FaultLine: Software-based Fault Injection on Memory Transfers

Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, Philippe Loubet Moundi

▶ **To cite this version:**

Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, Philippe Loubet Moundi. FaultLine: Software-based Fault Injection on Memory Transfers. 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), Dec 2021, Tysons Corner, France. pp.46-55, ⟨10.1109/HOST49136.2021.9702295⟩. ⟨hal-04672567⟩

HAL Id: hal-04672567

https://hal.science/hal-04672567v1

Submitted on 19 Aug 2024

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

FaultLine: Software-based Fault Injection on Memory Transfers

Joseph Gravellier[1], Jean-Max Dutertre[1], Yannick Teglia[2] and Philippe Loubet Moundi[2]
[1]Mines Saint-Etienne, CEA-Tech, Centre CMP. Gardanne, France
{joseph.gravellier, dutertre}@emse.fr
[2]Thales, La Ciotat, France
{yannick.teglia, philippe.loubet-moundi}@thalesgroup.com

***Abstract*—Today's integrated memory controllers use complex hardware such as delay-lines to monitor and control signal timings during external memory transfers. Because memory chips with different timing specifications may be used, delay-line tuning registers often remain accessible and programmable from the application processor.**

In this paper, we introduce FaultLine and the concept of delay-line-based fault injection. First, we demonstrate that by modifying the delay-line calibration value through a simple register access, a malware may induce faults in memory transfers and jeopardize the security of concurrently running assets. Then, we experimentally evaluate the fault injection on an OS-capable system-on-chip by exposing cryptographic applications to corrupted data and retrieving their secret keys. We finally discuss why delay-line-based fault injection should be systematically considered as a potential threat in modern systems where entities with different privileges share external memories.

***Index Terms*—Fault Injection Attack, Software-based, External Memory, Delay-Lines, SoC, PFA, DFA.**

I. Introduction

A software-based hardware attack takes advantage of the security-oblivious cooperation between hardware and software resources [1]. Through a computer malware it aims at accessing and turning a hardware component into a side-channel analysis (SCA) vector or into a fault injection attack (FIA) mechanism. A software-based hardware attack differs from an usual hardware attack since it doesn't involve any equipment beyond the target itself. In that way, it enables hardware attacks on remote devices.

A physical attack is such a powerful threat since it bypasses the logical isolation implemented in system-on-chips (SoCs). Even if an asset (processor, cryptoprocessor, secure-element, FPGA) is efficiently protected against software accesses, it still shares the same resources and silicon die with the other SoC parts. It means, amongst others, that the asset activity affects its surroundings and vice-versa. Software-based power analysis attacks demonstrated this by proving that using an embedded power meter, a physical processor core can eavesdrop the activity of another one. For instance, SideLine attack on ARM devices [2] and Platypus attack on Intel computers [3] bypassed the logical isolation between processes using power consumption sensors. These attack vectors were used to carry out power attacks such as cryptographic key inferring.

In SideLine, delay-line components implemented in device memory controllers were studied as potential software-based hardware attack vectors. The authors exploited the delay-line relationship with voltage fluctuations in order to perform power side-channel attacks. By reading the delay-line state and turning it into leakage information, SideLine demonstrated that unsuspected hardware components may be misused by a malware in order to conduct remote hardware attacks.

In our paper, we introduce a novel fault injection mechanism suitable for software-based hardware attacks. Similarly to SideLine, we focus on programmable delay-lines components widely available in modern systems that use external memories. However, instead of monitoring the delay-line state, we force it to a faulty value. Therefore, we turn the passive side-channel attack vector into an active fault injection medium. Our contributions are described below:

- We reveal that delay-lines components available in a wide range of memory controllers can be turned into memory transfer fault vectors.
- We carry out an extensive characterization of the fault vector. We provide guidelines to control the fault and statistics on the gathered errors.
- On a Xilinx Zynq development board, we evaluate the attack vector in bare-metal mode. We deploy two programs, victim and attacker, and inject faults on the SDRAM accesses launched by the victim.
- We then evaluate the fault injection vector while running a Linux-based OS and deploy attacks on

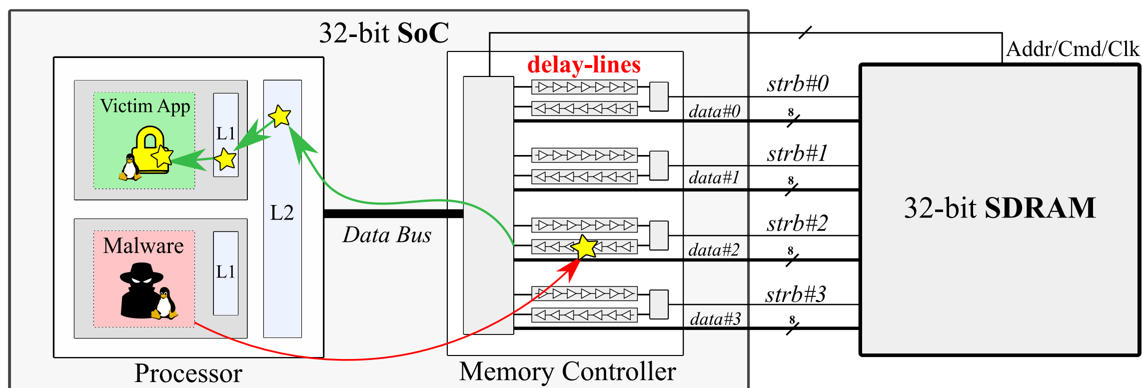

Fig. 1: Memory transfer organisation between a SoC and an external SDRAM memory chip.

AES encryption and RSA signature applications.

The remainder of this paper is organized as follows. In section II, we provide background information on fault injection attacks and describe the threat model. Then, we present the experimental setup and the fault model in section III. Sections IV and V are dedicated to the deployment of the attack scenarios. We discuss performances, limitations, countermeasures in section VI. Section VII concludes the paper.

Responsible Disclosure: We responsibly disclosed our findings to Xilinx on March 4th, 2021 which acknowledged and agreed on the publication of these results. Please keep in mind that FaultLine has been performed on this processor for demonstration purposes but the concept is generic and any devices that embed delay-lines can be affected.

II. Preliminaries

This section covers the technical background around the delay-line-based fault injection. It also introduces how it could be used to jeopardize SoC security.

A. Monitoring Memory Transfers

Figure 1, illustrates the typical components involved in a memory transfer operation between a SoC and an external SDRAM memory. The keystone of this structure is the integrated memory controller. Its main role is to collect and schedule memory access requests. Moreover, it ensures proper timings for the data signals flowing from the SoC to the memory, and vice-versa.

A dual data rate (DDR) memory fetches data on both rising and falling-edges of a strobe signal (the strobe is the clock signal for the data lines, each data byte has its own strobe). The high throughput achieved by DDR3+ memories increases the risks of timing errors during a data fetch especially if the data lags behind the strobe signal. The signal lag depends on the length of the PCB tracks which connect the SoC to the external memory. Moreover, it also changes dynamically with voltage and temperature variations. To preserve memory operations from timing errors induced by these lags, recent memory controllers use a dedicated hardware block usually called PHY controller. The PHY implements, amongst others, hardware synchronization mechanisms such as delay-lines (depicted in figure 1) to preserve memory operations from timing errors. The idea is to delay the strobe signal with respect to the data in order to make sure that fetching only occurs when the data is ready.

Because the applied delay may vary from a board to another, it should programmable. Usually the delay-line calibration is automatically done at boot-time through memory tests and dynamically updated over time to counteract PVT variations. Almost every memory controller that support DDR3+ memories implements timing control mechanisms. As a result, a wide-range of modern devices contain delay-lines. Static delay-lines are often used along with dynamic elements such as delay-locked-loops and phase-locked-loops. Such elements are not only DDR-related, they can be found in some high-speed Flash and SDMMC memory controllers.

B. Faulting Memory Transfers

Because delay-line settings can be accessed and modified during run-time, it may be maliciously used to corrupt memory transfers. Figure 1 illustrates a simple scenario in which a malware (in red) and a victim application (in green) run concurrently on the SoC processor cores. Both apps occasionally generate memory accesses that are handled by the integrated memory controller. In this particular example, the malware aims at corrupting the data loaded by the victim app. To that end, it modifies the delay-line calibration value for a short period of time (red arrow). The calibration chosen is a critical value that is known to induce faults during reads. With any luck, the

victim app is simultaneously performing a load operation which therefore results in a corrupted data read. The fault is then loaded into cache memory (green arrow) and remains persistent until the faulted data is evicted. Depending on the data loaded, the fault injection may jeopardize the victim app and its security. Several attack scenarios presented in section IV and V arise from this fault model.

C. Threat Model

Modern systems are more and more heterogeneous. They embed, processors, microcontrollers, FPGAs, secure elements, etc. All these assets may use external memories and thus share the memory controller in order to access data. While logical isolation theoretically prevent processes from accessing the others, they cannot detect tampered data transfers induced by a malware.

Here, we further specify the threat model by illustrating it with the example of a SoC embedding both a user and a secure processor. The first one "user processor" implements a rich OS and contains the user data. The second one, "secure processor" performs security related computations: OS verification, login, encryption/decryption. The sensitive data belonging to the secure processor such as cryptographic keys and truslets are protected by a trusted execution environment based for instance on TrustZone [4] and/or a hardware security module. Let's suppose that the user processor is compromised by a hardware level malware or a malicious OS which aims at extracting a protected key. In an attempt to bypass the security isolation, the user processor leverages its access to delay-lines to corrupt memory transfers handled by the secure processor. A fault injected inside a crypto-algorithm can then be exploited to expose its cryptographic key. To that end, the malicious processor leverages traditional fault analysis methods such as differential fault analysis (DFA [5]) or persistent fault analysis (PFA [6]).

III. Experimental Setup and Fault Parameters

FaultLine is generic as delay-lines are implemented in a wide-range of modern devices. As other fault injection vectors it is configurable and should be characterized to maximize the fault rate/success.

A. Experimental Setup

The target adopted to demonstrate our fault mechanism is the ZYBO development board from Digilent `PB200-351 REV B` [7]. This board embeds a Xilinx Zynq SoC `XC7Z010-1CLG400C` [8] and two 16-bit DDR3 memory chips of 512 MB each.

The Zynq processor contains two ARM Cortex-A9 cores cadenced at 666 MHz and an Artix-7 FPGA (not used in this work). Figure 1 illustrates the simplified view of the Zynq memory organization. Each processor core contains independent 32 KB level 1 instruction and data caches. It also integrates a 512 KB level 2 cache which is shared between the two processors.

Two software setups were considered in this work:
a) ***bare-metal*** *setup:* These experiments were built using Xilinx Vitis version 2020.2. We leveraged the possibility to load one executable per CPU core to build dual-core attacks. The bare-metal configuration requires a hardware platform specification file for the ZYBO and the board support package project generated in Vitis. Note that the used Vitis and Vivado versions should not impact the attack results. The algorithm attacked in this scenario is the TinyAES available on GitHub [9]. Bare-metal attacks are described in section IV.
b) ***Linux-based*** *setup:* We implemented the Linaro Debian Linux OS version *linaro-jessie-developer-20161117-32* on the Zybo board. Our Linux-based attacks were not constrained, we did not perform any kernel modification. We did not limit the execution of background processes or kernel processes. In this scenario we targeted two algorithms: the TinyAES and the OpenSSL crypto-library [10]. Linux-based attacks are described in section V.

All the documentation about delay-lines was found in the Xilinx Zynq 7000 technical reference manual [8] chapter *DDR memory controller*, section *PHY controller*. We provide all the bare-metal and linux-based attack codes in the following GitHub repository: https://github.com/Remote-HWA/FaultLine.

B. Deeper View of the Fault Mechanism

When a read operation is initiated by the memory controller, the SDRAM first collects the data in the proper bank and then outputs the data signals and the strobe signals edge-aligned. These signals then travel through the PCB tracks and reach the SoC and its memory controller. Inside the SoC, delay-lines are used to delay the strobe signal with respect to the data.

The top part of figure 2 illustrates the correct sampling of consecutive reads. Here, the black signal represents the data line 0 out of the 32 total data lines. The strobe signal depicted in red is used as a reference clock to fetch the data. In nominal operation, a 90° phase shift is added to delay the strobe signal propagation. This ensures that the data bit is ready and stable before its capture by the strobe edge. The bottom part of figure

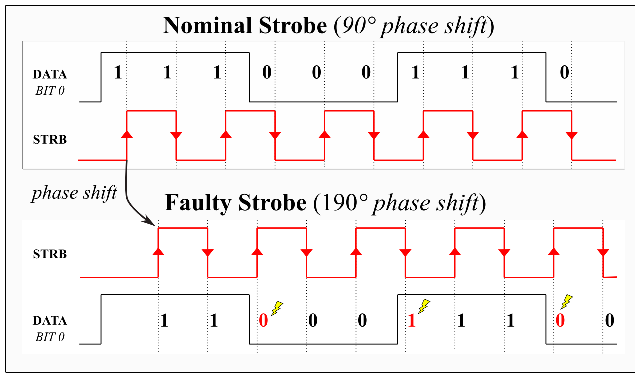

Fig. 2: Nominal vs faulty strobe phase-shifts

2 illustrates a different delay-line calibration for the same read sequence. This time the strobe phase-shift was increased to 190° while the data phase remained stable. This resulted in read errors in the fetching operation as the strobe delay was too high.

The delay-line calibration is controlled from software and the fault can be finely tuned to affect a limited number of read or write operations.

C. Shaping the Glitch

The typical faults obtained using this method are bit flips. Their apparition rate, the number of bits flipped and the number of faulted read and write accesses can be precisely controlled using software parameters. As for power or electromagnetic glitches, delay-line-based fault injection can be finely calibrated to maximize the fault success and limit the number of mutes (processor crashes caused by invalid instructions and segmentation faults). In addition to usual parameters such as glitch strength and width, delay-line-based fault injection bring additional controls on fault direction and faulted-byte. The following list introduces five parameters that need to be tuned using software to inject precise faults in memory transfers.

Listing 1: Fault Injection Pseudo-Code (bare-metal)

```
// register addr depends on direction and byte
addr = get_delayline_addr(direction ,byte);
for(i = 0 ; i < init_delay ; i++){} // delay
writereg(addr,stress);  // start glitch
for(i = 0 ; i < width ; i++){} // glitch width
writereg(addr,nominal); // stop glitch
```

The pseudo-code 1 illustrates how these parameters are controlled from the malware. We comment them in order of appearance:

1. `direction`**:** Because there is necessarily a separate delay-line for read and for write operations, the attacker can choose the direction of the fault. This can be controlled by changing the accessed delay-line register (line 2 in script 1). On the Zynq processor, we induce a write fault by modifying the `phy_wr_dqs_cfg0-3` registers and we induce a read fault by modifying the `phy_rd_dqs_cfg0-3` registers.

2. `byte`**:** The fault can be even more targeted as there is a distinct strobe per memory data byte. That is, 4 strobes for 4 bytes. On the Zynq processor, 4 separate registers can be modified to inject a fault on the 4 distinct data bytes. The byte choice parameter added to the fault direction parameter provide a fine and powerful control to the attacker as he can focus on a specific read or write data byte over the 4 existing ones.

3. `init_delay`**:** This parameter (line 3) acts as a temporization medium prior to the glitch injection. It can be programmed to perform temporal fault injection mappings. In bare-metal mode, we finely tune it using for loops. These are easily programmable and are quite accurate (~1 increment per clock cycle).

4. `stress`: This parameter defines the delay-line calibration value, that is, the phase-shift applied on the strobe signal. To generate read or write faults, the malware modifies the stress in order to induce a faulty phase-shift (line 4). This unstable phase-shift can be reached by increasing or decreasing the stress until a first faulted memory transfer occurs. If the stress is too distant from the nominal operation the risk of obtaining mutes soars. Please note that the optimal delay-line calibration value may slightly change from a board to another due to process variations.

5. `width`**:** The width parameter determines the glitch duration, that is, the time during which the `stress` will be applied on the delay-line. It is controlled by inserting a programmable delay between the glitch injection (line 4) and the glitch relaxation (line 6). In bare-metal this delay is also controlled using for-loops (line 5). We evaluate its impact in section IV.

IV. FaultLine on a Bare-Metal Device

This section describes fault evaluation and attacks conducted using two parallel programs running within the Zynq processor cores.

A. Characterizing the Injected Faults

The delay-line calibration registers can be set and forced to values ranging from 0 (the shortest delay) to 512 (the longest delay). Within this range, we aim at distinguishing the proper delay calibrations from the faulty ones. To that end, we deployed two C programs running on separate CPU cores. The memory

Listing 2: Victim memory transfers pseudo-code

```
int ref_array[nAttack]; //nAttack = 1,000,000
int store_array[nAttack];
DisableCacheL1_L2(); // force SDRAM access
FillArray(ref_array,0x00000000,0xFFFFFFFF);
// The fault is injected within this loop
for(int i = 0 ; i < nAttack ; i++)
    store_array[i] = ref_array[i];
```

caches L1 and L2 were disabled using the Xilinx API `Xil_DCacheDisable()` to ensure that each load or store operation would result in an SDRAM memory access. The injection program runs in CPU#0, It implements a looped version of the listing 1. The victim program runs in CPU#1, it implements the pseudo-code given in listing 2.

The one million elements `test_array` (line 1) is filled with two successive values: `0x00000000` on even indexes and `0xFFFFFFFF` on odd indexes (line 4). Then a loop copies the `test_array` content within the `store_array` (line 6-7). Thanks to the adopted repetitive pattern, a continuous sequence of zeros and ones will travel through each data line when the array is stored or load. We chose this pattern as a higher number of data line state transitions should maximize the fault rate. Figure 2 illustrates this behavior.

The fault injection occurs during the array copy operation. If the fault direction is *read*, it will affect the `test_array` loading from SDRAM. If the fault direction is *write*, it will affect the export of the `store_array` inside the SDRAM. Based on this setup we deployed several tests:

Glitch stress sweep test: For each delay-line calibration values available we performed one million writes to SDRAM. Figure 3.a highlights the results obtained from the smallest to the greatest phase-shift applied. Three different colours corresponding to three different behaviours can be distinguished. The yellow one at the center represents delay-line calibration values which gave correct writes (the write-eye) `store_array = ref_array`. On the edges, the red one represents the delay values which induced mutes (communication with the SoC is lost until reset). Finally, the values displayed in orange are the one that generated usable faults `store_array != ref_array`. Only few values are suitable for fault injection. These values correspond to the boundary between proper and faulty operation. That is, a maximum negative phase-shift (orange left) or a maximum positive phase-shift (orange right) between the strobe and the data. At the operating boundary, the error

a) Glitch stress sweep test results (write)

	mute	fault	write eye	fault	mute
Byte 0	0-26	27-28	29-77	78-80	81-512
Byte 1	0-26	27-29	30-84	85-86	87-512
Byte 2	0-35	36	37-87	88-90	91-512
Byte 3	0-26	27-28	29-74	75-77	78-512

Delay Applied →

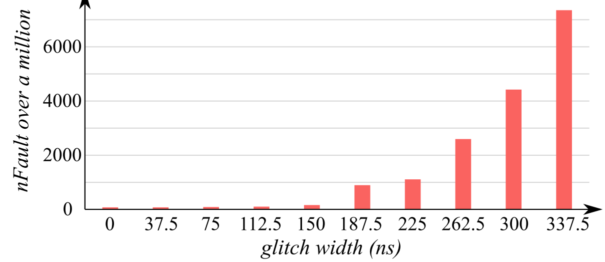

Fig. 3: `stress` and `width` characterization

rate highly depends on PVT variations that make the strobe signal phase unstable. This instability generates both proper and faulty fetches. To avoid mutes, we try to limit the fault rate to less than 1% by finely tuning the glitch parameters. Under this rate, the attack is almost mute free.

Glitch width sweep test: A linear increase of the glitch width do not necessarily involves a linear increase of the fault number. Figure 3.b depicts this behavior obtained on read tests. The fault rate remains quite stable (around 100 over a million) until the glitch width reaches 150 ns. Then it increases much faster between 150 ns and 337.5 ns, higher values generate mutes. From these results, we can deduce that there is a minimal glitch `width` inherent to the hardware. This explains why the fault rate do not depends on the glitch width applied when it is smaller than 150 ns. Once reached and after 225 ns, the fault rate increase becomes roughly linear. In our following experiments, we tuned the glitch `width` according to the glitch `stress` applied. Proper `stress` and `width` pairs were established using sweep tests.

B. Differential Fault Analysis Attack on AES

The first attack application proposed in this paper consists in conducting a core-vs-core Piret & Quisquater (P&Q) DFA attack [5]. The attacker and victim programs run in separate CPUs. The victim CPU#1 computes Tiny AES encryptions on attacker CPU#0 request. The programs are loaded through JTAG inside SDRAM at address 0x5000000 for the attacker and 0x6000000 for the victim. To facilitate the DFA Attack, we disabled the data caches L1 and L2. This relaxes the need to perform

```
[victim] pt: 8620914039f68c7c5a33509aecda185a
[victim] ct: 00000000000000000000000000000000
[attacker] byte 0, stress: 1, delay: 1620
[victim] ct: 0000A800008E00003300000000000090
[victim] ct: 00004900003A00008200000000000066
[attacker] byte 1, stress: 2, delay: 1640
[victim] ct: 2A000000000000D10000450000940000
[victim] ct: A4000000000000820000510000 9F0000
[attacker] byte 2, stress: 56, delay: 1640
[victim] ct: B40000000000007200003900005A0000
[attacker] byte 3, stress: 52, delay: 1620
[victim] ct: 00000089000056000045000 0AA000000
[victim] ct: 000000100000FD00004500001C000000
```

TABLE I: Variations in faulty ciphertexts obtained depending on the delay-line byte choice (read direction)

a cache flush before each fault injection as the Sbox table would have been automatically loaded into cache memory.

Here is how the attacker proceeds: first, an encryption from CPU#1 is requested. Shortly after, it generates a delay-line glitch according to listing 1. We chose to conduct a simple DFA attack in AES round 9 as described in [5]. To that end, the glitch must be injected between the penultimate and ultimate *MixColumns* transformations. A fault success can be easily detected as exactly 4 bytes of the faulted ciphertext will differ from those of the correct one.

To find the proper injection timings, our malware gradually increased the `init_delay` with steps of 150 ns. As expected, the first faults injected within the AES computation modified all the ciphertext bytes. Once the injection timing reached the penultimate *MixColumns* we started obtaining 4-byte ciphertext faults. Table I shows several faulty ciphertexts collected with faults injected on the 9th AES round. For visualization ease, we chose a constant plaintext `pt` which generates a zero ciphertext `ct` on nominal AES encryption. We also used the `byte` parameter as an advantage to inject faults in different AES columns. Indeed, the P&Q attack requires at least 2 faults in each AES column to succeed. The control of the delay-line `byte` targeted allowed us to choose the column under attack (faulty positions in red). This additional control maximizes the chances to obtain new groups of faulty ciphertexts and thus increases the attack speed. Based on our experimental results, it took 100,000 injections (2.4 minutes) and around 500 AES faults in average to retrieve the full AES secret key. Among these 500 faults, 8 only were required for DFA the others were duplicates or faults injected within the same column.

C. Persistent Fault Analysis Attack on AES

The P&Q attack presented upwards requires memory caches disabling and thus makes the attack hardly feasible in practice. This is why we introduce a second application of delay-line-based fault injection which relaxes the attacker prerequisites: the persistent fault analysis (PFA) [6]. The PFA attack leverages the processor capability to temporarily store data and instructions into memory caches instead of performing time-consuming SDRAM memory accesses every time. If a fault is injected during a memory access, it will remain persistent in the cache until the data is evicted. It has been experimentally demonstrated in [6] that this effect can be maliciously employed to break symmetric crypto-algorithms.

We deployed PFA on our experimental setup. Again, the CPU#1 computes Tiny AES encryptions on CPU#0 request. However, this time, both L1 and L2 memory caches are enabled. Because the data loaded into the cache memory remains cached, a fault injected during the first Sbox loading from SDRAM will persist over time. Therefore, on AES, by faulting a single Sbox reading, we can expect to continuously obtain faulty ciphertexts. This requires only one successful fault injection, success is easy to detect and more importantly this doesn't require fine synchronization. Here is how the attacker may proceed:

1. Flushing: The attacker flushes the cache memory to remove all the Sbox indexes potentially stored during previous encryptions. The Zynq board support package comes with several APIs to flush the data and instruction caches. Here we used the `Xil_DCacheFlushRange()` function to flush the AES variables.

2. Collecting: The attacker then asks for 20,000 AES encryptions to the victim process (without cache flush). If a fault is detected, the attacker saves the encryptions for post-treatment, else he returns to the first step.

3. Analysing: Using the collected ciphertexts, the attacker computes the Sbox candidates according to the method described in [6]. If a single Sbox fault was injected he can retrieve the full AES key using less than 2,000 faulty ciphertexts.

In practice, it took less than 10 seconds and around 1,000 glitches to obtain a suitable fault. Based on this faulted Sbox, we computed and collected 17,053 faulty ciphertexts. An illustration of the faulty ciphertext distribution obtained for the first four AES key bytes is provided in figure 4. The c_j^{min} apparition rate (in red) remains zero because its associated Sbox output was removed by the fault. c_j^{min} emerges from the global distribution after around 1,200 ciphertexts. With this information, we can retrieve the missing Sbox index

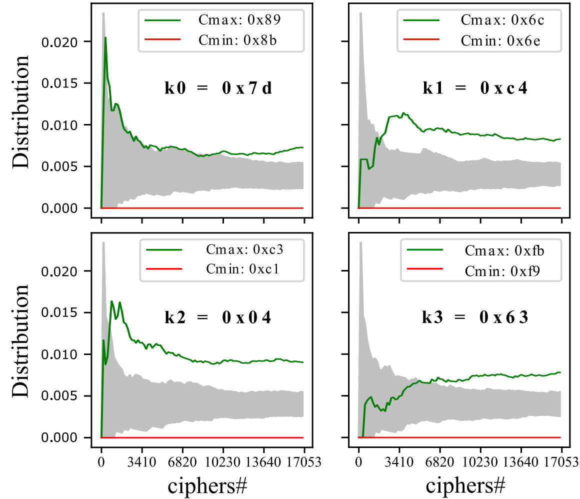

Fig. 4: Progression of the PFA distribution over 17,053 faulty ciphertexts. The `Cmin` distribution (in red) results from the missing Sbox. The `Cmax` distribution (in green) results from the duplicated Sbox value.

and value. Then we use the following equation $c_j^{min} = Sbox[i] \oplus K_j$ to retrieve the AES key bytes K_j.

D. Conclusion on Bare-Metal Results

In this section, we demonstrated that a simple SoC register modification can lead to a complete disclosure of cryptographic secrets. We demonstrated that DFA and PFA attacks are feasible on a Zynq device using delay-line-based fault injection. The fact that memory caches are enabled doesn't mitigate the attack as long as the attacker finds a method to flush them. Delay-line-based fault injection also provides additional fault parameters such as direction and byte choice. These were used to facilitate both the DFA and PFA attacks.

V. FaultLine on a Device Running Linux

The bare-metal setup presented in section IV is convenient for fault characterization as it relaxes the attack complexity. With a Linux Debian OS now deployed on the Cortex-A9 processors, we aim at generalizing, comparing and demonstrating that our attack medium remains functional in real-life complex OS scenarios.

A. Attack Setup

Each attack scenario presented in this section involves two Linux applications: a `malware` and a victim `encrypt` app. The following paragraphs describe the attack scenario.

Attack script: Listing 3 illustrates the three commands required to inject a fault. First of all the attack

Listing 3: Linux OS Attack Script

```
sync; echo 3 > /proc/sys/vm/drop_caches
./malware -dly -val -width -dir -byte &
./encrypt -nEnc &
```

flushes the entire memory cache to make sure that the victim data will be loaded from SDRAM. Here, we used the system call `drop_caches` to flush the data stored in memory caches. Dropping all the memory caches is time-consuming and drastically reduces the fault injection rate. However, to our knowledge, this is the only flushing method available on this platform that do not involve any kernel modification. Then, the two following script lines successively launch the `malware` and the victim `encrypt` apps. The presence of the & symbol at the end of each system call enables concurrent application running. This ensures that the malware and the encryption apps will run simultaneously.

Malware App: On listing 3 line 2, the malware app takes the glitch parameters as arguments. When launched, it first maps the delay-line physical addresses using the `mmap()` privileged system call. Then, it injects the glitch with respect to the pseudo-code 1 and exits. The `init_delay` mechanism previously implemented using for-loops was improved with the use of `clock_gettime()`. This system call directly accesses the hardware performance counters and is therefore less affected by temporal jitter caused by OS interruptions. We leveraged this stable delay medium to perform fine temporal mappings. That is, we gradually delayed the fault injection with respect to the encryption app execution until a fault occurred.

External Computer: An external computer (not required but used for experimentation ease) automates the attack and resets the Zybo board on mutes. Python Paramiko package is used to script the SSH communication between the computer and the Zybo. The bash script 3 is continuously relaunched with new arguments. Once the injection campaign is completed, a post-attack analysis is conducted on the computer side to retrieve the successful errors.

B. Simple Data Byte Corruption

We first aimed to validate whether the delay-line-based fault injection was still feasible with the presence of an OS. The adopted method is equivalent to the one described in the bare-metal experiments IV-A, that is, a victim application loads an array from SDRAM while the attacker injects a delay glitch.

This time, memory caches L1 and L2 were enabled and the default OS kernel applications were running simultaneously to the experimental applications. To avoid array caching, the victim application directly instantiates the array within the SDRAM by mapping memory addresses using `mmap`. By doing so, the victim application is forced to load data from SDRAM: `array_value = *(volatile int)(virt_SDRAM_addr)`.

The results below illustrate the array values loaded from SDRAM without and with the presence of a delay glitch (these array values were randomly initialized by the victim app). In this experiment, the attacker app focused the delay-line controlling the strobe for the data byte #0. By XORing the results, we can indeed observe that the faults only impact the array byte #0. This observation was experimentally tested and confirmed on the 3 other data bytes and for both read and write attack directions.

```
[attacker] Byte 0, stress: 1, delay: 1620
[attacker] fault: 683dede1 -> 683dede3 - XOR: 00000002
[attacker] fault: 08bbb4a7 -> 08bbb4ef - XOR: 00000048
[attacker] fault: 61bef407 -> 61bef405 - XOR: 00000002
[attacker] fault: 34a787be -> 34a787b4 - XOR: 0000000a
```

The OS kernel is a potential collateral victim of our experiments. We observed that the kernel was sometimes subjected to memory errors such as page faults or NULL pointer dereference. These errors usually induce mutes and necessitate a board reset to restart the fault campaign. Despite this new constraint, it usually took less than 10 seconds to obtain a successful fault as long as the glitch and timing parameters were properly calibrated. To conclude, the fault model remains consistent with or without the presence of an OS although the latter is more likely to be subjected to mutes.

C. Persistent Fault Analysis Attack on AES

PFA fits particularly well with the adopted Linux setup as it relaxes the need for time-synchronization which cannot easily be controlled on Linux because of the timing jitter constantly induced by kernel interruptions. The AES executable evaluated implements the Tiny AES source code and takes the number of encryption requested as argument. We chose a number of 6,000 encryptions to conduct the PFA. For each launch, the application computed 6,000 AES encryptions using a predefined plaintext sequence and stored the associated ciphertexts in a text file. The presumably faulted output file was each time compared with a correct file in order to detect a divergence. If so, we saved it. Based on this setup, we conducted 10,000 injections ($\sim$ 2 hours) and obtained 26 files containing faulty ciphertexts.

All the 26 files contained results suitable for PFA. However, 25 of them were subjected to an additional difficulty brought by the OS which causes the faulty Sbox to get evicted from cache after a while. In these cases the faulty Sbox didn't remained cached during enough encryptions and the PFA attack couldn't be completed (PFA requires at least 1,641 faulty ciphertexts [6]). In this campaign, a unique file met the PFA requirements and the AES key was successfully retrieved using 4,375 faulty ciphertexts. Later on, we were able to reproduce this attack with different data and key.

D. Bellcore Attack on OpenSSL Signatures

Using the OpenSSL v1.1.0a version available on GitHub, we deployed a signature application written in C. This simple application reads an input text file, signs it using the OpenSSL `RSA_private_encrypt()` function and stores the computed signature within an output file. The `rsa_ossl_mod_exp()` function used for modular exponentiation implements the Chinese Remainder Theorem (CRT). CRT speeds up the RSA computation by splitting the signature calculation into two simpler partial signature computations which are combined afterward.

The Bellcore attack [11] demonstrates that a fault injection corrupting the calculation of a partial signature may leak the RSA private key. If the attacker is able to collect a correct signature S and a faulty signature S', he may retrieve the entire RSA private key by computing $gcd(S'-S, N)$. According to the listing 3, we conducted 1,500 injections on our RSA signature application and collected 45 faulty signatures. However, as in [12], none of them led to a successful Bellcore attack.

The `rsa_ossl_mod_exp()` function includes a protection against Bellcore attacks. Before outputting the computed signature, it verifies that $S^e - M = 0$. Where S is the computed signature, e the public exponent and M the file to sign. To check whether this verification was the cause of our failure, we removed it and recompiled OpenSSL. By injecting 100,000 glitches ($\sim$ 20 hours) on the unprotected version, we obtained 5,713 various errors including 14 faulty signatures . All of them led to RSA full private key recovery. This result demonstrates that delay-line-based fault injection is not restricted to DFA and PFA but can also be used to conduct Bellcore attacks on unprotected RSA implementations.

On the protected OpenSSL version, we tried to inject double faults in order to generate a faulty signature and then bypass the protection. However, considering the fact that the success rate for a single fault was 0.00014%,

a double fault event was very unlikely to occur. This limitation is discussed in section VI.

E. Conclusion on OS Results

With an OS implemented, additional difficulties arise such as synchronisation control or cache eviction constraints. Despite these restrictions, we demonstrated that data byte corruption, PFA and Bellcore fault attacks were feasible. The following section addresses the overall performance and limitations of our attack vector and compares it with existing works.

VI. Discussion

A. Related Works

In 2014, Rowhammer [13], [14] was the first hardware vulnerability to be exploited using software programming only. Due to an intrinsic leakage phenomenon in DRAM memories, a malware could induce bit flips in DRAM without accessing it. This threat directly exposed millions of devices to hardware attacks. Inspired by the Rowhammer potential, the research works related to software-based hardware attacks soared. In 2017, a new family of software-based fault attacks arose with CLKscrew [1]. This work demonstrated that by manipulating power and frequency regulators using OS-kernel drivers, an attacker could inject power and frequency glitches inside processes. Again, this hardware attack vector bypassed software isolation and was used to break state-of-the-art security features such as TrustZone on ARM devices. This attack was also demonstrated against SGX [15] on Intel processors with Voltjockey, Plundervolt and V0ltpwn [16], [12], [17]. In addition to SoCs, several temperature and voltage glitch attacks were conducted on FPGA and demonstrated the dangers of multi-tenants FPGA cloud datacenters [18]. Software-based hardware attack topic is growing fast and represents a frightening threat for connected device security. The increasing number of ethical disclosures, CVEs and security patches deployed by industrial traduces the extent of the threat.

B. Advantages and Limitations over Prior Methods

In the following paragraph, we list the strengths and weaknesses of delay-line-based fault injection by comparing it to other existing attack vectors.

Fault Privilege: Because it accesses hardware registers to inject the fault, FaultLine may require root privileges depending on the implemented OS. Although being not as reproducible as the Rowhammer attack which can be launched from user-space, multiple real-world scenarios fit with its prerequisites. All modern systems where entities with different privileges share external memories are potential targets.

Fault Surface: 1) FaultLine affects all the memory accesses regardless of their origin. This phenomenon enables the fault injection on other processes and thus the attack. However, this could sometimes induce errors on untargeted processes. These collateral damages may cause mutes when applied to kernel processes.

2) Even if our experiments used the delay-lines located in the DDR memory controller, FaultLine should not be regarded as SDRAM specific and could be conducted on other components embedding delay-lines in the future (e.g Flash, SD and MMC memory controllers).

Fault Synchronization: As for typical glitch injection, FaultLine success depends on timing synchronization. This differs from Rowhammer whose success is more related to the bit flip location in DRAM. In FaultLine, the right timings were found through try-and-error using delay routines (for-loops, hardware performance counters). This could be improved by the use of cache side-channel or software-based power side-channel [**?**] to finely time the victim operations.

Fault Rate: The delay-line glitch success rate on memory transfers can be rounded to 0.1%. To improve it, one could inject wider glitches but would be submitted to more mutes. Even if V0LTpwn achieves an impressive 99% success rate on OpenSSL hash computations [17], this limited success ratio is recurrent in SoC attacks and exists in Rowhammer and Plundervolt (it takes around 500,000 iterations for Plundervolt to inject a multiplication error [12]).

Fault Precision: Delay-line registers can be finely tuned. They provide full control on the byte and the direction (read/write) chosen by the attacker. These parameters enable precise injection and facilitate attacks such as DFA. They are inherent to FaultLine and offer an improved fault control in comparison to voltage/frequency glitch injection methods.

C. Countermeasures

In the following, we describes three potential approaches to mitigate FaultLine.

Disabling access to delay-lines: As stated previously, synchronization components such as delay-lines cannot be removed and have to remain programmable. However, they should only be accessible during the DRAM training sequence at boot time. A simple mitigation could act at system level by preventing the access to the delay-line registers by unauthorized software entities. Hence, only the OS for instance would have access to this resource.

Another method would be to use a TrustZone aware device that can prevent the delay-line access from the normal world users.

Fault-Resistant Software: As described in the OpenSSL RSA attack experiments, traditional software fault injection countermeasures such as signature verification may effectively mitigate FaultLine. Indeed, the limited FaultLine success rate prevents it from injecting double faults in a reasonable amount of time and thus from defeating this type of protections. However, this might not be true for all the existing countermeasures. For instance, detection based countermeasures such as duplicate encryption [19] may fail due to the persistent behaviour of the fault injected (e.g PFA on AES). To remain effective, these mitigation methods would require cache invalidation between each encryption leading thus to significant time overheads.

Error-Correction Code: The use of software or hardware ECC stands as an efficient method to detect and correct some of the errors induced by FaultLine. By computing a function which depends on the data stored (e.g Hamming Code), the ECC mechanism is able to detect corrupted memory transfers. However, it was proven in [20] that ECC can be defeated with the injection of precise faults on both the data and the computed ECC function. Because our attack vector provide a fine control on the fault applied it could be employed as Rowhammer to defeat this kind of memory protection.

VII. CONCLUSION

In this work we evaluated the use of delay-lines-based components as a potential fault injection mechanism suitable for software-based hardware attacks. By leveraging the open access to delay-lines in a Zynq processor, we were able to induce glitches in external memory accesses and more significantly to corrupt data transfers, RSA signatures and even to retrieve AES keys from cryptographic applications. Delay-line fault injection is unprecedented, it provides new controls and parameters that can be used to finely shape the glitch applied. It thus stands as a powerful novel alternative to the exiting software-based fault mechanisms. Delay-lines are implemented in a wide range of electronic devices from microcontrollers to complex processors. Because malware exploitation of this threat could emerge in a near future, delay-lines should now be considered as a potential threat and systematically protected against malicious modifications.

REFERENCES

[1] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management," in *26th USENIX Security Symposium*, 2017.

[2] J. Gravellier, J.-M. Dutertre, Y. Teglia, and P. L. Moundi, "Side-Line: How Delay-Lines (May) Leak Secrets from your SoC," in *Constructive Side-Channel Analysis and Secure Design*, 2021.

[3] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *IEEE Symposium on Security and Privacy (SP)*, 2021.

[4] L. ARM, "Building a Secure System using TrustZone Technology," tech. rep., 2008.

[5] G. Piret and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, 2003.

[6] F. Zhang, Y. Zhang, H. Jiang, X. Zhu, X. Zhao, Z. Liu, D. Gu, and K. Ren, "Persistent Fault Attack in Practice," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 2, pp. 172–195, 2020.

[7] D. Inc., *Zybo FPGA Board Reference Manual*. 2016.

[8] Xilinx Inc., *Zynq-7000 SoC Technical Reference Manual*. 2018.

[9] Kokke, "https://github.com/kokke/tiny-AES-c," 2018.

[10] OpenSSL Foundation, "https://www.openssl.org/," 2002.

[11] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Lecture Notes in Computer Science*, 1997.

[12] K. Murdock, D. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," *41st IEEE Symposium on Security and Privacy*, 2020.

[13] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *41st International Symposium on Computer Architecture*, 2014.

[14] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," *CoRR*, 2015.

[15] Intel, *Software Guard Extensions Prog. Reference*. 2014.

[16] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," *ACM Conference on Computer and Communications Security*, 2019.

[17] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0LTpwn: Attacking x86 Processor Integrity from Software," *CoRR*, 2019.

[18] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "FPGAhammer : Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[19] H. Seo, T. Park, J. Ji, and H. Kim, "Lightweight Fault Attack Resistance in Software Using Intra-instruction Redundancy, Revisited," in *Lecture Notes in Computer Science*, 2018.

[20] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, may 2019.