# Cycle-Accurate Power Side-Channel Analysis Using the ChipWhisperer: a Case Study on Gaussian Sampling

Nils Wisiol[1][0000−0003−2606−614X], Patrick Gersch[1][0000−0001−8898−7353], and Jean-Pierre Seifert[1,2]

[1] Security in Telecommunications, Department of Electrical Engineering and Computer Science, TU Berlin, D-10587 Berlin, Germany
[2] FhG SIT, D-64295 Darmstadt, Germany

**Abstract.** This paper presents an approach to uncover and analyze power side-channel leakages on a processor cycle level precision. By carefully designing and evaluating the measurement setup, accurate trace timing is enabled, which is used to overlay the trace with the corresponding assembly code. This methodology allows to expose the sources of leakage on a processor cycle scale, which allows for evaluating new implementations. It also exposes that the default ChipWhisperer configuration for STM32F4 targets used in prior work includes wait cycles that are rarely used in real-world applications, but affect power side-channel leakage.

As an application for our setup, we target the widely used Sign-Flip function of Gaussian sampling code used in multiple Post-Quantum Key-Exchange Mechanisms and Signature schemes. We propose new implementations for the Sign-Flip function based on our analysis on the original implementation and further evaluate their leakage.

Our findings allow the conclusion that unmasked cryptographic implementations of schemes based on Gaussian random numbers for STM32F4 cannot be secure against power side-channel, and that masking just the Gaussian sampler is not a viable option.

**Keywords:** Power side-channel analysis · ChipWhisperer · Processor cycle level analysis · Gaussian sampler · Sign-Flip · GALACTICS · FALCON · FrodoKEM

## 1 Introduction

The US National Institute of Standards and Technology (NIST) started the Post-Quantum Cryptography (PQC, [23]) process in November 2017 to select quantum-resistant key-exchange mechanisms and signature schemes as a preemptive response to the emergence of large-scale quantum computers. As part of this process, NIST also considers the resistance to side-channel attacks an important criterion for algorithm selection; submitters are encouraged to provide implementations of their schemes optimized for microcontrollers and FPGAs. The ARM Cortex-M4 turned out as a baseline for many performance comparisons, because of projects like [15] and works based on it [16, 31]. Therefore, this paper will also focus on this specific microcontroller.

The Round 3 finalist FALCON [11], Round 3 alternative candidate FrodoKEM [2] and BLISS [8] (the constant-time GALACTICS implementation [9]), one of the earliest post-quantum schemes, all make use of Gaussian sampling. In FALCON and BLISS, Gaussian random numbers are required for signature generation; in FrodoKEM, they are required for key encapsulation and decapsulation. All of their Gaussian sampler implementations are based on a Sign-Flip function. Multiple attacks on BLISS implementations are based on side-channel leakage that revealed information about the signs of the Gaussian samples used [33, 21]. In this paper, we turn our attention to leakage observable via the power side-channel, which is a typical attack scenario against embedded and IoT devices. The ChipWhisperer [29] provides a complete side-channel lab setup to enable rapid analysis.

*Contributions.*

- We show that the Gaussian sampling routines of the GALACTICS implementation of BLISS as well as FALCON and FrodoKEM on the STM32F4 all leak information about the generated samples. We propose several new variants of these implementations, and show that the leakage can be reduced. However, based on our findings in this case study, we argue that the number representation format of the STM32F4 results in side-channel leakage that cannot be mitigated by modifications to the sampling routine.
- Using the ChipWhisperer, we present a novel approach for in-depth power analysis on the processor cycle level. We demonstrate a detailed breakdown of the required setup and condition to execute cycle level analysis on the STM32F4. The core of this cycle level analysis is to accurately overlay the traces with the corresponding assembly instructions.
- We demonstrate a flaw in the latest release version of the ChipWhisperer, potentially affecting previous power analyses based on the ChipWhisperer STM32F4 setup.

*Organization of this paper.* In Sec. 3, we describe our measurement and analysis setup in detail and argue for its validity. Furthermore, we argue that the current default configuration of the ChipWhisperer STM32F4 target board does

not reflect real-world scenarios, which may influence the validity of previous power side-channel analyses based on the ChipWhisperer setup (Sec. 4). Afterwards, in Sec. 5, we apply our analysis setup to the Sign-Flip subroutines used by GALACTICS, FALCON and FrodoKEM and demonstrate that information about the sampled sign is leaked. This section also covers advancements of the Sign-Flip implementation that are able to reduce the amount of leakage, but cannot fully mitigate it. Finally, we argue that the number representation format of the STM32F4 makes it impossible to implement a side-channel resistant Sign-Flip function. We discuss our findings in Sec. 6.

*Related Work.* Many works use the ChipWhisperer platform to evaluate implementation for their power side-channel resistance [30, 14, 21].

The implementation of the PQC candidate SIKE for the ARM Cortex-M4 was successfully attacked by Genêt and Kalułterović using the ChipWhisperer through single-trace power analysis [13]. A masked implementation of Saber for the Cortex-M4 was also successfully attacked using the ChipWhisperer. The leakage is thought to be related to the Hamming distance between the old and new values of pipeline registers [28]. Askeland and Rønjom attack the NTRU implementation with a ChipWhisperer. They identify the Hamming weight of secrets as a major reason for power side-channel leakage in implementations [5].

Leakage simulation aims to construct the power trace for a given set of assembly instruction. ELMO is a prominent leakage simulator for the Cortex-M0/M4 processor. Even though such simulators can give a good estimation of the power leakage, only experiments on real hardware can confirm their accuracy [22].

Tibouchi and Wallet present a timing attack on the Sign-Flip in BLISS, resulting in a full exposure of the signing key [33]. A power side-channel leakage in the Sign-Flip is used for two attacks against the Gaussian sampler resulting in full key recovery in Marzougui et al. [21].

## 2 Preliminaries

*Notation.* Binary strings are represented as a string with a bit symbol pre-pended, e.g. b'1000 1010' and hexadecimal numbers with a '0x' pre-pended, e.g. 0x1234. Numbering is in $\mathrm{LSB}_0$ scheme, meaning it starts at zero for the least significant bit (LSB). The LSB is the right-most bit. The most significant bit (MSB) is thus the highest-order bit, which is the left-most bit.

*Negative number representation.* In writing, negative numbers are usually represented by a minus sign. However, in digital circuits, different ways to represent negative numbers in binary exist. First, the *sign-magnitude* represents negative numbers using a single sign bit, often the most-significant bit. Second, the *ones' complement* representation performs a bitwise NOT operation to get from positive to negative numbers. Third, the *two's complement* representation transforms from positive to negative numbers by applying a bitwise NOT operation and adding one afterwards. The preferred choice for modern processors is the

*two's complement*, as it allows for efficient arithmetic operations and has only one representation of zero.

In Tab. 1, we give examples of 16-bit two's complement number representation along with their Hamming weight. The Hamming weight is a metric for the number of 1's in a binary string. Especially the difference in Hamming weight between positive and negative numbers should be noted.

| Number | 16-bit two's complement | Hamming weight |
|---|---|---|
| -4 | b'11111111 11111100' | 14 |
| -3 | b'11111111 11111101' | 15 |
| -2 | b'11111111 11111110' | 15 |
| -1 | b'11111111 11111111' | 16 |
| 0 | b'00000000 00000000' | 0 |
| 1 | b'00000000 00000001' | 1 |
| 2 | b'00000000 00000010' | 1 |
| 3 | b'00000000 00000011' | 2 |
| 4 | b'00000000 00000100' | 1 |

Table 1: *Two's complement* numbers from -4 to 4 in 16-bit representation

*Sign-Flip.* Lattice-based cryptography such as FALCON [11], FrodoKEM [2], and BLISS [8] requires random numbers $X$ drawn from discrete Gaussian distribution with mean $\mu = 0$ and variance $\sigma^2$, i.e.

$$X \sim \lceil \mathcal{N}(\mu, \sigma) \rceil.$$

$$\rho_{\mu,\sigma}(x) = e^{-\frac{|x-\mu|^2}{2\sigma^2}} \tag{1}$$

$$D_{\mathbb{Z},\mu,\sigma}(x) = \frac{\rho_{\mu,\sigma}(x)}{\sum_{z\in\mathbb{Z}} \rho_{\mu,\sigma}(z)} = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{|x-\mu|^2}{2\sigma^2}} \tag{2}$$

(Note that the definitions for the distributions vary slightly across the different schemes.)

Several implementations for generation of random Gaussian numbers exist. Implementations based on the cumulative distribution table (CDT) have been shown to achieve the highest throughput and the smallest resource utilization [17]. Such implementations contain a hard-coded, precomputed table of the cumulative distribution table. Taking advantage of the symmetry of the distribution about the mean, the table size can be reduce by half to reduce resource consumption by only storing values for positive $x$. In this case, a sample is first drawn from distribution $|\lceil \mathcal{N}(\mu, \sigma) \rceil|$. In a second step, the Sign-Flip function

$$\text{Sign-Flip}(x, c) = \begin{cases} -x & c = 0 \mod 2, \\ x & \text{otherwise}, \end{cases} \tag{3}$$

is applied to determine the sign, resulting in a sample from $\lceil \mathcal{N}(\mu, \sigma) \rceil$. In Eq. 3 the sample from the distribution $|\lceil \mathcal{N}(\mu, \sigma) \rceil|$ is $x$ and $c$ is a sample from a uniformly random distribution. The side-channel security of CDT samples has previously been studied [18].

The Sign-Flip function is used by GALACTICS [9], FrodoKEM [2] and FALCON [11] as part of their discrete CDT Gaussian sampling procedure. Tab. 2 shows the output for a chosen set of example inputs, represented in two's complement.

| Input $x$ | if $c = 0 \pmod 2$ | if $c = 1 \pmod 2$ |
|---|---|---|
| 0 | b'00000000 00000000' (0) | b'00000000 00000000' (0) |
| 1 | b'11111111 11111111' (-1) | b'00000000 00000001' (1) |
| 2 | b'11111111 11111110' (-2) | b'00000000 00000010' (2) |
| 3 | b'11111111 11111101' (-3) | b'00000000 00000011' (3) |

Table 2: Sign-Flip function value table. Output is shown in decimal and two's complement representation.

*Side-channel analysis.* Kocher et al. [19] have shown that *power side-channel attack* can reveal information about the internal state of hardware that processes cryptographic algorithms, including the secret key, by measuring and analyzing the power consumed by the hardware over time (*power trace*). Subsequently, Chari et al. [7] proposed *template attacks*, where the attacker has full control to a device identical to the device under attack. This device can be used to build a model of power-consumption behavior in dependence of the internal state, which afterwards is deployed to deduce the internal state from the measured power consumption. Lerman et al. [20] proposed to use machine learning techniques for this modeling. This work also uses a machine learning based template attack; for creating the model, we use a multilayer perceptron.

## 3 Measurement Setup

This section details how we collected power traces using the ChipWhisperer for this work. It focuses on high temporal precision to enable matching of measured power consumption of the hardware to the instructions executed at the given time. Our complete analysis was conducted using the ChipWhisperer-Lite and an UFO target board with an STM32F4 target mounted. In the standard ChipWhisperer configuration that we use, the power measurement is started using a dedicated GPIO pin of the target board.

### 3.1 Target: STM32F4

The STM32F4 target (CW308T-STM32F, [27]) was mounted on a ChipWhisperer CW308 UFO target board (NAE-CW308-04, [26]). The UFO board includes an oscillator, power supply and on-board LC low-pass filter for the connected target board. A 7.3728 MHz crystal was used for all experiments and set as default clock source. Therefore, each processor clock cycle takes approximately 135ns. Based on the STM32F4, the STM32F405RGT6 [32] target board was designed to conform with the UFO target board requirements. The power consumption of the STM32F4 target is measured via a voltage measurement on a shunt resistor.

The STM32F4 is an ARM Cortex-M4 [3] architecture, implementing the 32-bit *Armv7E-M* instruction set architecture (ISA). The *Armv7E-M* has 16 32-bit registers named *r0-r15*. The Cortex-M4 has a 3-stage pipeline (fetch, decode, execute).

**GPIO Toggle Speed** A GPIO pin from the STM32F4 is used to signal the start and end of the power trace by inserting toggle commands using software. Therefore, the toggle speed of the STM32F4 is of utmost importance as the accuracy of the start and end of the power trace relies on it. If the toggle of a GPIO pin takes longer than one clock cycle of the processor, this delay needs to be considered when matching the power trace against the executed program.

The GPIO toggle speed of the STM32F4 depends on the OSPEEDRy hardware register (b'10'), the capacitive load and the operation voltage ($3.3V$) [32, datasheet p.117]. Depending on the capacitative load, the rise/fall time lies between 4ns and 6ns and thus does certainly not exceed the cycle time of 135ns. The GPIO toggle speed can thus be ignored for the purpose of matching power consumption against executed instructions in our setup.

**Memory Caching & ART Accelerator** Memory caches are important to speed up memory access of the processor, but highly stateful. Memory access times can vary dramatically depending on the current state of the cache, with cache hits being processed much faster than cache misses. To match the executed instructions against the recorded power trace, we turned off all caching on our target board.

We disable the cache as enabling the cache raises the question how to initialize the cache state. As in practical attacks, the cache state depends on prior usage of the device, there is no generally valid answer to this question. For real-time applications, the cache may be turned off even in real-world scenarios.

The used STM32F405RGT does not include a flexible static memory controller (FSMC) [32, datasheet p. 14]. There is also no cache in between the ARM Cortex and the SRAM. All code will be placed on the Flash memory. The Flash memory is accessed from the processor by the adaptive real-time memory (ART) accelerator with a build in cache. Sec. 4 will discuss the ART in greater details.

**Compiler Instruction Re-Ordering** For all experiments, the *arm-none-eabi-gcc* compiler on version 9.2.1 has been used. It supports different compiler optimization flags to improve various performances like memory usage, code size and program speed. Optimization level 3 (*-O3*) is the one which is most often used for NIST PQC candidates [15], resulting in fast performance of the compiled binaries. However, due to instruction re-ordering by the compiler, this may falsify side-channel leakages.

This is a particular problem when using the GPIO-triggering used by the default ChipWhisperer [25] configuration, as instructions for triggering the start and end of the trace could be moved to earlier or later locations in the program.

Using the ChipWhisperer, tracing a function might look like "*trigger_high(); func_under_test(); trigger_low();*". As *trigger_high()* and *trigger_low()* operate on the same hardware registers, the compiler is aware that they depend on each other. The same is not true for the function we are trying to trace. Therefore, the compiler might re-order the *func_under_test()* before or after the triggers. In some situations, the compiler might even inline a *func_under_test()* if the function is relatively short (depending on compiler flags and limits). After inlining the function, the compiler can re-order the inlined instructions with the trigger functions. As a result, the tracing might not capture some instructions of the *func_under_test()*, not trace the function at all, or include leakage caused by code originally intended to run before the start of the trigger or after the intended end of the trigger.

If the compiler supports the function attributes *no_reorder* or *noinline* then these can be used to mitigate the issue. Unfortunately the used *arm-none-eabi-gcc* compiler does not and therefore simply ignores these attributes. The only way to reliably prevent the re-ordering is the *optimize(2)* function attribute to set the optimization level lower for the code section of our *func_under_test*. While instruction reordering is one of the fundamental concepts to enable most optimizations, compilers do not have a model for time, but only for the result of instructions. Thus, reordering only guarantees that the outcome of the program is according to the code, but not intermediate steps [6].

### 3.2 Trace: ChipWhisperer

The ChipWhisperer-Lite (NPCB-CWLITECAP-01 CW1173 [24]) in our experiments collects traces using a high-gain Low Noise Amplifier (LNA) with a 10-bit Analog Digital Converter (ADC). In our setup, the tracing is triggered when the ChipWhisperer detects a rising edge on it's trigger input. I.e., the tracing starts when signal goes from *low* to *high* and stops when switching back from *high* to *low*.

**Trigger Speed** After examining the toggle speed of the GPIO in Sec. 3.1, we will have a look at how accurate the ChipWhisperer-Lite notices the trigger and therefore starts/stops the tracing. Any delay by one or more clock cycle needs to be known to overlay the trace with the assembly code. To evaluate the delay

at which the ChipWhisperer starts tracing after the rising edge on the trigger signal, we ran the code shown in Listing 1.1 on the target board.

After preparation, it uses a single instruction (line 8) to output a rising edge on the GPIO pin that we connected to the ChipWhisperer trigger input. In the instruction immediately after, the GPIO is set to low again.

Listing 1.1: ARM assembly code to toggle the trigger GPIO

```
1  # r11 stores the MMIO address of the trigger GPIO
2  # r10 stores the value to set the trigger GPIO to high
3  # r12 stores the value to set the trigger GPIO to low
4  ldr r11, =0x40020000
5  ldr r10, =0x1000
6  ldr r12, =0x10000000
7  .align 4
8
9  # Using a single 'STR' instruction
10 # to set the trigger GPIO to high
11 str r10, [r11, #24]
12
13 # Pre-calculated address and value in r11 and r12
14 # to set the trigger GPIO to low in one instruction
15 str r12, [r11, #24]
```

A single 'STR' instruction on an STM32F4 takes 2 cycles [4] to execute, but every consecutive 'STR' instruction takes only 1 cycle. We thus expect that the trigger GPIO is high for exactly 1 cycle. We call this assembly version of toggling the trigger GPIO using fixed constants in specific registers *fast_trigger*, as opposed to the ChipWhisperer's standard implementation, which is wrapped in C functions and hence requires multiple cycles to execute.

Our experimental results show that the ChipWhisperer collects a power trace of length 4. The default configuration of the ChipWhisperer collects 4 samples each processor cycle. Therefore, the trigger GPIO is high for exactly 1 cycle according to the ChipWhisperer-Lite trigger speed. In experiments that traced the power consumption of a single instruction, we confirmed that the triggering of the ChipWhisperer does not incur significant delay by comparing the power traces of an instruction with high and low Hamming weight. Also the findings presented in Sec. 5.2 show high correlation between Hamming distance of register updates and power trace, providing evidence that there is no delay introduced by the triggering. We therefore conclude that no extra offset or precaution needs to be taken when working with the ChipWhisperer-Lite trigger mechanism.

### 3.3 Analyze: Overlaying the Trace With Corresponding Assembly Code

To remove a side-channel vulnerability from an implementation, it is required to determine the code location that caused the leakage of secret information. In this

work, we locate leaking code locations by matching the collected power traces against the assembly instructions executed at the time. After leakage locations in the trace and program have been identified, this information can be used to improve attack performance and accuracy (by restricting attention to affected areas) or to harden the implementation against side-channel attacks (by avoid using leaking instructions in the code).

After making sure that the start and end of the power trace is accurate (see Sec. 3.1 and 3.2), no caches are activated (Sec. 3.1) and the instruction order is preserved (Sec. 3.1) a simple mapping between clock cycle in the power trace and instructions in the assembly code can be applied. Each instruction takes a predefined amount of clock cycles according to the microcontroller's documentation [4].

For simplicity, we use this methodology only on branch free code. For code containing loops, we collect the power-trace of a single iteration instead of the full loop or unroll the loop. For code containing function calls, we trace the function separately or inline it.

## 4 ChipWhisperer Firmware

The ChipWhisperer comes with an open-source software package [25] to assist in side-channel analysis. At the time of writing, the latest release is version 5.5.2 from May 5, 2021. The ChipWhisperer firmware for the STM32F4 target prepares the device for side-channel analysis, which includes configuring the ART (adaptive real-time memory) accelerator, which handles memory accesses from the processor to the flash memory. The memory is accessed upon execution of load instructions (D-Code) and to read the next instructions of the program in execution (I-Code).

To access data from flash memory, the address is sent to the ART accelerator, which reads 128-bit of data from memory[3] and caches it. For subsequent memory access, if the requested data is present in the cache, it can be read out without any delays. However, if not, a fixed number of cycles are needed for the ART accelerator to read the Flash memory and provide the data.

The ChipWhisperer 5.2.2 firmware configures the ART with disabled data cache, disabled instruction cache, disabled prefetching, and five wait states latency. This means that after 128-bit of instructions, 5 cycles of waiting are required to read the next 128-bit from memory. 128-bit of instructions can be comprised of 4 instructions of 32-bit (in ARM mode), 8 instructions of 16-bit (in thumb mode), or a mixture of both. We observed the presence of wait states in the length of the power traces that we collected during our experiments.

However, the microcontroller can also be operated without wait states at the voltage and frequency used by the ChipWhisperer platform [32, reference manual p. 80]. Given the significant performance loss caused by the wait states, it is

[3] To the best of our knowledge, the alignment of this process is not documented. However, our experiments provide some evidence that a 4-word alignment is used on our target device.

unlikely that real-world applications would use a configuration with wait states. Accordingly, on 21 September 2021, the ChipWhisperer development branch was updated (commit 5863217) to configure STM32F4 target boards to not apply wait states (0 WS).

However, at the time of writing, this change was not yet contained in any released ChipWhisperer software package; the ART configuration with wait states is default since the introduction of STM32F4 support in ChipWhisperer in March 2017. Consequently, power traces of the STM32F4 target board collected using the ChipWhisperer contain wait states unless the target board ART configuration was changed manually or, after 21 September 2021, the ChipWhisperer development code was used instead of installing the software as described in the documentation [25].

The presence of wait cycles in the execution of the program influences the internal state of the processor, which may influence the leakage of implementations in side-channel analysis. This casts some doubt on the validity of previous side-channel analysis of the STM32F4 on the ChipWhisperer platform.

As a case study on the impact of wait states on side-channel leakage, we compare template attacks on the Sign-Flip function running on the STM32F4 using five wait states (5 WS) and no wait states (0 WS). Marzougui et al. [21] demonstrated that a generic MLP classifier could recover the sampled sign in 99.9% and 100.0% of cases, respectively, when using 5 WS. For direct comparison, we re-ran this experiment with an identical setup (Sec. 3) and machine learning attack (see Sec. 5.1 for details on the machine learning attack) for both the 5 WS and 0 WS configuration. The firmware version with 0 WS results in 98.0% (Sec. 5.1) while the version with 5 WS reaches 99.9%. A detailed breakdown of the classification performance depending on training set size can be seen in Fig. 1.

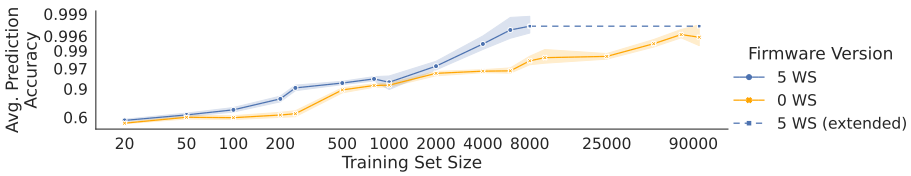


Fig. 1: Classifier comparison for the firmware with 5 WS and 0 WS for different training sizes. Validation set size is fixed at 1000. Each data point shown is based on multiple runs of the attack. The darker colored lines show the mean value, while the lighter areas show the range of accuracies. The accuracy for 5 WS with a training set size of 8000 to 98000 is constant as it already reaches 0.999% at a size of 8000.

While this reduced prediction accuracy, it is still very high and continues to pose a security threat to GALACTICS on the STM32F4. Of the two attacks presented by Marzougui et al. [21, Sec. 6.1, 6.3] that use leakage of the Sign-Flip

function, one will need slightly more samples of the target under attack, and one will remain largely unaffected as predictions are the result of majority voting across a very large population.

## 5 Sign-Flip Analysis

In this section, we will first present three different implementations of the Sign-Flip function as found in three different post-quantum cryptography schemes. Next, we will show that all of them are vulnerable to profiling power analysis attacks using machine learning (see Sec. 5.1).

Subsequently, in Sec. 5.2, we present advanced and modified implementations of the Sign-Flip function and attack using the same profiling attack. We find that, even though some implementations leak less under this attack, none of them is immune to our attack.

Additionally, we present an implementation of a two-sided CDT Gaussian sampler, which avoids the need for a Sign-Flip function at the cost of increasing the CDT table size (Sec. 5.3). Our findings show that also this approach cannot remove significant leakage on the sign of the sampled values.

### 5.1 Power Analysis of Different Implementations

We analyze the Sign-Flip implementations of GALACTICS [10] and of the reference implementations of FrodoKEM [1] and FALCON [12] by subjecting them to identical attacks.

First, for each scheme, we located and isolated the code implementing the Sign-Flip function. While similar, the schemes all use slightly different implementations of the same functionality, as shown below. (Variable names have been changed to match the Sign-Flip$(x, c)$ definition of Eq. 3.)

In GALACTICS, given `uint32` inputs `x` and `c`, Sign-Flip returns `int32` defined by

```
(x & -(c & 1)) ^ (-x & ~(-(c & 1))).
```

In FrodoKEM, given `uint32` inputs `x` and `c`, Sign-Flip returns `int32` defined by

```
(x ^ -c) + c.
```

In FALCON, given `uint16` inputs `x` and `c`, Sign-Flip returns `int16` defined by

```
(-c ^ x) + c.
```

Note that the implementations of FrodoKEM and FALCON differ in the used data types. Note that the `-` and `~` have higher operator precedence than `&`.

Second, to be able to run the identified code snippet independently from the rest of the scheme, we determined the random distribution of values for `x` and `c` in each scheme.

Third, using the identified code and input random distributions, we ran each snippet 10,000 times and recorded input, output, and power traces using the setup described in Sec. 3.

Finally, to run an attack, the recorded data is randomly partitioned in a training set of 9,000 examples and a test set of 1,000 samples. (Fig. 2 displays a subset of collected traces for each scheme.) Using the training set, the sections of the power trace showing large difference across the recorded output sign are identified. Then, an neural network is trained given the identified sections and output sign values. Our neural network uses the *adam* optimizer, a batch size of 200 and one hidden layer with 20 neurons. Using the test set, the prediction accuracy of the trained network is evaluated. While other parameters for the neural network may increase performance or prediction accuracy, we found that these parameters already reach high accuracy and thus reveal sensitive information.

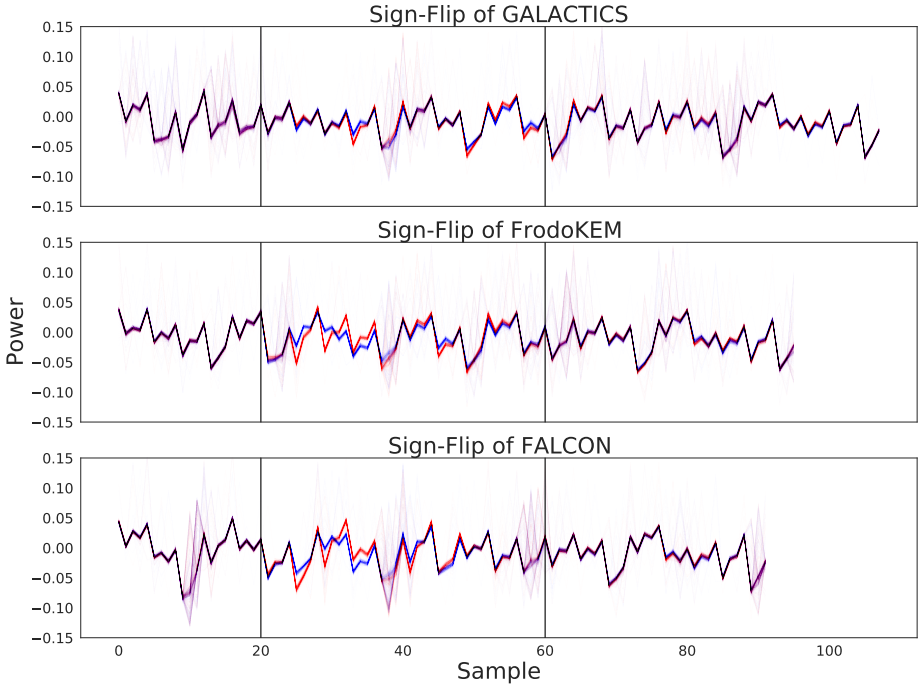


Fig. 2: 1,000 power traces of Sign-Flip implementations. Red and blue lines indicate negative and positive output, respectively. The section between the vertical black lines is used for training the neural network.

We find that for all three schemes, our attack yields near-perfect prediction accuracy of the sampled sign. The Sign-Flip implementation of GALACTICS was previously under attack using a profiling power analysis attack [21]. For GALACTICS, our experiment resulted in a prediction accuracy of 98.0%.

In the case of FALCON and FrodoKEM we observed similar-looking power traces, which is expected due to the similar implementations. We obtained prediction accuracy of 99.7% and 100.0% for FALCON and FrodoKEM, respectively.

The Sign-Flip implementations in FALCON and FrodoKEM are originally not encapsulated in a subroutine. To confirm that extracting the code snippet and running it individually did not influence the leakage significantly, we re-ran our experiment on the full FALCON and FrodoKEM Gaussian sampler. These experiments confirm our findings, with prediction accuracy reaching 97.3% and 100.0%, respectively. We note that the Hamming distance for register updates in FALCON is smaller than in FrodoKEM, as 16 bit values are used, which could explain the difference in prediction accuracy [28].

### 5.2 Analyzing Various New Versions

To harden cryptographic schemes based on Gaussian random numbers against power side-channel analysis, we explore candidate implementations of the Sign-Flip function in this section. To understand the issue in the existing implementation, we matched the power trace against the instruction executed at the given time as described in Sec. 3.3.

As the implementation used by GALACTICS showed the least leakage in above analysis, we chose it as a starting point. It's assembly code is shown in Tab. 3, together with example register values for the cases $c = 0$ and $c = 1 \pmod 2$. The distribution for input value $x$ is a half-normal distribution (positive half) with mean zero and variance $\sigma = \frac{205}{256}$ [9, p. 14], input value $c$ is uniformly distributed in $\{0, \ldots, 255\}$.

| instruction | pseudo-code | computed function | assigned register values if $c = 0 \mod 2$ | otherwise |
|---|---|---|---|---|
| 1. and.w r1, r4, #1 | r1 = r4 & 1 | $= c$ & 1 | 0x00000000 | 0x00000001 |
| 2. subs r4, r1, #1 | r4 = r1 - 1 | $= (c$ & 1) - 1 | 0xFFFFFFFF | 0x00000000 |
| 3. negs r5, r3 | r5 = -r3 | $= -x$ | $-x$ | $-x$ |
| 4. negs r1, r1 | r1 = -r1 | $=$ -($c$ & 1 ) | 0x00000000 | 0xFFFFFFFF |
| 5. and.w r0, r1, r3 | r0 = r1 & r3 | $=$ [-($c$ & 1)] & x | 0x00000000 | $x$ |
| 6. ands r5, r4 | r5 = r5 & r3 | $= -x$ & [-($c$ & 1)] | $-x$ | 0x00000000 |
| 7. eors r5, r0 | r5 = r5 & r0 | $=$ — $\oplus$ ([-($c$ & 1) & $x$) | $-x$ | $x$ |

Table 3: Assembly instruction of the GALACTICS Sign-Flip implementation, displayed along with assigned register values for the cases of $c = 0$ and $c = 1 \pmod 2$. The code expects $x$ in r3 and $c$ in r4 and stores the result in r5.

In Fig. 3, we display a number of power traces of the Sign-Flip implementation, colored by the sign of the return value. The power trace is labeled with the instruction currently in the execution stage of the processor's pipeline. As this stage is responsible for executing the calculation on the arithmetic logic unit (ALU), we assume that the leakage comes from there. However, we note that

the fetch and decode stage of the pipeline might also leak information [22]. The plot is shorter than the original trace (Fig. 2) as the *fast_trigger* from Sec. 3.2 was used. This results in 0 cycles following the GPIO *high* instruction and only 1 cycle for the GPIO *low* instruction (denoted as "Trigger").

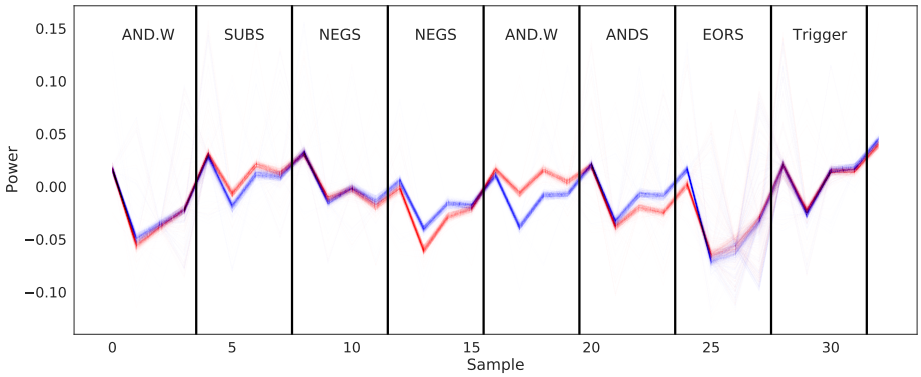


Fig. 3: GALACTICS Sign-Flip power trace annotated with the corresponding assembly instructions in the execute stage of the pipeline. Traces shown in red and blue had negative and positive return value, respectively.

Our power traces show that the instructions 1, 2, 4, 5, and 6 have visible leakage. From Tab. 3, we know that instructions 1, 2, and 4 differ by 15 and 16 in Hamming weight, and instructions 5 and 6 are expected to differ by 8, when comparing $c = 0$ and $c = 1 \pmod 2$ cases. Applying the same methodology, we found similar patterns of leakage in the Sign-Flip implementations of FALCON and FrodoKEM.

To reduce leakage, we propose four ARM assembly implementations of the Sign-Flip function aimed at avoiding large difference in Hamming weight when comparing $c = 0$ and $c = 1 \pmod 2$ cases. An implementation based on the MUL instruction is shown in Tab. 4, an implementation based on shifting in Tab. 5. Tab. 6 shows an implementation built on the XOR operation, and finally, Tab. 7 shows an implementation based on the SXTH instruction.

We base one alternative Sign-Flip implementation on the MUL instruction, shown in Tab. 4. First, depending on $c$, the value 2 or 3 is generated. (Note the similar Hamming weight.) Then, the value $-2x$ is computed by negating $x$ and summation. Finally, $x$ is multiplied with 1 or 3 (depending on $c$), and the result is added to $-2x$.

While this reduces the precision of $x$ by 2 bits, even in the case of FALCON, only 5-bit precision are needed, as $x$ only is in the range of $[0, 18]$. FrodoKEM and GALACTICS use even less precision. While this avoids some of the large Hamming weight differences seen in the GALACTICS implementation, instruc-

| instruction | pseudo-code | computed function | assigned register values if $c = 0 \mod 2$ | otherwise |
|---|---|---|---|---|
| 1. lsl r4, r4, #1 | r4 = r4 <<1 | = c <<1 | c <<1 | c <<1 |
| 2. and.w r1, r4, #3 | r1 = r4 & 0x3 | = c & 0x3 | 0x00000000 | 0x00000002 |
| 3. orr r1, r1, #1 | r1 = r1 \| 0x1 | = (c & 0x3) \| 0x1 | 0x00000001 | 0x00000003 |
| 4. negs r5, r3 | r5 = -r3 | $= -x$ | $-x$ | $-x$ |
| 5. add r5, r5, r5 | r5 = r5 + r5 | = (-x) + (-x) | $-2x$ | $-2x$ |
| 6. mul r6, r1, r3 | r6 = r1 * r3 | = ((c & 0x3) \| 0x1) * x | $x$ | $3x$ |
| 7. add r5, r5, r6 | r5 = r5 + r6 | | $-x$ | $x$ |

Table 4: Sign-Flip implementation based on the MUL instruction. The code expects $x$ in r3 and $c$ in r4 and stores the result in r5.

tion 7 still shows Hamming weight significantly differing between the cases of $c = 0$ and $c = 1 \pmod 2$.

| instruction | pseudo-code | computed function | assigned register values if $c = 0 \mod 2$ | otherwise |
|---|---|---|---|---|
| 1. negs r5, r3 | r5 = -r3 | = -x | -x | -x |
| 2. add r6, r5, r5 | r6 = r5 + r5 | = (-x) + (-x) | -2x | -2x |
| 3. add r5, r6, r5 | r5 = r6 + r5 | = ((-x) + (-x)) + (-x) | -3x | -3x |
| 4. and.w r1, r4, #1 | r1 = r4 & 1 | = c & 1 | 0x00000000 | 0x00000001 |
| 5. add r1, r1, 1 | r1 = r1 + 1 | = (c & 1) + 1 | 0x00000001 | 0x00000002 |
| 6. lsl r6, r3, r1 | r6 = r3 <<r1 | = x <<((c & 1) + 1) | 2x | 4x |
| 7. add r5, r5, r6 | r5 = r5 + r6 | | -x | x |

Table 5: Sign-Flip implementation based on the SHIFT instruction. The code expects $x$ in r3 and $c$ in r4 and stores the result in r5.

Second, we propose an implementation of Sign-Flip based on the SHIFT instruction, shown in Tab. 5. Like the implementation based on MUL above, it avoids the computation of (c & 1) - 1 and hence does not show large Hamming weight difference between the two cases. Also it requires a reduction in precision of $x$ by 3 bit as we need to compute $4x$. After computing computing $-3x$ in instructions 1 - 3, the values $2x$ or $4x$ are generated in instructions 4 - 6 using a left shift operation. Then, the result is calculated by adding these results in instruction 7, which also results in the majority of leakage which we observed. We remark that we also observed leakage of the left shift operation, even though there is no difference in Hamming distance between $c = 0$ and $c = 1 \pmod 2$.

Third, in Tab. 6, we propose an implementation based on the XOR-Operation. In this implementation, the constants 0xAAAAAAAA and 0x55555555 are used to "mask" the values $x$ and $-x$. As both constants have Hamming weight 16, this equalizes the Hamming distance for assigned register values across the cases $c = 0$ and $c = 1 \pmod 2$. On the downside, it uses twice the number of instructions that the MUL and SHIFT implementations require. Also, we found

| instruction | pseudo-code | assigned register values<br>if $c = 0 \mod 2$ | otherwise |
|---|---|---|---|
| 1. mov r5, 0xAAAAAAAA | r5 = 0xAAAAAAAA | 0xAAAAAAAA | 0xAAAAAAAA |
| 2. and.w r1, r4, #1 | r1 = r4 & 1 | 0x00000000 | 0x00000001 |
| 3. negs r4, r3 | r4 = -r3 | r4 = -x | r4 = -x |
| 4. lsr r6, r5, r1 | r6 = r5 <<r1 | 0xAAAAAAAA | 0x55555555 |
| 5. and.w r7, r5, r3 | r7 = r5 & r3 | b'x[31]0 ... x[1]0' | b'x[31]0 ... x[1]0' |
| 6. and.w r8, r5, r4 | r8 = r5 & r4 | b'(-x)[31]0 ... (-x)[1]0' | b'(-x)[31]0 ... (-x)[1]0' |
| 7. mov r5, #0x55555555 | r5 = 0x55555555 | 0x55555555 | 0x55555555 |
| 8. and.w r9, r5, r4 | r9 = r5 & r4 | b'0x[30] ...0x[0]' | b'0x[30] ... 0x[0]' |
| 9. and.w r10, r5, r3 | r10 = r5 & r3 | b'0(-x)[30] ... 0(-x)[0]' | b'0(-x)[30] ... 0(-x)[0]' |
| 10. lsl r5, r5, r1 | r5 = r5 <<r1 | 0x55555555 | 0xAAAAAAAA |
| 11. eor r7, r7, r9 | r7 = r7 $\oplus$ r9 | b'x[31](-x)[30] ... x[1](-x)[0]' | b'x[31](-x)[30] ... x[1](-x)[0]' |
| 12. eor r8, r8, r10 | r8 = r8 $\oplus$ r10 | b'(-x)[31]x[30] ... (-x)[1]x[0]' | b'(-x)[31]x[30] ... (-x)[1]x[0]' |
| 13. and.w r5, r7, r5 | r5 = r7 & r5 | b'0(-x)[30] ... 0(-x)[0]' | b'x[31]0 ... x[1]0' |
| 14. and.w r6, r8, r6 | r6 = r8 & r6 | b'(-x)[31]0 ... (-x)[1]0' | b'0x[30] ...0x[0]' |
| 15. eor r5, r5, r6 | r5 = r5 $\oplus$ r6 | -x | x |

Table 6: Sign-Flip implementation based on the XOR instruction. The code expects $x$ in r3 and $c$ in r4 and stores the result in r5.

significant leakage of the instructions 10 and 15. Instruction 10 is leaking even though there is no difference in assigned Hamming weight, as it shifts depending on the value of $c$. Instruction 15 is leaking as it has unavoidable Hamming weight difference required by the Sign-Flip function's definition.

Fourth and last, in Tab. 7, we propose an implementation based on the SXTH instruction. While this implementation avoids the SHIFT instruction and thus the leakage it caused in the SHIFT and MUL versions above, we can still observe leakage of the ROR instruction. Again, we are also observing leakage of the final SXTH instruction, as it extends the computed 16 bit value of $x$ and $-x$, respectively, having large Hamming weight difference across the cases $c = 0$ and $c = 1 \pmod 2$.

| instruction | pseudo-code | assigned register values<br>if $c = 0 \mod 2$ | otherwise |
|---|---|---|---|
| 1. negs r7, r3 | r7 = -r3 | -x | -x |
| 2. and.w r8, r4, #1 | r8 = r4 & 1 | 0x00000000 | 0x00000001 |
| 3. lsl r8, r8, #4 | r8 = r8 <<4 | 0x00000000 | 0x00000008 |
| 4. pkhbt r6, r7, r3, LSL #16 | r6 = (r3<31:16>,r7<15:0>) | (x<31:16>,-x<15:0>) | (x<31:16>,-x<15:0>) |
| 5. ror r5, r6, r8 | r5 = r6 ROR r8 | (x<31:16>,-x<15:0>) | (-x<31:16>,x<15:0>) |
| 6. sxth r5, r5 | r5 = r5<15:0>→ r5 <31:0> | r5 = -x | r5 = x |

Table 7: Sign-Flip implementation based on the SXTH instruction. The code expects $x$ in r3 and $c$ in r4 and stores the result in r5.

We subjected all four implementation of the Sign-Flip function to the power side-channel attack presented in Sec. 5.1. We find that while the four novel versions can reduce attack success to some extend, all proposed implementations are still vulnerable to attacks with significant prediction accuracy. In our tests, the the SXTH implementation achieved the best results by reducing the prediction

accuracy of the attack to 91.6%. A detailed comparison of all presented Sign-Flip implementations with respect to the predictive power of our attack can be found in Tab. 8; for completeness, we also include the result of an implementation with branching instructions. It is possible that different hyperparameters of the neural network, more measurements per clock cycle, and/or more examples in the training set can further increase the predictive power of attacks on our proposed implementations, but the presented finding are enough to argue that none of the implementation is sufficiently secure.

| Algorithm | Branching | Post-Quantum Crypto | | | This Work | | | |
|---|---|---|---|---|---|---|---|---|
| | | GALACTICS | FALCON | FrodoKEM | MUL | SHIFT | XOR | SXTH |
| Accuracy | 99.8% | 98.0% | 99.7% | 100.0% | 95.2% | 94.4% | 93.2% | 91.6% |

Table 8: Comparison of all different Sign-Flip versions using an MLP classifier. Trivial accuracy would be 50.0% as it is a coin flip.

### 5.3 Two-Sided CDT Gaussian Sampler

The Sign-Flip function used by the CDT Gaussian sampler in GALACTICS, FALCON and FrodoKEM allows reducing the CDT size by half and thus increases performance and decreases required randomness (Sec. 2).

To avoid leakage of the function, the Sign-Flip function can be avoided at the cost of doubling the CDT table size by adding negative numbers. We modified the FrodoKEM implementation of the Gaussian sampler [1] to operate as a two-sided CDT sampler without a Sign-Flip function.[4]

Again using the attack shown in Sec. 5.1, we found that the resulting prediction accuracy on the two-sided CDT sampler is 100.0%. The recorded power traces are displayed in Fig. 4.

## 6 Conclusion

In this paper, we demonstrated that the implementations of the Sign-Flip function in GALACTICS, FALCON and FrodoKEM all leak information on the sign of the produced Gaussian sample when attacked using power analysis with a simple, generic neural network classifier. Novel implementations that avoid large differences in Hamming weight during the execution of the Sign-Flip code could only reduce the leakage under our attack to a small extent.

We draw the following conclusions:

[4] While confirming that the random distribution did not change and is still according to the specification of $\sigma = 2.8$, we found that, due to the 16-bit constraint on the table entries, the original one-sided implementation has $\sigma \approx 2.8146$ and our two-sided adaptation has $\sigma \approx 2.8138$.

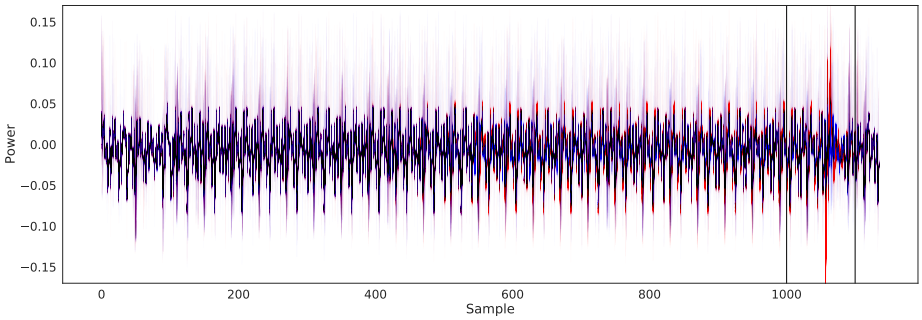


Fig. 4: 1,000 power traces of the FrodoKEM Gaussian sampler, modified to act as a two-sided sampler. Runs resulting in a negative sample are drawn in red, others in blue. The section marked by vertical lines shows the section used for training the classifier.

1. The identified weakness in the original implementations is caused by the Hamming weight distance between the register assignments in each instruction, compared for the cases of $c = 0$ and $c = 1 \pmod 2$.
2. Every single assembly instruction needs to be analyzed to evaluate the power side-channel resistance of a function; in our experiments, the Hamming weight is a good indicator for leakage.
3. Even when carefully crafting an assembly version of the Sign-Flip or the whole Gaussian sampler, leakage on the sign of the Gaussian sample cannot be significantly reduced.
4. The *two's complement* number representation has large difference in Hamming weight for negative and non-negative numbers and thus plays a major role in the leakage of the Sign-Flip function. This is specifically true for a subset like $[-12, 12]$ in FrodoKEM, because of the high Hamming weight distance, as can be seen in Tab. 1.

From these findings, we conclude that, on the STM32F4, no secure implementation of the Sign-Flip function exists as long as the return value is the unmasked sample value. However, as the sample value typically is added to other internal values, this requires either unmasking the sample (which leaks information) or masking the entire scheme.

## References


1. Alkim, E., Bos, J.W., Ducus, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: Frodokem implementation. https://github.com/Microsoft/PQCrypto-LWEKE (2021)
2. Alkim, E., Bos, J.W., Ducus, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: Frodokem learning with errors key encapsulation. https://frodokem.org/files/FrodoKEM-specification-20210604.pdf (2021)
3. ARM: ARM Cortex-M4. https://developer.arm.com/Processors/Cortex-M4
4. ARM: ARM Cortex-M4 instruction cycle count. https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC
5. Askeland, A., Rønjom, S.: A side-channel assisted attack on ntru. Cryptology ePrint Archive, Paper 2021/790 (2021), https://eprint.iacr.org/2021/790, https://eprint.iacr.org/2021/790
6. Carruth, C.: Why statement order can not be enforced. Stackoverflow (2016), https://stackoverflow.com/a/38025837, https://stackoverflow.com/a/38025837
7. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 13–28. Springer (2002)
8. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal gaussians. Cryptology ePrint Archive, Paper 2013/383 (2013), https://eprint.iacr.org/2013/383, https://eprint.iacr.org/2013/383
9. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal gaussians. Cryptology ePrint Archive, Report 2013/383 (2013), https://ia.cr/2013/383
10. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Galactics implementation. https://github.com/espitau/GALACTICS (2019)
11. Fouque, P.A., Hoffstein, J., Kirchner, P., Luybashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier lattice-based compact signatures over ntru. https://falcon-sign.info/falcon.pdf (2020)
12. Fouque, P.A., Hoffstein, J., Kirchner, P., Luybashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon implementation. https://falcon-sign.info/ (2020)
13. Genêt, A., Kalułterović, N.: Single-trace clustering power analysis of the point-swapping procedure in the three point ladder of cortex-m4 sike. Cryptology ePrint Archive, Paper 2022/364 (2022), https://eprint.iacr.org/2022/364, https://eprint.iacr.org/2022/364
14. Kamucheka, T., Fahr, M., Teague, T., Nelson, A., Andrews, D., Huang, M.: Power-based side channel attack analysis on pqc algorithms. Cryptology ePrint Archive, Paper 2021/1021 (2021), https://eprint.iacr.org/2021/1021, https://eprint.iacr.org/2021/1021
15. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4
16. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Paper 2019/844 (2019), https://eprint.iacr.org/2019/844, https://eprint.iacr.org/2019/844
17. Khalid, A., Howe, J., Rafferty, C., Regazzoni, F., O'Neill, M.: Compact, scalable, and efficient discrete gaussian samplers for lattice-based cryptography. In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS) (2018). https://doi.org/10.1109/ISCAS.2018.8351009

18. Kim, S., Hong, S.: Single trace analysis on constant time cdt sampler and its countermeasure. Applied Sciences **8**(10) (2018). https://doi.org/10.3390/app8101809, https://www.mdpi.com/2076-3417/8/10/1809
19. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual international cryptology conference. pp. 388–397. Springer (1999)
20. Lerman, L., Bontempi, G., Markowitch, O., et al.: Power analysis attack: an approach based on machine learning. Int. J. Appl. Cryptogr. **3**(2), 97–115 (2014)
21. Marzougui, S., Wisiol, N., Gersch, P., Krämer, J., Seifert, J.: Machine-learning side-channel attacks on the GALACTICS constant-time implementation of BLISS. CoRR **abs/2109.09461** (2021), https://arxiv.org/abs/2109.09461
22. McCann, D., Oswald, E., Whitnall, C.: Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. Cryptology ePrint Archive, Paper 2016/517 (2016), https://eprint.iacr.org/2016/517, https://eprint.iacr.org/2016/517
23. National Institute of Standards and Technology (NIST): Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography
24. NewAE Technology Inc.: ChipWhisperer-Lite 32-Bit. https://www.newae.com/products/NAE-CWLITE-ARM
25. NewAE Technology Inc.: ChipWhisperer software. https://github.com/newaetech/chipwhisperer
26. NewAE Technology Inc.: CW308 UFO Target Board. https://www.newae.com/products/NAE-CW308
27. NewAE Technology Inc.: STM32F4 Target for CW308. https://www.newae.com/ufo-target-pages/NAE-CW308T-STM32F4
28. Ngo, K., Dubrova, E., Johansson, T.: Breaking masked and shuffled cca secure saber kem by power analysis. Cryptology ePrint Archive, Paper 2021/902 (2021), https://eprint.iacr.org/2021/902, https://eprint.iacr.org/2021/902
29. O'Flynn, C., Chen, Z.D.: Chipwhisperer: An open-source platform for hardware embedded security research. Cryptology ePrint Archive, Paper 2014/204 (2014), https://eprint.iacr.org/2014/204, https://eprint.iacr.org/2014/204
30. Park, J., Anandakumar, N.N., Saha, D., Mehta, D., Pundir, N., Rahman, F., Farahmandi, F., Tehranipoor, M.M.: Pqc-sep: Power side-channel evaluation platform for post-quantum cryptography algorithms. Cryptology ePrint Archive, Paper 2022/527 (2022), https://eprint.iacr.org/2022/527, https://eprint.iacr.org/2022/527
31. Ravi, P., Roy, D.B., Bhasin, S., Chattopadhyay, A., Mukhopadhyay, D.: Number "not used" once - practical fault attack on pqm4 implementations of nist candidates. Cryptology ePrint Archive, Paper 2018/211 (2018), https://eprint.iacr.org/2018/211, https://eprint.iacr.org/2018/211
32. STMicroelectronics: STM32F405/415. https://www.st.com/en/microcontrollers-microprocessors/stm32f405-415.html#overview
33. Tibouchi, M., Wallet, A.: One bit is all it takes: A devastating timing attack on bliss's non-constant time sign flips. Cryptology ePrint Archive, Paper 2019/898 (2019), https://eprint.iacr.org/2019/898, https://eprint.iacr.org/2019/898