

Instruction	Definition
`ld [r1+r2], r3`	Loads a 32-bit word into register `r3` from data memory (D-Mem) address `r1+r2`.
`xor r1, r2, r3`	Bit-wise XOR operation on `r1` and `r2` Result is written to register `r3`.
`cmp r1, r2`	Compares registers `r1` and `r2` and updates conditional flags accordingly.
`be offset`	PC-relative conditional jump: If zero-flag is set, PC will be `PC + offset`. Otherwise, PC will be `PC + 4`.

is the PC value if the branch is taken. Otherwise, it will not affect any destination.

Table 4 provides an example list of propagated fault effects for each (*instruction, micro-architecture block*) pair. In this example, we assume a *single bit-flip fault* in any micro-architectural block. A fault induced in I-Memory, I-Fetch, or I-Decode block would affect syntax (i.e., opcode and operands) and/or semantics (i.e., the operation to be applied) of an instruction independent from the type of the instruction. Thus, Table 4 shows the propagated fault effects for these blocks in a single cell. Faults induced in the other blocks would cause errors in the instruction-specific computation of a correctly fetched and decoded instruction:

– **I-Mem, I-Fetch, I-Decode:** If the fault manifests in the opcode part of the faulty instruction, another instruction will be executed. If the fault affects the addresses of source operands, they will be fetched from an incorrect location. Similarly, the result of an instruction will be written into a wrong location if the fault hits the destination address. Finally, the next instruction will be fetched from an incorrect location if the PC calculation gets faulty.
– **O-Fetch:** For the `ld` instruction, a single-bit fault in this block affects the value of register `r1` or `r2` fetched from register file. The fault then causes D-Mem address to be faulty. As a result, a single-bit fault in either `r1` or `r2` may induce an arbitrary number of faults in the destination register `r3` because the result will be fetched from an incorrect D-Mem location.

  For the `xor` instruction, the fault will affect a single bit of `r1` or `r2`, which will be propagated to `r3` as a single-bit fault.

  For the `cmp` instruction, the single-bit fault may affect the result of the comparison, which will alter the conditional flags based on the modified comparison result.

  For the `be` instruction, the fault will not have any effect because this instruction does not fetch anything from the register file.
– **Execute:** For the `ld`, `xor`, and `cmp` instructions, the effects of the fault will be same as the effects described in the O-Fetch case.

  For the `be` instruction, the fault will change the single bit of the computed branch address. If the branch is taken, the destination address will be wrong. Otherwise, the faulty branch address will not affect the program. The fault may also change the direction of the branch instruction from taken branch to non-taken branch, or vice versa.
– **Store:** For the `ld` instruction, the fault will cause a single-bit error in the correctly computed D-Mem address `[r1+r2]`. For the `xor` and `cmp` instructions, the effects of the fault will be same as the effects described in the O-Fetch case. For the `be` instruction, the fault will change the value of the PC if it is a taken branch.
– **D-Mem:** As none of the instructions use a value from the data memory, the fault in D-Mem will not affect any of the considered instructions.
– **Register File:** For the `ld`, `xor`, and `cmp` instructions, the effects of the fault will be same as the effects described in the O-Fetch case. As the `be` instruction does not use this block, the fault will not have any effect on this instruction.
– **Conditional Flags:** The fault in conditional flags will affect only the `be` instruction as the other instructions do not use the conditional flags.

Table 4 Propagated effects to software layer for each faulty micro-architectural block (with 1-bit fault) and instruction

Faulty block	Propagated fault effects			
(1-bit fault)	`ld [r1+r2], r3`	`xor r1, r2, r3`	`cmp r1, r2`	`be dest`
I-Mem I-Fetch (PC, IR) I-Decode	Execution of a wrong instruction due to opcode-field corruption Fetching operands from wrong location due to source-operand-location corruption Updating a wrong destination due to destination-operand-location corruption Fetching next instruction from a wrong address due to PC corruption			
O-Fetch (1-bit fault in `r1` or `r2`)	Arbitrary # of faults in `r3` (Faulty D-Memory address)	1-bit fault in `r3` (Faulty XOR input(s))	Faulty update of conditional flags	No effect
Execute	Arbitrary # of faults in `r3` (Faulty D-Memory address)	1-bit fault in `r3` (Faulty XOR operation)	Faulty update of conditional flags	1-bit fault in jump address or Inversion of branch
Store	1-bit fault in `r3` (Faulty update of `[r1+r2]`)	1-bit fault in `r3` (Faulty update of `r3`)	Faulty update of conditional flags	1-bit fault in jump address
D-Mem	No effect			
Register File	Fetching wrong source operands from register file			No effect
Conditional Flags	No effect			No jump to `dest`

6 Fault Exploitation Techniques

This section presents main fault exploitation techniques, which have been proposed to break the security of both cryptographic and non-cryptographic security mechanisms protecting embedded software. Each exploitation technique relies on a fault model, which is a high-level assumption for the effects of physical fault injection on the execution of the target software. Thus, we start with commonly used fault models in practice. Then, we will briefly explain fault exploitation techniques.

6.1 Fault Models

In the design phase of a fault attack, an adversary makes a fault model assumption and develops an exploitation strategy based on the fault model. This assumption generally includes the location of the fault in the data or control flow of the target program, the timing of the fault with respect to the duration of the target program, size of the fault, and effect of the fault. The fault models can be described in algorithm level, source code level, or instruction level. The following paragraphs provide an example list of commonly used fault models.

The most of fault-based cryptanalysis techniques on symmetric and asymmetric cryptography assume faults on data flow of a target program that corrupt a single bit, single byte, multiple bytes, or a single word of a security-critical variable in various ways (e.g., flip, set, reset, random) [6, 7, 40].

On the control flow, the most popular fault models are to skip the execution of a specific instruction (i.e., instruction skip) [42, 43], multiple instruction skips [44, 45], replacing an instruction with another one (i.e., instruction modification) [21, 46], changing the result of a conditional branch [47, 48], and tampering with loop counters [49, 50].

In the implementation of a fault attack, the adversary aims at inducing the fault effects assumed in the fault model via fault injection, fault manifestation, and fault propagation processes. Therefore, a fault model can be realized through different combinations of fault injection, fault manifestation, and fault propagation. The following sections provide a list of commonly used fault exploitation techniques to breach the security of embedded software.

6.2 Cryptanalysis using Fault Injection

Using fault injection for cryptanalysis has been extensively studied on the implementations of symmetric-key, public-key, and post-quantum cryptography algorithms [6, 7, 14, 39, 40].

Differential Fault Analysis (DFA) is the most widely used fault-based cryptanalysis technique. The main principle of DFA is to exploit the difference between the faulty and fault-free outputs of a cryptosystem. In a typical DFA attack, an adversary collects two outputs (e.g., ciphertexts) from a cryptosystem (e.g., encryption) that are generated for the same input (e.g., plaintext) and secret variable (e.g., encryption key). One of the outputs is collected without fault injection. During the generation of the second output, the adversary injects a certain fault into the execution of the cryptosystem. Then, the adversary analyzes the propagation of this fault differential to the output and reveals the secret variable. DFA attacks assume specific fault during differential analysis of the faulty and fault-free outputs. Various DFA techniques have been successfully demonstrated on block ciphers [51], stream ciphers [52], public-key algorithms [7, 53], and post-quantum cryptography [54]. To illustrate how the DFA works, the following two paragraphs briefly explain a previously proposed DFA attack on Advanced Standard Encryption (AES) algorithm.

In this example attack [55], the assumed fault model is a single-bit flip in the input of the last AES round. Figure 3 shows the propagation of a single bit fault through the last round of AES. Due to the structure of AES, the single bit flip ($v*$) is propagated to the ciphertext as a single fault byte

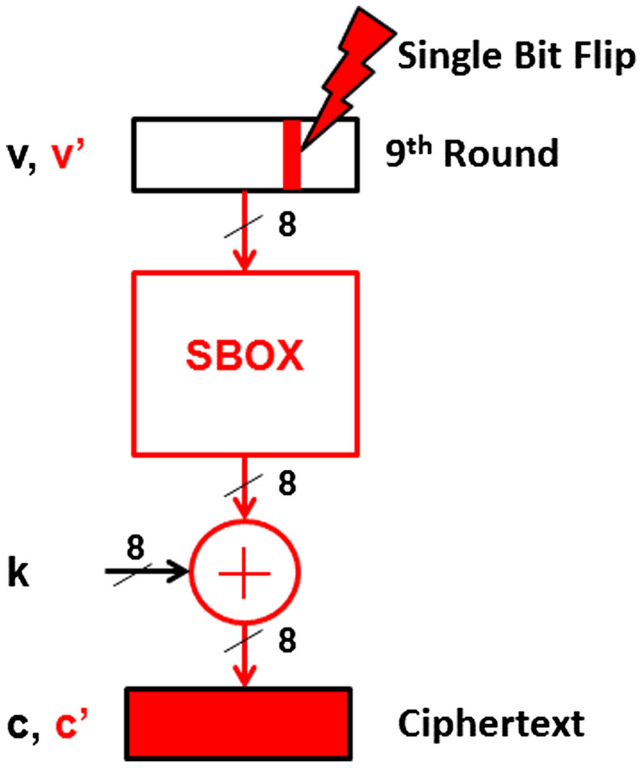

Fig. 3 Propagation of a single-bit flip fault through the last round of AES. The fault causes a single faulty byte in the ciphertext

($c*$). For the fault-free (c) and faulty ($c*$) ciphertexts, we can write

$$c = \mathtt{SBOX}(v) \oplus k$$
$$c^* = \mathtt{SBOX}(v^*) \oplus k$$
$$\Delta = v \oplus v^*$$

where Δ denotes the injected fault differential. The adversary is able to observe the values of $c*$ and c. As the assumed fault model is to flip a single bit of v, the Hamming weight (HW) of the injected fault differential Δ is assumed to be 1. The purpose of the adversary is to reveal the value of the corresponding byte (k) of the last round key. The adversary achieves this through an exhaustive search on the possible key values. For each possible key hypothesis $\tilde{k}$ for actual key byte k, the adversary first computes the corresponding fault differential $\tilde{\Delta}$ as follows:

$$\tilde{v} = \mathtt{SBOX}^{-1}(c \oplus \tilde{k})$$
$$\tilde{v}^* = \mathtt{SBOX}^{-1}(c^* \oplus \tilde{k})$$
$$\tilde{\Delta} = \tilde{v} \oplus \tilde{v}^*$$

The adversary then checks whether the computed (i.e., hypothesized) differential $\tilde{\Delta}$ is equal to the injected (i.e., assumed) differential Δ. For our example, the hypothesized key $\tilde{k}$ is a possible candidate for the actual key k if the Hamming weight of $\tilde{\Delta}$ is 1. Otherwise, the hypothesized key $\tilde{k}$ is discarded.

After testing all of the possible key hypotheses, the set of possible key candidates contains eight elements on the average [56]. Therefore, two fault injection experiments on a given input byte of the AES key. The remaining bytes of the last round key can be revealed by repeating the same steps for the remaining input bytes of the last round. As a result, the explained attack requires 32 fault injection experiments to retrieve the whole 16-byte last-round key of AES-128. The adversary can then calculate other round keys by applying AES key scheduling algorithm on the retrieved last round key. For further information on the DFA techniques and their comparison, the reader may refer to existing works in the literature [7, 57, 58].

Biased Fault Analysis attacks [59–63] exploit biased fault behavior: Because of the correlation between the fault behavior of a target program and the applied physical fault intensity, the distribution of fault models is non-uniform. They allow an adversary to treat fault behavior as a side-channel signal, which relaxes the strict fault model requirements of the previous attacks [12]. As an example of biased fault analysis, we will demonstrate Differential Fault Intensity Analysis (DFIA) on AES, which was proposed by Ghalaty et al. [59].

Similar to the previous DFA example, the DFIA assumes faults in the input of the AES last round. Unlike DFA attacks, DFIA does not make a precise assumption on the injected faults. Instead, DFIA assumes that the injected faults are biased: The adversary adjusts the fault intensity during the fault injection step such that the number of faulty bits in the input of the last AES round (v) is minimal. For this specific DFIA attack, it is assumed that the number of faulty bits in the input byte v is less than 4 [59]. DFIA retrieves the value of the corresponding key byte as follows. The adversary first computes the fault differential $\tilde{\Delta}$ for each key hypothesis $\tilde{k}$. If the correct key hypothesis is made, the Hamming weight of the hypothesized fault differential $\tilde{\delta}$ is small (i.e., $HW(\tilde{\delta}) < 4$). Under a wrong key hypothesis, the expected Hamming weight of the fault differential is large because of highly non-linear design of SBOX of AES.

It is possible that a single fault is insufficient to uniquely determine the correct key. In that case, the adversary can inject multiple biased faults, under a gradually increasing fault intensity, each time recording the faulty ciphertext $c*$. For each key hypothesis $\tilde{k}$ and injected fault, the adversary computes the Hamming weight of the corresponding fault differential. For the correct key hypothesis, the sum of all Hamming weights is still minimal [59]. Ghalaty et al. demonstrated that, on the average, DFIA requires 4.6 faults to retrieve one byte of the last round key and 68 faults to retrieve all bytes of it. In conclusion, DFIA relaxes fault model requirements and more suitable than DFA when fault injection is hard to control.

Safe Error Analysis (SEA) attacks exploit the dependence between the use of a faulty data and the value of a secret variable [64]. An adversary first identifies a target intermediate variable, of which use depends on the value of a secret variable. Then, the adversary injects a specific fault into the target variable and observes whether the output is faulty or not. If the output is faulty, it means that the faulty target variable is used and the secret variable has a specific value. The advantage of the SEA is that it requires only a single-bit information from fault observation: If the faulty value has been used or not. Fault injection may be used to check if a specific computation is executed (C-safe errors [65]) or if a specific memory location is accessed (M-safe errors [66]). SEA attacks have been successfully demonstrated on symmetric-key [7, 67] and public-key [7, 65] algorithms. Yen and Joye describe a form of safe-error analysis that is based on collisions [65]. By forcing a value on an intermediate value with data dependency on the output, and by checking if the output is affected or not, a *collision* between the forced value and the original secret value can be detected. For this reason, for example, write-only cryptographic key registers should never allow partial update; otherwise, the attacker can test a partial key guess by detecting these collisions.

Algorithm-Specific Fault Analysis uses fault injection to exploit algorithm-specific properties. For instance, the public-key cryptography algorithms such as RSA and ECC rely on a hard-to-solve mathematical problem. An adversary may use fault injection to alter the mathematical foundations of the problem and convert the problem into an easy-to-solve one.

In the infamous Bellcore Attack, Boneh et al. demonstrate that the security of the RSA cryptosystem can be broken with a single faulty computation [68]. In the RSA, a message M is signed by computing $S = M^d \, mod N$, where d is the secret exponent and $N = pq$ is a product of two large prime integers. The security of the system relies on the difficulty of factoring the modulus N. An efficient implementation of RSA is RSA-CRT, in which $S_1 = x^d \, mod p$ and $S_2 = x^d \, mod q$ are computed first and then the Chinese remainder theorem (CRT) is used to combine S_1 and S_2 to obtain $S = M^d \, mod N$. In the Bellcore Attack, a single random fault is assumed in the computation of either S_1 or S_2. If a fault occurs during the computation of S_1, the modulus N can be easily factored using the equation $gcd(S - \hat{S}) = q$. In this equation, S and $\hat{S}$ denote the fault-free and faulty signatures, respectively. Similar algorithm-specific analysis attacks have been mounted on several public-key systems including RSA and ECC [7, 69, 70].

6.3 Using Fault Injection to Assist Side-Channel Analysis

Another use of fault injection is to assist side-channel analysis for reducing the complexity of side-channel attacks or thwarting the countermeasures. Side-Channel Analysis, introduced by Kocher et al. [71], is a major category of the implementation attacks used for cryptanalysis of secure embedded software. While fault attacks actively manipulate the physical operating conditions of a target device, side-channel attacks exploit physical leakage (e.g., power consumption, and electro-magnetic radiation) emanating from the device during a security-critical operation. Side-channel attacks are usually partitioned into two categories. *Simple Side-Channel Analysis (SSCA)* exploits a single observation of the physical leakage of the device during a cryptographic operation. *Differential side-channel analysis (DSCA)* collects multiple observations of the physical leakage and retrieves the secret information by applying statistical tests on these observations. In the last 20 years, various SSCA and DSCA methods have been demonstrated on all forms of cryptography [72–74]. Similarly, developing countermeasures against SSCA and DSCA attacks have been extensively investigated [4, 75–78]. The advancements in the countermeasure design motivated *fault-assisted side-channel attacks*, which utilize fault injection to break the security of systems protected against SCAs.

In 2006, Skorobogatov [79] used a laser source to illuminate a specific area of an SRAM memory to increase the side-channel leakage of the illuminated area. In 2007, Amiel et al. [80] proposed a fault-assisted side-channel attack on an RSA implementation resistant to both side-channel and fault attacks. In this attack, the injected fault modifies a secret variable such that the modified variable leaks information via SSCA. A similar attack was also developed by Clavier et al. [81] on an AES implementation protected with first-order masking, a DSCA countermeasure. Roche et al. [82] also demonstrated a combined attack on an high-order masked and DFA-resistant AES implementation. Based on the work of Roche et al. [82], Dassance et al. [83] developed combined attacks on the key schedule of a protected AES implementation. In 2010, Schmidt et al. [84] both demonstrated novel fault-assisted side-channel attacks and countermeasures on them. Later, Feix et al. proposed novel attacks that are capable of breaking the countermeasures proposed by Schmidt et al. [84]. In 2018, Yao et al. [85] proposed a fault-assisted side-channel attack that utilize fault injection to weaken a DSCA-resistant masking scheme and breaking its security with a first-order DSCA.

6.4 Fault-Enabled Logical Attacks

In addition to their use in cryptanalysis, fault attacks can also be used to trigger logical attacks (e.g., control flow hijacking, privilege escalation, subverting memory isolation) on smartcards and general-purpose processors. Classic logical attacks such as buffer overflow tamper with the inputs of a program to exploit a security bug in the implementation of the program. A well-known example is the HeartBleed bug [86]. In the absence of such an exploitable software bug, it is not possible for an adversary to mount a logical attack by just modifying inputs. In such a case, an adversary can inject faults to dynamically create required conditions to mount a logical attack. A straightforward application of this idea are attacks on input/output routines, which copy a portion of an internal memory region to the outputs of the chip. By glitching the end condition of the input/output routine, an adversary can force dumping of the entire internal data memory region, rather than just the portion allocated to the input/output buffer. Similarly, an attacker can also utilize fault injection to dump the source/binary code of the target program in case the code is normally not available to the attacker [87, 88]. This enables the attacker to analyze the source/binary code for identifying and subsequently exploiting the software vulnerabilities. The following paragraphs briefly explain fault-enabled logical attack examples from the literature.

Barbu et al. [42] demonstrated two fault-enabled logical attacks on a Java Card. In the demonstrated attacks, the adversary uses a laser-induced instruction-skip model to create type confusion. Then, the adversary exploits the induced type confusion to load an unverified adversary-controlled code on the Java Card. Type confusion also enables an adversary to access other applications' memory space. Vetillard et al. [47] and Bouffard et al. [89] also demonstrate similar attacks on Java Card, in which they employed fault injection to bypass run-time security checks and execute malicious code on the platform.

The first fault-enabled fault attack on a general-purpose processor has been demonstrated by Govindavajhala et al. [24]. In the demonstrated attack, the adversary designs and runs a software program on a Java Virtual Machine (JVM) on a desktop computer. The malicious program is designed such that a bit error in the data space of the program allows the adversary take full control over JVM. To induce those exploitable faults, the authors overheat the memory chips.

Nashimoto et al. [45] proposed a fault-enabled buffer overflow (BOF) attack on a buffer overflow countermeasure, which limits input size. The authors demonstrated the proposed attack on an 8-bit AVR ATmega163 and a 32-bit ARM Cortex-M0+ microcontroller. Their fault models were single and multiple instruction-skip, which are induced by clock glitching.

Timmers et al. [21] demonstrated two ARM-specific, fault-enabled logical attacks which are based on setting the program counter (PC) of a microprocessor to an adversary-controlled value. The authors alter the execution of a memory-load instruction (i.e., instruction replacement) via voltage glitching to set PC to an adversary-controlled value. The authors provide two case studies to demonstrate the use of such an attack. In the first case, the authors bypass a secure-boot mechanism and run their own unverified program on the processor. In the second case, the authors subvert the hardware-enforced isolation mechanism of a Trusted Execution Environment (TEE) and run their code program with the highest privileges on the processor.

Vasselle et al. [90] demonstrated a fault-enabled logical attack on a Quad-core ARM Cortex-A9 processor, which bypasses secure boot mechanism and allows an adversary to get highest privileges on the processor. The authors achieved privilege escalation by resetting the privilege-level-specifying bit of the Secure Configuration Register of the processor via laser fault injection.

Timmers et al. [91] proposed three fault-attack enabled logical attacks on a Linux Kernel to gain kernel-level execution privileges. The authors demonstrated their attacks on an ARM Cortex-A9 processor through voltage glitching. In the demonstrated attacks, the authors request system calls from the user space and, then, inject faults during the execution of system calls for privilege escalation. The gained privileges may allow an adversary to run an arbitrary code on the device and access the memory space of other applications.

The software-controlled fault injection methods such as triggering memory disturbance errors broaden the scope of fault attacks as they allow remote fault attacks. For example, in Rowhammer attacks [34], an adversary-controlled program (running in a user space) injects bit-flip faults into security-sensitive DRAM memory cells by repeatedly accessing nearby cells. In 2015, Seaborn [92] demonstrated two practical Rowhammer attacks. The first attack induces bit-flips to escape from Google Native Client (NaCl) sandbox. The second attack use bit-flips in DRAM for privilege escalation. Gruss et al. [35] successfully mounted a Rowhammer attack from web browsers on four off-the-shelf laptops. Similarly, van der Veen et al. [36] achieved privilege escalation on Android-running mobile platforms. Razavi [37] demonstrated a Rowhammer attack in a cloud setting, in which a malicious virtual machine induces memory disturbance errors to gain unauthorized access to memory space of a co-hosted virtual machine. Kurmus et al. [38] and Cai et al. [33] demonstrated that software-controlled memory disturbance errors can be triggered on Multi-cell (MLC) NAND Flash memories to mount fault-enabled logical attacks.

Finally, Tang et al. [32] exploited security-oblivious dynamic voltage and frequency scaling (DVFS) interface to induce faults in a smartphone. They demonstrated two software-controlled fault attacks. The first attack allows a malicious user-space program to inject faults into the operation of an encryption program running in Trustzone environment and to reveal the value of secret key stored in Trustzone environment. In the second attack, an adversary bypasses an authentication mechanism running in Trustzone to load an unauthorized program into Trustzone environment. These two attacks show that fault injection may enable an adversary to subvert hardware-enforced isolation mechanisms such as ARM Trustzone.

6.5 Using Fault Injection to Assist Reverse Engineering

Another potential use of fault injection is to assist reverse engineering. San Pedro et al. [93], Le Bouder et al. [94], and Clavier et al. [95] employed fault injection to reverse engineer specifications of block ciphers similar to Data Encryption Standard (DES) and Adavanced Encryption Standard algorithms. For instance, San Pedro et al. [93] propose the FIRE attack that employs fault injection to reverse engineer SBox specification of DES-like and AES-like block ciphers. In the AES version of the FIRE, single-byte faults are injected into the penultimate round of AES and faulty output data are collected. Then, the faulty data are converted into a set of linear Boolean equations. Finally, the equation system is solved using the Gaussian elimination and the SBox is reverse-engineered. Similarly, Jacob et al. [96] induce faults into the execution of an obfuscated cipher and retrieve the secret key. Courbon et al. [97] also demonstrated a method to reverse-engineer gate-level structure of a hardware implementation of Advanced Encryption Standard (AES) algorithm, in which laser fault injection and image processing are combined.

7 Fault Attack Evaluation and Certification

Given the abundance and diversity of attacks on hardware products, the question arises for individual products: What is the attack resistance, and how secure does the product need to be? This question has been addressed by the global security certification community, resulting in the Common Criteria (CC) [98]. This standard defines levels of security and a methodology for evaluation. A consortium of vendors, certification bodies, and labs maintain a procedure for attack rating.

Any product with secure hardware, for which a vendor seeks certification, can be evaluated according to this methodology. The aim is to provide sufficient assurance that the product remains secure during several years of operation. In this section, we first briefly explain the steps and actors involved in the certification process. We then demonstrate how the fault attack resistance of a product is evaluated.

7.1 The Common Criteria Certification Process

Three main roles exist in the certification process:

– **Vendor** is the manufacturer that develops the product to be certified.
– **Evaluator** is the security evaluation lab that reviews and tests the product design and implementation.
– **Certification body** is the authority that certifies the product after successful completion of the evaluation.

Additional roles may be involved for separate software/hardware vendors, for product issuers, and evaluation sponsors, but these roles do not significantly change the basic model.

Figure 4 depicts a simplified view of how the three main actors work together. The process takes the following steps:

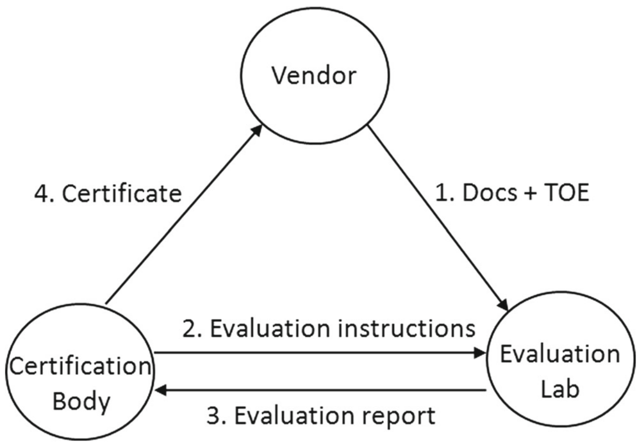

Fig. 4 The Common Criteria (CC) Certification Process

1. A vendor who wants to have a product CC certified provides an evaluator with all relevant documents and the product to be certified (i.e., Target of Evaluation (TOE)). Vendors need to create and maintain a significant set of documents describing the system design and implementation in detail. These documents should also include proof that the implementation meets its specifications. The first task for the evaluator is to review all documents and decide whether they conform to the CC methodology and its requirements.
2. The certification body provides instructions to the evaluator as to how to evaluate the product. The evaluator performs all relevant tests to prove resistance of the product and informs the certification body. Strictly, the evaluator verifies that the product conforms to its security target, but the scope of that must be approved by the certification body. If any blocking issues are found, the product may need revision and re-evaluation.
3. The evaluator sends the Evaluation Technical Report (ETR) to the certification body (and to the vendor) who then reviews the evidence reported by the evaluator. Before issuing a certificate, the certification body may mandate additional testing if new threats surfaced, or when the test results gave reason to doubt.
4. If no objections remain, the certification body issues the certificate.

A product typically consists of multiple layers (hardware, operating system, applications) which can be independently certified. This would start with the hardware certification, after which composite evaluations can be done, where a new layer is certified in conjunction with a certified platform.

The CC certification process is known to be cumbersome; it is lengthy and costly [99]. Although completion of the process may take a couple of months in an ideal situation, it often takes much longer. Apart from cost and time-to-market, this also carries a security penalty: Vendors may be hesitant to make security improvements to a certified system since changes break the certification. In this way, vulnerabilities may remain longer in products than needed.

While Common Criteria certifications enjoy popularity for high assurance products, it is not the only evaluation methodology. Other schemes exists such as EMVCo [100] and FIPS 140-2 [101], many of whom are lighter in execution. However, the processes described here are similar across many schemes and serve well to explain the ecosystem. Next, we will take a closer look into the evaluation process.

7.2 Evaluation of Fault Attack Resistance

Products are always evaluated in "white-box" style. This means an evaluation lab gets access to all product design and implementation information. This will reduce evaluation cost (no need for lengthy reverse engineering) and reduce the risk of missing big security weaknesses. Typically, an evaluation consists of two phases:

1. Vulnerability Analysis (VA)
2. Penetration Testing (PT)

During the VA, the lab reviews the design and implementation code of a product and weighs this against applicable threats. In the PT, a product is tested against a number of attacks to measure its actual resistance. All successful attacks are rated, and when their scores are sufficiently high, the product qualifies for certification.

The evaluation methodology distinguishes between Identification and Exploitation. The former defines the cost of demonstrating that the attack works on the product, while the latter looks at the cost of repeating the attack. Both aspects are important. For instance, an attack with very high initial cost will scare away low-budget attackers, while a high repetitive cost will prohibit attack scaling.

Parameters used in the attack rating are (1) time, (2) expertise, (3) product knowledge, (4) number of the target samples, (5) equipment, and (6) ability to configure target. The parameter time is extremely important during an evaluation as this is most often the limiting aspect during the penetration testing phase. An evaluator cannot afford to lose time if several attacks are to be executed within the typical time frame of a few months.

For efficiency, the PT starts with an investigation of sensitivity to different fault injection methods. While voltage glitching is typically the simplest method, this is often prevented by sensors. Alternative methods like EM and optical glitching are more complex, but also harder to prevent. During this sensitivity analysis, the evaluator uses test software on the target that runs loops and typical instructions that may be affected. The test software accelerates the detection of hardware weaknesses, and supports finding optimal attack parameters, such as glitch intensity and duration. Figure 5 shows how the right combination of glitch voltage and glitch length can be found. The green dots represent experiments that did not affect the chip. Alternatively, the yellow dots represent experiments where the glitch was too strong and resulted in a reset of the chip. Finally, the red dots represent successful glitch parameters that resulted in an observable effect in the test code. Figure 6 shows how effective temporal offsets are found (represented as wait cycles on the x-axis).

Once a weakness is established, a setup is built to demonstrate that the weakness can be exploited on the real product software. This setup includes equipment for generating glitches at the right time, and solutions to minimize the effect of countermeasures. A highly automated setup runs for several days and uses dedicated control procedures to manage smooth repetition and fault logging. Ultimately, an evaluation results in a report, where successful attacks are rated, and a discussion is given on the attack risks and potential mitigation strategies.

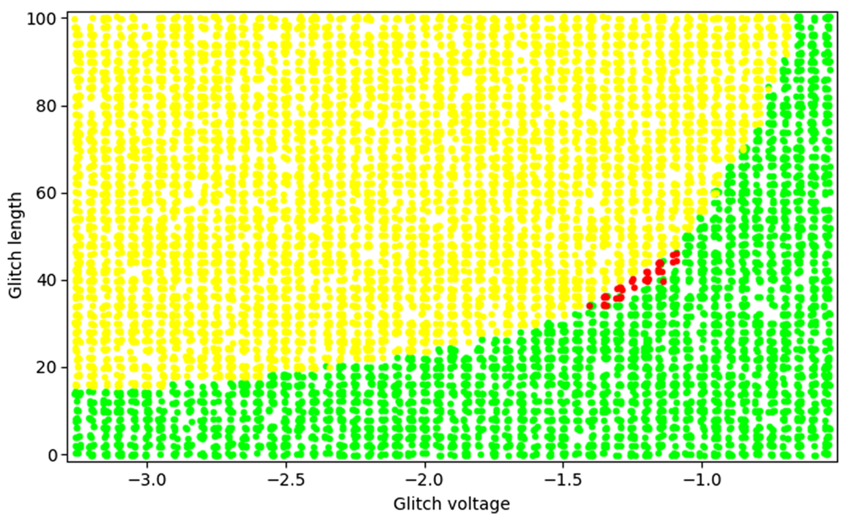

Fig. 5 Relation between the glitch length, glitch voltage, and fault behavior

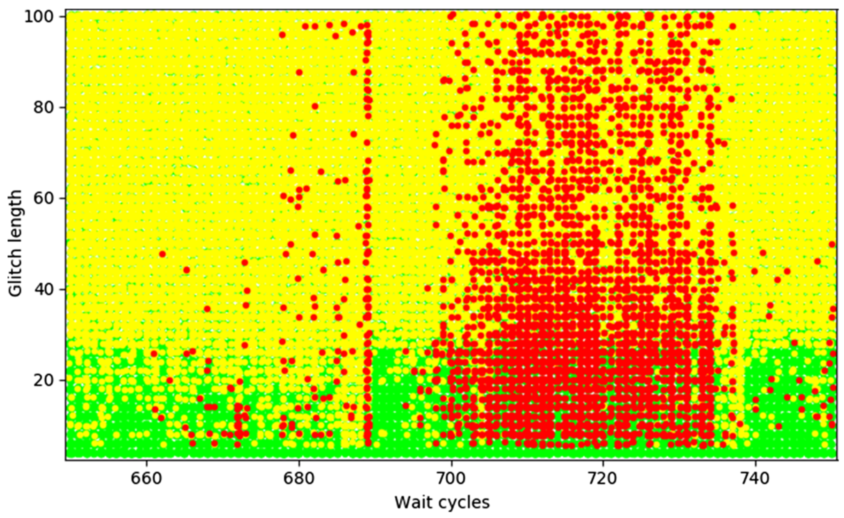

Fig. 6 Relation between the glitch length, temporal offset (i.e., wait cycles), and fault behavior

8 Conclusions

This paper provides a review on mechanisms, implementation, and evaluation of hardware-based fault attacks that aims at breaking the security of embedded software. Our review shows that multiple abstraction layers (software, instruction-set, and hardware layers) take part in fault attacks and each abstraction layer may be a target for an adversary. We also observe that fault attacks break a fundamental assumption made by secure embedded software: The hardware layer of the embedded systems ensures the correct execution of the embedded software. Therefore, fault attacks pose a serious security threat to any kind of security-critical software (firmware, operating system, user applications, and cryptography) running on embedded devices:

– In a fault attack on embedded software, the target of fault injection is the hardware layer while the target of exploitation is the software layer. Thus, it is not always possible to mitigate the fault attack threat with software-only countermeasures.
– Fault attacks do not require presence of a software bug in the embedded software because the fault attacks dynamically alter the behavior of the underlying hardware through fault injection. As our review shows, this enables fault attacks to dynamically induce software bugs, to thwart countermeasures, and to enable other attacks.
– Although fault attacks traditionally require expertise and expensive equipment, they tend to become more accessible because of the advancement in the field. As we demonstrate, today, it is possible to inject faults via inexpensive hardware equipment (less than 500$) or via only software programs.

Considering the increasing role of the embedded Internet of Things (IoT) devices in our daily life and critical infrastructure, we believe that embedded hardware and software developers need to put additional effort on mitigation and evaluation of the fault attack risk on the embedded systems.

Acknowledgements The authors would like to thank Dennis Vermoen from Riscure Security Lab for his help and support.

Funding Information During this work, the first author was supported in part through the National Science Foundation Grant 1441710 and 1314598, and in part through the Semiconductor Research Corporation.

References

1. Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown, arXiv:1801.01207
2. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2018) Spectre attacks: exploiting speculative execution, arXiv:1801.01203
3. Piessens F, Verbauwhede I (2016) Software security: vulnerabilities and countermeasures for two attacker models. In: Design Automation &, test in Europe conference & exhibition (DATE), pp 990–999
4. Witteman M, Oostdijk M (2008) Secure application programming in the presence of side channel attacks. In: RSA Conference, vol 2008
5. Yuce B, Ghalaty NF, Deshpande C, Patrick C, Nazhandali L, Schaumont P (2016) FAME: fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In: Hardware and Architectural Support for Security and Privacy (HASP). ACM, p 8
6. Barenghi A, Breveglieri L, Koren I, Naccache D (2012) Fault injection attacks on cryptographic devices: theory, practice, and countermeasures. Proc IEEE 100(11):3056–3076
7. Joye M, Tunstall M (eds) (2012) Fault analysis in cryptography, ser. Information security and cryptography. Springer, Berlin
8. Galathy NF, Yuce B, Schaumont P (2017) A systematic approach to fault attack resistant design. In: Fundamentals of IP and SoC security, pp 223–245. Springer

9. Moro N, Dehbaoui A, Heydemann K, Robisson B, Encrenaz E (2013) Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: 2013 Workshop on fault diagnosis and tolerance in cryptography (FDTC), pp 77–88. IEEE
10. Courbon F, Loubet-Moundi P, Fournier JJ, Tria A (2014) Adjusting laser injections for fully controlled faults. In: International Workshop on constructive side-channel analysis and secure design, pp 229–242. Springer
11. Yuce B, Ghalaty NF, Schaumont P (2015) Improving fault attacks on embedded software using risc pipeline characterization. In: Proc. of FDTC'15, pp 97–108
12. Li Y, Sakiyama K, Gomisawa S, Fukunaga T, Takahashi J, Ohta K (2010) Fault sensitivity analysis. In: Proc. of CHES'10, pp 320–334
13. Bhattacharya S, Mukhopadhyay D (2017) Formal fault analysis of branch predictors: attacking countermeasures of asymmetric key ciphers. J Cryptogr Eng 7(4):299–310
14. Bar-El H, Choukri H, Naccache D, Tunstall M, Whelan C (2006) The sorcerer's apprentice guide to fault attacks. Proc IEEE 94(2):370–382
15. Guilley S, Sauvage L, Danger J-L, Selmane N, Pacalet R (2008) Silicon-level solutions to counteract passive and active attacks. In: 5th Workshop on fault diagnosis and tolerance in cryptography, 2008. FDTC'08. IEEE, pp 3–17
16. Zussa L, Dutertre J-M, Clédiere J, Robisson B, Tria A et al (2012) Investigation of timing constraints violation as a fault injection means. In: 27th Conference on design of circuits and integrated systems (DCIS). Avignon
17. Korak T, Hoefler M (2014) On the effects of clock and power supply tampering on two microcontroller platforms. In: Proc. of FDTC'14, pp 8–17
18. Riscure Inspector FI https://www.riscure.com/security-tools/inspector-fi/, Online; Accessed 18 May 2017
19. O'Flynn C, Chen ZD (2014) ChipWhisperer: an open-source platform for hardware embedded security research. In: Constructive side-channel analysis and secure design. Springer, pp 243–260
20. Barenghi A, Bertoni G, Parrinello E, Pelosi G (2009) Low voltage fault attacks on the RSA Cryptosystem. In: 2009 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 23–31
21. Timmers N, Spruyt A, Witteman M (2016) Controlling PC on ARM using fault injection. In: Fault diagnosis and tolerance in cryptography (FDTC), pp 25–35
22. Hutter M, Schmidt J-M (2013) The temperature side channel and heating fault attacks. In: International conference on smart card research and advanced applications. Springer, pp 219–235
23. Skorobogatov S (2009) Local heating attacks on flash memory devices. In: IEEE International workshop on hardware-oriented security and trust. 2009. HOST'09. IEEE, pp 1–6
24. Govindavajhala S, Appel AW (2003) Using memory errors to attack a virtual machine. In: 2003 Symposium on security and privacy, 2003. Proceedings. IEEE, pp 154–165
25. Korak T, Hutter M, Ege B, Batina L (2014) Clock glitch attacks in the presence of heating. In: 2014 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 104–114
26. Skorobogatov S, Anderson RJ (2002) Optical fault induction attacks. In: Revised Papers from the 4th international workshop on cryptographic hardware and embedded systems. Springer-Verlag, pp 2–12
27. Schmidt J-M, Hutter M Optical and EM fault-attacks on CRT-based RSA: concrete results
28. Van Woudenberg JG, Witteman MF, Menarini F (2011) Practical optical fault injection on secure microcontrollers. In: 2011 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 91–99
29. Maistri P, Leveugle R, Bossuet L, Aubert A, Fischer V, Robisson B, Moro N, Maurine P, Dutertre J-M, Lisart M (2014) Electromagnetic analysis and fault injection onto secure circuits. In: 2014 22nd International conference on very large scale integration (VLSI-SoC). IEEE, pp 1–6
30. Moro N, Dehbaoui A, Heydemann K, Robisson B, Encrenaz E (2014) Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller, CoRR, vol. abs/1402.6421. [Online]. Available: arXiv: 1402.6421
31. Velegalati R, Van Spyk R, van Woudenberg J (2013) Electro magnetic fault injection in practice. In: International Cryptographic module conference (ICMC)
32. Tang A, Sethumadhavan S, Stolfo S (2017) CLKSCREW: exposing the perils of security-oblivious energy management. In: 26th USENIX security symposium (USENIX Security 17). Vancouver, BC: USENIX Association, pp 1057–1074. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang
33. Cai Y, Ghose S, Luo Y, Mai K, Mutlu O, Haratsch EF (2017) Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. In: 2017 IEEE International symposium on high performance computer architecture (HPCA). IEEE, pp 49–60
34. Kim Y, Daly R, Kim J, Fallin C, Lee JH, Lee D, Wilkerson C, Lai K, Mutlu O (2014) Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In: ACM SIGARCH Computer architecture news, vol 42, no 3. IEEE Press, pp 361–372
35. Gruss D, Maurice C, Mangard S (2016) Rowhammer. js: a remote software-induced fault attack in javascript. In: Detection of intrusions and malware, and vulnerability assessment. Springer, pp 300–321
36. van der Veen V, Fratantonio Y, Lindorfer M, Gruss D, Maurice C, Vigna G, Bos H, Razavi K, Giuffrida C (2016) Drammer: deterministic rowhammer attacks on mobile platforms. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM, pp 1675–1689
37. Razavi K, Gras B, Bosman E, Preneel B, Giuffrida C, Bos H (2016) Flip feng shui: hammering a needle in the software stack. In: USENIX Security symposium, pp 1–18
38. Kurmus A, Ioannou N, Papandreou N, Parnell T (2017) From random block corruption to privilege escalation: a filesystem attack vector for rowhammer-like attacks. In: USENIX Workshop on offensive technologies (WOOT)
39. Karaklajic D, Schmidt J, Verbauwhede I (2013) Hardware designer's guide to fault attacks. IEEE Trans VLSI Syst 21(12):2295–2306
40. Otto M (2005) Fault attacks and countermeasures. Ph.D. dissertation, University of Paderborn
41. Anceau S, Bleuet P, Clèdière J, Maingault L, Rainard J, Tucoulou R (2017) Nanofocused x-ray beam to reprogram secure circuits. In: Cryptographic hardware and embedded systems (CHES), pp 175–188
42. Barbu G, Thiebeauld H, Guerin V (2010) Attacks on java card 3.0 combining fault and logical attacks. Smart Card Research Adv Appl, 148–163
43. Dehbaoui A, Mirbaha A-P, Moro N, Dutertre J-M, Tria A (2013) Electromagnetic glitch on the AES round counter. In: International Workshop on constructive side-channel analysis and secure design. Springer, pp 17–31
44. Riviere L, Najm Z, Rauzy P, Danger J-L, Bringer J, Sauvage L (2015) High precision fault injections on the instruction cache of ARmV7-m architectures. In: 2015 IEEE International

symposium on hardware oriented security and trust (HOST). IEEE, pp 62–67
45. Nashimoto S, Homma N, Hayashi Y-i, Takahashi J, Fuji H, Aoki T (2017) Buffer overflow attack with multiple fault injection and a proven countermeasure. J Cryptogr Eng 7(1):35–46
46. Balasch J, Gierlichs B, Verbauwhede I (2011) An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: Workshop on fault diagnosis and tolerance in cryptography (FDTC 2011), pp 105–114. [Online]. Available: https://doi.org/10.1109/FDTC.2011.9
47. Vétillard E, Ferrari A (2010) Combined attacks and countermeasures. In: International conference on smart card research and advanced applications. Springer, pp 133–147
48. Potet M-L, Mounier L, Puys M, Dureuil L (2014) Lazart: a symbolic approach for evaluation the robustness of secured codes against control flow injections. In 2014 IEEE Seventh International conference on software testing, verification and validation (ICST). IEEE, pp 213–222
49. Choukri H, Tunstall M (2005) Round reduction using faults. FDTC 5:13–24
50. Dutertre J-M, Mirbaha A-P, Naccache D, Ribotta A-L, Tria A, Vaschalde T (2012) Fault round modification analysis of the advanced encryption standard. In: 2012 IEEE International symposium on hardware-oriented security and trust (HOST). IEEE, pp 140–145
51. Biham E, Shamir A (1997) Differential fault analysis of secret key cryptosystems. In: Advances in cryptology—CRYPTO'97. Springer, pp 513–525
52. Hoch JJ, Shamir A (2004) Fault analysis of stream ciphers. In: International Workshop on cryptographic hardware and embedded systems. Springer, pp 240–253
53. Biehl I, Meyer B, Müller V (2000) Differential fault attacks on elliptic curve cryptosystems. In: Annual International cryptology conference. Springer, pp 131–146
54. Taha M, Eisenbarth T (2015) Implementation attacks on post-quantum cryptographic schemes, Cryptology ePrint Archive, Report 2015/1083. http://eprint.iacr.org/
55. Giraud C (2004) DFA on AES. In: International conference on advanced encryption standard. Springer, pp 27–41
56. Ferretti C, Mella S, Melzani F (2014) The role of the fault model in DFA against AES. In: Proceedings of the workshop on hardware and architectural support for security and privacy (HASP). ACM, p 4
57. Sakiyama K, Li Y, Iwamoto M, Ohta K (2012) Information-theoretic approach to optimal differential fault analysis. IEEE Trans Inf Forens Secur 7(1):109–120
58. Ali SS, Mukhopadhyay D, Tunstall M (2013) Differential fault analysis of AES: towards reaching its limits. J Cryptogr Eng 3(2):73–97
59. Ghalaty NF, Yuce B, Taha M, Schaumont P (2014) Differential fault intensity analysis. In: 2014 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 49–58
60. Li Y, Ohta K, Sakiyama K (2012) New fault-based side-channel attack using fault sensitivity. IEEE Trans Inf Forens Secur 7(1):88–97
61. Liu Y, Zhang J, Wei L, Yuan F, Xu Q (2015) Dera: yet another differential fault attack on cryptographic devices based on error rate analysis. In: Design Automation conference (DAC). ACM, p 31
62. Fuhr T, Jaulmes E, Lomné V, Thillard A (2013) Fault attacks on AES with faulty ciphertexts only. In: 2013 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 108–118
63. Järvinen K, Blondeau C, Page D, Tunstall M (2012) Harnessing biased faults in attacks on ECC-based signature schemes. In: 2012 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 72–82
64. Joye M, Jean-Jacques Q, Sung-Ming Y, Yung M (2002) Observability analysis-detecting when improved cryptosystems fail. In: Cryptographers' track at the RSA conference. Springer, pp 17–29
65. Yen S-M, Joye M (2000) Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans Comput 49(9):967–970
66. Karaklajic D, Fan J, Verbauwhede I (2012) A systematic M safe-error detection in hardware implementations of cryptographic algorithms. In: 2012 IEEE International Symposium on hardware-oriented security and trust (HOST), pp 96–101
67. Blömer J, Seifert J-P (2003) Fault based cryptanalysis of the advanced encryption standard (AES). In: Computer Aided verification. Springer, pp 162–181
68. Boneh D, DeMillo RA, Lipton RJ (1997) On the importance of checking cryptographic protocols for faults. In: International Conference on the theory and applications of cryptographic techniques. Springer, pp 37–51
69. Ciet M, Joye M (2005) Elliptic curve cryptosystems in the presence of permanent and transient faults. Des Codes Cryptograph 36(1):33–43
70. Fouque P-A, Lercier R, Réal D, Valette F (2008) Fault attack on elliptic curve montgomery ladder implementation. In: 5th Workshop on Fault diagnosis and tolerance in cryptography. 2008. FDTC'08. IEEE, pp 92–98
71. Kocher P, Jaffe J, Jun B (1999) Differential power analysis. In: Advances in cryptology—CRYPTO'99. Springer, pp 789–789
72. Fan J, Guo X, De Mulder E, Schaumont P, Preneel B, Verbauwhede I (2010) State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures. In: 2010 IEEE International Symposium on hardware-oriented security and trust (HOST). IEEE, pp 76–87
73. Oswald D (2013) Implementation attacks: from theory to practice, Ph.D dissertation
74. Spreitzer R, Moonsamy V, Korak T, Mangard S (2017) Systematic classification of side-channel attacks: a case study for mobile devices. IEEE Communications Surveys & Tutorials
75. Tillich S, Herbst C (2008) Attacking state-of-the-art software countermeasures—a case study for AES. Lect Notes Comput Sci 5154:228–243
76. Rivain M, Prouff E (2010) Provably secure higher-order masking of AES. Cryptograph Hardware Embedded Syst CHES 2010:413–427
77. Grosso V, Standaert F-X, Faust S (2014) Masking vs. multiparty computation: how large is the gap for AES? J Cryptogr Eng 4(1):47–57
78. Chevallier-Mames B, Ciet M, Joye M (2004) Low-cost solutions for preventing simple side-channel analysis: side-channel atomicity. IEEE Trans Comput 53(6):760–768
79. Skorobogatov S (2006) Optically enhanced position-locked power analysis. Cryptograph Hardware Embedded Syst-CHES 2006:61–75
80. Amiel F, Villegas K, Feix B, Marcel L (2007) Passive and active combined attacks: combining fault attacks and side channel analysis. In: Workshop on Fault diagnosis and tolerance in cryptography, 2007. FDTC 2007. IEEE, pp 92–102
81. Clavier C, Feix B, Gagnerot G, Roussellet M (2010) Passive and active combined attacks on AES combining fault attacks and side channel analysis. In: 2010 Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 10–19
82. Roche T, Lomné V, Khalfallah K (2011) Combined fault and side-channel attack on protected implementations of AES. Smart Card Res Adv Appl, 65–83

83. Dassance F, Venelli A (2012) Combined fault and side-channel attacks on the AES key schedule. In: 2012 Workshop on Fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 63–71
84. Schmidt J-M, Tunstall M, Avanzi RM, Kizhvatov I, Kasper T, Oswald D (2010) Combined implementation attack resistant exponentiation. LATINCRYPT 6212:305–322
85. Yao Y, Yang M, Patrick C, Yuce B, Schaumont P (2018) Fault-assisted side-channel analysis of masked implementations (to appear). In IEEE International Symposium on hardware oriented security and trust (HOST), 2018. IEEE, pp 72–77
86. Durumeric Z, Kasten J, Adrian D, Halderman JA, Bailey M, Li F, Weaver N, Amann J, Beekman J, Payer M, Paxson V (2014) The matter of heartbleed. In: Internet Measurement conference (IMC), pp 475–488
87. Obermaier J, Tatschner S (2017) Shedding too much light on a microcontroller's firmware protection. In: USENIX Workshop on offensive technologies (WOOT)
88. Scott ME Glitchy descriptor firmware grab, https://www.youtube.com/watch?v=TeCQatNcF20, Online; Accessed 14 Nov 2017
89. Bouffard G, Iguchi-Cartigny J, Lanet J-L (2011) Combined software and hardware attacks on the java card control flow. In CARDIS, vol 7079. Springer, pp 283–296
90. Vasselle A, Thiebeauld H, Maouhoub Q, Morisset A, Ermeneux S (2017) Laser-induced fault injection on smartphone bypassing the secure boot. In: 2017 Workshop on Fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 41–48
91. Timmers N, Mune C (2017) Escalating privileges in Linux using voltage fault injection. In: Fault Diagnosis and tolerance in cryptography (FDTC), pp 25–35
92. Seaborn M, Dullien T (2015) Exploiting the dram rowhammer bug to gain kernel privileges. Black Hat
93. San Pedro M, Soos M, Guilley S (2011) Fire: fault injection for reverse engineering. In: WISTP. Springer, pp 280–293
94. Le Bouder H, Guilley S, Robisson B, Tria A (2014) Fault injection to reverse engineer DES-like cryptosystems. In: Foundations and practice of security. Springer, pp 105–121
95. Clavier C, Wurcker A (2013) Reverse engineering of a secret AES-like cipher by ineffective fault analysis. In: 2013 Workshop on Fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 119–128
96. Jacob M, Boneh D, Felten E (2002) Attacking an obfuscated cipher by injecting faults. In: Digital Rights management workshop, vol 2696, pp 16–31
97. Courbon F, Fournier JJ, Loubet-Moundi P, Tria A (2015) Combining image processing and laser fault injections for characterizing a hardware AES. IEEE Trans Comput-aided Des Integr Circ Syst 34(6):928–936
98. Common Criteria Community https://www.commoncriteriaportal.org, Online Sccessed 18 Jan 2018
99. United States Government Accountability Office, Information assurance, national partnership offers benefits, but faces considerable challenges, Technical Report GAO-06-392, 2006. http://www.gao.gov/new.items/d06392.pdf
100. EMVCo Product Approval Processes http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf, Online Accessed 18 Jan 2018
101. National Institute of Standards and Technology (NIST), Security requirements for cryptographic modules, FIPS PUB 140-2, 2001. https://www.emvco.com/processes-forms/product-approval/