

A practical approach to Side Channel Attacks and Fault Injection

Étienne Richard
GCC-ENSIBS
Vannes, France
e2200238@etud.univ-ubs.fr

Matthieu CIZDIEL
GCC-ENSIBS
Vannes, France
e2201238@univ-ubs.fr

Clément CARDIN
GCC-ENSIBS
Vannes, France
e2201215@univ-ubs.fr

Antoine BOURY--LE-STRAT
GCC-ENSIBS
Vannes, France
e2403267@etud.univ-ubs.fr

Abstract—Energy consumption, electromagnetic radiation, and execution time are physical data exploited by what are known as side-channel attacks in order to extract sensitive information or disrupt the normal functioning of an embedded device. As part of the CSAW Embedded Security Challenge, we put several of these attacks into practice, including power and timing analysis, and here we present the resolution methods and experimental results. This led us to propose a reflection, illustrated by proofs of concept, on the integration of machine learning techniques to improve the accuracy and effectiveness of side-channel attacks.

Index Terms—ChipWhisperer, Side-Channel Attack, Power Analysis, Fault Injection, Timing Analysis, Artificial Intelligence

I. INTRODUCTION

In modern cybersecurity, hardware attacks have become a critical concern as they bypass traditional software protections by directly exploiting the physical properties of embedded systems. Among these, Side-Channel Analysis (SCA) and fault-injection (FI) techniques stand out for their ability to extract or alter sensitive information without breaking cryptographic algorithms themselves. As connected and resource-constrained devices proliferate, understanding and mitigating such physical attacks has become an essential skill for security researchers and engineers alike.

Within this broader context, the CSAW Embedded Security Challenge provides a unique opportunity to explore these techniques in a controlled, hands-on environment. By engaging with realistic attack scenarios, participants move beyond theory to apply practical methods of hardware exploitation and defense.

In this work, we investigate several SCA and FI challenges proposed during the CSAW Embedded Security Challenge finals using a ChipWhisperer Nano (CW-Nano) platform and an STM32F0 target. Our approach focuses on identifying representative categories of passive and active attacks, analyzing the mechanisms that make them effective, exploit the weakness using the ChipWhisperer API, and discussing mitigation strategies and design principles that can strengthen embedded systems against such threats.

II. METHODOLOGY

To seek vulnerabilities in the different challenges, we used a well-defined pattern. First, we searched the code for points of interest we could identify. These portions are easily recognized by calls to the functions `trigger_high` and `trigger_low`, which are essential for the proper capture operation of the board. These functions send a signal the board can use to trigger routines and later analyze parameters or inject faults. Once our Point of Interest (POI) was identified, we looked for a way to reach that part of the code. For some complex programs, we employed an SMT solver (Z3Prover) to automatically derive the symbolic expressions of intermediate values. While simple puzzles could be handled manually, certain constructions—such as two rounds of ChaCha20—quickly became too intricate to model by hand. Having determined where and how to proceed, we asked what the program actually does and what role the POI plays in the system context. This reverse-engineering step was done by hand thanks to the short length of the programs and the fact that we had part of the C code used in the firmware. In more complex situations, we would use reverse-engineering frameworks such as Ghidra or IDA. Finally, we considered the Device Under Test's (DUT) interaction with the physical world. This step aims to determine which types of SCA we can use to retrieve data or to divert the program from its original flow by performing FI. Once these four questions were answered, we had a clear idea of what to do to solve the puzzles and which methodology to follow for each challenge.

III. TIMING ANALYSIS

Although performing timing analysis on the provided platform is highly challenging, we investigated how a timing-based approach could nevertheless address some of the challenge scenarios. To illustrate the issue, consider the following code from `Gatekeeper` function:

```

for (int i = 0; i < password_len; i++)
{
    if (user_input[i] == password[i]) {
        matched_chars++;
        for (volatile uint32_t j = 0; j <
            (uint32_t)(5000 / (c+1)) - i*125; j++)
            // Highly power consuming loop
        } else {
            break;
        }
    }
}

```

From a timing viewpoint, this snippet exhibits a clear vulnerability: an attempt whose first character differs from the password completes noticeably faster than an attempt where the first 12 characters match. The same flow can appear in contexts where a successful authentication triggers further actions.

One plausible hypothesis is that the for loop in the example is functionally analogous to an authentication routine (e.g., session setup). In that case the loop produces both a power consumption pattern and a timing signature that can be correlated with the number of matching characters, making it possible to recover the password. Therefore, we argue that power-consumption patterns should be correlated with execution time to reveal such leaks.

We used these correlations to obtain the flag in both :

- Dark Gatekeeper – ESC{J0lt_Th3_G473},
- Gatekeeper 1 – gk1{l0g1npwn}

challenges. However, this approach failed on Gatekeeper 2, as the “value” of the challenge (represented by the `c` variable in the above listing) is bigger than in the first challenge. So, it reduce the number of iteration in the loop when the character is correct. We manage to distinguish the two first characters of the flag : `gk2{7r` but not more.

To mitigate this leakage, the implementation must perform password verification in constant time. Removing the for loop that creates a timing-dependent delay is insufficient: while this change may make remote timing attacks harder when measurements are noisy, an attacker using power analysis—specifically simple power analysis, as described in a later section of this paper—can still recover the password. A more robust countermeasure is to use a constant-time function to compare strings. For example, MbedTLS supplies such an implementation: [in the following file](#).

Another challenge involving timing attacks was Ghost Blood, this challenge involve a lightweight version of the ChaCha20 stream cipher. Indeed, all word are represented over 16 bits instead of 32 bits in the Bernstein’s version [1]. The “nothing up my sleeve” numbers have also been modified; however, analyzing whether this modification introduces a backdoor is out of scope.

Let $C_i (0 \leq i < 4)$ being the nothing up my sleeve numbers of this lightweight ChaCha20, $N_i (0 \leq i < 4)$ being a fixed known nonce and $K_i (0 \leq i < 7)$ being the secret key. Note that all of these C_i, N_i and K_i are in $\mathbb{Z}/2^{16}\mathbb{Z}$.

Here is the initial state (i.e. the array named `x` in the original C code):

C_0	K_0	K_1	K_2
K_3	C_1	N_0	N_1
N_2	N_3	C_2	K_4
K_5	K_6	K_7	C_3

The C implementation given as a challenge do the standard Add Rotate Xor (ARX) operations on this state, with the standard number of round (10×2), i.e. :

- Repeat 10 times :
 - Quarter Round in column.
 - Quarter Round in row.
 - Add the initial state with the transformed state to avoid all the operations from being invertible.
- However, we can notice two major changes :
- Each Quarter Round use 4 rotations values, the user *can choose them*.
 - The ROTL function (i.e. left rotation) is not constant time, indeed :

```

uint16_t ROTL(uint16_t a, uint8_t b)
{
    uint16_t ret ;
    rotls++ ;
    if (rotls == thresh)
    { trigger_high() ; }

    if(b > 16){ return a ; }
    else if (a >= (1<<(16-b)))
    {
        ret = ((a) << (b)) | ((a) >> (16 -
        (b))) ;
    }
    else
    {
        ret = (a) << (b) ;
    }
    if (rotls == thresh)
    { trigger_low() ; }
    return ret ;
}

```

An attacker—who is able to measure the full execution time of multiple call to ROTL —can distinguish whether the execution flow went to : the first, the second or the third if / else statement. This is because the code try to optimize “the right rotation by b ” for integer less than 2^{16-b} while a correct implementation use CPU instruction (e.g. the ROL instruction on x86) or constant-time implementation when using architecture without dedicated instruction, e.g. see: [MbedTLS implementation](#).

In the context of ChaCha20 the ROTL function takes, during the first round, the sum of : a piece of the key (i.e. a K_i , true for all i), and a known constant (can be both C_i or N_i , also true for all i). Based on the timing leak of the ROTL function, the attacker can distinguish if the sum was greater or less than 2^{16-b} . Moreover, the attacker can control b , thus the attacker can iterate over all possible value for b

before seeing of change of timing. Here is aaaaaan example : In the first round, the first Quarter Round is called with the following arguments: (C_0, K_3, N_2, K_5) , thus, the first call to ROTL has the sum $K_5 + C_0 \bmod 2^{16}$ as an argument. The attacker can then iterate over all possible b to detect when a change of timing will occur, if the change of timing appear between $b = 2$ and $b = 3$ we can deduce : $2^{12} < K_5 + C_0 < 2^{14}$ which implies $2^{12} - C_0 < K_5 < 2^{14} - C_0$. Repeating this method for all call to Quarter Round of the first round, the attacker can bound, between two successive powers of two, each piece of the key (i.e. all K_i 's, $0 \leq i < 8$).

Assuming the key was randomly generated, without any bias over an uniform distribution, the average of the values is $\frac{2^{16}}{2} = 2^{15}$ thus reducing the brute force from 128 bits to 120 bits.

We can take this further: by examining the values of subsequent rounds, we can combine more inequalities with others to further reduce the search space (by intersecting inequalities). We implemented a Z3Prover version of the lightweight ChaCha20 and added constraints based on timing leak results. However, Z3Prover was unable to find a solution to the SMT we provided, even though the possible range for each part of the key had been reduced.

In this first attempt, we used a very weak SCA—timing analysis—which did not sufficiently reduce the key's entropy. We will discuss a second attempt using a more powerful SCA in the CPA section.

IV. POWER ANALYSIS

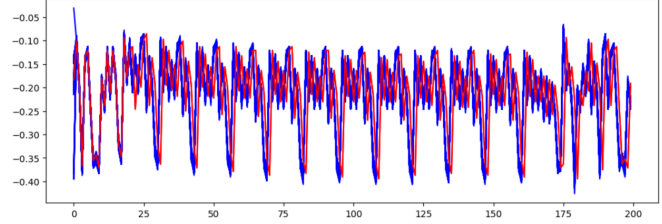
A. Simple Power Analysis

Simple Power Analysis (SPA), as explained in [2], is the most accessible form of power analysis attacks, so we start with it. The goal of this type of attack is to relate the device's power consumption to the operations performed by the DUT. In our case, we analyze consumption during character-comparison operations and correlate it with the branches taken by the program during execution. Our case study is the second part of the GateKeeper challenge—that we didn't manage to solve using timing analysis—which can be expressed in the following pseudocode:

```
for (int i = 0; i < password_len; i++)
{
    if (user_input[i] == password[i]) {
        matched_chars++;
        for (volatile uint32_t j = 0; j <
            (uint32_t)(5000 / (c+1)) - i*125; j++)
            // Highly power consuming loop
        } else {
            break;
        }
    }
}
```

It is clear that when a character in `user_input` matches the character in `password`, a costly loop is executed, resulting in a high power consumption. This consumption can be used

to distinguish a correct guess from an incorrect one, which makes recovering the password straightforward.



In the example above, the region highlighted in red shows an anomalously high power usage (observed between sample 10 and sample 50). We can exploit this observation to build an automated attack that recovers password characters by analyzing power consumption. As stated in the section III, there might be links between power consumption and time, link we exploited in the DarkGateKeeper challenge to retrieve the flag `ESC{J0lt_Th3_G473}` as the evaluation loop ends faster if an invalid character is provided.

Procedure used:

- 1) Capture a reference trace (`ref_trace`) — the power trace produced by a character that is not in the password. In our experiment, the password consisted of 12 characters, each chosen from uppercase letters, lowercase letters, and digits, so we used the out-of-charset character “?” as the reference.
- 2) For each candidate character in the charset, capture its trace and compute the difference with the reference trace. If the difference is small, the tested character is likely incorrect; if the difference is large, it is likely the correct character.

We can automate this process with the following pseudocode:

```
ref_trace = capture_trace("?")
for char in charset:
    trace = capture_trace(char)
    diff = sum(abs(ref_trace - trace))
    # the char with the largest diff as
    the best guess
```

Repeat this procedure for every position of the password to reconstruct the full string and retrieve the flag. Although this method succeeded for Dark GateKeeper, we were unable to reproduce it on GateKeeper — Part 2. The capture buffer was too small to hold the complete trace; additionally, the CW-Nano platform does not allow applying an offset prior to capture (see: <https://chipwhisperer.readthedocs.io/en/v6.0.0b/Capture/ChipWhisperer-Nano.html>), so we could not recover the full flag.

A Simple power analysis is also present in the Echoes of Chaos challenge in a data-dependent merge routine used during sorting. In the `clyde()` merge function a conditional comparison `if (casper[i] <= wendy[j])` is immediately followed by a data-dependent delay loop (`for(volatile del = 0; del < 100 + i; del++) ;`), producing a reproducible timing and power signature

that correlates directly with the outcome of each comparison. However, our attempts to implement this vulnerability have all failed.

To defend against this class of attacks, a simple mitigation is to remove the high-power-consuming loop that produces an easily analyzable trace; but, as discussed in previous section this is not sufficient and a constant time implementation shall be used. In practice, firmware—or hardware-level counter-measures—for example: introducing controlled consumption noise, or applying alignment mitigations can also make the attack substantially harder to carry out.

B. Correlation Power Analysis

Correlation Power Analysis (CPA) is an advanced form of power analysis. The role of CPA is to link the power consumption of the DUT to the sensitive information that is being manipulated by the device. To do so, we can use different leakage models — from the least significant bit (LSB) to the most significant bit (MSB) — but the most effective is usually the Hamming Weight model. This model assumes that there is a correlation between the binary representation of the sensitive information and the power consumption. This assumption is relevant because a logical one is often associated with a higher voltage while a logical zero is associated with a low or zero voltage. The use of MSB or LSB can still be relevant, but we lose a great deal of information by keeping only one bit of an eight-bit variable.

In our case we analyze the consumption during the execution of the XOR loop of the Hyperspace Jump Drive and Alchemist Infuser challenges. Here is the loop of Hyperspace :

```
trigger_high();

volatile uint8_t* secret_bytes =
(uint8_t*)secret_ignition_sequence;
int total_bytes =
sizeof(secret_ignition_sequence);
volatile uint8_t temp_result ;

for (int i = 0; i < total_bytes; i++) {
    temp_result = inversion_mask ^
secret_bytes[i];
    simple_delay_separator();
}
total_bytes += temp_result ;

trigger_low();
```

The *inversion_mask* variable is defined by the user, so we can send all possible values for *inversion_mask* and capture the power consumption during the XOR loop. The consumption traces that have been captured reveal a pattern repeated twelve times, corresponding to the length of the key we want to retrieve. The same key is used across the entire sequence, so we must conduct twelve CPAs on the twelve patterns found in the trace. In our case, when attacking the XOR phase

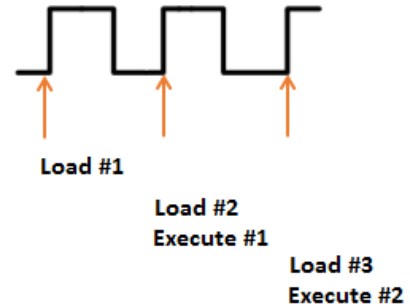
of the target we make 256 assumptions for the key byte value, compute the XOR result of each assumption with the *inversion_mask*, and then compute the correlation between the Hamming Weight of the result and the power consumption at all trace points. The correct value should show a higher correlation. However, when attacking the XOR operation we can encounter difficulties in retrieving the correct key because XOR is a linear function.

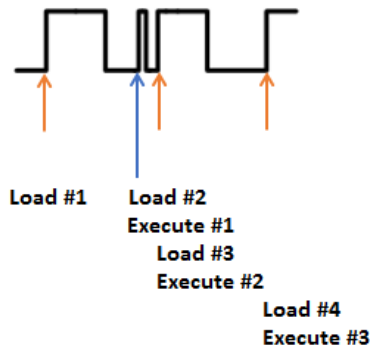
V. FAULT INJECTION

Since the beginning of this paper we have considered only passive side-channel attacks. By “passive” we refer to techniques that do not directly perturb the execution of the target program but instead recover secrets by analyzing information leaked through the device’s physical side channels (power consumption, electromagnetic emissions, timing, etc.). In the following section we address a class of active attacks in which faults are deliberately induced in the device to cause deviations from the intended control flow.

We focus here on electromagnetic fault injection (EMFI). In EMFI, an external disturbance — typically in the form of pulsed or continuous electromagnetic fields — is used to induce transient faults in the DUT. Such faults can affect storage elements or logic (for example, transiently corrupting register contents or altering branch outcomes), and thereby modify program behavior in ways that may be exploitable. Because the CW-Nano has limited capability for EM-based injections, we do not present experimental EMFI results with this platform; nonetheless we discuss EMFI conceptually because it targets memory and logic elements that are not always affected by other fault-injection methods, and thus raises distinct threat and mitigation considerations.

The second category of active fault injection targets the timing subsystem of the device, specifically its clock. Every processor synchronizes its operations to an internal clock that determines the timing and frequency of instruction execution. If the clock signal is momentarily disrupted or shortened, the processor may not allocate sufficient time for an instruction to complete. As a result, the instruction can be skipped or executed incorrectly, leading to deviations in the program’s control flow.





In the simplified timing diagram above, the Load#2/Execute#1 stage is skipped because Load#3/Execute#2 is triggered before the processor has completed the previous operation. This illustrates how a brief clock perturbation—often called a clock glitch—can directly alter instruction sequencing.

Within the context of our experiments and the CW-Nano platform, the generation of such glitches is controlled through the `trigger_high()` and `trigger_low()` functions. By adjusting the offset between these two signals, one can define the precise instant at which the glitch is introduced, as well as the repetition rate of the fault event. This timing control is central to reproducing deterministic and analyzable glitch effects during experimentation.

We take the Critical Calculation challenge as a representative case study to illustrate the principles of clock-glitch fault injection. This example constitutes a textbook instance of how precise timing perturbations can alter program execution and thereby expose logical vulnerabilities in embedded systems.

```
volatile uint32_t cnt=0;
trigger_high();
for(i = 0; i < 100; i++){
    for(j = 0; j < 40; j++){
        cnt += 2;
    }
}
trigger_low();

if (cnt != 8000) {
    simpleserial_put('r', flag_len,
flag);
}
```

The example illustrates that, in the presence of a clock glitch that effectively implements a `cnt += 2` update, the final authentication condition may fail to validate and the program can consequently reveal the flag `cc1{C0RRUPT3D_C4LCUL4T10N}` with an offset of 418 and a repeat of 12. The primary challenge is achieving adequate temporal alignment of the perturbation: the fault must occur within a tightly constrained execution window. This requires fine-grained control of the glitch timing and repetition parameters exposed by the measurement platform.

This class of attack is not unique to the present challenge. The Dark Gatekeeper task can be addressed using the same principle, but with substantially higher precision requirements. While the previous challenge tolerated the omission of one instruction among roughly 4,000, the Dark Gatekeeper scenario demands perturbation of a single, specific instruction (a single-bit or single-value shift). The C source for that challenge is given below.

```
char access_granted = 1;
for (int i = 0; i < KEY_LENGTH; i++) {
    if (incoming_key[i] != master_key[i])
    {
        access_granted = 0 ;
    }
}

if (access_granted) {
    simpleserial_put('r',
RESPONSE_LENGTH, flag);
}
else {
    simpleserial_put('r',
RESPONSE_LENGTH, msg_failure);
}
```

that we could convert in the following assembly code:

```
loop:
cmp rbx,rcx
jne invalid
add rbx, 8
add rcx, 8
cmp rax, KEY_LENGTH
jne loop
jmp out

invalid:
mov rdx, [rsp-24]
sub rdx, 1
jmp loop

out:
mov rdx, [rsp-24]
cmp rdx, 1
je print_flag

print_flag:
call print_flag
```

In the example above, a successful glitch of the `sub rdx, 1` instruction would cause the program to print the flag. Although this attack vector is feasible in principle, we did not demonstrate it experimentally due to the limited timing and spatial precision of the CW-Nano platform. Nevertheless, we consider it important to describe this alternative exploitation path, because it highlights how a single transient fault in a loop counter or comparison can bypass authentication logic.

In the example above, a successful glitch of the `sub rdx, 1` instruction would cause the program to print the flag.

Another example can be found in the `Echoes of Chaos` challenge, where we could try to reverse the result of this comparison (`correct == 0`). However, it is not found between the `trigger_high` and `low` calls.

Although this attack vector is feasible in principle, we did not demonstrate it experimentally with one comparison, due to the limited timing and spatial precision of the CW-Nano platform. Nevertheless, we consider it important to describe this alternative exploitation path, because it highlights how a single transient fault in a loop counter or comparison can bypass authentication logic.

VI. AUTOMATION AND INTEGRATION OF ARTIFICIAL INTELLIGENCE

Some tasks can be automated using Artificial Intelligence (AI). For example, in the `DarkGate Keeper` challenge a simple power analysis showed that, for each key byte, the correct guess produced a distinct power-consumption signature. We used the machine-learning algorithm Isolation Forest to detect anomalies in the dataset. In the `Jump Drive` challenge, points of interest within the consumption trace were identified by computing the signal-to-noise ratio (SNR). We detected twelve SNR peaks and partitioned the traces into twelve segments corresponding to these peaks; this step was automated using the `scipy.signal.find_peaks` function. AI can therefore help automate attack workflows, but it can also be employed as an offensive tool. In the `JumpDrive` and `AlchemistInfuser` challenges we performed correlation power analysis (CPA). CPA can be replaced by non-profiled deep-learning attacks such as MOC or MOR [3]. We evaluated these methods but obtained poor results; we attribute this outcome to the linearity of the targeted XOR operation and expect better performance when attacking nonlinear primitives.

VII. UNINTENDED SOLUTIONS

Designing reliability-oriented code in an embedded context does not, on its own, guarantee system robustness: if the underlying hardware infrastructure does not have the components necessary for this reliability, the very principle of software robustness is compromised or undermined. The series of challenges `Sorters Song` illustrate well this idea, by implementing time based randomization element. Because the CW-Nano lacks a reliable entropy source, the firmware seeds its PRNG with the current time (i.e. the result of `time(NULL)`). Not only is this poor practice—time is an inadequate source of entropy—but it is exacerbated on the CW-Nano because the platform provides no clock: `time(NULL)` therefore always fails (i.e. returns the `-1` value), producing a constant seed to the `srand` function. As a consequence the PRNG and the generation of both secret arrays are fully deterministic. This permitted us to offline-reconstruct the arrays using the same algorithms present in the supplied C code. This means that these elements can be reliably regenerated from a firmware without

requiring explicit storage of the generated data. We can then input the tables generated back into the challenges to get both flags `ss1{y0u_g0t_it_br0!}` and `ss2{!AEGILOPS_chimps}` with following inputs:

- `0x070c2b3439425068717c8893acb1db,`
- `0x66038a2422400149e14b5c699c7089878790f694729a3aa665b316c9dbd4.`

VIII. CONCLUSION

The experimental approach of side-channel analysis applied to the ChipWhisperer-Nano platform during the CSAW Embedded Security Challenge allows us to identify three key conclusions. First, it confirms the feasibility of extracting sensitive information from simple power and execution time measurements, highlighting the inherent vulnerability of low-cost embedded devices. Second, the limited precision of the CW-Nano’s Analog-to-Digital Converter (ADC) constrains the applicability of advanced timing or FI based-attacks, underlining the importance of hardware capabilities in side-channel research. Finally, this work opens the way for future investigations into the use of machine learning techniques to strengthen the correlation between traces and secret keys, thereby improving the efficiency and accuracy of next-generation attacks.

APPENDIX 1 - TABLE OF FLAGS.

Challenge	Flag	Points
GateKeeper 1	gk1{!0g!npwn}	100
GateKeeper 2	✗	150
Sorters Song 1	ss1{y0u_g0t_it_br0!}	150
Sorters Song 2	ss2{!AEGILOPS_chimps}	200
Critical Calculation	cc1{C0RRUPT3D_C4LCUL4T10N}	100
Dark GateKeeper	ESC{J0lt_Th3_G473}	200
Hyperspace Jump Drive	b64{c3RhcmR1c3Q=}	200
Ghost Blood	✗	200
Alchemist Infuser	a1c{Wh1teDragonT}	200
Echoes of Chaos	✗	200

REFERENCES

- [1] Y. Nir and A. Langley, “ChaCha20 and Poly1305 for IETF Protocols.” [Online]. Available: <https://www.rfc-editor.org/rfc/rfc7539>
- [2] P. C. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” Technical Report, 1998. [Online]. Available: <https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf>
- [3] N.-T. Do, P.-C. Le, V.-P. Hoang, V.-S. Doan, H. G. Nguyen, and C.-K. Pham, “MO-DLSCA: Deep Learning Based Non-profiled Side Channel Analysis Using Multi-output Neural Networks,” in *2022 International Conference on Advanced Technologies for Communications (ATC)*, 2022, pp. 245–250. doi: 10.1109/ATC55345.2022.9943024.