TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

# Soft Analytical Side-Channel Attacks on the Number Theoretic Transform for Post-Quantum Cryptography

Custers, Frank J.A.

*Award date:*
2022

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

**Take down policy**
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 21. Mar. 2025

Eindhoven University of Technology

NXP Semiconductors

Master Industrial and Applied Mathematics

# Soft Analytical Side-Channel Attacks on the Number Theoretic Transform for Post-Quantum Cryptography

Master Graduation Project

*Author:*
Frank Custers (1013292)
f.j.a.custers@student.tue.nl

*Supervisor TU/e:*
dr. Andreas Hülsing
a.t.huelsing@tue.nl

*Supervisors NXP:*
dr. Christine Cloostermans
christine.cloostermans@nxp.com

dr. Joost Renes
joost.renes@nxp.com

March 31, 2022

**Abstract**

In this thesis, we investigate a subclass of side-channel attacks, called soft analytical side-channel attacks (SASCA). Our target for this attack is the number theoretic transform (NTT), which consists of a network of so-called butterfly operations. The NTT is a transform method that is used for efficiently multiplying polynomials, and is used in various post-quantum cryptography schemes. A SASCA attack consists of an online and offline part, where in the online part side-channel information is obtained via template matching. For this thesis we simulate the online step and focus on the offline step, which revolves around solving a marginalization problem using the efficient belief propagation algorithm. We investigate previous works on this topic and note required further improvements. Most notably is the $\mathcal{O}(q^2)$ runtime of belief propagation on this target, where $q$ is a parameter fixed by the target scheme. For schemes using a value of $q$ of size around 12 bits this is feasible but slow, while for schemes with a larger $q$ this becomes intractable. Therefore we propose various methods for improvement to the runtime.

## Acknowledgements

This master thesis is the result of my time doing research at the Eindhoven University of Technology, which was in collaboration with the company NXP Semiconductors.
First of all, I would like to thank my supervisors Andreas Hülsing (TU/e), Christine Cloostermans (NXP) and Joost Renes (NXP) for their extensive feedback and insights during our weekly update sessions, as well as always being available to ask questions to. Next I want to thank NXP, particularly the Security Concepts group, for welcoming me and involving me in the group. Also, I want to thank Melissa Azouaoui (NXP) for sharing her knowledge and providing an implementation of the Belief Propagation algorithm. Finally, I would like to thank Marko Boon (TU/e) for being a second reader and being in the assessment committee.

# Contents


**1 Introduction** 4
- 1.1 Notation, abbreviations and figures . . . 5

**2 History of side-channel attacks** 9
- 2.1 Timing attacks . . . 9
- 2.2 Simple and Differential Power Analysis . . . 9
- 2.3 More Divide & Conquer Side-Channel Attacks . . . 12
- 2.4 Template Attacks . . . 12
- 2.5 Algebraic Side-Channel Attacks . . . 14
- 2.6 Countermeasures . . . 14

**3 Soft Analytical Side-Channel Attacks** 16
- 3.1 Template matching and modeling simulated leakage . . . 16
- 3.2 Belief Propagation . . . 19
- 3.3 Post-processing . . . 28

**4 Use case: applying Soft Analytical Side-Channel Attacks to the Number Theoretic Transform in Post-Quantum Cryptography schemes** 29
- 4.1 Post-Quantum Cryptography and Learning With Errors . . . 29
- 4.2 The Number Theoretic Transform . . . 31
- 4.3 Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption . . . 35
- 4.4 More Practical Single-Trace Attacks on the Number Theoretic Transform . . . 37
- 4.5 Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber . . . 39

**5 Experimental results of simulated SASCA attacks an NTT structure** 40
- 5.1 Details on the implementation . . . 40
- 5.2 Simulations to test performance . . . 44
- 5.3 Complexity and butterfly message creation . . . 47

**6 Improving SASCA attacks on an NTT structure** 50
- 6.1 Improving success rate . . . 51
- 6.2 Revisiting the factor graph with split butterfly node . . . 52
- 6.3 Computations modulo smaller primes and recombining with the Chinese Remainder Theorem . . . 56
- 6.4 Faster butterfly message creation via non-negligible support . . . 57

**7 Discussion** 64
- 7.1 Conclusion . . . 64
- 7.2 Shortcomings and suggestions for further research direction . . . 65

**Appendices** 71

**A NTT multiplication example** 71

**B LBP on the XOR-AND example** 72

# 1 Introduction

With the upcoming threat of large scale quantum computers, the cryptography community has started preparing schemes that resist against quantum adversaries. In 2017 the National Institute of Standards and Technology (NIST) started a Post-Quantum Cryptography (PQC) competition [47], which at the time of writing is in the third and final round, and aims to standardize a new set of quantum-resistant public-key schemes that can replace the ones currently being used. The largest family of schemes remaining in this round are the ones that are based on hard lattice problems.

Simultaneously while new schemes are being designed, also new attacks are discovered and applied. Since these new schemes are designed with more security knowledge, it becomes harder for an attacker to find flaws to exploit in the design. This however gives rise to attackers being more creative and designing new types of attacks. One such class is what is called a Side-Channel Attack (SCA), which targets physical information that leaks when a device runs a cryptographic scheme.

In this thesis we focus on a subclass of side-channel attacks, called Soft Analytical Side-Channel Attacks (SASCA) [65]. More specifically, we focus on applying this type of attack to the Number Theoretic Transform (NTT) which is used in various lattice-based PQC schemes. A SASCA attack consists of two main parts: collecting side-channel information and doing computations with this information to obtain the secret such as a private key or plaintext. We will simulate this first step and focus on the second step, which revolves around using an algorithm called Belief Propagation (BP), which is an efficient message-passing algorithm that acts on graphs and solves the marginalization problem. The goal of this research is to collect and analyze what has already been researched for applying SASCA on the NTT, such that we can further improve it. Improving can be done in two directions: improving the success rate or improving the runtime of an attack. Even though BP is an efficient algorithm, it still is slowed down by the large parameters that are used in cryptographic schemes.

In Section 2 we give an overview of various side-channel attacks. Then in Section 3 we give a full description of soft analytical side-channel attacks and the belief propagation algorithm. In Section 4 we describe the use case of this thesis, which is applying SASCA to the NTT. To do so we first describe the basics of lattice-based post-quantum cryptography schemes and their underlying hard problem, as well as the number theoretic transform and its efficient implementation. Then a series of 3 papers that demonstrate these attacks are summarized. In Section 5 we describe the details of our own implementation of such an attack and do some experimental results by varying the relevant parameters. This implementation is then used in Section 6 to investigate various improvements. Finally, in Section 7 we discuss our results, as well as shortcomings and recommendations for further research.

## 1.1 Notation, abbreviations and figures

Throughout the thesis we will use the following notation:

| Notation | Meaning |
|---|---|
| $\cdot$ | Regular multiplication of two integer/real/complex values or polynomials |
| $*$ | Point-wise multiplication of two integer/real/complex valued vectors |
| $\star$ | Convolution between two integer/real/complex valued vectors |
| $a \mod q$ | An integer $a$ in the range $0, ..., q-1$ by using modular reduction |
| $\mathcal{D}$ (calligraphic capital) | A set or domain |
| $\mathbb{Z}_q$ | The ring of integers with modulo $q$ |
| $\mathcal{D}[x]/(f)$ | The polynomial ring with coefficients in $\mathcal{D}$ and quotient polynomial $f$ |
| $\mathbf{a}$ (boldface) | A vector, explicitly written with $[\cdot]$ brackets |
| $\tilde{\mathbf{a}}$ | A vector in the NTT domain |
| $\mathbf{a}[i]$ | The value in the vector $\mathbf{a}$ with index $i$, starting at 0 |
| $\mathbb{P}[x = a \vert l]$ | The probability that an intermediate value $x$ is equal to $a$, given the leakage information $l$ |
| $\mathcal{N}(a, \sigma^2)$ | The normal (Gaussian) distribution with mean $a$ and standard deviation $\sigma$ |
| $\mathcal{U}$ | Discrete uniform distribution, typically defined on a set $\mathcal{D}$ |
| $HW(a)$ | The Hamming weight of the integer value $a$, i.e. the number of ones in the binary representation |
| $\mathcal{O}(\cdot)$ | Big O notation for algorithmic runtime |
| $\mathbf{q}_{x_n \to f_m}$ or $\mathbf{q}_{n \to m}$ | A message from a variable node $x_n$ to a factor node $f_m$ |
| $\mathbf{r}_{f_m \to x_n}$ or $\mathbf{r}_{m \to n}$ | A message from a factor node $f_m$ to a variable node $x_n$ |

Throughout the thesis we will use the following abbreviations:

| Abbreviation | Meaning |
|---|---|
| BKZ | Block Korkine-Zolotarev |
| BP | Belief Propagation |
| CPA | Correlation Power Analysis |
| D&C | Divide & Conquer |
| DES | Data Encryption Standard |
| DPA | Differential Power Analysis |
| EM | Electromagnatic |
| FFT | Fast Fourier Transform |
| HW | Hamming Weight |
| HW-leakage | Hamming Weight leakage model |
| ID-leakage | Identity leakage model |
| LBP | Loopy Belief Propagation |
| LPR-scheme | Cryptosystem by Lyubashevsky, Peikert and Regev |
| LWE | Learning With Errors |
| MIA | Mutual Information Analysis |
| MLWE | Module Learning With Errors |
| NIST | National Institute of Standards and Technology |
| NTT | Number Theoretic Transform |
| PQC | Post-Quantum Cryptography |
| RLWE | Ring Learning With Errors |
| RSA | Rivest–Shamir–Adleman |
| SASCA | Soft Analytical Side-Channel Attack |
| SAT | Satisfiability |
| SCA | Side-Channel Attack |
| SPA | Simple Power Analysis |
| TA | Template Attacks |
| XOR | Exclusive-OR |

# List of Figures


1 SPA trace showing an entire DES operation . . . 10
2 DPA traces for DES . . . 11
3 Example of a probability vector from a simulated Hamming weight leakage . . . 18
4 Example of a probability vector from a simulated ID-leakage . . . 18
5 Visualization of XOR example and its factor graph . . . 20
6 Visualization of the BP algorithm steps on the XOR example . . . 24
7 Two factor graphs of the XOR-AND example . . . 27
8 The message passing scheme for LBP on the XOR-AND example factor graph . . . 27
9 The marginal probability of being equal to 0 of the 4 variables in the XOR-AND example . . . 28
10 Visualization of a single butterfly operation in an in-place Cooley-Tukey NTT . . . 34
11 Visualization of an in-place Cooley-Tukey NTT as a network of butterflies, with $n = 8$ 35
12 A single butterfly and its factor graph as in [53] . . . 36
13 The factor graph of a single butterfly as used in [29, 52] . . . 39
14 The factor graph of our implementation in the case $n = 4$, with a loop of length 8 highlighted in red . . . 42
15 The message passing scheme for BP in the case of an NTT with $n = 4$ . . . 44
16 Plot showing the effect of $n$ on the success rate of a SASCA attack on the NTT . . . 45
17 Plot showing the effect of $q$ on the success rate of a SASCA attack on the NTT using a HW-leakage . . . 46
18 Plot showing the effect of $q$ on the success rate of a SASCA attack on the NTT using an ID-leakage . . . 47
19 Distribution of the coefficients of the private key of the LPR-scheme and Kyber, before and after applying the NTT . . . 52
20 Distribution of the elements of the private key in Kyber before (left, centered binomial with $\eta = 2$) and after applying an NTT (right, nearly uniform) where $n = 256$ and $q = 7681$. . . . 52
21 Comparison of computing the marginals directly or with BP using a split and merged butterfly, when $n = 2$ . . . 54
22 Comparison of computing the marginals directly or with BP using a split and merged butterfly, when $n = 4$ . . . 55
23 Comparison of computing the marginals directly or with BP using a split and merged butterfly, when $n = 2$, using an example where only a single solution has a non-negligible probability . . . 55
24 Example of mapping and ID-leakage vector to smaller modulos . . . 57
25 Example of mapping and HW-leakage vector to smaller modulos . . . 57
26 Example of a message computation via non-negligible support trick . . . 60
27 Average success rate of 120 runs of our optimized butterfly message creation using a non-negligible support with ID-leakage . . . 61
28 Average success rate of 120 runs of our optimized butterfly message creation using a non-negligible support with HW-leakage . . . 61
29 Average computation time of our optimized butterfly message creation using a non-negligible support with ID-leakage . . . 63

30 Average computation time of our optimized butterfly message creation using a non-negligible support with HW-leakage . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
31 Example of an NTT computation using Cooley-Tukey butterflies . . . . . . . . . . . 72

# 2 History of side-channel attacks

Attacking a cryptographic scheme can be done in many different and creative ways. In mathematical cryptanalysis, an attacker focuses on weaknesses of the scheme itself, where they try to find flaws that can be exploited. Another type of attack is a Side-Channel Attack (SCA), where the attacker focuses on the implementation of a scheme. When a cryptographic scheme is implemented on a physical device, information can be leaked from this device through physical means. This includes factors like time, power consumption, electromagnetic radiation and so on. In this section we will discuss the development of side-channel attacks and countermeasures, starting at timing attacks in Section 2.1, followed by simple and differential power analysis attacks in Section 2.2 and other divide and conquer attacks in Section 2.3. In Section 2.4 we discuss template attacks in more detail. Section 2.5 gives some information about algebraic side-channel attacks and finally in Section 2.6 we discuss some countermeasures.

## 2.1 Timing attacks

The first paper that introduced a side-channel attack, is a paper by Kocher which was published in 1996, titled "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems" [35]. Here they apply timing attacks on a computation that involves the private key. For Diffie-Hellman and RSA private key operations consist of computing a value value $K$ from a private key $x$ via a computation $K \equiv y^x \mod n$, where $n$ and $y$ public variables. The computation of such modular exponentiation can be done via a square-and-multiply method. In such method, the base value $y$ is repeatedly squared interleaved with a possible multiplication with $y$, where this multiplication happens based on the bit sequence of the secret value $x$. Since this additional multiplication will take some time, an attacker can use this timing side-channel to learn whether the additional multiplication was performed or not. This then implies the value of a bit of the secret value of $x$, and if the attacker learns this for every bit they learn the value of $x$. Furthermore, [35] also shows experimental results by attacking an RSA implementation on a physical device. They discuss how to overcome issues when methods like Montgomery Multiplication and Chinese Remainder Theorem are used. They also discuss countermeasures such as fixed timing implementations and blinding methods. This paper by Kocher [35] laid the foundation of timing attacks, which were further explored by e.g. [22, 30].

## 2.2 Simple and Differential Power Analysis

Subsequently, another side-channel technique was published in 1999, by Kocher, Jaffe and Jun, titled "Differential power analysis" [34]. In this paper, the authors utilize the fact that on a physical device, during computations electrons move through the device. This results in difference of power consumption, as well as a difference in emitted electromagnetic (EM) radiation. These are side-channels that can be used to perform attacks. To measure power consumption on a device, we can apply a small resistor on the power or ground input, which together with the voltage difference can be used to compute the current. If we sample many voltage differences over time during a cryptographic computation, we get what is called a trace. Since this adding the resistor requires some modification to the target device, this method is called semi-invasive. An alternative non-invasive method to obtain a trace is measuring electromagnetic radiation via a probe. Extracting information from a single trace is called Simple Power Analysis (SPA). See Figure 1 for an example

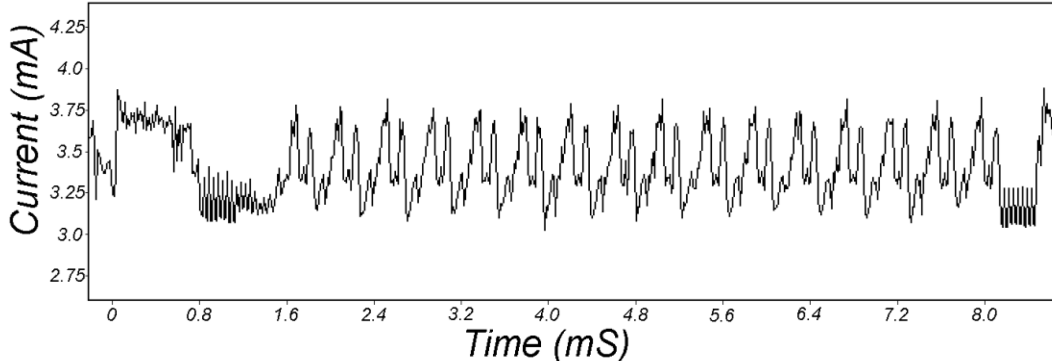


Figure 1: SPA trace showing an entire DES operation. Source: [34]

of an SPA trace during an entire Data Encryption Standard (DES) operation, which clearly shows the 16 rounds that happen in a DES computation. SPA can be used to break cryptographic schemes, by combining that a trace will show the sequence of instructions that are executed and that the values in the trace will depend on the involved data (e.g. the key or other intermediates). In the DES example, differences in the trace can be seen by e.g. rotations in the 28-bit key registers, bit permutations which depend on conditional branching, and string or memory comparisons which again use conditional branching for finding mismatches. Furthermore, modular multiplications will leak a lot of information due to the computation intensity strongly depending on the operands. The same holds for the modular exponentiation from the timing attack example, as it also leaks information on the value of $x$ via SPA since an additional multiplication will have increased power consumption. An attacker can also collect multiple traces and combine the information to learn something about the secret values used. This can be useful to cancel out measurement noise or other forms of noise caused by the physical device. Furthermore, using more traces is useful when the difference in power is too small to derive information on the secret from a single trace. The usage of power measurements leakage from multiple traces is called Differential Power Analysis (DPA) [34]. We will briefly give the intuition behind such an attack by an example on DES, for a more rigorous explanation see [34]. DES uses 16 rounds of a Feistel network, where in every round there are two 32-bit registers $L$ and $R$, and on the $R$ register a Feistel function is applied which is then exclusive-ORed onto $L$. The Feistel function contains 8 S-box operations, where every S-box has as input 6 bits of $R$ exclusive-ORed with 6 bits of a subkey, which is a 48-bit round key derived from the secret key and is only used in a single round. For the attack we define a DES selection function $D(C, b, K_s)$, defined as computing the value of the $b$th bit of the DES intermediate $L$ at the beginning of the 16th round for the ciphertext $C$, with $K_s$ being the 6-bit input to the S-box. If $K_s$ is incorrect, $D(C, b, K_s)$ will give a correct result for the $b$th bit of $L$ with probability approximately a half. The attacker observes $m$ encryptions by recording the traces and saving the ciphertexts. The goal is to determine whether a key block guess $K_s$ is correct. This is done by computing a differential trace. This differential trace is computed as the difference between the average of the traces for which $D(C, b, K_s) = 1$ and the average of the traces for which $D(C, b, K_s) = 0$. For an incorrect $K_s$, the bit $b$ is uncorrelated between the traces and thus the differential trace will go to

zero for increasing $m$. For a correct guess of $K_s$, we have that $D(C, b, K_s) = 1$ with probability 1, thus the differential trace will have a spike at the values that involve this bit $b$. See Figure 2 for an example of the traces, note the spike in the second trace for the correct $K_s$ guess. The full DES key is then recovered by performing this DPA attack on all subkeys. This idea of DPA seemed promising especially on small devices such as smart cards, and was further developed and formalized by Messergers, Dabbish and Sloan [44]. DPA attacks are currently among the most used SCA attacks.

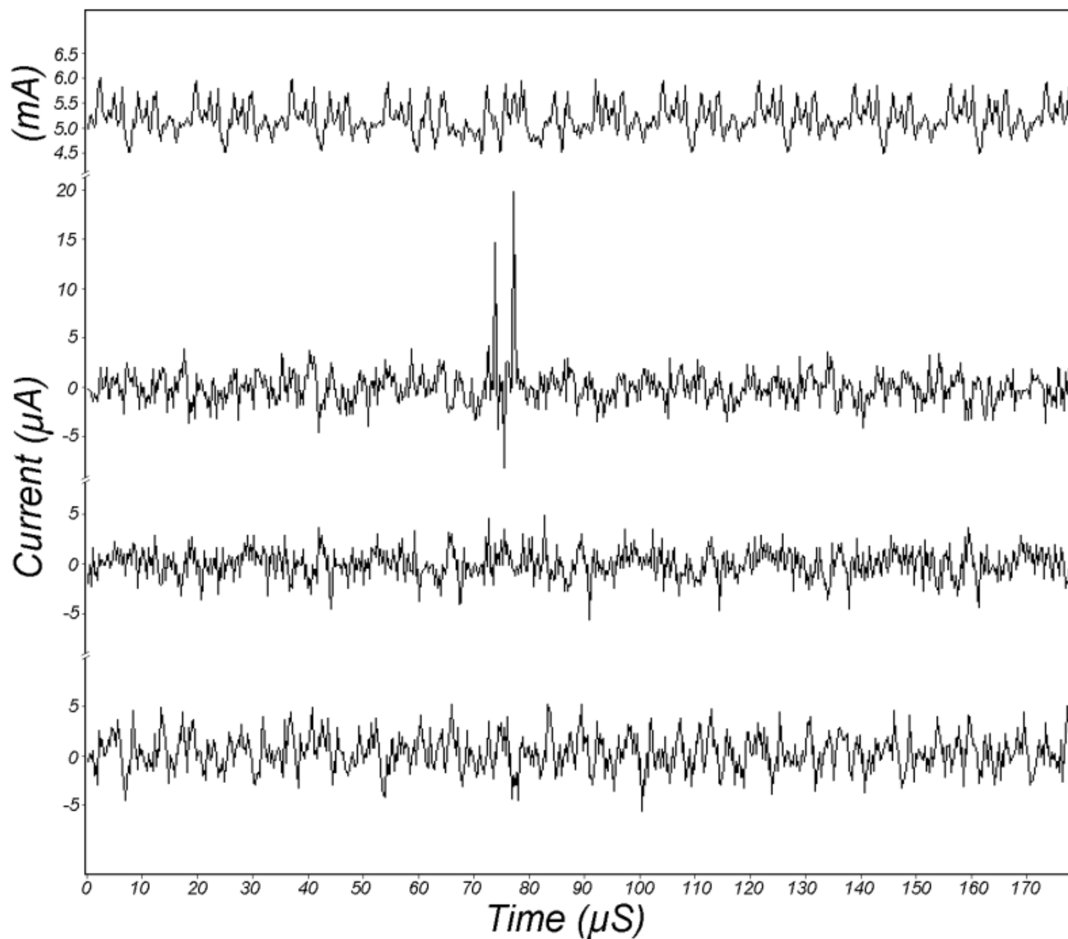


Figure 2: DPA traces for DES, using the selection function, using $m = 1000$ recorded traces. On top is the average of all recorded traces, second is the differential trace for a correct $K_s$ guess and the two below are with incorrect $K_s$ guesses. Source: [34]

## 2.3 More Divide & Conquer Side-Channel Attacks

The attacks described in Section 2.1 and 2.2 typically target smaller parts of some secret value, and later recombine this to obtain the full secret. This is called a Divide & Conquer (D&C) technique, where attacking the smaller parts such as bit values or subkeys is dividing, and recombining is the conquering. Targeting a smaller part of a secret separately means that the target space is smaller and thus an attack will be computationally cheaper, but due to requiring one or often multiple traces, they are expensive in terms of the required data. In [34, 44] it has already been noted that the obtained traces have a strong correlation to the Hamming weight[1] of the observed intermediates. This idea of using Hamming weights is further analyzed in e.g. [19, 41, 50] in what is called the Hamming weight model and its generalization the Hamming distance model.

The DPA attacks described in Section 2.2 use a statistical test to distinguish characteristics in a power trace, namely a difference of means. Subsequently, it has been shown that other statistical tests as distinguishers can be used for power analysis side-channel attacks. The Pearson correlation test is used as distinguisher in Correlation Power Analysis (CPA) attacks, which has been studied by Brier, Clavier and Olivier in [11]. Here they demonstrate CPA attacks on unprotected implementations of algorithms such as DES and AES and they use the Hamming distance model. Since the values in a trace are a result of both the secret values and noise, they can be modeled via probability distributions. This can be used for a Bayesian Classification statistical test as a distinguisher, which is most commonly used in Template Attacks (TA) [14]. See the next section for more details on TA. A more theoretical distinguisher based on Information Theory was proposed by Micali and Reyzin [45], which was further developed by Standaert, Malkin and Yung [63] by making use of Mutual Information which measures the average amount of information present in measurements. This has been generalized by Gierlichs, Batina, Tuyls and Preneel [27] in a power analysis attack named Mutual Information Analysis (MIA). Another noteworthy D&C attack was introduced by Schindler, Lemke and Paar [57], which describes an attack that uses a stochastic model for side channel attacks. This large variation of D&C attacks show that a lot of information can be gained from side-channel information.

## 2.4 Template Attacks

Template Attacks (TA) were introduced by Chari, Rao and Rohatgi [14], and can be considered a D&C power analysis attack with a Bayesian classification as a distinguisher. However, as will be shown in Sections 2.5 and 3, template matching methods can be more broadly used. Since they are a crucial step in Soft Analytical Side-Channel Attacks, we will discuss them here in more detail. TAs are considered the strongest possible form of power analysis side-channel attacks when the attacker has access to only a small number of samples, since they use all the information available from the traces, in contrary to e.g. DPA attacks where a lot of information is discarded due to averaging. A template attack consists of three steps; first in the template building step templates are computed, which are used in the second step where a measurement is compared to all the templates, which gives probabilistic information which then requires a third step where pruning is used to obtain the secret values from the probabilistic information. The first and third step can be done offline, which means that no access to the physical device is required.

[1]The Hamming Weight of an integer is the number of ones in its binary representation

**Template building** An important assumption for a TA is that the attacker has an identical experimental device which can be programmed. The template building is done on the identical device and therefore constitutes an offline step. To do this, we model a power or EM radiation measurement as the sum of the actual power consumption or EM radiation and some noise. This noise is modeled as a sample from a probability distribution, where most commonly the Gaussian distribution is used. The templates aim to capture the characteristics of the dependencies of a specific data input to an operation and the power consumption or EM radiation during that operation. Let $O_i$ be an operation with a corresponding data input $i$, and let there be $K$ different data inputs, thus we have the set of operations $\{O_1, ..., O_K\}$. To build a template for a specific input $i$, the attacker samples $L$ operations of that input and records their traces as $\mathbf{T}_i = [\mathbf{t}_{i,1}, ..., \mathbf{t}_{i,L}]$. For the ease of notation, we will drop the index $i$, so $\mathbf{T} = [\mathbf{t}_1, ..., \mathbf{t}_L]$. Since a trace contains many data points, only a small sample of the $N$ most important points are selected (which can for instance be chosen via the statistical t-test), i.e. $\mathbf{t}_1 = [t_{1,1}, ..., t_{1,N}]$. The values of these $N$ points in a trace are modeled as an $N$-variate Gaussian distribution with mean vector $\mathbf{m}$ and covariance matrix $\mathbf{C}$:

$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N |\mathbf{C}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{m})^T \mathbf{C}^{-1}(\mathbf{x} - \mathbf{m})\right),$$

where $\mathbf{m}$ is approximated as the average of the $L$ traces and $\mathbf{C}$ the covariance matrix of these $L$ traces, where every trace consists of $N$ data points:

$$\mathbf{m} = \frac{1}{L}\sum_{j=1}^{L} \mathbf{t}_j, \quad \mathbf{C}_{u,v} = \operatorname{cov}([t_{1,u}, ..., t_{L,u}], [t_{1,v}, ..., t_{L,v}]), \quad u, v = 1, ..., N$$

A template now consists of the pair $\mathbf{m}$ and $\mathbf{C}$, thus for every input $i$ we build the template $h_i = (\mathbf{m}, \mathbf{C})_i$.

**Template matching** After the offline template building phase, the attacker does an online side-channel attack and obtains a trace $\mathbf{t}$ that corresponds to exactly one operation in the set $\{O_1, ..., O_K\}$. The goal is to determine the correct operation, which is done by creating a hypothesis for every operation. The probability of the hypotheses being correct is computed via the templates, which are modeled via a multivariate Gaussian distribution, thus the attacker evaluates their probability density function:

$$p(\mathbf{t}|h_i) = p(\mathbf{t}, (\mathbf{m}, \mathbf{C})_i) = \frac{1}{\sqrt{(2\pi)^N |\mathbf{C}|}} \exp\left(-\frac{1}{2}(\mathbf{t} - \mathbf{m})^T \mathbf{C}^{-1}(\mathbf{t} - \mathbf{m})\right).$$

After this evaluation for every $i$, the attacker has the size $K$ vector $l = \left[p(\mathbf{t}|h_i)\right]_{i=1}^{K}$, where the $i$th entry represents the probability that operation $O_i$ happened.

**The pruning step** From the vector of probabilities, the attacker has to choose which input was most likely to have been used. If the noise were truly Gaussian, the most optimal method is to simply select the input which resulted in the highest probability, which is called the maximum likelihood decision. It is also possible that the attacker is interested in a probability distribution over all templates, which can be computed via Bayes' theorem:

$$p(h_i|\mathbf{t}) = \frac{p(\mathbf{t}|h_i) \cdot p(h_i)}{\sum_{j=1}^{K} p(\mathbf{t}|h_j) \cdot p(h_j)},$$

where $p(h_x)$ is the probability that input $x$ was used. Typically this is uniform (i.e. $p(h_x) = \frac{1}{K}$), from which the above equation becomes a normalization such that the values in the vector sum up to 1. From this probability distribution vector the attacker can then prune the results. For example threshold pruning or prune all values that are not in a radius close to the maximum value. The goal of this pruning step is to narrow down the number of possible operations that occurred during the measurement, ideally down to one.

## 2.5 Algebraic Side-Channel Attacks

As noted before, D&C attacks are efficient attacks that make minimal use of the working of the attacked scheme because they attack smaller parts of a secret separately. However, these smaller secrets might be related via known computations, which can be formulated via equations. This leads to attacks aiming to recover the full key at once. One such attack is an SPA attack by Mangard [40] which aims to recover an AES key by greatly reducing the number of possible keys, which is done by an SPA attack on the key expansion step. Another series of attack papers started when Schramm, Wollinger and Paar showed how to combine collision attacks with side-channel information such as SPA [61]. Collision attacks were already known in cryptanalysis as attacks on hash functions, see e.g. [24] for using collision attacks to successfully break MD4. In collision attacks on hash functions, an internal collision propagates to the final output, but for keyed schemes this is not the case. However, it has been shown that the attack can still be extended to keyed schemes by combining the internal collisions with side channel attacks. The first attack showing this technique was demonstrated on DES [61], but collision attacks combined with DPA were also demonstrated to be applicable to AES [10, 60]. In these attacks it is assumed that internal collisions which occur for particular plaintext encryptions are somehow correlated to the secret key. The term Algebraic Side-Channel Attacks (ASCA) was coined by Renauld and Standaert [55], which aimed to combine side-channel information with algebraic cryptanalysis methods such as a SAT-solver. They noted that previous SCA attacks only exploit information from the target schemes to a bare minimum, by e.g. only using the first or last round of a cipher. Furthermore their proposed attack methods do not require knowledge on the plain- or ciphertext, and they require far less side-channel information. The attacks are divided in two phases: an online and offline phase. In the online phase, side channel information is gathered. In the offline phase, this side-channel information is used with algebraic cryptanalysis methods, in a black box setting [21]. The main tool in this second phase is the use of automated SAT solvers as in [6, 20], which solves the satisfiability problem, formed by formulating the target scheme as a set of equations. The challenge here is to form these equations, as this is non-trivial for any modern cryptographic scheme. They demonstrate their methods by attacking the cipher PRESENT due to its simple algebraic structure, which was followed by an attack on AES [56].

## 2.6 Countermeasures

Side-channel information leaks due to physical devices handling secret data during cryptographic operations. Countermeasures against SCAs therefore include reducing the information leaked as well as removing the correlation between the handled data and the secret data. Reducing the amount of information leaked can be done in various ways, e.g. reducing electromagnetic emissions, adding random delays between operations to counter timing attacks, increasing the electromagnetic noise that is emitted, or constant time implementations. Removing or decreasing the correlation

between handled data and secret data can be done via e.g. blinding or masking. Blinding can only be used for specific schemes and is most commonly used for blind signatures. A blind signature is a signature made of a disguised message, such that it can be verified against the original message. This disguise randomizes the data, which can also be done more generally via masking. Masking an operation means that instead of working with a secret value, we split it up in shares such that the operation is applied to every share, after which the outputs are recombined again. For this to be useful, the shares must contain no information of the secret. For example, let $x$ be the secret value and let $r$ be some random value, then via the XOR function we can compute two shares $x' = r$ and $x'' = x$ XOR $r$. Note that $x'$ XOR $x'' = x$, furthermore note that if $r$ is drawn uniformly random on the same domain as $x$, then also $x'$ and $x''$ are uniform random elements from this domain if we look at them separately, while their joint distribution is only uniform if $x$ was drawn uniform. Applying the operation to the masked shares $x'$ and $x''$ gives the masked outputs $y'$ and $y''$ respectively. From these two masked outputs, the correct unmasked output $y$ can be computed, where the computation depends on the applied operation. Instead of boolean masking via the XOR, we can also use additive masking via addition and subtraction. Instead of two shares any arbitrary number of shares can be used.

# 3 Soft Analytical Side-Channel Attacks

From the previous section we saw that we can classify side-channel attacks in two main groups: divide & conquer attacks and algebraic attacks. Generally, D&C attacks have a low computational complexity but high data complexity, while algebraic attacks have a high computational but low data complexity. Soft Analytical Side-Channel Attacks [65] (SASCA) aim to use the best of both worlds by formulating a class of attacks that sits between the two groups, resulting in a practical level of computation and data complexity. To do this, a suitable set of operations that depend on secret input data is required. In such a set of operations, intermediate values are computed that are used in further computations. Via side-channel attacks we can gain probabilistic information on these intermediates. Combining this probabilistic information with knowledge of the operations, i.e. how the intermediates are related, we can formulate a marginalization problem which can be efficiently solved with the Belief Propagation algorithm. If sufficiently much information was obtained from the side-channel, the attacker learns the true values of the intermediates, which depending on the targeted operations can be used to compute the secret target. So to summarize, a SASCA attack starts with a template matching phase to gain probabilistic information on a set of intermediates, which are related to each other in a known way, followed by solving a marginalization problem, and optionally some post-processing is required. In Section 3.1 we describe the template matching step as well as how to model simulated leakage. Then in Section 3.2 we give the marginalization problem and belief propagation algorithm with an example, and finally Section 3.3 gives information about the post-processing phase.

## 3.1 Template matching and modeling simulated leakage

In Section 2.4 template attacks are described, which consist of a template building and matching phase, followed by a pruning phase. We can however also exclude this pruning phase, such that we end up with a collection of probability vectors of every intermediate value. Let $x$ be an intermediate value, which we define as a discrete random value on a domain $\mathcal{D}$, and let $l$ denote the information available from a side-channel, then we use the following notation to denote the probability that an intermediate is equal to a value $a \in \mathcal{D}$:

$$\mathbb{P}[x = a|l].$$

After the template matching phase we have a set of intermediate values $\{x_1, x_2, ...\}$, which each have a probability vector $\left[\mathbb{P}[x = a \mid l]\right]_{a \in \mathcal{D}}$. In a real attack, the attacker performs this step by doing measurements on a physical device. However, in order to get experimental results as well as gain reproducibility, we can simulate this step. One realistic way to do this is using the aforementioned Hamming weight model. In such a model, the attacker gets for every intermediate the sum of the Hamming weight of the true value and some noise, which is sampled from some distribution with a certain variance parameter. Generally we will use the Gaussian distribution for this noise, with mean zero and variance $\sigma^2$, which is a parameter that can be chosen arbitrarily. More generally, instead of one, we can allow the attacker to obtain $m$ leakage values for a single intermediate. These multiple leakage values can come from either multiple leakages within the same trace (e.g. consider a loading and storing operation of the same intermediate), or from multiple traces. So for an intermediate $x$, which has true value $a$, the leakage has the following form:

$$l_x = l_{x,1}||...||l_{x,m} = HW(a) + \mathcal{N}(0, \sigma^2)||...||HW(a) + \mathcal{N}(0, \sigma^2),$$

where the $m$ values of $\mathcal{N}(0, \sigma^2)$ are independently sampled. We will refer to this Hamming weight leakage model as HW-leakage from now on. We can formulate an alternative leakage model by not using the Hamming weights, but assume that side-channel leaks the true value with some error, thus:

$$l_x = a + \mathcal{N}(0, \sigma^2)||...||a + \mathcal{N}(0, \sigma^2).$$

We will call this identity leakage model ID-leakage. In general any function $f : \mathcal{D} \to \mathcal{C}$ with $|\mathcal{C}| \leq |\mathcal{D}|$ can be used to create an $f$-leakage model:

$$l_x = f(a) + \mathcal{N}(0, \sigma^2)||...||f(a) + \mathcal{N}(0, \sigma^2).$$

The template building and matching is then replaced by the following computation:

$$\left[\mathbb{P}_f[x = a|l_x]\right]_{a \in \mathcal{D}} := \left[\mathbb{P}[f(x) = f(a)|l_x]\right]_{a \in D}, \tag{1}$$

where we assumed that $f(x)$ comes from a normal distribution with variance $\sigma$ and the mean can be estimated by taking the mean from all the leakage values. Therefore every element from the vector of the right hand side of Equation 1 is computed as an evaluation of the probability density function of the normal distribution:

$$\mathbb{P}[f(x) = f(a)|l_x] = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{f(a) - \frac{1}{m}\sum_{i=1}^{m} l_{x,i}}{\sigma}\right)^2\right).$$

Note that we don't have to do this computation for every $a \in \mathcal{D}$, but only for every element in the range of $f$. See Figure 3 for an example of a HW-leakage with $m = 1$, $\mathcal{D} = \mathbb{Z}_{97}, \sigma = 0.8$ where the true value is 62, which has Hamming weight 5 and simulated leakage value $l = 5.4422$, thus the sampled noise is 0.4422. As expected, all values that have the same Hamming weight have the same amplitude in the probability vector, therefore all values in $\mathbb{Z}_{97}$ that also have hamming weight 5 (i.e. 31, 47, 55, 59, 61, 62, 79, 87, 91, 93, 94) have an equal probability value. For an example of the ID-leakage see Figure 4, with $m = 1$, $\mathcal{D} = \mathbb{Z}_{97}, \sigma = 3$ and true value 62, with simulated leakage $l = 64.2569$, thus the sampled noise is 2.2569. Here the non-negligible probability values are centered around the leakage value.

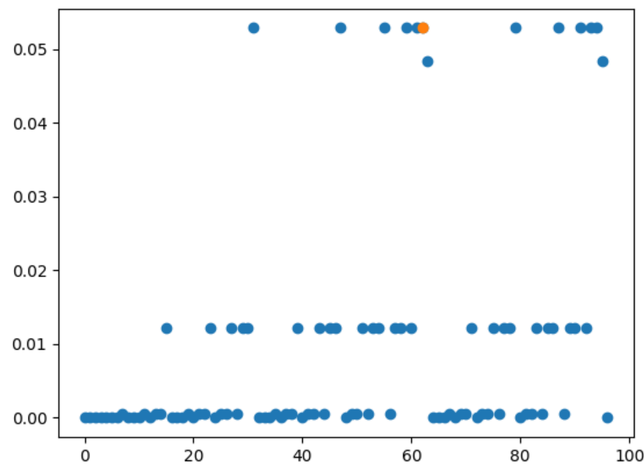


Figure 3: Example of a probability vector from a simulated Hamming weight leakage, where $\mathcal{D} = \mathbb{Z}_{97}, \sigma = 0.8$ and the true value (in orange) is 62.

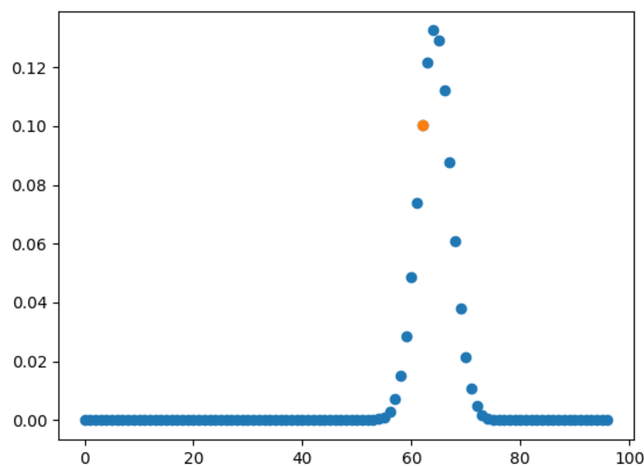


Figure 4: Example of a probability vector from a simulated ID-leakage, where $\mathcal{D} = \mathbb{Z}_{97}, \sigma = 3$ and the true value (in orange) is 62.

The usage of simulated leakage allows comparison of attack methods by verifying for up to which $\sigma$ (which regulates the amount of noise) the attack is successful. Note that simulated leakage

assumes that leakage from any side-channel gains information via the same type of leakage. Furthermore, this makes no assumptions of the architecture of the attacked physical device. If we want to target a specific device we need to make more assumptions, or alternatively we can use something as the ELMO project [42], which is a statistical leakage simulator for specifically the Cortex Arm M0 microprocessors family. Simulated leakage should only be used for analysis purposes, to guarantee the success of a physical attack it should be tested with an experimental setup.

## 3.2 Belief Propagation

In the first step of a SASCA attack we gained probabilistic information on intermediates. The goal of the second step is to use information of the system on how these intermediates are related, to obtain a most likely value for all intermediates in the whole system. This is done by computing the marginal probabilities for every intermediate value, where we want to compute this by marginalizing over all other intermediates. Therefore we want to solve what is called the marginalization problem. This can be computed directly, but it is inefficient since it has a runtime that is exponential in the number of variables. Therefore we use the Belief Propagation (BP) algorithm, which is more efficient since it is a message passing algorithm that does local computations on a graph. To demonstrate the marginalization problem and Belief Propagation we will use the following example:

**Example 1** (Exclusive-OR)**.** *Given are three binary variables $x_1, x_2$ and $x_3$, which are related via an exclusive-OR (XOR) function: $x_1$ XOR $x_2 = x_3$, see the table below for the XOR function.*

| $x_1$ | $x_2$ | $x_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*Via a template matching side-channel attack we gained probabilistic information on the intermediates:*

| $i$ | $\mathbb{P}[x_i = 0\|l]$ | $\mathbb{P}[x_i = 1\|l]$ |
|---|---|---|
| 1 | 0.8 | 0.2 |
| 2 | 0.1 | 0.9 |
| 3 | 0.4 | 0.6 |

*Looking at only $x_3$ there is not much confidence for its actual value being 0 or 1. However, this example is small enough to see that due to $x_1$ being most likely 0 and $x_2$ most likely being 1, one might expect $x_3$ to be 1.*

This is just a small example where all variables are directly related to each other via the XOR function. In a larger example $x_3$ can be related to some other intermediate $x_4$ and so on. With the direct relations we can form a graph which embeds the intermediates and the relations between them. We let the intermediates be variable nodes and the functions that relate the intermediates be factor nodes. Furthermore, the information from the leakage is added via factor nodes as well, which will be connected to exactly one variable node. The graph that is created is called a factor graph. See Figure 5 for an example of the factor graph of the exclusive-OR example. Note that a

factor graph will always be bipartite, since variable nodes nor factor nodes are ever interconnected. Factor nodes that embed how variable nodes are related are binary functions, they return 1 if the relation between the variables is met and 0 otherwise. In the XOR example we have the function $f_{\mathrm{XOR}}$ that relates three variables:

$$f_{\mathrm{XOR}}(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_1 \text{ XOR } x_2 = x_3 \\ 0 & \text{otherwise} \end{cases}$$

The leakage nodes are functions defined as $f_l(a) = \mathbb{P}[x = a|l]$.

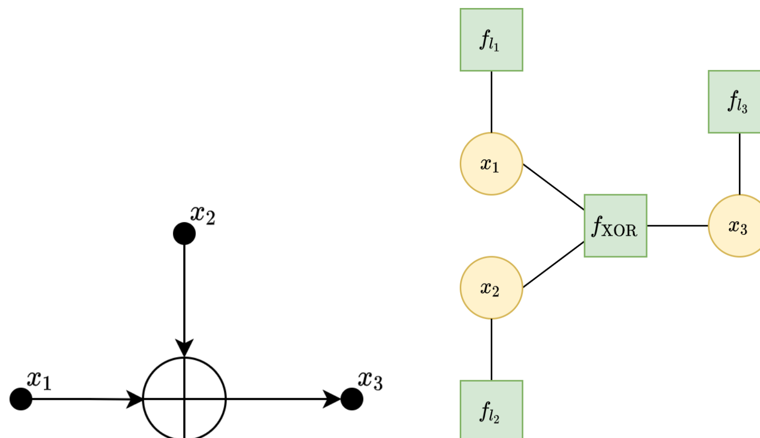


Figure 5: Three intermediate values $x_1, x_2$ and $x_3$ related via an XOR (left) and their factor graph (right).

For consistency with previous works (e.g. [29, 52, 53]), we will follow the notation as introduced in [39]. Some required notation:

- $\mathbf{x} \equiv \{x_n\}_{n=1}^N$ is the set of all variables,
- $x_n$ is a single variable defined on a domain $\mathcal{D}$,
- a factor is denoted by $f$, where the $m$th factor is denoted by $f_m$,
- $\mathbf{x}_m$ is the subset of $\mathbf{x}$ that partake in the factor $f_m$, if we want to exclude a neighbor $n$ we write $\mathbf{x}_m \setminus n$,
- $\mathcal{N}(m)$ is the set of indices of the variables that are connected to the factor $f_m$, likewise $\mathcal{M}(n)$ is the set of indices of the factors that are connected to the variable $x_n$,
- a message from a variable $x_n$ to factor $f_m$ is denoted by $\mathbf{q}_{n\to m}[x_n]$ and message from a factor $f_m$ to a variable $x_n$ is denoted by $\mathbf{r}_{m\to n}[x_n]$

**The marginalization problem** Using the above notation we can first define the problem we want to solve:

**Definition 1** (Marginalization problem)**.** *Given is a function $P^*$ defined on the set of $N$ variables $\mathbf{x}$, which is defined as the product of its $M$ factors:*

$$P^*(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m).$$

*The problem is to compute the marginalization $Z_n$ for every individual variable $x_n$:*

$$Z_n(x_n) = \sum_{\mathbf{x}\setminus x_n} P^*(\mathbf{x}).$$

*Optionally these marginals can be normalized:*

$$P_n(x_n) = \frac{1}{Z} Z_n(x_n), \quad Z = \sum_{\mathbf{x}} \prod_{m=1}^{M} f_m(\mathbf{x}).$$

Since our XOR example only has 3 variables and all with a domain of size 2, we can compute the marginals directly. Starting with defining the complete function $P^*$:

$$P^*(\mathbf{x}) = P^*(x_1, x_2, x_3) = \prod_{m=1}^{M} f_m(\mathbf{x}_m) = f_{l_1}(x_1) f_{l_2}(x_2) f_{l_3}(x_3) f_{\mathrm{XOR}}(x_1, x_2, x_3).$$

First we compute the normalization factor $Z$:

$$\begin{aligned} Z = \sum_{\mathbf{x}} \prod_{m=1}^{M} f_m(\mathbf{x}) &= \sum_{x_1, x_2, x_3 \in \{0,1\}} f_{l_1}(x_1) f_{l_2}(x_2) f_{l_3}(x_3) f_{\mathrm{XOR}}(x_1, x_2, x_3) \\ &= f_{l_1}(0) f_{l_2}(0) f_{l_3}(0) + f_{l_1}(0) f_{l_2}(1) f_{l_3}(1) + f_{l_1}(1) f_{l_2}(0) f_{l_3}(1) + f_{l_1}(1) f_{l_2}(1) f_{l_3}(0) \\ &= 0.8 \cdot 0.1 \cdot 0.4 + 0.8 \cdot 0.9 \cdot 0.6 + 0.2 \cdot 0.1 \cdot 0.6 + 0.2 \cdot 0.9 \cdot 0.4 = 0.548 \end{aligned}$$

Note that in the computation we omit the triples $x_1, x_2, x_3$ such that $f_{\mathrm{XOR}}$ is 0. Next we compute the marginals per variable:

$$Z_1(x_1) = \sum_{\mathbf{x}\setminus x_1} P^*(\mathbf{x}) = \sum_{x_2, x_3 \in \{0,1\}} P^*(x_1, x_2, x_3).$$

So for $x_1 = 0$ we find

$$Z_1(0) = f_{l_1}(0) f_{l_2}(0) f_{l_3}(0) + f_{l_1}(0) f_{l_2}(1) f_{l_3}(1) = 0.8 \cdot 0.1 \cdot 0.4 + 0.8 \cdot 0.9 \cdot 0.6 = 0.464.$$

Here we again omit the computations for the inputs where the $f_{XOR}$ function is zero. After normalizing we find $P_1(0) = \frac{1}{Z} Z_1(0) = \frac{0.464}{0.548} \approx 0.847$. Doing similar computations five more times we find the following results:

| $i$ | $P_i(0)$ | $P_i(1)$ |
|---|---|---|
| 1 | 0.847 | 0.153 |
| 2 | 0.080 | 0.920 |
| 3 | 0.190 | 0.810 |

An interesting observation is that if we use the computed marginals rather than just the leakage on $x_3$, as can be expected, $x_3$ is now more likely to be equal to 1.

**The relation between the marginalization problem and a SASCA attack** If we look at our example how we computed $Z_1(0)$, we see that we sum up two values, which are both computed by multiplying probabilities that describe an initial distribution of the variables. In the case of a SASCA attack, these initial distributions come from the template matching step. The binary function $f_{XOR}$ ensures that we only sum up combinations of $(x_1, x_2, x_3)$ such that they form a correct solution to the system, which in this case means that it must hold that $x_1$ XOR $x_2 = x_3$. There are 3 variables with domain size 2, so there are $2^3 = 8$ possible solutions to the system, though only 4 are correct, namely $(x_1, x_2, x_3) \in \{(0,0,0),(0,1,1),(1,0,1),(1,1,0)\}$. Since we are computing for $x_1 = 0$ we take only the two solutions where this requirement is met. Now the probability of a solution is the multiplication of all the initial probabilities of the values in this solution, normalized by the sum of the probabilities of all the solutions (which is computed as $Z$). So when we compute a value $Z_n(i)$, we sum up the probabilities of all the solutions where $x_n = i$. In a SASCA attack, we want to obtain the solution to the system that was actually being computed by the physical device. To obtain this solution, the attacker has to make a guess based on the marginals. This guess is computed by taking the value with the largest probability for every variable independently. In the ideal situation, there is a single solution with a very high probability compared to all other solutions. In this ideal case the marginals all have a value with probability almost 1. However, if there is too much noise during the template matching step, there might be multiple solutions with a non-negligible probability, and the marginals will not be distributed nicely. In such case the attack will fail since the attacker does not obtain the correct values for the intermediates and generally it is hard to guess which values are correctly guessed, if any.

**Belief Propagation** The goal of BP is now to compute these marginals in a more efficient way. It does this by using that variables only directly depend on a small subset of the other variables, namely the ones they are connected to in the factor graph via the factor nodes. This local computation is done by passing messages over the edges in the graph, where every edge connects a variable node to a factor node. These messages are also called beliefs, since they contain information of what the sending node believes the marginals of a variable node should be, based on local information from its neighbors. In the case that the sending node is a variable node, the message contains the beliefs of this variable node, while in the case that the sending node is a factor node, the message contains the beliefs of the receiving variable node. These messages are computed via two rules, one for messages from a variable to a factor node and one vice versa. The rule for computing messages

from a variable node $x_n$ to a factor node $f_m$ is defined as[2]:

$$\mathbf{q}_{n\to m}[x_n] = \prod_{m' \in \mathcal{M}(n) \setminus m} \mathbf{r}_{m' \to n}[x_n],$$

which states that a message is computed element-wise as the product of values in all of the messages that $x_n$ received, except the one it received from $f_m$. The rule for computing a message from a factor node $f_m$ to a variable node $x_n$ is defined as:

$$\mathbf{r}_{m\to n}[x_n] = \sum_{\mathbf{x}_m \setminus n} \left( f_m(\mathbf{x}_m) \prod_{n' \in \mathcal{N}(m) \setminus n} \mathbf{q}_{n' \to m}[x_{n'}] \right),$$

which states that the message is computed element-wise as the product of values in all of the incoming messages to the factor $f_m$, except the one it received from $x_n$, but then marginalized over all those neighbors such that it forms a correct local solution with respect to $f_m$. Before sending the message, it is normalized such that all the values in the message sum up to 1. Since a node will receive a message from all its neighbors, we say it has an inbox of messages. From the message computation rules we can see that BP computes the marginals of the variables but only locally. A node is a leaf if it only has one neighbor. For a leaf variable node $x_n$ with only one neighbor factor $f_m$, the set $\mathcal{M}(n) \setminus m$ is empty and thus we set $\mathbf{q}_{n\to m}[x_n] = 1$. For a leaf factor node $f_m$ with only neighbor the variable $x_n$, the set $\mathcal{N}(m) \setminus n$ is empty and $\mathbf{x}_m \setminus n$ is also empty, so we set $\mathbf{r}_{m\to n}[x_n] = f_m(x_n)$. Leakage information is added to a variable via a leaf factor node. Note that in our XOR example, the factor graph is a tree. This allows for a very simple message passing scheme, namely by defining a root node of the tree and start sending messages from the leaves towards the root, and then back again. In such message passing scheme, on every edge exactly 2 messages are passed, one in each direction. At this point, every variable node has received a message from all its neighboring factor nodes, from which the marginals can be computed as the element-wise product of all the messages, followed by a normalization. See Figure 6 for a visualization on how to solve the marginalization problem of the XOR example via the BP algorithm.

[2]Note that we use the notation $n \to m$ in a subscript to represent a message sent from a variable node $x_n$ to a factor node $f_m$. Alternatively we will use the notation of $x_n \to f_m$ as subscript to clarify which variable and factor node we are dealing with. Typically the $n$ are integer values while the $m$ are named values, such as $XOR$ in our example.

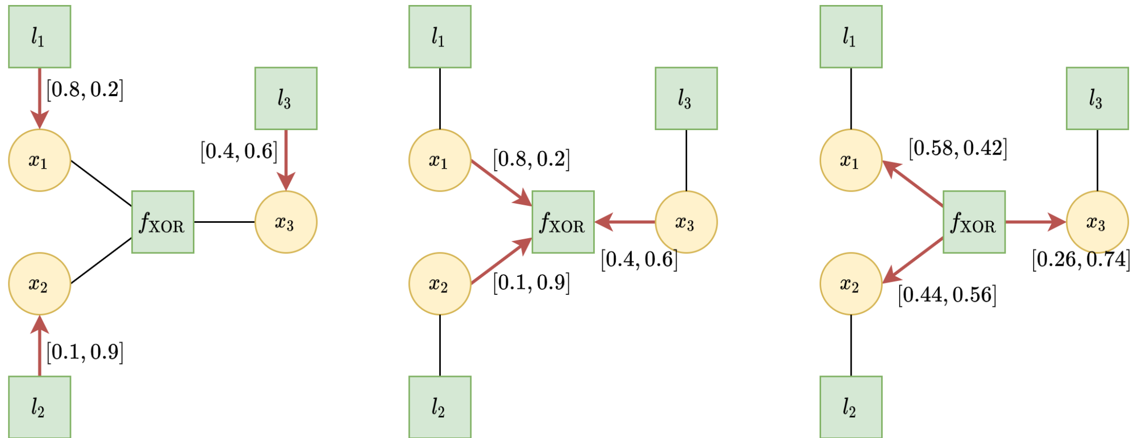


Figure 6: From left to right, the steps in the BP algorithm on the XOR example via the leaf-to-root-and-back passing scheme with $l_1, l_2$ and $l_3$ as leaves and $f_{XOR}$ as root.

In this example we let the factor nodes $l_1, l_2$ and $l_3$ be our leaves and the $f_{XOR}$ factor node be the root and we use the leaf-to-root-and-back message passing scheme. First the leakage factor nodes send their messages to their variable nodes:

$$\mathbf{r}_{l_1 \to x_1} = [0.8, 0.2]$$
$$\mathbf{r}_{l_2 \to x_2} = [0.1, 0.9]$$
$$\mathbf{r}_{l_3 \to x_3} = [0.4, 0.6]$$

Next the variable nodes send their message towards to root node. In this case they only have one message in their inbox, so the outgoing message is exactly this message in the inbox:

$$\mathbf{q}_{x_1 \to f_{XOR}} = [0.8, 0.2]$$
$$\mathbf{q}_{x_2 \to f_{XOR}} = [0.1, 0.9]$$
$$\mathbf{q}_{x_3 \to f_{XOR}} = [0.4, 0.6]$$

Now the root node $f_{XOR}$ has 3 messages in its inbox, and can start sending messages back to the leaves. To compute a message to a variable node, it uses the two messages that came from the

other variable nodes:

$$\begin{aligned}
\mathbf{r}_{f_{XOR}\to x_1} &= \Bigg[\sum_{x_2,x_3} f_{XOR}(0,x_2,x_3)\cdot \mathbf{q}_{x_2\to f_{XOR}}[x_2]\cdot \mathbf{q}_{x_3\to f_{XOR}}[x_3], \\
&\quad \sum_{x_2,x_3} f_{XOR}(1,x_2,x_3)\cdot \mathbf{q}_{x_2\to f_{XOR}}[x_2]\mathbf{q}_{x_3\to f_{XOR}}[x_3]\Bigg] \\
&= [0.1\cdot 0.4+0.9\cdot 0.6, 0.1\cdot 0.6+0.9\cdot 0.4] = [0.58, 0.42] \\
\mathbf{r}_{f_{XOR}\to x_2} &= \Bigg[\sum_{x_1,x_3} f_{XOR}(x_1,0,x_3)\cdot \mathbf{q}_{x_1\to f_{XOR}}[x_1]\cdot \mathbf{q}_{x_3\to f_{XOR}}[x_3], \\
&\quad \sum_{x_1,x_3} f_{XOR}(x_1,1,x_3)\cdot \mathbf{q}_{x_1\to f_{XOR}}[x_1]\cdot \mathbf{q}_{x_3\to f_{XOR}}[x_3]\Bigg] \\
&= [0.8\cdot 0.4+0.2\cdot 0.6, 0.8\cdot 0.6+0.2\cdot 0.4] = [0.44, 0.56] \\
\mathbf{r}_{f_{XOR}\to x_3} &= \Bigg[\sum_{x_1,x_2} f_{XOR}(x_1,x_2,0)\cdot \mathbf{q}_{x_1\to f_{XOR}}[x_1]\cdot \mathbf{q}_{x_2\to f_{XOR}}[x_2], \\
&\quad \sum_{x_1,x_2} f_{XOR}(x_1,x_2,1)\cdot \mathbf{q}_{x_1\to f_{XOR}}[x_1]\cdot \mathbf{q}_{x_2\to f_{XOR}}[x_2]\Bigg] \\
&= [0.8\cdot 0.1+0.2\cdot 0.9, 0.8\cdot 0.9+0.2\cdot 0.1] = [0.26, 0.74]
\end{aligned}$$

At this point, all variable nodes have received a message from their respective neighboring factor nodes. Using these messages, the marginal values are computed:

$$\begin{aligned}
x_1 &\sim [0.58\cdot 0.8, 0.42\cdot 0.2] = [0.464, 0.084] \sim [0.847, 0.153] \\
x_2 &\sim [0.44\cdot 0.1, 0.56\cdot 0.9] = [0.044, 0.504] \sim [0.080, 0.920] \\
x_3 &\sim [0.26\cdot 0.4, 0.74\cdot 0.6] = [0.104, 0.444] \sim [0.190, 0.810]
\end{aligned}$$

Here these marginals are computed as the product of the incoming messages and are normalized to get the final probability distribution. Note that since all three variables in this example are locally connected, the BP algorithm will not be more efficient then the direct computations as computed earlier, since the efficiency comes from doing only local computations.

**Loopy Belief Propagation and short loops** Unfortunately, in practice factor graphs will contain loops. The leaves-to-root-and-back message passing scheme does not work anymore. Furthermore, a cyclic dependency issue arises when computing messages since a message that a node sends is computed from its incoming messages. Luckily these problems can be overcome by what is called Loopy Belief Propagation (LBP). This is an iterative version of BP where messages are initialized. The default method to do this, is to initialize all messages that the variables send to 1, thus $\mathbf{q}_{n\to m}[x_n] = 1$. After this initialization, all factor nodes can send their messages in some arbitrary order, after which the variable nodes can send their new messages. Due to the initialization, the marginals that can now be computed are only an approximation. However, we can do more iterations of sending messages over all edges such that after every iteration the computed marginals are a closer approximation to the true marginals. To do so, a new message is computed from only the newest messages in the inbox of a node. These approximated marginals may converge

to values that are close the true marginals. This convergence is not guaranteed, but it often does in practice. When LBP does not converge, oscillations may take place where the computed marginals oscillate between different solutions. How close the approximations of the converged marginals are will depend on the structure of the graph, namely it depends on how many loops are in the graph and how long these loops are. In general, short loops are bad because they cause for messages that a node receives to be more dependent on the message it had sent, thus confirming its own bias.

To demonstrate LBP, we extend our XOR example. Assume there is a fourth intermediate value $x_4$, that is being computed as $x_1$ AND $x_2 = x_4$, and gained from the template matching step the probabilities $\mathbb{P}[x_4 = 0|l] = 0.7$ and $\mathbb{P}[x_4 = 1|l] = 0.3$. We can extend the factor graph of the XOR example by introducing a factor node for the AND function:

$$f_{AND}(x_1, x_2, x_4) = \begin{cases} 1 \text{ if } x_1 \text{ AND } x_2 = x_4 \\ 0 \text{ otherwise.} \end{cases}$$

See the left side of Figure 7 for the factor graph, and note the loop in this graph of length 4. Since a factor graph is bipartite, a length 4 loop is of minimal length. Computing the marginals directly we find the following exact results:

| $i$ | $P_i(0)$ | $P_i(1)$ |
|---|---|---|
| 1 | 0.9155 | 0.0845 |
| 2 | 0.0868 | 0.9132 |
| 3 | 0.1240 | 0.8760 |
| 4 | 0.9391 | 0.0609 |

If we use LBP instead for 5 iterations, we find the following approximations:

| $i$ | $P'_i(0)$ | $P'_i(1)$ |
|---|---|---|
| 1 | 0.9209 | 0.0791 |
| 2 | 0.0814 | 0.9186 |
| 3 | 0.1361 | 0.8639 |
| 4 | 0.9334 | 0.0666 |

Here we used the default message passing scheme for LBP, so first all variable nodes send the uniform message $[0.5, 0.5]$ to their neighboring factor nodes as initialization, then in every iteration first all factor nodes send messages and then all variable nodes send messages, see Figure 8. See Appendix B for the messages that are sent during LBP on the extended XOR-AND example, and see Figure 9 for the convergence of the marginals during 5 iterations. We terminated LBP after 5 iterations because the marginals have converged to a value that are close approximations to the correct marginals. In some cases it is possible to remove short loops by merging factor nodes. In the XOR-AND example this is also possible by merging $f_{XOR}$ and $f_{AND}$ into a single node $f_{XA}$ that takes as input all 4 variables:

$$f_{XA}(x_1, x_2, x_3, x_4) = \begin{cases} 1 \text{ if } x_1 \text{ XOR } x_2 = x_3 \text{ and } x_1 \text{ AND } x_2 = x_4 \\ 0 \text{ otherwise.} \end{cases}$$

See the right side of Figure 7 for the factor graph using $f_{XA}$. Merging factor nodes can eliminate loops, but the downside is that the new merged node will have a higher number of neighbors which can negatively impact the runtime of BP.

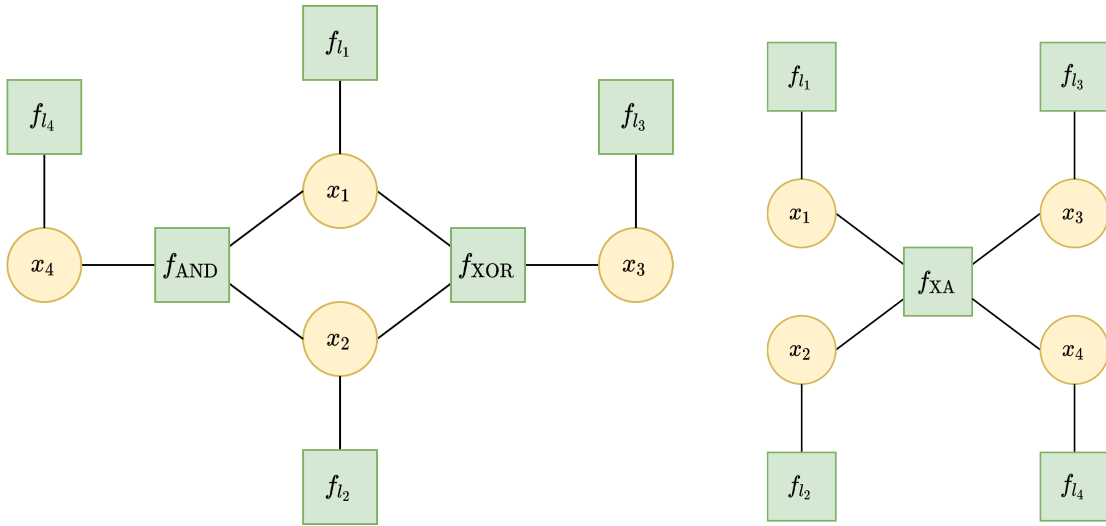


Figure 7: Two factor graphs of the XOR-AND example, left one has a loop of length 4.

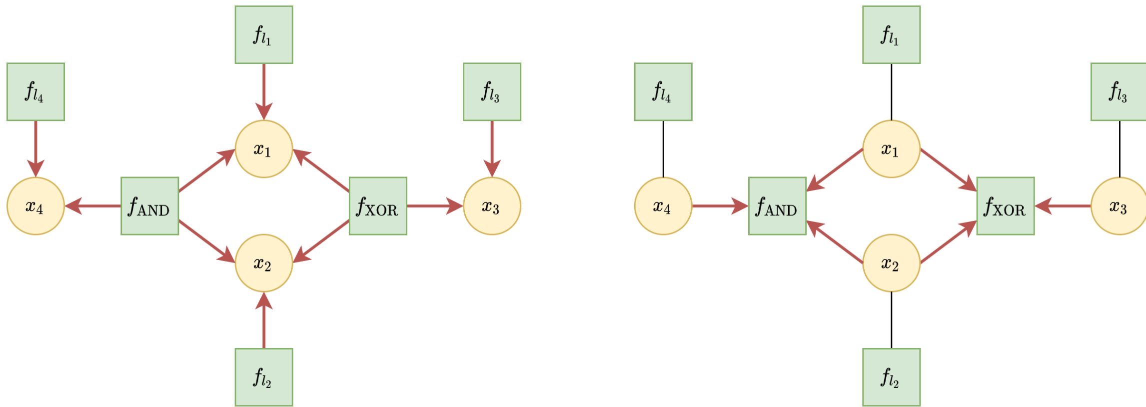


Figure 8: The message passing scheme for LBP on the XOR-AND example factor graph. In every iteration all the factor nodes send messages to the variable nodes (left), followed by all the variable nodes sending messages to the factor nodes except the leakage factor node (right).

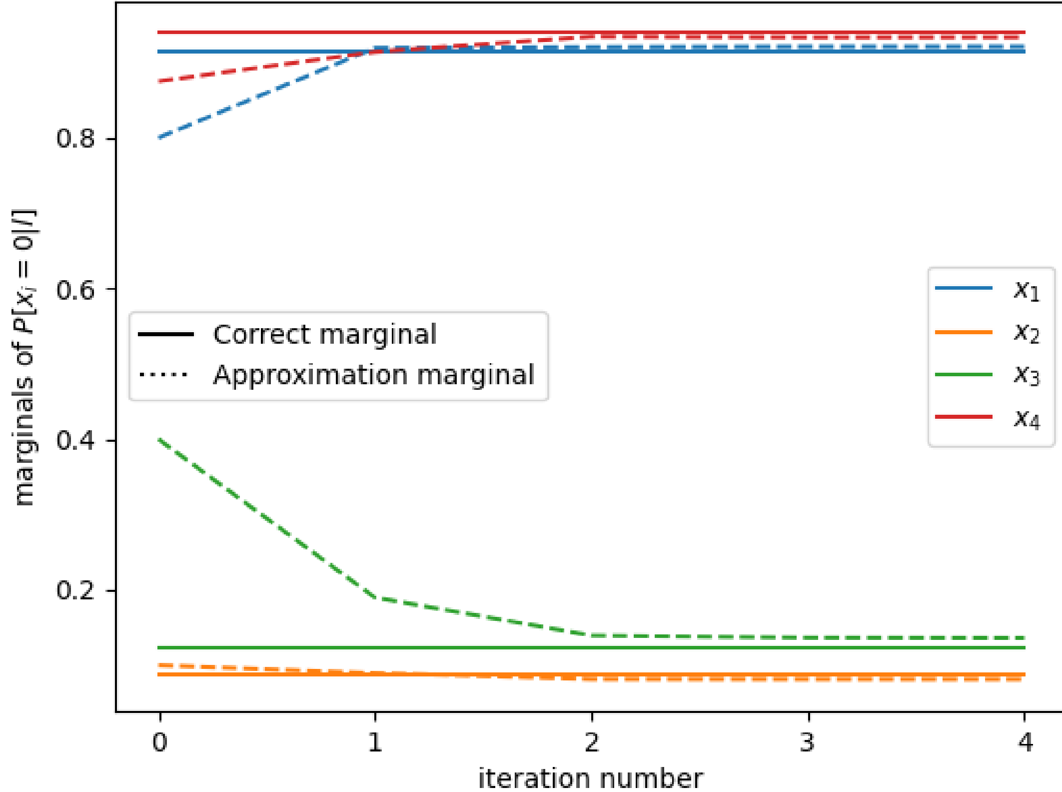


Figure 9: The marginal probability of being equal to 0 of the 4 variables in the XOR-AND example. The solid line is obtained by computing the marginals directly, while the dashed lines are the converging values from LBP over 5 iterations.

### 3.3 Post-processing

At the end of the second step in a SASCA attack, if successful, the attacker has obtained the true values of the intermediate values. It is possible that some of these intermediate values are what the attacker wants, see e.g. the SASCA attack on AES in [65] where some of the intermediates are values of the secret key $K$ or [52] where some of the intermediate values are the plaintext. However, it is expected that intermediates that contain the values of a secret key are better protected against side-channel attacks. Therefore it is sometimes better to target intermediate values that are somehow related to the secret key, as done in e.g. [29, 53], where the recovered intermediate values are the result of a multiplication with the secret key (in these examples they are also in another domain, see Section 4 for more details). In such case a post-processing computation step is needed to retrieve the secret value.

# 4 Use case: applying Soft Analytical Side-Channel Attacks to the Number Theoretic Transform in Post-Quantum Cryptography schemes

Soft Analytical Side-Channel Attacks have already proven to be successful on block ciphers, including the widely used AES [65]. Due to its nature of targeting structured sets of operations, a good target for SASCA attacks are schemes or algorithms that are very algebraic in nature. One such example is the class of lattice-based schemes that are proposed for the upcoming post-quantum schemes, because they use the Number Theoretic Transform (NTT) to speed up polynomial multiplications. In Section 4.1 we first discuss Post-Quantum Cryptography and the underlying hard problem of the lattice based schemes. Then in Section 4.2 we give all required information about the Number Theoretic Transform and an efficient algorithm to implement it. This algorithm has been the target of a series of three papers that apply SASCA attacks to it, which are described in Sections 4.3, 4.4 and 4.5.

## 4.1 Post-Quantum Cryptography and Learning With Errors

Modern cryptography that is currently widely used is based on the assumption that their underlying mathematical problems are hard to solve. However, the hardness of these problems assume an attacker with access to only a classical computer. If an attacker has access to a quantum computer, some classically hard problems become feasible to solve. The most prevalent example is Shor's algorithm [62], which is a quantum algorithm that can solve instances of the discrete log problem and factorization problem in polynomial time (where the fastest classical methods have a sub-exponential runtime), which are mathematical problems used in e.g. the public key schemes RSA, Diffie-Helman and Elliptic curve key exchange. Even though the best current quantum computers are not powerful enough and it is uncertain if it is feasible to build large scale quantum computers, the cryptographic community has started preparing for quantum-resistant cryptographic schemes.

**Post-Quantum Cryptography** In 2017 the National Institute of Standards and Technology (NIST) has started a standardization process of Post-Quantum Cryptography (PQC) [47]. Here new schemes have been submitted that are supposed to be quantum-resistant and can replace current schemes. At the time of writing this competition is in the third round and there are 15 candidates left, which are split up in finalists and alternatives, see Table 1 and 2. PQC schemes are based on problems that are conjectured to be hard even for large scale quantum computers, such as lattice problems, multivariate equations, hash functions, coding theory problems and supersingular elliptic curve isogenies. Among the finalists, we see that the most occuring problem that these schemes are based on, are the lattice based problems.

**(Ring) Learning With Errors** The hard lattice based problems that most NIST PQC finalists are based on, are variants of the Learning With Errors (LWE) problem. LWE was introduced by Regev [54], and later a variant which is based on rings was introduced [38], which we will denote by RLWE and will describe in this section. To describe the RLWE problem we need the following preliminaries:

- Let $\mathbb{Z}_q$ be the ring of integers modulo $q$.

Round 3 finalists

| Public-Key Encryption and Key-Establishments | | Digital Signatures | |
|---|---|---|---|
| Algorithm name | Type | Algorithm name | Type |
| Classic McEliece [8] | Coding theory | CRYSTALS-DILITHIUM [37] | Lattice |
| CRYSTALS-KYBER [4] | Lattice | Falcon [26] | Lattice |
| NTRU [16] | Lattice | Rainbow [23] | Multivariate |
| SABER [25] | Lattice | | |

Table 1: The round 3 finalists of the NIST PQC competition.

- Let $n$ be an integer, which is the degree of a polynomial $f$. Then we can define a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(f)$, which is the ring modulo the polynomial $f$ with coefficients from $\mathbb{Z}_q$. Typically $f = x^n \pm 1$ is used. An element $\mathbf{a} \in \mathcal{R}_q$ is characterized by the coefficients, $\mathbf{a} = a_0 + a_1 x + ... + a_{n-1}x^{n-1}$, so alternatively we can write $\mathbf{a} = [a_0, ..., a_{n-1}]$.
- Let $\chi$ be an error distribution, a distribution with a small support. Typically this is a discrete Gaussian or centered binomial distribution.
- Let $\mathcal{U}$ be the uniform distribution with base $\mathbb{Z}_q$.
- When we draw a polynomial $\mathbf{a} \in \mathcal{R}_q$ from a distribution, we mean that every coefficient is drawn from that distribution.

**Definition 2** (Ring Learning With Errors problems)**.** *Let $\mathcal{R}_q, \chi$ and $\mathcal{U}$ be as above. Let $\mathbf{a}$ be a known polynomial from $\mathcal{R}_q$, where every coefficient $a_i$ is drawn randomly from $\mathcal{U}$. Let $\mathbf{e}$ be randomly drawn from $\chi$ and fix some $\mathbf{s} \in \mathbb{Z}_q^n$. An RLWE tuple is defined as $(\mathbf{a}, \mathbf{b})$ with $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \in \mathcal{R}_q$. A challenger is given $\mathbf{a}$ and an arbitrary number of pairs $(\mathbf{a}, \mathbf{b})$, where in every pair $\mathbf{s}$ is the same and $\mathbf{e}$ is freshly generated. The RLWE search problem is now defined as finding $\mathbf{s}$. We can also define it as a decision problem, where the challenger has to distinguish whether they are given RLWE pairs where $\mathbf{b}$ is constructed as above, or randomly generated.*

A related class of problems is the Module-LWE (MLWE) where instead of elements of $\mathcal{R}_q$, we work with vectors and matrices of size $k \in \mathbb{N}$ and $k \times k$ respectively, where every element in the vector/matrix is an element from $\mathcal{R}_q$. The RLWE and MLWE problems (either search or decision) are a more structured version of the general LWE problem, due to the added ring structure. MLWE and RLWE are preferred for cryptographic implementations since this added structure allows for more efficient computations and smaller key sizes. The hardness of LWE has been proven in [54] and [51] by a reduction proof, showing that LWE is at least as hard as GapSVP, which is a well understood NP-hard lattice problem. The added structure of MLWE and RLWE however invalidates these proofs, and only quantum reductions are currently known [38].

**An RLWE encryption scheme** The most straightforward way to implement RLWE as an encryption scheme was proposed by Lyubashevsky, Peikert and Regev [38], which we will call the LPR-scheme. Note that this is not one of the NIST schemes, but the lattice-based NIST schemes follow a similar approach while the LPR-scheme is an easier example to explain. It uses the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ and has a parameter set $(n, q, \sigma)$. Here $n$ is the degree of the polynomials and thus the size of the elements in $\mathcal{R}_q$, $q$ is the the modulus of the field $\mathbb{Z}_q$ and thus

Round 3 alternative candidates

| Public-Key Encryption and Key-Establishments | | Digital Signatures | |
|---|---|---|---|
| Algorithm name | Type | Algorithm name | Type |
| BIKE [3] | Coding theory | GeMSS [13] | Multivariate |
| FrodoKEM [2] | Lattice | Picnic [15] | ZKP/Hash |
| HQC [43] | Coding theory | SPHINCS+ [9] | Hash |
| NTRU Prime [12] | Lattice | | |
| SIKE [5] | Isogeny | | |

Table 2: The round 3 alternatives of the NIST PQC competition.

the maximum value of the coefficients of an element of $\mathcal{R}_q$, and $\sigma$ is the standard deviation of the discrete Gaussian distribution $D_\sigma$ which is used as the error distribution. Typical parameter sets are $(n, q, \sigma) = (256, 7681, 4.51)$ and $(n, q, \sigma) = (512, 12289, 4.86)$, which were introduced in [28]. The LPR-scheme consists of a key generation, encryption and decryption algorithm:

- **KeyGen(a):** Key generation with an input $\mathbf{a} \in \mathcal{R}_q$. Sample two polynomials $\mathbf{r}_1, \mathbf{r}_2 \in \mathcal{R}_q$ from the error distribution $D_\sigma$ and use them to compute $\mathbf{p} = \mathbf{r}_1 - \mathbf{a} \cdot \mathbf{r}_2$. The public key is the pair $(\mathbf{a}, \mathbf{p})$ and the private key is $\mathbf{r}_2$. The polynomial $\mathbf{r}_1$ is no longer needed and should be discarded.
- **Enc(a, p, m):** Encryption with inputs the public key $\mathbf{a}, \mathbf{p} \in \mathcal{R}_q$ and message $\mathbf{m}$. The message $\mathbf{m}$ is first encoded to $\bar{\mathbf{m}}$ via an encoder function. Sample three polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in \mathcal{R}_q$ from the error distribution $D_\sigma$ and compute the polynomials $\mathbf{c}_1 = \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2$ and $\mathbf{c}_2 = \mathbf{p} \cdot \mathbf{e}_1 + \mathbf{e}_3 + \bar{\mathbf{m}}$. The ciphertext that is the output is the pair $(\mathbf{c}_1, \mathbf{c}_2)$.
- **Dec($\mathbf{c}_1, \mathbf{c}_2, \mathbf{r}_2$):** Decryption with inputs the ciphertext pair $\mathbf{c}_1, \mathbf{c}_2$ and the secret key $\mathbf{r}_2$. First compute $\mathbf{m}' = \mathbf{c}_1 \cdot \mathbf{r}_2 + \mathbf{c}_2 \in \mathcal{R}_q$ and decode this via a decoding function to obtain the plaintext $\mathbf{m}$.

The most simple encoder function would be one that maps the coefficients of a binary message $m$ that are 1 to $(q-1)/2$ and the 0 coefficients stay 0. The corresponding decoding function then first maps the coefficients of $\mathbf{m}'$ to the interval $(-q/2, q/2]$ and decodes a coefficient $m_i'$ to 1 if $|m_i'| > q/4$ and to 0 otherwise. We can verify the correctness of the scheme:

$$\begin{aligned}
\mathbf{m}' = \mathbf{c}_1 \cdot \mathbf{r}_2 + \mathbf{c}_2 = (\mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2) \cdot \mathbf{r}_2 + \mathbf{p} \cdot \mathbf{e}_1 + \mathbf{e}_3 + \bar{\mathbf{m}} \\
&= \mathbf{a} \cdot \mathbf{r}_2 \cdot \mathbf{e}_1 + \mathbf{r}_2 \cdot \mathbf{e}_2 + \mathbf{r}_1 \cdot \mathbf{e}_1 - \mathbf{a} \cdot \mathbf{r}_2 \cdot \mathbf{e}_1 + \mathbf{e}_3 + \bar{\mathbf{m}} \\
&= \bar{\mathbf{m}} + \mathbf{r}_2 \cdot \mathbf{e}_2 + \mathbf{r}_1 \cdot \mathbf{e}_1 + \mathbf{e}_3 \approx \bar{\mathbf{m}}
\end{aligned}$$

The approximation follows from the fact that $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{r}_1, \mathbf{r}_2$ are all sampled from an error distribution which have small values for the coefficients, and after decoding this small error vanishes. The security of the scheme is based on RLWE since we see that the ciphertext pair $(\mathbf{c}_1, \mathbf{c}_2)$ consists of two elements that are in the form of the RLWE problem.

## 4.2 The Number Theoretic Transform

In the above RLWE public key encryption scheme we saw that we are working with elements from a polynomial ring $\mathcal{R}_q = \mathbb{Z}[x]/(f)$ with $f$ typically either $x^n \pm 1$. Addition between two elements

$\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$ is a relatively cheap operation since it is element wise, i.e. we compute $\mathbf{c} = \mathbf{a} + \mathbf{b}$ as $c_i \equiv a_i + b_i \mod q$ for $i = 0, ..., n-1$, which has a runtime of $\mathcal{O}(n)$. Multiplication however is not as cheap, since to compute a polynomial multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ naively (i.e. schoolbook) we compute $c'_i = \sum_{j+k=i} a_j b_k \mod q$ for $i = 0, ..., 2n-1$, which has a runtime of $\mathcal{O}(n^2)$, followed by a polynomial reduction. Reducing modulo the polynomial $x \pm 1$ is cheap since it is computed as $c_i = c'_i \mp c'_{i+n}$ for $i = 0, ..., n-1$. There are however more efficient methods for integer multiplication and polynomial multiplications, such as Karatsuba [32], Toom-Cook [64], Fast Fourier Transform [49], Schönhage-Strassen [59], Nussbaumer [48] and the Number Theoretic Transform [1], for an overview of multidigit multiplication methods see [7]. For an analysis of the efficiency of these polynomial multiplication in use for the PQC schemes, see [36]. The Number Theoretic Transform method requires an NTT-friendly ring $\mathcal{R}_q$ and is the preferred choice for the schemes Kyber and Dilithium. Furthermore it has been shown that schemes using unfriendly rings (e.g. NTRU, Saber and NTRU Prime) can still benefit from the NTT [17]. The choice of $q$ defines whether a ring is NTT friendly, for $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$ it is required that $q$ is of the form $k \cdot n + 1$ and for $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ we need $q = 2k \cdot n + 1$.

**Point-value methods and Fast Fourier Transform** The NTT is similar to the Fast Fourier Transform (FFT), which is a point-value method, but it is defined for finite rings. A point value method means that we evaluate a polynomial in sufficient many points such that the evaluations uniquely define the polynomial. This evaluation can be seen as a transformation from the time domain to the frequency domain. Multiplication in the frequency domain is done point-wise. So if we want to multiply two polynomials $\mathbf{a}, \mathbf{b}$ using a transform function $T$, we first transform them both to the frequency domain, multiply them point-wise and then transform this solution back to the time domain:

$$\mathbf{c} = T^{-1}(T(\mathbf{a}) * T(\mathbf{b})).$$

Point-wise multiplication is efficient since it has a runtime of $\mathcal{O}(n)$, so in order for a point-value form method to be efficient, the transform needs to be efficient. Simply evaluating $n$ distinct points will result in a runtime of $\mathcal{O}(n^2)$, but since the points are free to choose, a clever choice can speed up the transform. For the FFT, this choice of points are the $n$ complex $n$th primitive roots of unity. A value $\alpha$ is a $j$th root of unity if $\alpha^j = 1$, and it is primitive if $\alpha^k \neq 1$ for $k = 1, ..., j-1$. The FFT acts on complex values, thus the choice of points to evaluate are the complex $n$th roots $e^{ik2\pi/n}$ for $k = 0, 1, ..., n-1$. Using these points we can apply the efficient Cooley-Tukey algorithm [18], which is a recursive algorithm that via a divide-and-conquer design splits a size $n$ multiplication in two size $n/2$ multiplications. This results in a runtime of $\mathcal{O}(n \log n)$ per transform, and after a small modification this can be used for the inverse transform with also a runtime of $\mathcal{O}(n \log n)$. So for the polynomial multiplication we have 2 forward and one inverse transform, as well as a point-wise multiplication, giving a total runtime of $\mathcal{O}(n \log n)$. The Cooley-Tukey algorithm can be optimized to be an in-place algorithm. See Algorithm 1 for a pseudocode of an in-place FFT via Cooley-Tukey. Here bit-reversal is applied such that the output is in natural order, this can optionally be ignored as long as this is consistent with the inverse transform. Replacing $\omega_m \leftarrow \exp(-2\pi/m)$ by $\omega_m \leftarrow \exp(2\pi/m)$ and a scaling of $n^{-1}$ of the final result gives the algorithm for the inverse FFT. The $\omega$ factors are called the twiddle factors.

**Modifying the FFT to get the NTT** For this thesis, we are interested in the NTT. The NTT is a modified FFT such that it acts on elements from $\mathbb{Z}_q$, a ring of integers with modulus $q$. The

**Algorithm 1** Fast Fourier Transform via in-place Cooley-Tukey

**Input:** Array $\mathbf{a}$ of $n$ complex values
**Output:** Array $\tilde{\mathbf{a}}$ FFT transform of $\mathbf{a}$

```
bit-reverse-copy(a, ã)
n ← a.length
for s = 1 to log n do
    m ← 2^s
    ω_m ← exp(−2π/m)
    for k = 0 to n − 1 by m do
        ω ← 1
        for j = 0 to m/2 − 1 do
            t ← ω ã[k + j + m/2]
            u ← ã[k + j]
            ã[k + j] ← u + t
            ã[k + j + m/2] ← u − t
            ω ← ω ω_m
        end for
    end for
end for
```

complex $n$th roots of unity of the FFT need to be replaced by $n$th roots of unity from $\mathbb{Z}_q$. In order for this to exist, $q$ needs to be of the form $k \cdot n + 1$. If $r$ is a primitive $(q-1)$th root of unity, then a $n$th root of unity $\alpha$ can be computed as $\omega_n \equiv r^k \mod q$. We can then compute $n$ of these $n$th roots of unity as $\omega_n^j$ for $j = 0, 1, ..., n-1$. Using these roots as twiddle factors we have an optimal in-place NTT algorithm via Cooley-Tukey. This algorithm has a nice structure to it, since it consists of many so-called butterfly operations. Every butterfly has as a fixed twiddle factor $\omega$, as input two values $x_1$ and $x_2$, and it outputs two values $y_1$ and $y_2$, which are computed as $x_1 + \omega x_2 \equiv y_1 \mod q$ and $x_1 - \omega x_2 \equiv y_2 \mod q$. See Figure 10 for a visualization of a single butterfly operation. The in-place NTT algorithm consists of $\log_2 n$ layers of butterflies, with in each layer $n/2$ butterflies, see figure 11 for an example with $n = 8$. For a pseudocode of the NTT and its inverse, see Algorithm 2, where we assume for optimization purpose that the twiddle factors are precomputed and act as input. This NTT algorithm, which we will denote by $NTT(\cdot)$ and its inverse $INTT(\cdot)$, can be directly used for multiplications in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$:

$$\mathbf{c} = INTT(NTT(\mathbf{a}) * NTT(\mathbf{b}))$$

See Appendix A for an example of a polynomial multiplication in this ring with $n = 8$ and $q = 17$. For polynomial multiplications in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, we need to modify the algorithm. The inputs $\mathbf{a}$ and $\mathbf{b}$ need a scaling to obtain $\mathbf{a}'$ and $\mathbf{b}'$, the NTT and INTT need to be modified to work with $2n$th roots of unity as the twiddle factors, and the output $\mathbf{c}'$ after the INTT needs a scaling. We will denote the NTT and inverse NTT with the $2n$th roots of unity by $NTT_{\omega_{2n}}$ and $INTT_{\omega_{2n}}$ respectively. Note that this requires that the prime value $q$ is of the form $q = 2k \cdot n + 1$.

In total we find:

$$\mathbf{a}': \quad a'_i = a_i \cdot \omega_{2n}^i, \quad i = 0, ..., n-1$$
$$\mathbf{b}': \quad b'_i = b_i \cdot \omega_{2n}^i, \quad i = 0, ..., n-1$$
$$\mathbf{c}' = INTT_{\omega_{2n}}(NTT_{\omega_{2n}}(\mathbf{a}') * NTT_{\omega_{2n}}(\mathbf{b}'))$$
$$\mathbf{c}: \quad c_i = c'_i \cdot n^{-1} \cdot \omega_{2n}^{-i}, \quad i = 0, ..., n-1$$

**Algorithm 2** Fast NTT via in-place Cooley-Tukey

**Input:**
  Array $\mathbf{a} \in \mathbb{Z}_q^n$ of $n$ values
  Array $\boldsymbol{\omega}$ of $n/2$ twiddle factors
**Output:** Array $\tilde{\mathbf{a}} \in \mathbb{Z}_q^n$, NTT transform of $\mathbf{a}$
  bit-reverse-copy($\mathbf{a}, \tilde{\mathbf{a}}$)
  $n \leftarrow \mathbf{a}.length$
  **for** $s = 1$ **to** $\log_2 n$ **do**
    $m \leftarrow 2^s$
    **for** $k = 0$ **to** $n - 1$ **by** $m$ **do**
      **for** $j = 0$ **to** $m/2 - 1$ **do**
        $t \leftarrow \boldsymbol{\omega}[n/m * j] * \tilde{\mathbf{a}}[k + j + m/2] \mod q$
        $u \leftarrow \tilde{\mathbf{a}}[k + j]$
        $\tilde{\mathbf{a}}[k + j] \leftarrow u + t \mod q$
        $\tilde{\mathbf{a}}[k + j + m/2] \leftarrow u - t \mod q$
      **end for**
    **end for**
  **end for**

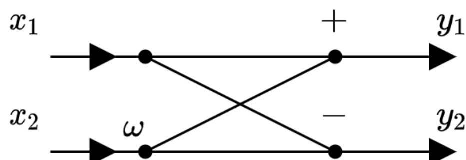


Figure 10: Visualization of a single butterfly operation in an in-place Cooley-Tukey NTT.

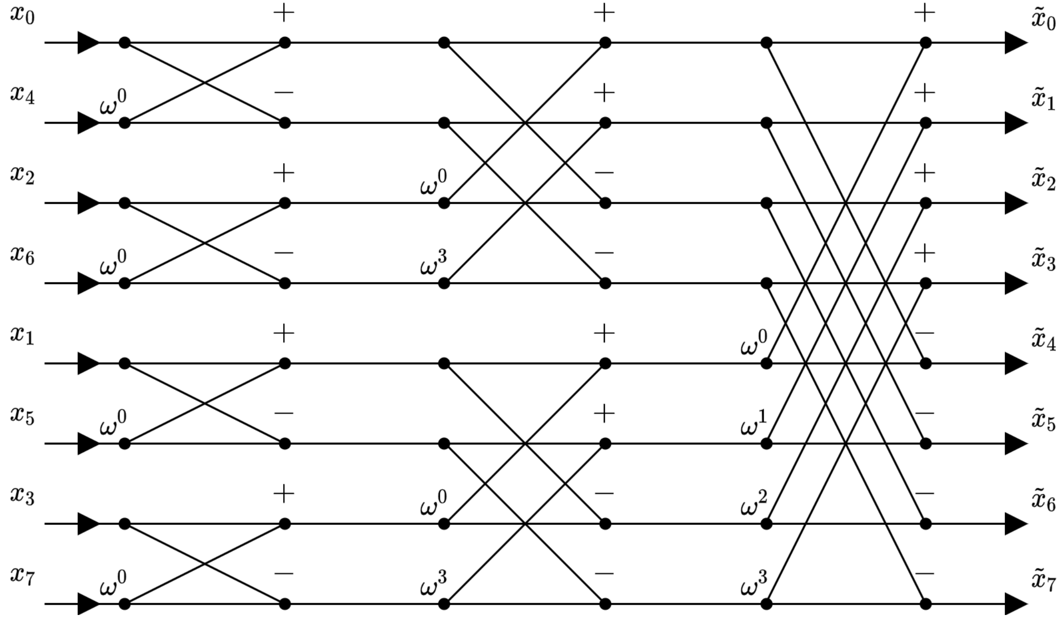


Figure 11: Visualization of an in-place Cooley-Tukey NTT as a network of butterflies, with $n = 8$.

## 4.3 Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption

In 2017, Primas and Mangard [53] demonstrated the first successful Soft Analytical Side-Channel Attack on an NTT. The target scheme is the LPR-scheme as described in Section 4.1, with parameters $n = 256$, $q = 7681$ and $\sigma = 4.51$. From this scheme the targeted set of operations is the inverse NTT in the decryption algorithm, where it is assumed that the polynomial multiplication $\mathbf{c}_1 \cdot \mathbf{r}_2$ is computed via the NTT as $INTT_{\omega_{2n}}(NTT_{\omega_{2n}}(\mathbf{c}_1) * NTT_{\omega_{2n}}(\mathbf{r}_2))$. This means that the input to the targeted INTT is $\tilde{\mathbf{c}}_\mathbf{1} * \tilde{\mathbf{r}}_\mathbf{2}$, where we can assume that $\tilde{\mathbf{c}}_\mathbf{1}$ is known and $\tilde{\mathbf{r}}_\mathbf{2}$ is the secret key that we want to obtain. The first step is transforming the operations that occur in an (I)NTT to a factor graph. To do that, we look at a single butterfly operation, since the (I)NTT is a structured collection of butterfly operations. See Figure 12 to see how in this attack three operations of a butterfly (i.e. the multiplication with twiddle factor, the addition and subtraction) are transformed into a factor graph.

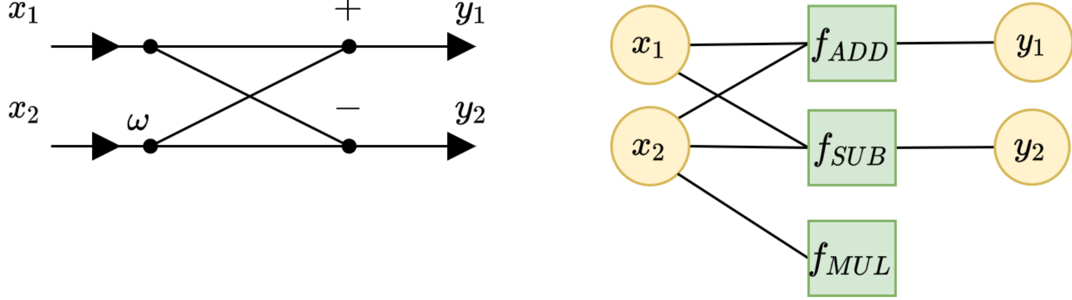


Figure 12: A single butterfly (left) and its factor graph as in [53] (right).

We have $\frac{n}{2}\log_2 n = 1024$ of these butterflies, totalling 3072 factor nodes, and $n(1+\log_2 n) = 2304$ variable nodes. The leakage is assumed to come from the modular multiplication, which happens when $x_2 \cdot \omega$ is computed. This means that in every layer of variables only half the variables get leakage information directly via a factor node, except for the final layer which gets no information at all. Furthermore it is assumed that via a timing side-channel the attacker knows when a modular reduction took place after the addition or subtraction. This gives the following five functions:

$$f_{MUL}(b) = \mathbb{P}[x_2 = b|l]$$

$$f_{ADD_{Red}}(a,b,c) = \begin{cases} 1 & \text{if } a + b\omega \equiv c \mod q \text{ and } a + (b\omega \mod q) \geq q \\ 0 & \text{otherwise} \end{cases}$$

$$f_{ADD_{NoRed}}(a,b,c) = \begin{cases} 1 & \text{if } a + b\omega \equiv c \mod q \text{ and } a + (b\omega \mod q) < q \\ 0 & \text{otherwise} \end{cases}$$

$$f_{SUB_{Red}}(a,b,d) = \begin{cases} 1 & \text{if } a - b\omega \equiv d \mod q \text{ and } a - (b\omega \mod q) < 0 \\ 0 & \text{otherwise,} \end{cases}$$

$$f_{SUB_{NoRed}}(a,b,d) = \begin{cases} 1 & \text{if } a - b\omega \equiv d \mod q \text{ and } a - (b\omega \mod q) \geq 0 \\ 0 & \text{otherwise,} \end{cases}$$

Here we use the notation of $a, b, c$ and $d$ representing some element of the domain that $x_1, x_2, y_1$ and $y_2$ respectively are defined on. The attack is demonstrated both as a real physical attack as well as a simulated attack. The target device is an ARM Cortex-M4F microcontroller, specifically the Texas Instruments MSP432 microcontroller on a MSP432P401R LaunchPad. A near-field probe is used to obtain information via an EM side-channel. There are $\frac{n}{2} = 128$ different twiddle factors and $q = 7681$ values for $x_2$, so for the template building step there are $\frac{qn}{2} = 983168$ different templates needed. To make the templates, 100 traces are prerecorded, totalling in roughly 100 million traces. Note that only a single trace needs to be recorded from the target device to be used in the template matching step. For the simulated leakage the Hamming weight leakage is used with 2 samples, one before and one after multiplication:

$$l = HW(b) + \mathcal{N}(0,\sigma^2) || HW(b\omega_n^i \mod q) + \mathcal{N}(0,\sigma^2).$$

Thus for the simulated leakage a 2-variate template matching is used. For the BP algorithm, the slowest step is the message creation that the addition and subtraction factor nodes perform. Naively this step has a runtime of $\mathcal{O}(q^3)$, but can easily be reduced to $\mathcal{O}(q^2)$, see section 5.3 for a more in-depth analysis. This can be reduced to $\mathcal{O}(q \log q)$ for the specific factor graph via an FFT trick, see Section 6.2 for more details on this FFT trick. Applying the BP algorithm to the full graph gives no good results, which is caused by the uneven availability of the side-channel information as well as the observation that when the twiddle factor $\omega_n^0 = 1$ is used, less information is obtained. To overcome this the BP algorithm is applied to three smaller subgraphs such that the amount of side-channel information is relatively high in every sub-graph. Using this method an attacker will get in the best case the correct values of 192 out of 256 variables in a single layer. The other 64 values are then obtained via a post-processing step where they use lattice decoding via the lattice reduction algorithm BKZ [58]. As long as the recovered 192 intermediate values are correct, the post-processing step will always give the correct result. For the real device attack, 20 iterations of the BP algorithm are used and all the experiments were successful. For the simulated leakage the attack methods are tested with increasing values for the HW-leakage parameter $\sigma$, which results in a successful attack for up to $\sigma = 0.4$. Finally the attack is also tested on masked implementations which gave similar results.

## 4.4 More Practical Single-Trace Attacks on the Number Theoretic Transform

As a follow up of the above publication, Pessl and Primas [52] improved the attack method in 2019. Rather than the RLWE based LPR-scheme, the target is the MLWE based scheme Kyber, which is one of the NIST PQC finalists. For the targeted parameter set of Kyber768 we have the parameters $n = 256$ as before, $k = 3$ for the module dimension, $q = 7681$ as the field size and $\eta = 4$ for the error distribution which is used in Kyber, namely the centered binomial distribution. The target NTT is the one in the encryption algorithm of Kyber, which computes the NTT of a sampled noise $\mathbf{r} \in \mathcal{R}_q^k$. From this value $\mathbf{r}$ the plaintext can be computed. This means that the target secret is an ephemeral secret and the attack has to be redone for every message that an attacker wants to recover. The benefit of this target is that $\mathbf{r}$ is defined on a much smaller support of $[-\eta, \eta]$ due to it being sampled from a centered binomial distribution, compared to a support of $\mathbb{Z}_{7681}$ that all other (I)NTT inputs are defined on. Since $\mathbf{r}$ is a vector of $k$ polynomials, every polynomial needs to be targeted separately. To improve the previous work, its issues are pinpointed and improved on:

- As can be seen in Figure 12, the factor graph will contain many loops of length 4, which is the worst case since for a bipartite graph the minimal loop length is 4. To overcome this, the addition and subtraction factor nodes are merged into a single butterfly factor node. The downside is that the fast FFT trick for message creation is not possible anymore. This decreases the number of loops in the total graph and only gives loops of length at least 8, which will help for the marginal approximation of the LBP algorithm. This gives a new function $f_{BF}$ defined on 4 variables:

$$f_{BF}(a, b, c, d) = \begin{cases} 1 & \text{if } a + b\omega = c \mod q \text{ and } a - b\omega = c \mod q, \\ 0 & \text{otherwise.} \end{cases}$$

  Furthermore assuming the knowledge of whether a modular reduction took place is removed, since this requires a timing side-channel which in practice is often not possible due to constant

time implementations.

- As already noted in the previous work, the side-channel information is unevenly spread. Rather than obtaining side-channel information from the modular multiplication, it can be obtained from load/store operations, such that now every variable gets added information via a leakage factor node. See Figure 13 for the improved factor graph of a single butterfly.
- Targeting the load or store operations reduces the number of required precomputed traces since the different twiddle factors do not have to be taken into account. This number is reduced further by using Hamming weight templates, i.e. only making a template for every different Hamming weight. This reduces the number of templates from a hundred million down to a few hundred.
- The message passing scheme within BP is improved, such that in every single iteration every pair of variables communicate with each other. Due to this improved scheme, the number of required iterations of BP is greatly reduced. See Section 5.1 for more information on message passing schemes for the NTT.
- To improve the approximation of the computed marginals in case of oscillations, message damping is used. This means that to create a new messages a weighted average of new and old incoming messages are used.

As noted above, the smaller support of $[-\eta, \eta]$ of the input of the target NTT increases the success rate of the attack. The attack of Section 4.3 gives a high success rate for up to $\sigma = 0.9$. After applying all of the above improvements, this parameter can be increased up to $\sigma = 1.5$. The improved attack is also demonstrated on a masked implementation, which is also improved by not targeting the shares separately, but in a combined factor graph. Even with no noise ($\sigma = 0$), the independent factor graphs have only a small success probability, while for the combined factor graph a noise of up to $\sigma = 0.3$ gives a high success probability.

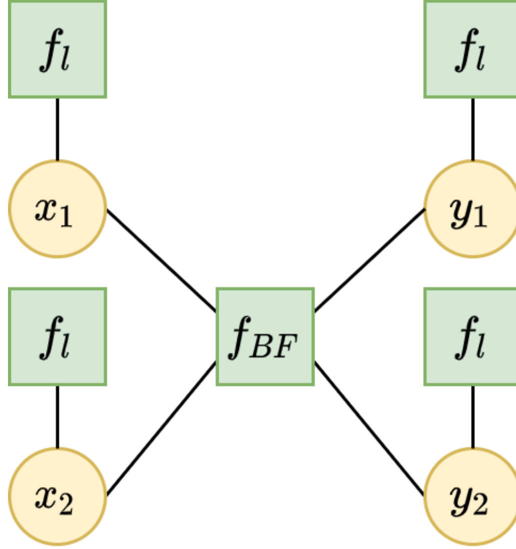


Figure 13: The factor graph of a single butterfly as used in [29, 52] and Section 5

### 4.5 Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber

In 2021, another paper by Hamburg, Hermelink, Primas, Samardjiska, Schamberger, Streit, Strieder and van Vredendaal [29] was published on SASCA attacks on NTT structures. They present further improvements on the attacks from Sections 4.3 and 4.4. This is achieved due to a new approach, where the ciphertext is not arbitrarily chosen, but it is constructed such that it gives a sparse (I)NTT input. The target scheme is Kyber, as in the attack from Section 4.4, but the target NTT is the INTT in the decryption algorithm, similar to the attack from Section 4.3. The targeted Kyber parameter sets use $q = 3329$ for the field size, which lacks a 512th root of unity, so the NTT algorithm is also slightly modified to have one less layer and the pointwise multiplication becomes a pairwise-pointwise multiplication. The input to the targeted INTT is $\tilde{\mathbf{s}} * \tilde{\mathbf{u}}$, where $\tilde{\mathbf{s}} \in \mathcal{R}_q^k$ is the secret key in the NTT domain and $\tilde{\mathbf{u}} \in \mathcal{R}_q^k$ is the compressed ciphertext in the NTT domain. Since $\tilde{\mathbf{u}}$ is both compressed and in the NTT domain, it is not trivial to create a sparse INTT input by being free to choose the ciphertext. They propose two methods: the first method uses the BKZ algorithm, and finds sparse vectors such that the non-zero values are randomly distributed. Their second approach is much faster but has the non-zero values in contiguous blocks, which turns out to be detrimental for the BP algorithm. For the BP step, the attack is similar to the attack of Section 4.4, but without the message damping. If the BP step is successful, an attacker learns the value of $\tilde{\mathbf{u}} * \tilde{\mathbf{s}}$, but due to the sparseness of $\tilde{\mathbf{u}}$ the attacker does not know all values of $\tilde{\mathbf{s}}$ yet. To obtain the remaining values of $\tilde{\mathbf{s}}$ the BKZ algorithm can be applied. The number of non-zero values of the NTT-input needs to be chosen such that the post-processing step is successful, which will also depend on which parameter set is being used, as well as the number of required traces. Using the sparse inputs on as many inputs as possible, the attacks were successful with high probability with noise up to $\sigma = 2.2$.

# 5 Experimental results of simulated SASCA attacks an NTT structure

In this section we will describe experimental results of Soft Analytical Side-Channel Attacks on the Number Theoretic Transform. Section 5.1 explains the details of our implementation including the relevant parameters, the structure of the factor graph and the message passing scheme. Then in Section 5.2 we analyze the performance of the attack by varying the relevant parameters. Finally in Section 5.3 we analyze the complexity of the belief propagation algorithm in this specific case.

## 5.1 Details on the implementation

We do all of our implementations in Python 3, and eliminate the usage of any physical target device by simulating the template matching step as described in Section 3.1. The NTT structure we target is the one that implements a transform used for multiplying elements in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$, since an NTT for this polynomial ring does not require any additional scaling and more values for $q$ can be used. We furthermore assume that $q$ is chosen such that a full NTT is possible (i.e. $q$ is of the form $k \cdot n + 1$). For the factor graph representing this NTT for Belief Propagation we choose one consisting of butterfly nodes and leakage factor nodes on all variables, as in Figure 13 which is also used in [29, 52].

**Parameters** In order to get experimental results, we can use smaller parameters than used in practice such that we can generate results in a reasonable time. We have the following parameters:

- The degree $n$ of the quotient polynomial in the polynomial ring, which is an integer and for simplicity we assume it is a power of 2. If we define an element $\mathbf{a} \in \mathcal{R}_q$ as a vector of its coefficients, then $n$ is the length of this vector. Therefore $n$ also defines the number of input values to the NTT, which consequently defines the number of variable and factor nodes in the factor graph. The factor graph contains $\log_2(n) + 1$ layers of variable nodes, with $n$ variables per layer, totaling both $n(\log_2(n) + 1)$ variable nodes and $n(\log_2(n) + 1)$ factor leakage nodes. Between every layer there are $\frac{n}{2}$ butterfly factor nodes, totalling $\frac{n}{2} \log_2(n)$ butterfly factor nodes.
- The prime $q$ which is defined as the modulus of the integers of the coefficients in the polynomial ring $\mathcal{R}_q$. For the usage of NTT it is required that $q$ is of the form $q = k \cdot n + 1$ for some value $k \in \mathbb{N}$ such that $q$ is prime. The value of $q$ determines the size of the domain of the intermediate values, and consequently also defines the size of the messages sent in the BP algorithm.
- The leakage type, we consider both an ID-leakage and HW-leakage. In the template matching step ID-leakage is sampled as $x + y$ where $x \in \mathbb{Z}_q$ is the true value and $y$ is sampled from $\mathcal{N}(0, \sigma_{ID}^2)$, and similarly HW-leakage is sampled as $HW(x) + y$ with y sampled from $\mathcal{N}(0, \sigma_{HW}^2)$. We choose these two leakage models, since the Hamming weight model is considered a realistic model since it turns out that in practice the power or EM radiation measurements have a strong correlation with the Hamming weight. Furthermore, a template has to be made only for every possible Hamming weight, which makes a real attack significantly more practical. We also consider the ID-leakage type since this can be considered an ideal situation, where every possible value can be distinguished by the templates.

- The amount of noise, which is regulated by $\sigma$, where we make a distinction for the different leakage types. A higher value means more noise, which reduces the amount of side-channel information, making it harder for an attacker to succeed.

**Random input and termination conditions** Besides these parameters, we need to choose an input to the NTT. To do so we sample an element $\mathbf{a} \in \mathcal{R}_q$ by sampling its $n$ coefficients uniformly and independently from $\mathbb{Z}_q$. The true values of all the other intermediate values are computed by evaluating the NTT and saving all the values after a butterfly operation. Note that the true values are only used to simulate the template matching and are later used for verification of correctness, an attacker does not know any of these values. Our factor graph will contain loops of length at least 8 when $n \geq 4$, see Figure 14. Therefore we use Loopy BP, for which we need to determine a termination condition. We use the following 3 conditions:

- If the current guess for all intermediate values at the end of an iteration is correct we let it terminate. To verify this we use our knowledge of the true values, which a real attacker does not have. It may however be possible for an attacker to verify correctness indirectly, depending on what secret they obtain. For example, if the attacker learns the secret key from the NTT input, they can verify this key if they have a plaintext-ciphertext pair.

- With the above condition, BP would never terminate in the case when it does not converge to the correct solution. Therefore we check the absolute difference of the marginals of all intermediate between the last 2 iterations, and we terminate if the sum of all these marginals is below some threshold value.

- In the case that the marginals oscillate or some other issue occurs, we set a maximum number of iterations such that BP will eventually always stop. It is important that we set this value high enough such that it will not terminate due to this condition in a case where it would have still converged to the correct solution when given more iterations.

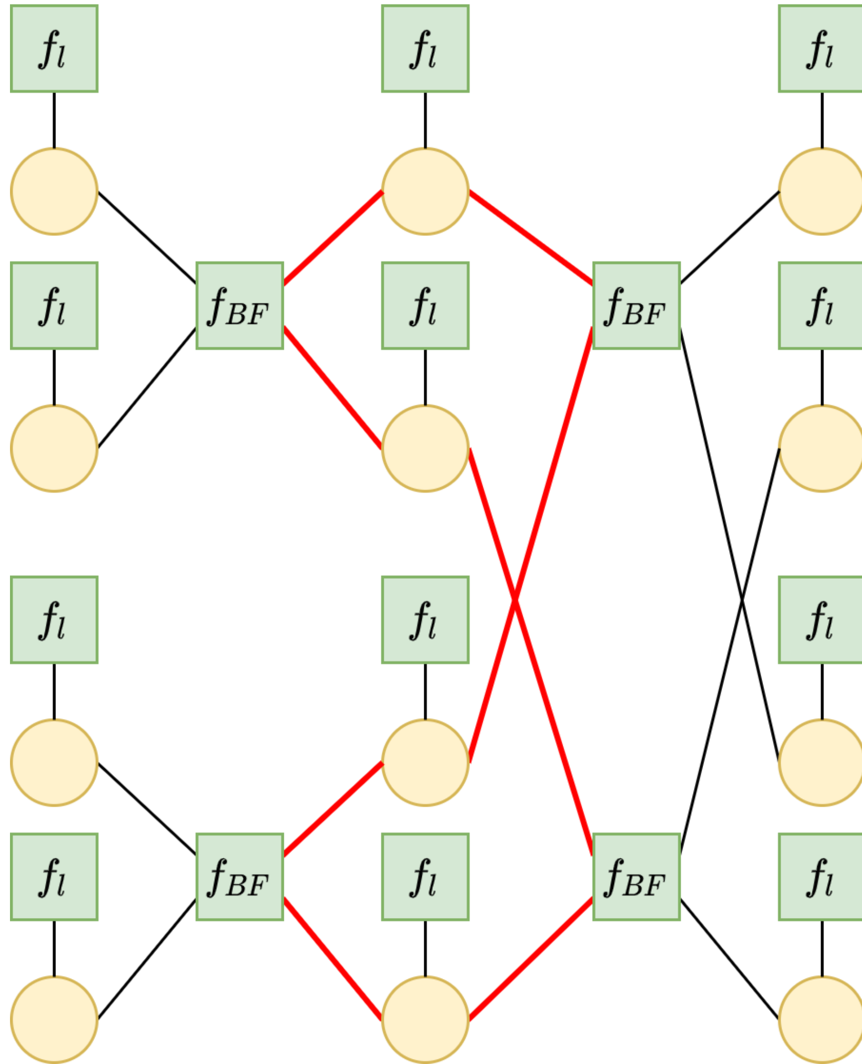


Figure 14: The factor graph of our implementation in the case $n = 4$, with a loop of length 8 highlighted in red.

**Success criterion** After BP has terminated we have the computed marginals for every intermediate value, which is a vector containing probabilities. From these probabilities we compute the guess, which is the value with the highest probability. Note that only a single layer of variables need to be guessed correctly, since all remaining values can be reconstructed from a single layer (though an attacker will only be interested in either the input or output of the NTT, i.e. the first or last layer of variables). Therefore we consider an attack to be successful if at least one layer of variables is correctly guessed. Since both the true values of the intermediates and the noise are randomly sampled, the success of an attack can depend on these values. So to get accurate results

we do multiple runs with the same parameter sets but with different input and noise and average the success rate over all runs.

**Message passing scheme** As explained in Section 3.2, the default order for sending messages in loopy BP is initializing all the outgoing messages from the variables as uniform, followed by sending messages from all factor nodes. However, simply adopting this message passing scheme will result in unnecessarily many iterations until convergence, as has been noted by [52]. This is because in the default message passing scheme, first all factor nodes send messages and then all variable nodes send their messages. This causes for only neighbouring layers of variables to communicate in a single iteration. We therefore use the message passing scheme from [52], which can be described as a layer-by-layer scheme, where messages are passed per layer from the left to the right and then back again. See Figure 15 for a visualization of this scheme on a factor graph with $n = 4$. Note that no messages are sent back to the leakage factor nodes since these messages will never be used. Using this message passing scheme all variables communicate with each other in a single iteration, resulting in significantly less iterations needed until convergence if compared to the default message passing scheme.

**Hardware used for simulations** Our implementation of the attack is made such that it runs on a single core, however we use a 12-core AMD Ryzen 9 3900XT, meaning that when doing multiple simulations we can run 12 of them in parallel.

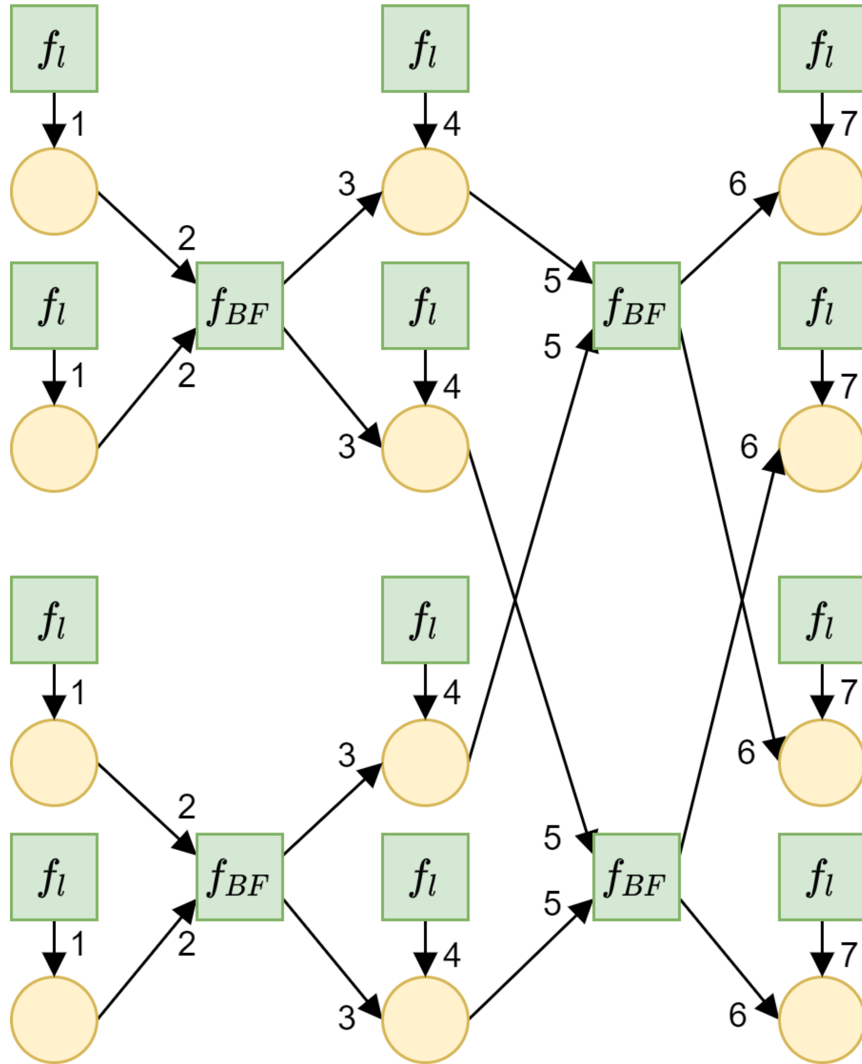


Figure 15: The message passing scheme for BP in the case of an NTT with $n = 4$, where the messages are sent per layer from left to right in order from 1 up to 7, and then back from right to left by reversing the directions of numbers 6, 5, 3 and 2, respectively.

### 5.2 Simulations to test performance

First we want to analyze the effect that the parameters have on the success rate of an attack. We start by fixing a value for $q$ and use increasing values of $n$ and test for convergence with HW-leakage for various values of $\sigma_{HW}$. Next we fix a value of $n$ and use increasing values of $q$ and test convergence for both ID and HW-leakage.

**The effect of $n$** A higher value for $n$ means that there are more intermediate values that we gain side-channel information on and more information on how the intermediate values are connected, thus we expect that this would result in a higher success rate. The only disadvantage is that more intermediate values need to be guessed correctly. The smallest possible value is $n = 2$ which is an NTT with only a single butterfly, and the highest value we see in practice being used is 512, so we test all powers of 2 between 2 and 512. We choose to fix $q = 97 = 3 \cdot 2^5 + 1$, note that for $n$ larger than $2^5 = 32$ this choice does not make sense in terms of an NTT, but for testing performance this does not matter as long as $q$ is a prime value. For the leakage we use HW-leakage with $\sigma_{HW} = 0.1$ up to 0.8 in steps of 0.1. We do 120 simulations per parameter set with different random inputs and leakage values and compute the average success rate over the 120 runs, see the results in Figure 16, which confirms our expectation that a larger $n$ has a positive effect on the success rate.

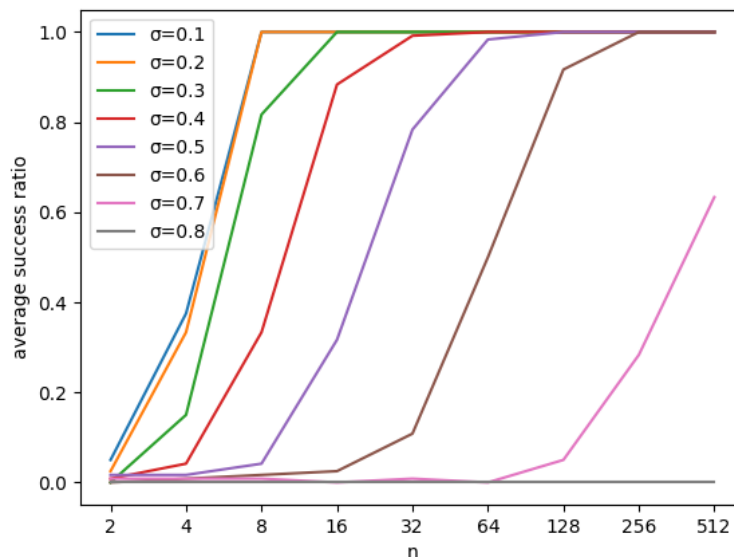


Figure 16: Average success rate over 120 runs per parameter set plotted against increasing values of $n$ in powers of 2, using $q = 97$ and HW-leakage with various values of $\sigma_{HW}$.

**The effect of $q$ and leakage type** Next we want to test the effect of $q$ on the average success rate of the attack. First we test this with a HW-leakage, and compare this with an ID-leakage since it can be expected that the results are different based on the leakage type. To see this, compare Figure 3 and 4, and imagine if we were to increase the value of $q$. For the ID-leakage the candidate values with a non-negligible probability will be centered close to the true value, independent of the size of $q$. However, for the HW-leakage every value that has the same or close to the same Hamming weight as the true value will get a non-negligible probability, and with a larger value of $q$ the true value can be larger for which there will be more values with a same or close to the same Hamming weight value. We choose to fix $n = 32$ and take prime values of $q$ of the form $q = k \cdot 32 + 1$. See Figure 17 for the results in the case of a HW-leakage and note that as expected the average success

rates will go to zero for increasing value of $q$ for every value of $\sigma_{HW}$. For an ID-leakage see Figure 18 and note that in this case the success rate does not go to zero for increased values of $q$.

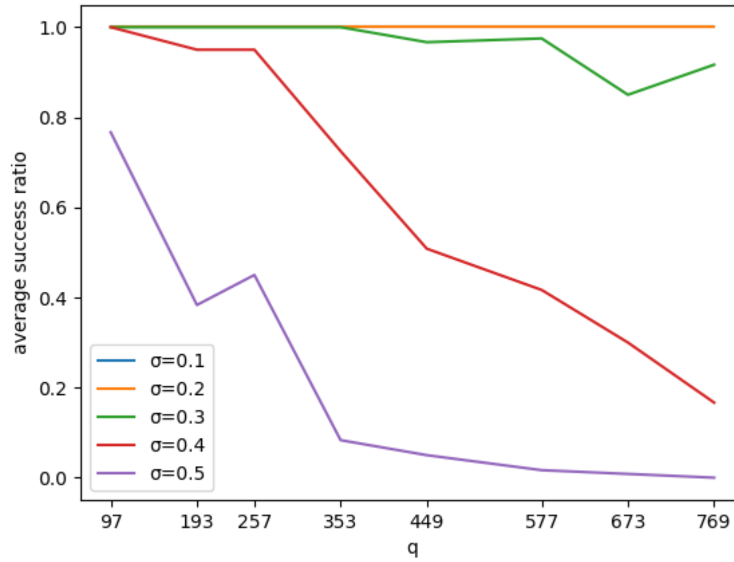


Figure 17: Average success rate over 120 runs per parameter set plotted against increasing values of $q$, using $n = 32$ and HW-leakage with various values of $\sigma_{HW}$.

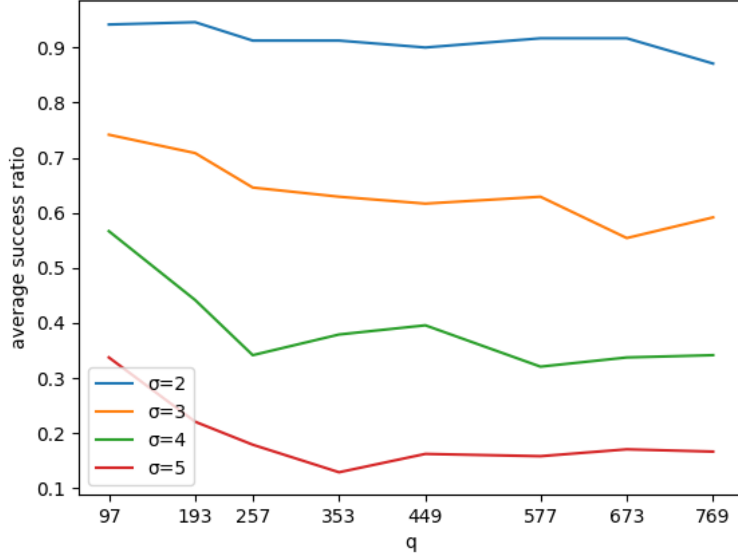


Figure 18: Average success rate over 240 runs per parameter set plotted against increasing values of $q$, using $n = 32$ and ID-leakage with various values of $\sigma_{ID}$.

### 5.3 Complexity and butterfly message creation

To study the complexity of the BP algorithm on an NTT we summarize the most important parts of how BP works: the BP algorithm is an iterative message passing algorithm, where at every iteration every variable node sends a messages to all its neighboring factor nodes and vice versa, via the following two rules,

$$\mathbf{q}_{n\to m}[x_n] = \prod_{m'\in\mathcal{M}(n)\setminus m} \mathbf{r}_{m'\to n}[x_n],$$

$$\mathbf{r}_{m\to n}[x_n] = \sum_{\mathbf{x}_m\setminus n} \left( f_m(\mathbf{x}_m) \prod_{n'\in\mathcal{N}(m)\setminus n} \mathbf{q}_{n'\to m}[x_{n'}] \right).$$

Note that every variable node has a domain $\mathcal{D} = \{0, ..., q-1\}$ of size $q$, and thus all messages that are sent during the algorithm are vectors of length $q$. Creating the messages that the variables send to factor graphs has a low complexity, since every node is connected to either 1 (for variable nodes at the edge layers) or 2 (for variable nodes at the center layers) butterfly nodes and 1 leakage node, thus an outgoing message is either one of the messages in its inbox, or is computed as the element-wise product of 2 messages in its inbox. Therefore the complexity of this step is $\mathcal{O}(q)$ per variable node, or $\mathcal{O}(n\log_2(n)q)$ for all variable nodes in one BP iteration. Since leakage factor nodes only have one neighbor, the message they sent simply is the leakage vector. The rule for generating messages from factors to variables unfortunately is not as fast for the butterfly nodes. Every butterfly has 4 neighbors, which we denote by $x_1, x_2, y_1$ and $y_2$ as in Figure 13, and assume

that the butterfly is sending a message to $x_1$. Then for every value $a$ in the domain of $x_1$ we need to compute

$$\mathbf{r}_{BF\to x_1}[a] = \sum_{b,c,d\in\{0,1,\dots,q-1\}} f_{BF}(a,b,c,d)\cdot \mathbf{q}_{x_2\to BF}[b]\cdot \mathbf{q}_{y_1\to BF}[c]\cdot \mathbf{q}_{y_2\to BF}[d].$$

Doing this computation naively like this, the sum goes over $q^3$ value combinations via $b, c$ and $d$, and this needs to be computed for all $q$ values of $a$, giving a total complexity of $q\cdot\mathcal{O}(q^3) = \mathcal{O}(q^4)$. However, note that $f_{BF}$ is 0 on most inputs. More precisely, if we fix 2 inputs, only exactly one input combination of the remaining $q^2$ combinations will result in an output of 1. We can compute this combination by using the relations between the variables: $x_1 + x_2\omega \equiv y_1 \mod q$ and $x_1 - x_2\omega \equiv y_2 \mod q$. In this example we go over all values of $x_1$ already via $a$, and can choose to go over all values of $x_2$ via $b$ and use the above 2 equations to compute the values for $y_1$ and $y_2$ as $c$ and $d$ respectively, such that $f_{BF}(a,b,c,d) = 1$. So in the example of sending to $x_1$ we get:

$$\begin{aligned}\mathbf{r}_{BF\to x_1}[a] &= \sum_{b,c,d\in\{0,1,\dots,q-1\}} f_{BF}(a,b,c,d)\cdot \mathbf{q}_{x_2\to BF}[b]\cdot \mathbf{q}_{y_1\to BF}[c]\cdot \mathbf{q}_{y_2\to BF}[d] &(2)\\ &= \sum_{b\in\{0,1,\dots,q-1\}} \mathbf{q}_{x_2\to BF}[b]\cdot \mathbf{q}_{y_1\to BF}[a+b\omega \mod q]\cdot \mathbf{q}_{y_2\to BF}[a-b\omega \mod q]. &(3)\end{aligned}$$

Using this method we get a complexity of $q\cdot\mathcal{O}(q) = \mathcal{O}(q^2)$ per butterfly node, so a total complexity of $\mathcal{O}(n\log_2(n)q^2)$ per iteration. Since this is the dominating step, this complexity is also the complexity of one iteration of BP. See Algorithm 3 for the pseudo-code of how this trick is used to compute the outgoing messages from the butterfly node.

**Algorithm 3** Butterfly message creation

**Input:**

The recipient: $s \in \{x_1, x_2, y_1, y_2\}$,

The three incoming messages: $\mathbf{q}_{x_1 \to BF}$, $\mathbf{q}_{x_2 \to BF}$, $\mathbf{q}_{y_1 \to BF}$, $\mathbf{q}_{y_2 \to BF}$ except for $\mathbf{q}_{s \to BF}$,

The twiddle factor $\omega$ and modulus $q$

**Output:** The outgoing message to the recipient: $\mathbf{r}_{BF \to s}$

$\mathbf{r}_{BF \to s} \leftarrow [0, 0, ..., 0]$ of length $q$
**if** $s = x_1$ **then**
  **for** $b \in \{0, 1, ..., q-1\}$ **do**
    $b' \leftarrow b \cdot \omega \mod q$
    **for** $a \in \{0, 1, ..., q-1\}$ **do**
      $c \leftarrow a + b' \mod q$
      $d \leftarrow a - b' \mod q$
      $\mathbf{r}_{BF \to s}[a] \mathrel{+}= \mathbf{q}_{x_2 \to BF}[b] \cdot \mathbf{q}_{y_1 \to BF}[c] \cdot \mathbf{q}_{y_2 \to BF}[d]$
    **end for**
  **end for**
**end if**
**if** $s = x_2$ **then**
  **for** $b \in \{0, 1, ..., q-1\}$ **do**
    $b' \leftarrow b \cdot \omega \mod q$
    **for** $a \in \{0, 1, ..., q-1\}$ **do**
      $c \leftarrow a + b' \mod q$
      $d \leftarrow a - b' \mod q$
      $\mathbf{r}_{BF \to s}[b] \mathrel{+}= \mathbf{q}_{x_1 \to BF}[a] \cdot \mathbf{q}_{y_1 \to BF}[c] \cdot \mathbf{q}_{y_2 \to BF}[d]$
    **end for**
  **end for**
**end if**
**if** $s = y_1$ **then**
  **for** $b \in \{0, 1, ..., q-1\}$ **do**
    $b' \leftarrow b \cdot \omega \mod q$
    **for** $a \in \{0, 1, ..., q-1\}$ **do**
      $c \leftarrow a + b' \mod q$
      $d \leftarrow a - b' \mod q$
      $\mathbf{r}_{BF \to s}[c] \mathrel{+}= \mathbf{q}_{x_1 \to BF}[a] \cdot \mathbf{q}_{x_2 \to BF}[b] \cdot \mathbf{q}_{y_2 \to BF}[d]$
    **end for**
  **end for**
**end if**
**if** $s = y_2$ **then**
  **for** $b \in \{0, 1, ..., q-1\}$ **do**
    $b' \leftarrow b \cdot \omega \mod q$
    **for** $a \in \{0, 1, ..., q-1\}$ **do**
      $c \leftarrow a + b' \mod q$
      $d \leftarrow a - b' \mod q$
      $\mathbf{r}_{BF \to s}[d] \mathrel{+}= \mathbf{q}_{x_1 \to BF}[a] \cdot \mathbf{q}_{x_2 \to BF}[b] \cdot \mathbf{q}_{y_1 \to BF}[c]$
    **end for**
  **end for**
**end if**

# 6 Improving SASCA attacks on an NTT structure

In this section we will describe our new contributions by using our implementation described in Section 5 to improve Soft Analytical Side-Channel Attacks on an NTT structure. Improving such an attack can be done in two directions, namely improving the success rate and improving the speed of the attack. Improving the success rate in this context means the ability to recover all the secret values with a larger amount of noise in the template matching step. After analyzing known work in Section 4 and general behaviour of the parameters in Section 5, we can formulate the two following research questions:

- Is there more information available from the input of the (I)NTT that can be used to increase the success rate even further?
- Is it possible to improve the performance of the attack in terms of speed, without losing performance in the success rate?

To answer the first question, we analyze some ideas in Section 6.1. To answer the second question, we first explain here why improving the speed is important. The motivation of using the BP algorithm in a SASCA attack is because of its speed, since the aim of a SASCA attack is having both a low data complexity (compared to D&C attacks) and computation complexity (compared to algebraic side-channel attacks). However, in the case of targeting an NTT structure in a lattice-based post-quantum cryptography scheme, the BP algorithm still takes some computing power. This is due to the $\mathcal{O}(q^2)$ runtime as explained in Section 5.3. In [52] (see Section 4.4) they target Kyber with both $q = 3329$ and $q = 7681$ and report a time of approximately 2 and 8 minutes respectively, per iteration, using an implementation on a single core. In [29] (see Section 4.5) they also target Kyber with $q = 3329$ and report a time of approximately 20 minutes for the full BP algorithm with a multithreaded implementation on 24 cores. In both of these attacks they use the butterfly message creation with $\mathcal{O}(q^2)$ runtime, where $q$ is approximately 13 to 14 bits. If we want to apply the same attack for example to the lattice-based PQC scheme Dilithium with $q = 8380417 \approx 2^{23}$ we can expect that a $\mathcal{O}(q^2)$ runtime is not practical. Furthermore, it has been shown in [17] that also lattice-based PQC schemes with unsuitable NTT parameters can still benefit from the NTT by applying multiple tricks. One of these tricks include using an NTT structure with a larger value $q'$ than the value $q$ that is defined in the scheme. They demonstrate this on Saber and NTRU and their values of $q'$ are up to 23 bits large. Therefore increasing the speed of the BP algorithm on an NTT structure will allow a faster attack on Kyber and could potentially also allow attacks on Dilithium, Saber and NTRU. One interesting observation made in [53] is that for their implementation of a factor graph, the message creation is a convolution computation, which can be implemented using an FFT, reducing the runtime to $\mathcal{O}(q \log q)$. However, one of the issues they encountered was the low success rate due to the short loops in this graph. This was addressed in [52], but since here they make multiple changes at once it is interesting to revisit the split factor nodes to only analyze the difference this makes for the success rate and time, which is done in Section 6.2. Next in Section 6.3 we analyze an approach to significantly speed up the BP algorithm by doing computations over smaller fields. Finally we propose a more promising speed up technique in Section 6.4 for a faster method of creating butterfly messages in the merged butterfly factor node, by ignoring negligible computations.

## 6.1 Improving success rate

The success rate of a SASCA attack on an NTT structure is expressed as the average success rate when doing multiple simulations with the same parameter set, but with different values sampled for the input to the NTT and different values sampled for the noise. In the case where we target a specific scheme, most parameters are fixed. If we simulate the template matching step, the noise parameter $\sigma$ is still free to choose and can be used as a score, i.e. how high can we set $\sigma$ such that the average success rate is still 1 or close to 1 (e.g. at least 0.9). Increasing this score has been the main target for the papers that target the LPR and Kyber schemes. This is done in the following ways:

- Template matching target, in [53] the template matching is performed on the multiplication with the twiddle factor. This only gives side-channel information on less than half the intermediate values. Therefore [29, 52] target the load and store operations of the intermediates, resulting in side-channel information on all intermediate values. Since we simulate the template matching, we assume that the amount of side-channel information obtained is independent on the operation that template matching is applied to. Therefore having side channel information on all intermediates is considered optimal.
- The input to the NTT, in [53] they target an INTT with as input a uniformly random element, meaning that every variable in the first layer has a domain of size $q$, similar like every other variable in the factor graph. In [52] they target a forward NTT where the input has a much smaller support, namely $n$ elements with domain $[-\eta, \eta]$ with $\eta = 4$. Having variable nodes with smaller domain makes it easier for the attack to find the correct values. In [29] they again target an INTT with variables having a domain of size $q$, however they force the input to be sparse, resulting in many fixed values of 0 in the graph, making the attack significantly likelier to find the correct values.
- Since an NTT structure will contain loops, LBP is used. This technique only gives an approximation to the marginals. Both the length of the loops and the number of loops will effect how close the approximation is, which consequently effects the success rate. In [53] the factor graph contains short loops of length 4, while in [29, 52] the factor graph only contains loops of length 8. Due to the structure of the NTT these loops of length 8 can not be removed without throwing away a lot of information in the graph.

From the above we see that these areas for improving the success rate are already thoroughly explored and improved on. An alternative area to increase the success rate is by adding additional information to the factor graph. In [29, 53] they target an INTT where the input is an element-wise multiplication of a ciphertext and secret key, both in the NTT domain. The ciphertext is a public value, and the secret key is the target of the attack. For the LPR and Kyber schemes, the secret key is sampled from the error distribution, meaning that it has a much smaller support and follows a well structured and known distribution, which is information that can potentially be used to improve the success rate. However, note that since we target an INTT, the inputs are in the NTT domain. Therefore we first need to check whether applying an NTT to the secret key $\mathbf{s}$ still gives a nice distribution for the coefficients of $\tilde{\mathbf{s}}$. Unfortunately, due to the many modular reductions taking place, the NTT destroys the structure of the distribution, resulting in a distribution that is nearly uniform on $\mathbb{Z}_q$. For BP, adding a leakage node that sends a uniform message is equivalent to not sending anything at all. See Figure 19 for the distribution of the coefficients in the private

key **s** in the LPR scheme and its distribution after applying the NTT. Similarly, Figure 20 shows the distribution of the coefficients of **s** as in Kyber, before and after applying an NTT.

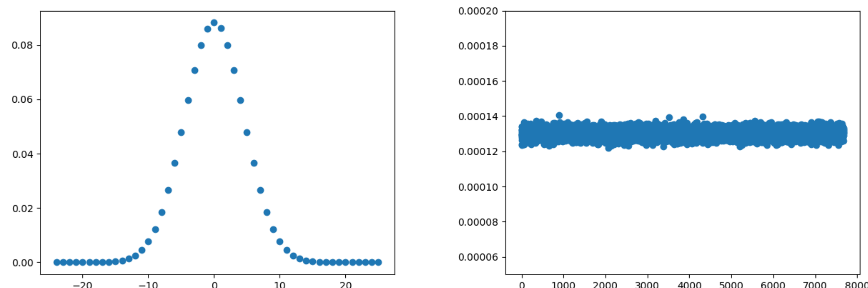


Figure 19: Distribution of the elements of the private key in the LPR scheme before (left, discrete Gaussian with $\sigma = 4.51$) and after applying an NTT (right, nearly uniform) where $n = 256$ and $q = 7681$.

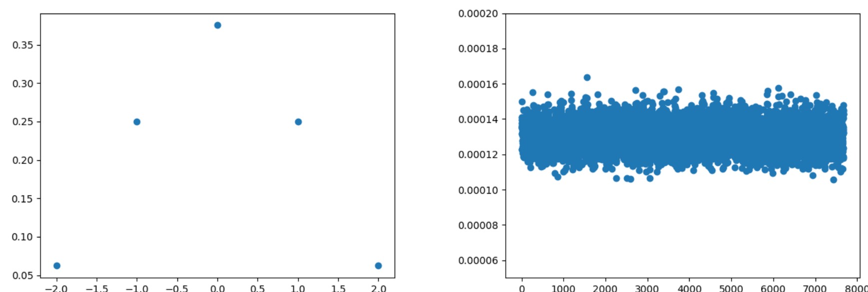


Figure 20: Distribution of the elements of the private key in Kyber before (left, centered binomial with $\eta = 2$) and after applying an NTT (right, nearly uniform) where $n = 256$ and $q = 7681$.

## 6.2 Revisiting the factor graph with split butterfly node

In [53] a factor graph is used where the butterfly node is split into two factor nodes, one for the addition and one for the subtraction that are performed in a butterfly computation. This introduces many short loops in the graph, which is addressed by [52]. In this section we analyze the impact of just switching the merged node to a split node approach. We are interested in this split butterfly approach since via an FFT trick it allows for a faster runtime for computing messages from the factors to the variable nodes.

**The FFT trick** Assume we want to send a message from the ADD factor node to $y_1$, then we need to compute

$$\begin{aligned}\mathbf{r}_{ADD\to y_1}[c] &= \sum_{a,b\in\{0,1,\ldots,q-1\}} f_{ADD}(a,b,c)\cdot \mathbf{q}_{x_1\to ADD}[a]\cdot \mathbf{q}_{x_2\to ADD}[b] \\ &= \sum_{a=0}^{q-1} \mathbf{q}_{x_1\to ADD}[a]\cdot \mathbf{q}_{x_2\to ADD}[(c-a)\omega^{-1}],\end{aligned}$$

for every value of $c$, resulting in a runtime of $\mathcal{O}(q^2)$. If we define a permutation function $\pi_\omega$ as[3]

$$\begin{aligned}\pi_\omega &: \mathbb{R}^q \to \mathbb{R}^q \\ & u[i] \mapsto u[i\omega \mod q], \quad i = 0, ..., q-1,\end{aligned}$$

and let $\mathbf{q}'_{x_2\to ADD} = \pi_\omega(\mathbf{q}_{x_2\to ADD})$ then we can rewrite this message computation as

$$\mathbf{r}_{ADD\to y_1}(c) = \sum_{a=0}^{q-1} \mathbf{q}_{x_1\to ADD}[a]\cdot \mathbf{q}'_{x_2\to ADD}[c-a].$$

This is exactly the definition of a discrete convolution of the two vectors $\mathbf{q}_{x_1\to ADD}$ and $\mathbf{q}_{x_1\to ADD}$, which we denote by

$$\mathbf{r}_{ADD\to y_1} = \mathbf{q}_{x_1\to ADD} \star \mathbf{q}'_{x_2\to ADD}.$$

The circular convolution theorem [31] now says that such convolution can be computed via the FFT:

$$\begin{aligned}\mathbf{r}_{ADD\to y_1} &= \mathbf{q}_{x_1\to ADD} \star \mathbf{q}'_{x_2\to ADD} \\ &= IFFT(FFT(\mathbf{q}_{x_1\to ADD}) * FFT(\mathbf{q}'_{x_2\to ADD})).\end{aligned}$$

Using this FFT trick the runtime is decreased to $\mathcal{O}(q\log q)$.

**Approximation precision** In the case $n = 2$, we only have a single butterfly and 4 variables. The merged butterfly will contain no loops and thus computes the marginals exactly, while the split butterfly will contain a single loop of length 4 and thus the computed marginals will be an approximation. See Figure 21 for an example of the computed marginals of $x_1$ with $n = 2$ and $q = 5$, and note that the split butterfly already has a significant difference to the exact marginals.

[3]Note that for this permutation to be properly defined, $q$ needs to be prime. This will always be the case for our value of $q$.

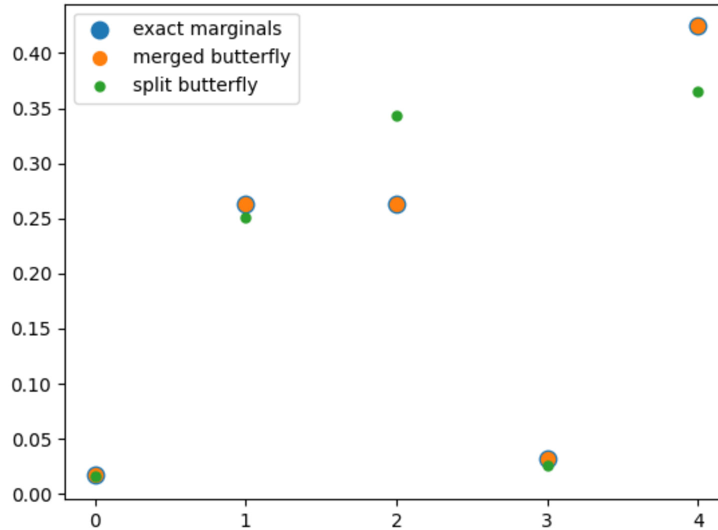


Figure 21: Example of the marginal probabilities of $x_1$ computed exactly, and computed via BP with merged and split butterfly, where $n = 2$ and $q = 5$.

Next, in the case $n = 4$ the merged butterfly factor graph will also contain a loop, but only of length 8, meaning that also here LBP needs to be used and thus this will result in an approximation. See Figure 22 for such an example with $n = 4$ and $q = 13$, and note that the merged butterfly is a much closer approximation, especially for the value 6. This difference becomes even clearer if there is only one solution with a non-negligible probability, see Figure 23 for such an example where in the exact marginals there is only one value with a probability of almost one. For the computed marginals via the merged butterfly we find a very close approximation since there also is only one value with a non-negligible probability. However, for the marginals computed via the split butterfly we see multiple values with a non-negligible probability, showing that already in a small case the approximation might not be sufficient for a successful SASCA attack.

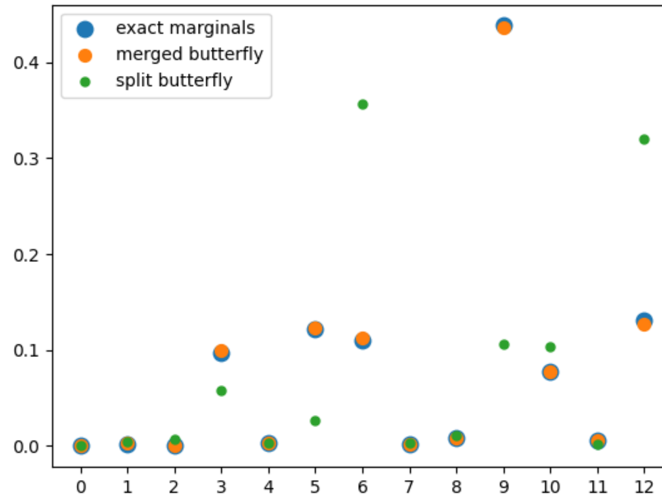


Figure 22: Example of the marginal probabilities of $x_1$ computed exactly, and computed via BP with merged and split butterfly, where $n = 4$ and $q = 13$.

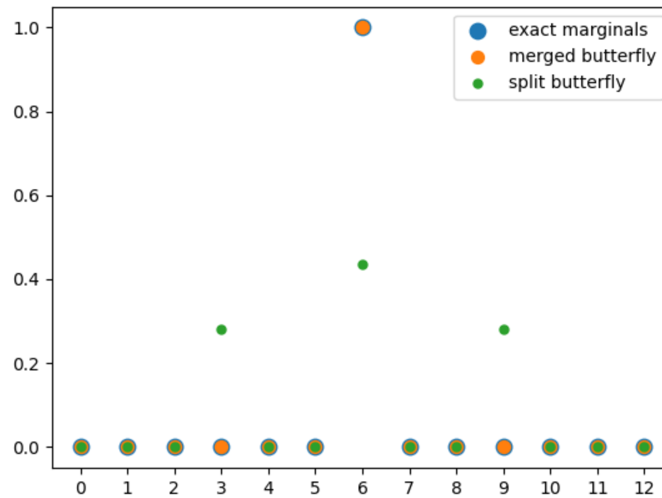


Figure 23: Example of the marginal probabilities of $x_1$ computed exactly, and computed via BP with merged and split butterfly, where $n = 4$ and $q = 13$ and only a single solution to the system with non-negligible probability.

**Simulations with split butterfly** To test the split butterfly, we run simulations with the parameter set $n = 128$, $q = 257$ and $\sigma = 0.2$ with a HW-leakage model. With this parameter set we would have an average success rate of 1 if we use the merged butterfly approach. With the split butterfly however, after 240 simulations we got an average success rate of 0. If we observe individual runs we note the following observations:

- BP only terminates due to the max iteration bound.
- After 5-7 iterations the current guess is always almost correct. Per layer of variables, between 115 and 125 out of the 128 intermediates are guessed correctly.
- In further iterations, one of the two cases follow. In the first case BP oscillates between two solutions where the difference of the number of correct guesses typically is less than 5, and none of the guesses is completely correct. The oscillations have a period of 1 iteration, i.e. it alternates between the two guesses after every iteration. In the second case, BP starts converging to a different, completely wrong solution.

Given these observations, one strategy is to terminate BP always after 7 rounds. The issue however is that, knowing that a large number of guesses are correct, it is impossible to know which ones are correct. If we increase the value of $\sigma$, but such that BP with the merged butterfly would still be successful, we find that this 'almost correct' occurrence does not happen anymore. Thus the above observations only occur for small noise. The same thing holds when increasing $q$.

## 6.3 Computations modulo smaller primes and recombining with the Chinese Remainder Theorem

The goal of a SASCA attack is to learn the true values of all intermediates. In our case these are values modulo $q$. What we instead can aim for is learning the correct values but then modulo some smaller value $q'$. If we do this for multiple values of $q'$, we can combine this information to learn the true value modulo $q$. To test this theory, we choose two values prime values $q_1$ and $q_2$ of approximately $\sqrt{q}$ such that $q_1 \cdot q_2 > q$. If we learn all the correct values modulo $q_1$ and $q_2$, we can then use the Chinese Remainder Theorem to compute the correct value modulo $q_1 q_2$, from which we can learn the correct value modulo $q$. Let $x'_i$ be the value corresponding to the intermediate $x_i$, but then modulo $q_1$, thus it has domain $\{0, ..., q_1 - 1\}$. We first need to map the vector of $q$ probabilities we obtained from the leakage to a vector of $q_1$ probabilities. Given the vector of probabilities $\left[\mathbb{P}[x_i = a|l]\right]_{a=0}^{q-1}$ for the intermediate $x_i$, we compute the vector containing the $q_1$ probabilities for intermediate $x'_i$ as

$$\left[\mathbb{P}[x'_i = a|l]\right]_{a=0}^{q_1-1} = \Big[\sum_{\substack{b \in \{0,\dots,q-1\}: \\ b \equiv a \mod q_1}} \mathbb{P}[x'_i = b|l]\Big]_{a=0}^{q_1-1}.$$

Using an ID-leakage this gives a nice mapping such that BP with modulus $q_1$ or $q_2$ could still converge, see for example Figure 24. However, for HW-leakage the structure in the leakage is completely destroyed by this mapping, resulting in a nearly uniform distribution over the smaller domain. A uniform distribution means that there is no information on the intermediate. See Figure 25 for an example.

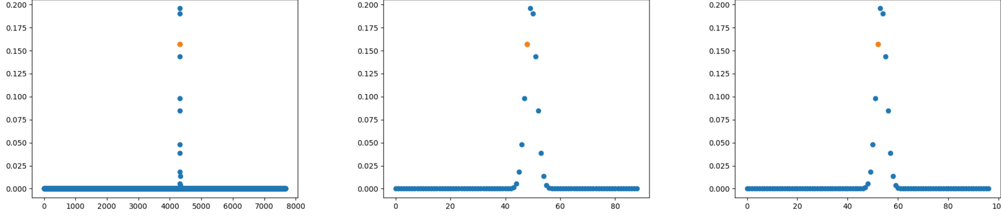


Figure 24: Example of mapping an ID-leakage vector with modulo $q$ to smaller modulos, with parameters $q = 7681$, $q_1 = 89$, $q_2 = 97$ and intermediate value 4320. The leakage is modeled via ID leakage with $\sigma = 2$. The first image is the probability vector modulo $q$, second image is modulo $q_1$ and third is modulo $q_2$. The orange dots are the true values.

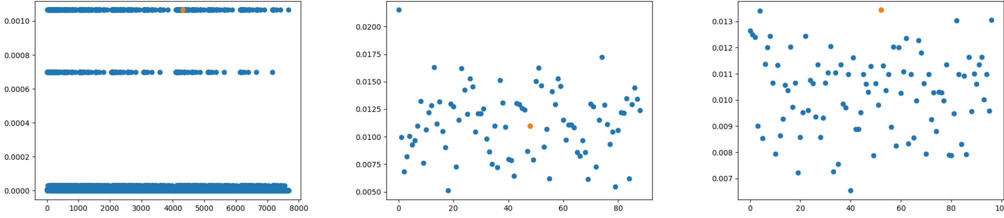


Figure 25: Example of mapping an HW-leakage vector with modulo $q$ to smaller modulos, with parameters $q = 7681$, $q_1 = 89$, $q_2 = 97$ and intermediate value 4320. The leakage is modeled via Hamming weight leakage with $\sigma = 0.5$. The first image is the probability modulus $q$, second image is modulus $q_1$ and third is modulus $q_2$. The orange dots are the true values.

After computing the new leakage vectors, we can create the new factor graph by replacing all $x_i$ by $x'_i$ and replacing the leakage factor nodes by the new ones. However, we quickly run into an issue with the butterfly nodes. For the BP algorithm we use the knowledge on how the intermediates are related, e.g. $x_1 + \omega x_2 \equiv x_3 \mod q$, however simply replacing $x_i$ by $x'_i$ this identity will generally not hold anymore. This is due to a reduction modulo $q$ taking place. For example, let $q = 7681, q_1 = 89$ and let the intermediates be $x_1 = 4320, x_2 = 983$ and $\omega = 2028$, then $x_3 = 4320 + 2028 \cdot 983 = 1997844 \equiv 784 \mod 7681$. We have $x'_1 = 48, x'_2 = 4$ and $x'_3 = 784 \equiv 72 \mod 89$. But $x'_1 + \omega x'_2 = 48 + 2028 \cdot 4 = 8160 \equiv 61 \mod 89 \equiv 1997844 \mod 89 \not\equiv x'_3 \mod 89$. Since $q$ will always be a prime value, we can not describe the relation between $x'_1, x'_2$ and $x'_3$ without knowing the true values of $x_1, x_2$ and $x_3$ and thus this attack method fails. Therefore, although this was a promising direction, it is an open problem whether such a mapping can still be used.

## 6.4 Faster butterfly message creation via non-negligible support

In order to speed up the BP algorithm on an NTT structure, we reanalyze the slowest step, namely the butterfly message creation from Algorithm 3. Note that any messages $\mathbf{q}_{x \to BF}$ that the butterfly node receives typically are very sparse, i.e. they are almost zero on most entries. This means that

when computing new messages as in Equation 3, which are a sum of products of values of the messages in the inbox, many of these products will also be nearly zero. Therefore when computing the sum, many of the summands are negligible. If we were to ignore all these negligible summands, computation time can be reduced while the sum will not change significantly.

**Defining negligible** To exploit this, we first need to define what it means for a probability in a message to be negligible, for which there are many ways to define this. For our approach, we look at the maximal probability in the message and define any probability that is smaller than $t$ times this maximal probability as negligible. So given a message $\mathbf{q}_{x \to BF}$, we define a function *support* with additional input $t$ that returns the indices of the values in the message that are larger than $t \cdot \max(\mathbf{q}_{x \to BF})$. Next we update the message creation step by modifying Algorithm 3, by now not going over all values but only going over non-negligible values. We also update the choice of which messages we iterate over, when sending a message to the right (i.e. sending to $y_1$ or $y_2$), we iterate over all the values of the non-negligible support of the messages from the left (i.e. from $x_1$ and $x_2$), and vice versa. See Algorithm 4 for the pseudo-code of the message creation step using non-negligible support. The new message that is computed will be an approximation, where a smaller value of $t$ will give a closer approximation. See Figure 26 for an example of this approximation in the case $t = 0.1$, where a message $\mathbf{r}_{BF \to y_1}$ is computed by using the non-negligible support of $\mathbf{q}_{x_1 \to BF}$ and $\mathbf{q}_{x_2 \to BF}$, and the corresponding values of $\mathbf{q}_{y_2 \to BF}$. Note here that the higher probabilities in $\mathbf{r}_{BF \to y_1}$ are closely approximated while the lower ones are not.

**Algorithm 4** Butterfly message creation using a non-negligible support

**Input:**

The recipient: $s \in \{x_1, x_2, y_1, y_2\}$,

The three incoming messages: $\mathbf{q}_{x_1 \to BF}$, $\mathbf{q}_{x_2 \to BF}$, $\mathbf{q}_{y_1 \to BF}$, $\mathbf{q}_{y_2 \to BF}$ except for $\mathbf{q}_{s \to BF}$,

The twiddle factor $\omega$ and modulus $q$

The factor $t$ to define which values are negligible

**Output:** The outgoing message to the recipient: $\mathbf{r}_{BF \to s}$

Precompute $2^{-1}$ and $\omega^{-1} \mod q$
$\mathbf{r}_{BF \to s} \leftarrow [0, 0, ..., 0]$ of length $q$
**if** $s = x_1$ **then**
  **for** $c \in \mathit{support}(\mathbf{q}_{y_1 \to BF}, \max(\mathbf{q}_{y_1 \to BF}) \cdot t)$ **do**
    **for** $d \in \mathit{support}(\mathbf{q}_{y_2 \to BF}, \max(\mathbf{q}_{y_2 \to BF}) \cdot t)$ **do**
      $a \leftarrow 2^{-1} \cdot (c + d) \mod q$
      $b \leftarrow (c - a) \cdot \omega^{-1} \mod q$
      $\mathbf{r}_{BF \to s}[a] += \mathbf{q}_{x_2 \to BF}[b] \cdot \mathbf{q}_{y_1 \to BF}[c] \cdot \mathbf{q}_{y_2 \to BF}[d]$
    **end for**
  **end for**
**end if**
**if** $s = x_2$ **then**
  **for** $c \in \mathit{support}(\mathbf{q}_{y_1 \to BF}, \max(\mathbf{q}_{y_1 \to BF}) \cdot t)$ **do**
    **for** $d \in \mathit{support}(\mathbf{q}_{y_2 \to BF}, \max(\mathbf{q}_{y_2 \to BF}) \cdot t)$ **do**
      $a \leftarrow 2^{-1} \cdot (c + d) \mod q$
      $b \leftarrow (c - a) \cdot \omega^{-1} \mod q$
      $\mathbf{r}_{BF \to s}[b] += \mathbf{q}_{x_1 \to BF}[a] \cdot \mathbf{q}_{y_1 \to BF}[c] \cdot \mathbf{q}_{y_2 \to BF}[d]$
    **end for**
  **end for**
**end if**
**if** $s = y_1$ **then**
  **for** $b \in \mathit{support}(\mathbf{q}_{x_2 \to BF}, \max(\mathbf{q}_{x_2 \to BF}) \cdot t)$ **do**
    $b' \leftarrow b \cdot \omega \mod q$
    **for** $a \in \mathit{support}(\mathbf{q}_{x_1 \to BF}, \max(\mathbf{q}_{x_1 \to BF}) \cdot t)$ **do**
      $c \leftarrow a + b' \mod q$
      $d \leftarrow a - b' \mod q$
      $r_{BF \to s}(c) += \mathbf{q}_{x_1 \to BF}[a] \cdot \mathbf{q}_{x_2 \to BF}[b] \cdot \mathbf{q}_{y_2 \to BF}[d]$
    **end for**
  **end for**
**end if**
**if** $s = y_2$ **then**
  **for** $b \in \mathit{support}(\mathbf{q}_{x_2 \to BF}, \max(\mathbf{q}_{x_2 \to BF}) \cdot t)$ **do**
    $b' \leftarrow b \cdot \omega \mod q$
    **for** $a \in \mathit{support}(\mathbf{q}_{x_1 \to BF}, \max(\mathbf{q}_{x_1 \to BF}) \cdot t)$ **do**
      $c \leftarrow a + b' \mod q$
      $d \leftarrow a - b' \mod q$
      $\mathbf{r}_{BF \to s}[d] += \mathbf{q}_{x_1 \to BF}[a] \cdot \mathbf{q}_{x_2 \to BF}[b] \cdot \mathbf{q}_{y_1 \to BF}[c]$
    **end for**
  **end for**
**end if**

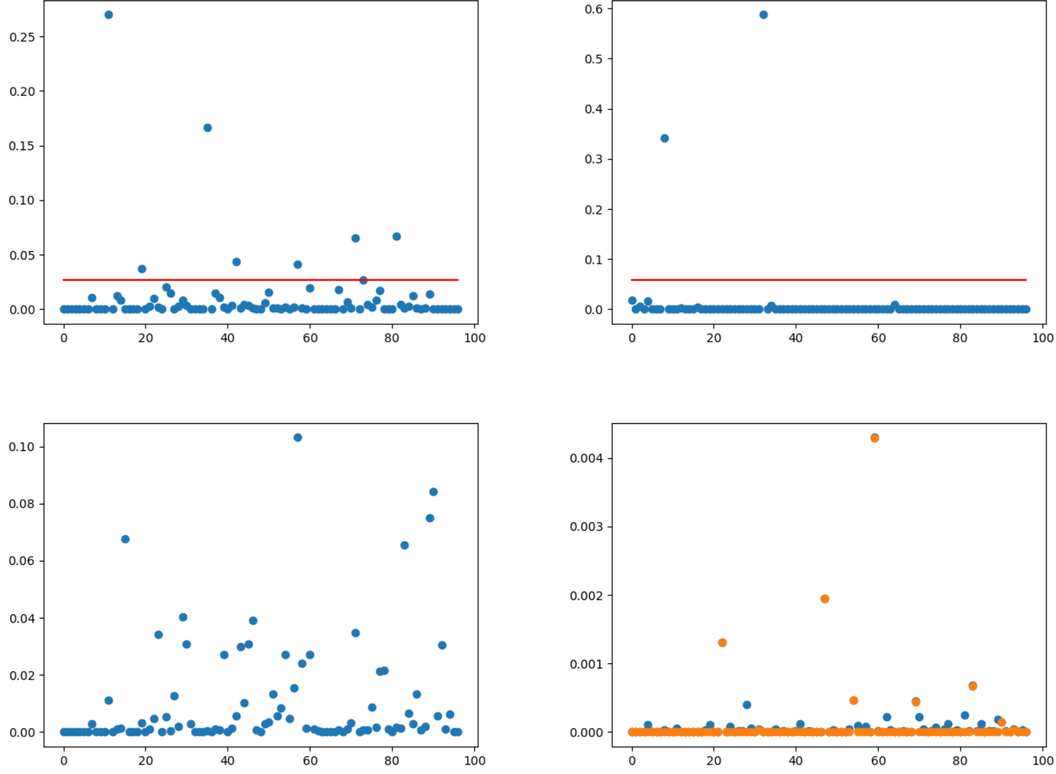


Figure 26: Example of how a message $\mathbf{r}_{BF\to y_1}$ is computed (bottom right) without optimization (in blue) and with optimization with $t = 0.1$ (in orange), where it is computed from the non negligible support of $\mathbf{q}_{x_1\to BF}$ (top left, all values above red line are non negligible) and $\mathbf{q}_{x_2\to BF}$ (top right) and the corresponding values of $\mathbf{q}_{y_2\to BF}$ (bottom left).

**Choosing the value of $t$** Next we need to choose the value of $t$. We want to set $t$ small enough such that we get the same or very close to the same success rate compared to if we do not use this optimization trick. However, the smaller we set $t$, the less performance is gained. We fix $n = 128$, and start with an ID-leakage model with $\sigma = 2$ up to 8 in increments of 2, and we vary $q = 257, 1409$ and 3329. We test $t = 10^{\tau}$ for $\tau = -1$ down to $-10$ in increments of $-1$. The results are in Figure 27. For $t$ small enough, we get back the original method, so by decreasing $t$ the average success rate will converge to the success rate for the original method. Therefore we want to take the largest value of $t$ such that the average success rate is at or very close to the rate it converges to. Based on Figure 27 the optimal choice for $t$ is somewhere around $10^{-5}$, independently of the choice for $\sigma$ and $q$. Next we verify if $t = 10^{-5}$ is sufficient in the case of a HW-leakage model too. We set $n = 128$ and $q = 257$ and vary $\sigma = 0.2$ up to 0.8 in steps of 0.2 and use the same values of $t$. See the results in Figure 28, from which we can conclude that the optimal value for $t$ is around $10^{-5}$ also for the HW-leakage model.

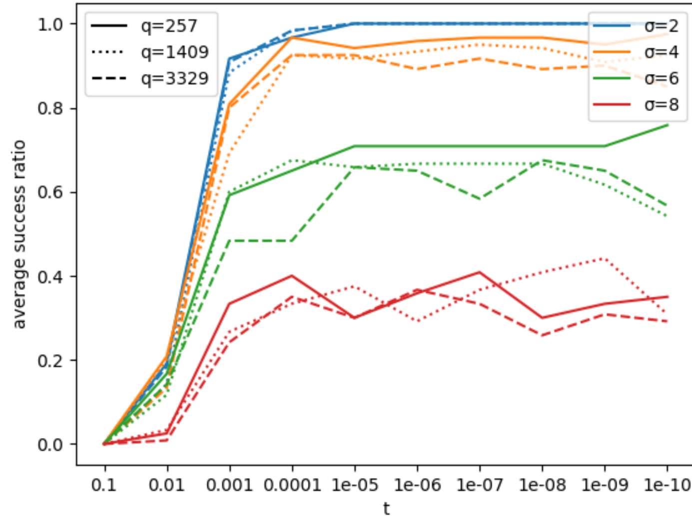


Figure 27: Average success rate of our optimized butterfly message creation using a non-negligible support with various values of $t, q$ and $\sigma$ using $n = 128$ and an ID-leakage model.

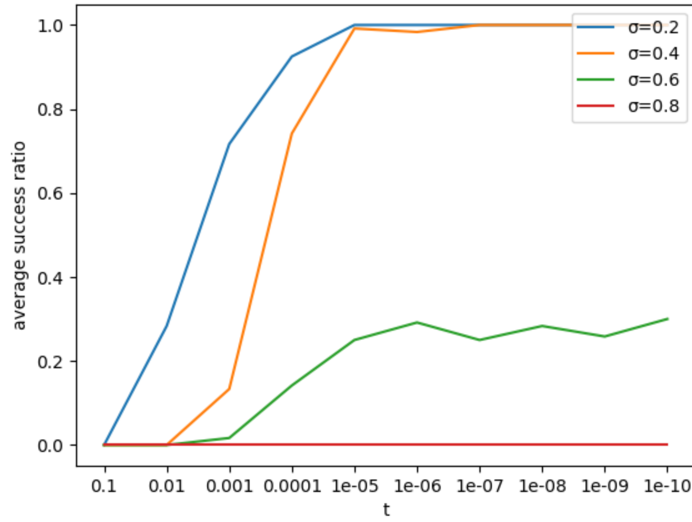


Figure 28: Average success rate of our optimized butterfly message creation using a non-negligible support with various values of $t$ and $\sigma$ using $n = 128, q = 257$ and a HW-leakage model.

**Performance gain** The amount of time saved with this method depends on the sparseness of the messages as well as the choice of $t$. Without this optimization, a new message is computed as $q$ sums, with in every sum $q$ summands where every summand is computed as a product of 3 values. So in total there are $2q^2$ product operations and $q^2$ addition operations. In the example of Figure 26, this would result in $2 \cdot 97^2 = 18818$ multiplications and $97^2 = 9409$ additions. Using $t = 0.1$ the messages $\mathbf{q}_{x_1 \to BF}$ and $\mathbf{q}_{x_2 \to BF}$ have only 7 and 2 values in their non-negligible support, resulting in $2 \cdot 7 \cdot 2 = 28$ multiplications and $2 \cdot 7 = 14$ additions. However, we saw that $t = 0.1$ is not small enough, so for $t = 10^{-5}$ we find 79 and 29 values in the non-negligible support, resulting in resulting in $2 \cdot 79 \cdot 29 = 4582$ multiplications and $79 \cdot 29 = 2291$ additions, which is still significantly less. The sparseness of a message depends on many factors:

- Convergence, the goal of BP in a SASCA attack is to find the true solution to the system. If BP converges to this solution, all messages sent will converge towards a message that contains a single probability of being 1 and all other probabilities being negligible. If BP does not converge to a single solution but a set of solutions, the messages become less sparse.
- Iteration number, in case when BP converges one or a few solutions, messages in at a higher iteration number will be more sparse.
- The leakage type, if we consider both an ID and HW-leakage model, we see that messages coming from the leakage factor nodes in an ID-leakage model are far more sparse than in an HW-leakage model. This propagates to the sparseness of the messages sent to the butterfly node.
- The value of $q$, in an ID-leakage model increasing $q$ makes messages relatively more sparse compared to a HW-leakage model.
- The value of $\sigma$, a larger amount of noise makes the messages less sparse since more solutions will have a non-negligible probability.

See Figures 29 and 30 for the average times of the first and second iteration of BP from the simulations from Figures 27 and 28 respectively. We only show the first and second iteration since when there is more than 2 iterations, the messages don't change significantly to have an impact on the average time, or the attack fails since BP does not converge to the true solution, in which case the times also don't change significantly. We want to compare these times to the times from the non-optimized method. Here the time is independent of the sparseness of the messages, so we only have to compute it for different values of $q$. We can estimate these values by timing a single butterfly message creation step, and multiply this with the number of messages being computed in a single iteration. Every butterfly computes 4 messages, and there are $\frac{n}{2} \log_2(n)$ butterfly nodes, so for $n = 128$ we have $64 \cdot 7 \cdot 4 = 1792$ messages being computed. For $q = 257, 1409$ and $3329$ we find an average time of $85.81, 2640.65$ and $14626.63$ seconds respectively per full iteration.

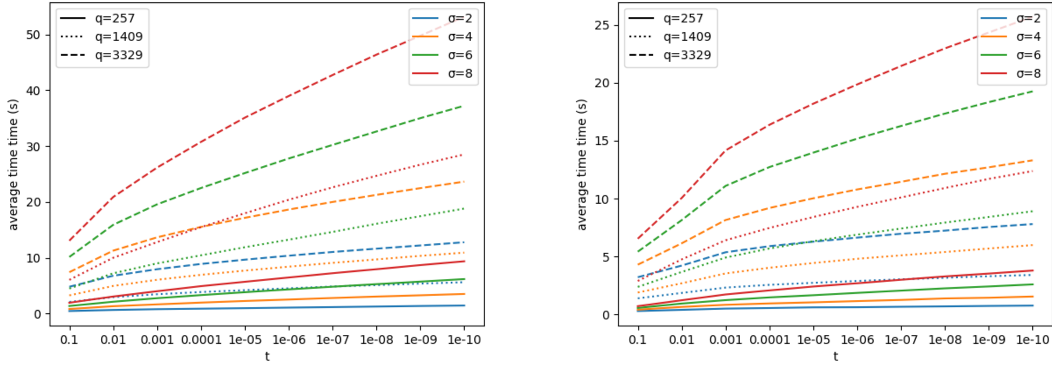


Figure 29: Time in seconds for the the first (left) and second (right) iteration from the simulations from Figure 27, where $n = 128$ with and ID-leakage model

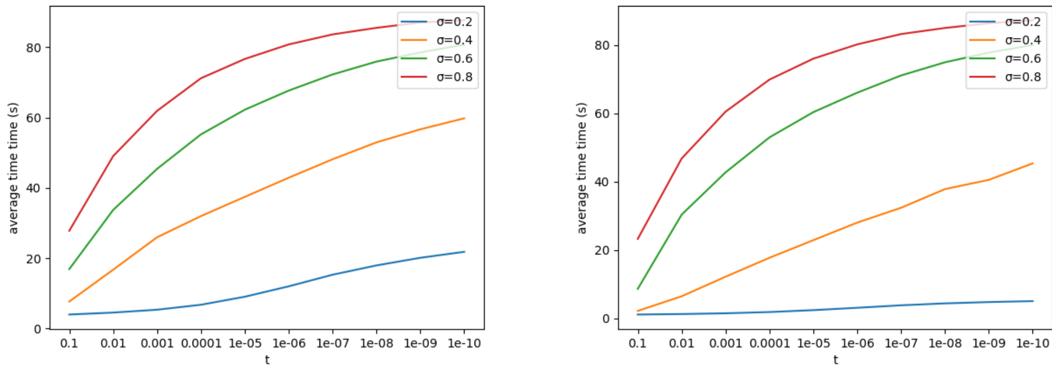


Figure 30: Time in seconds for the the first (left) and second (right) iteration from the simulations from Figure 28, where $n = 128, q = 257$ with and HW-leakage model

# 7 Discussion

In this section we will summarize and interpret the results of this thesis. The goal was to investigate SASCA attacks on an NTT structure. This is done by first analyzing and describing various types of side-channel attacks with emphasis on soft analytical side-channel attacks. Then we looked at a series of 3 papers [29, 52, 53] that attack specific post-quantum cryptography schemes by applying SASCA attacks to the NTT that these schemes use. We made our own implementation of such an attack, without assuming any specific target scheme but rather a generic NTT structure. Using this implementation we analyzed and improved the attack, for which we will summarise our findings in Section 7.1. Then in Section 7.2 we note some shortcomings of this thesis and give our recommendations for further research directions.

## 7.1 Conclusion

In Section 5 we discussed the details of our Python implementation of a SASCA attack on an NTT structure. We simulate the template matching step, so we mainly focus on the Belief Propagation step of the attack. We target an NTT that is used for multiplications in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$ since this requires no additional scaling and only requires the existence of a $n$th root of unity. However, the structure of the NTT does not change and thus the results are expected to be identical for an NTT for multiplications in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. The goal of this implementation is to get experimental results, by varying parameters and noting their effect on the success rate. For the size of the NTT, which is regulated by $n$, we saw that an increased value results in a higher success rate. This is due to having information on more intermediate values as well as more information on how these are connected. The effect of $q$ depends on the leakage type used. In the case of a Hamming weight leakage model, an increased value of $q$ will have a negative effect on the success rate. This is due to the fact that with a larger value of $q$, the number of values with a non-negligible probability in the template matching step increases as well. This is not the case for an identity leakage model, therefore a larger value of $q$ does not decrease the success rate there.

In section 6 we presented our contributions by investigating possible improvements of SASCA attacks on an NTT structure. Firstly, independently of our implementation, we test whether it is possible to improve the success rate for [29, 53] and similar targets where the input of the INTT depends on a private key. The small support and known distribution of this key is potentially useful information for the attack. However, due to the key being in the NTT domain we found that it is not useful anymore due to the NTT destroying the structure of the distribution.

We stress that the NTT target of our implementation is a generic NTT, because we do not assume any structure of the input to the NTT. This allows us to investigate possible improvements that can be applied to any SASCA attack on an NTT structure, independent of the cryptographic scheme in which the NTT is used. This however means that our implementation can not use additional information of a specific target to increase the success rate, such as forcing a sparse input or targeting an input with smaller support. Therefore we focus more on improving the speed of the attack. This is important due to the $\mathcal{O}(q^2)$ runtime, which will lead to intractable times for schemes with large values of $q$. Our first idea was to try and find the correct values of all intermediate values modulo some smaller prime, and repeat this for various smaller primes and then combine this to find the true value. However, this approach failed because the relations between the intermediate

value do not hold anymore, and due to modular reductions taking place when computing intermediate values, this can not be recovered. In our next approach we revisit an approach from [53], where they split up the butterfly factor node in two nodes. This allows for a $\mathcal{O}(q \log q)$ runtime per iteration, but causes issues with the success rate due to short loops in the factor graph. We found that having side-channel information on all intermediate values does not fix the issues the small loops cause, thus this approach requires further investigation. Lastly, we defined and implemented an optimization trick which ignores negligible computations. During the butterfly message creation computation, all values in a message that are smaller than $t$ times the maximal value are ignored. By choosing the value of $t = 10^{-5}$, no loss will be made for the average success rate, while still gaining in performance. The amount of performance gained depends on the sparseness of the messages. In the case of an ID-leakage model, the time saved is up to a factor 1000 for $q = 3329$, and this will scale positively for larger $q$. For a HW-leakage model however the time save will be much less, around a factor of 5, which is due to the messages being less sparse, and this is expected to not scale as positively for larger $q$.

## 7.2 Shortcomings and suggestions for further research direction

Due to the limited time frame available to write this thesis, it will have some shortcomings. Furthermore, since PQC schemes are still in an early stage and not yet standardized, there is still a lot to research in terms of attacks. This is especially the case for soft analytical side-channel attacks since this class of attacks is also relatively new. Our shortcomings and recommendations for further research can be summarized as follows:

- The Python implementation is purely for analysis purposes, and is not suitable for an actual real-world attack. To get realistic results for time and memory usage of the analyzed attack and improvements, an efficient implementation is required. We recommend to do this in a more efficient language such as C, C++ or Rust, while making use of their most optimal data structures.
- We simulate the template matching step, for which we made very general assumptions. To verify the correctness of an attack, this step needs to be performed with an experimental setup with a physical device.
- Due to simulating the template matching step, we mainly focused on the Belief Propagation part. However, further improvements can be made to this template matching step to increase the success rate of the complete SASCA attack.
- If we want to apply SASCA to an NTT with larger values of $q$ (e.g. Dilithiums $q$ with a size of 23 bits), other issues arise such as memory usage. The messages send in BP are vectors of size $q$, and for every edge in the graph there are 2 such messages per iteration. This will result in requiring to store many large vectors during BP. Therefore some smarter data structures are recommended. We suggest using a similar approach such as the non-negligible butterfly message computation trick. Rather than ignoring values in messages during this step, it is possible to check for negligibility earlier, namely by checking the current marginals of every variable in an iteration. If a value in the vector of probabilities for the computed marginal is negligible, this value can then be ignored. This means that every variable should keep track of which values have a non-negligible probability. Then when sending a message to a factor node, this message should only contain probabilities for those values. We therefore suggest

the messages to be a data structure similar to a Python dictionary, which would contain the non-negligible values as keys with the corresponding probability in the message as data.

- We target the NTT method for multiplying polynomials since this has a nice algebraic structure to it which helps for the Belief Propagation step. However, other similar multiplication methods such as Toom-Cook and Schönhage-Strassen also have a similar algebraic structure, which can be targeted as well. This will result in different factor graphs which can pose their own advantages and disadvantages, which can be researched.
- The FFT trick for fast message computation in the split butterfly case is due to the new message being a convolution of two messages. When computing the message for the merged butterfly, it is not exactly a convolution anymore, since now a third message is involved. However, this computation still has a very similar structure, so it is possible that a similar trick is possible to obtain a $\mathcal{O}(q \log q)$ runtime, even with the merged butterfly. We could not find such trick, but this does not imply it does not exist.
- The split butterfly approach fails in most cases due to the short loops. Belief Propagation is a widely used algorithm, and research has been done in improving the approximations in the case of short loops, see for example [33, 46]. Applying some of the tricks they suggest might make the split butterfly feasible even for larger values of $q$ and $\sigma$.

# References


[1] RC Agarwal and C Burrus. Fast convolution using fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974.

[2] Erdem Alkim, Joppe W Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, Douglas Stebila, et al. Frodokem learning with errors key encapsulation. *NIST, Gaithersburg, MD, USA, Tech. Rep*, 1130, 2017.

[3] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, et al. Bike: bit flipping key encapsulation. 2017.

[4] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4), 2017.

[5] Reza Azarderakhsh, Matthew Campagna, Craig Costello, LD Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, et al. Supersingular isogeny key encapsulation. *Submission to the NIST Post-Quantum Standardization project*, 152:154–155, 2017.

[6] Gregory V Bard, Nicolas T Courtois, and Chris Jefferson. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over gf (2) via sat-solvers, 2007.

[7] Daniel J Bernstein. Multidigit multiplication for mathematicians. 2001.

[8] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. Classic mceliece: conservative code-based cryptography. *NIST submissions*, 2017.

[9] Daniel J Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2129–2146, 2019.

[10] Andrey Bogdanov, Ilya Kizhvatov, and Andrey Pyshkin. Algebraic methods in side-channel collision attacks and practical collision detection. In *International Conference on Cryptology in India*, pages 251–265. Springer, 2008.

[11] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[12] Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. Ntru prime: round 3 20201007.

[13] Antoine Casanova, Jean-Charles Faugere, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. *GeMSS: a great multivariate short signature*. PhD thesis, UPMC-Paris 6 Sorbonne Universités; INRIA Paris Research Centre, MAMBA Team ..., 2017.

[14] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[15] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. The picnic signature scheme, 2020.

[16] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. Ntru algorithm specifications and supporting documentation. *Brown University and Onboard security company, Wilmington USA*, 2019.

[17] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. Ntt multiplication for ntt-unfriendly rings. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 159–188, 2021.

[18] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[19] Jean-Sébasticn Coron, Paul Kocher, and David Naccache. Statistics and secret leakage. In *International Conference on Financial Cryptography*, pages 157–173. Springer, 2000.

[20] Nicolas T Courtois and Gregory V Bard. Algebraic cryptanalysis of the data encryption standard. In *IMA International Conference on Cryptography and Coding*, pages 152–169. Springer, 2007.

[21] Nicolas T Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *International conference on the theory and application of cryptology and information security*, pages 267–287. Springer, 2002.

[22] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *International Conference on Smart Card Research and Advanced Applications*, pages 167–182. Springer, 1998.

[23] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. submission to the nist post-quantum cryptography standardization project (2020).

[24] Hans Dobbertin. Cryptanalysis of md4. In *International Workshop on Fast Software Encryption*, pages 53–69. Springer, 1996.

[25] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *International Conference on Cryptology in Africa*, pages 282–305. Springer, 2018.

[26] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5), 2018.

[27] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[28] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 512–529. Springer, 2012.

[29] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked cca2 secure kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 88–113, 2021.

[30] Helena Handschuh and Howard M Heys. A timing attack on rc5. In *International Workshop on Selected Areas in Cryptography*, pages 306–318. Springer, 1998.

[31] G Proakis John, G Manolakis Dimitris, and G Manolakis. Digital signal processing: principles, algorithms, and applications. *Pentice Hall*, 1996.

[32] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[33] Alec Kirkley, George T Cantwell, and MEJ Newman. Belief propagation for networks with loops. *Science Advances*, 7(17):eabf1211, 2021.

[34] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[35] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[36] BAM Babette Lips, BMM Benne de Weger, C Christine van Vredendaal, JR Joost Renes, HJM Hans Sterk, and LAM Berry Schoenmakers. The efficiency of polynomial multiplication methods for ring-based pqc algorithms of round 3 of the nist pqc competition.

[37] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. Crystals-dilithium. *Submission to the NIST Post-Quantum Cryptography Standardization [NIS]*, 2017.

[38] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 1–23. Springer, 2010.

[39] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms.* Cambridge university press, 2003.

[40] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.

[41] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92. Springer, 2000.

[42] David McCann, Carolyn Whitnall, and Elisabeth Oswald. Elmo: Emulating leaks for the arm cortex-m0 without access to a side channel lab. *IACR Cryptol. ePrint Arch.*, 2016:517, 2016.

[43] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. Hamming quasi-cyclic (hqc). *NIST PQC Round*, 2:4–13, 2018.

[44] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Investigations of power analysis attacks on smartcards. *Smartcard*, 99:151–161, 1999.

[45] Silvio Micali and Leonid Reyzin. Physically observable cryptography. In *Theory of Cryptography Conference*, pages 278–296. Springer, 2004.

[46] Joris Mooij, Bastian Wemmenhove, Bert Kappen, and Tommaso Rizzo. Loop corrected belief propagation. In *Artificial Intelligence and Statistics*, pages 331–338. PMLR, 2007.

[47] NIST. Post-quantum cryptography standardization. https: //csrc.nist.gov/projects/post-quantum-cryptography/ post-quantum-cryptography-standardization.

[48] Henri Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[49] Henri J Nussbaumer. The fast fourier transform. In *Fast Fourier Transform and Convolution Algorithms*, pages 80–111. Springer, 1981.

[50] Elisabeth Oswald. *On side-channel attacks and the application of algorithmic countermeasures.* na, 2003.

[51] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 333–342, 2009.

[52] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *International Conference on Cryptology and Information Security in Latin America*, pages 130–149. Springer, 2019.

[53] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 513–533. Springer, 2017.

[54] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[55] Mathieu Renauld and François-Xavier Standaert. Algebraic side-channel attacks. In *International Conference on Information Security and Cryptology*, pages 393–410. Springer, 2009.

[56] Mathieu Renauld, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Algebraic side-channel attacks on the aes: Why time also matters in dpa. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 97–111. Springer, 2009.

[57] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 30–46. Springer, 2005.

[58] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1):181–199, 1994.

[59] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3):281–292, 1971.

[60] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A collision-attack on aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 163–175. Springer, 2004.

[61] Kai Schramm, Thomas Wollinger, and Christof Paar. A new class of collision attacks and its application to des. In *International Workshop on Fast Software Encryption*, pages 206–222. Springer, 2003.

[62] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[63] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A formal practice-oriented model for the analysis of side-channel attacks. *IACR e-print archive*, 134(2006):2, 2006.

[64] Andrei L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[65] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 282–296. Springer, 2014.

# Appendices

## A NTT multiplication example

Let $n = 8$, $q = 17$ and $\mathcal{R}_q = \mathbb{Z}_{17}[x]/(x^8 - 1)$. We want to multiply $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$ to obtain $\mathbf{c} \in \mathcal{R}_q$. Let $\mathbf{a} = [15, 6, 9, 0, 10, 9, 8, 14]$ and $\mathbf{b} = [7, 14, 6, 3, 0, 12, 5, 10]$, corresponding to the polynomials $a(x) = 15 + 6x + 9x^2 + 10x^4 + 9x^5 + 8x^6 + 14x^7$ and $b(x) = 7 + 14x + 6x^2 + 3x^3 + 12x^5 + 5x^6 + 10x^7$ respectively. We have $q = k \cdot n + 1 = 2 \cdot 8 + 1$, so $k = 3$. We take $r = 3$ as a primitive 16th root of unity (it can be easily verified that $r^i \not\equiv 1 \mod 17$ for $i = 1, ..., 15$ and by Fermat's little theorem $r^{16} \equiv 1 \mod 17$), so we have as $n$th primitive root $\omega_n = r^k = 3^2 \equiv 9 \mod 17$. For the

forward NTT we need the 4 twiddle factors $[\omega_n^0, \omega_n^1, \omega_n^2, \omega_n^3] = [1, 9, 13, 15]$. See Figure 31 for the NTT transform of $\mathbf{a}$ to $\tilde{\mathbf{a}}$, as a butterfly network as in Figure 11.

Figure 31: The NTT transform of $\mathbf{a}$ as a butterfly network.

Applying the NTT to $\mathbf{a}$ and $\mathbf{b}$ we find $\tilde{\mathbf{a}} = [3, 2, 4, 8, 13, 0, 12, 10]$ and $\tilde{\mathbf{b}} = [6, 1, 12, 12, 13, 5, 14, 10]$ respectively. It can easily be verified that these values are equal to evaluating $a(x)$ and $b(x)$ for $x = \omega_n^0, ..., \omega_n^7$. After point-wise multiplication we find $\tilde{\mathbf{c}} = [1, 12, 14, 11, 16, 0, 15, 15]$. Applying the INTT to $\tilde{\mathbf{c}}$ with twiddle factors $[\omega_n^0, \omega_n^{-1}, \omega_n^{-2}, \omega_n^{-3}] = [1, 2, 4, 8]$ we find $\mathbf{c} = [5, 9, 12, 6, 15, 16, 2, 4]$. To verify the result we can compute $c(x) = a(x)b(x)$ via e.g. schoolbook multiplication and find

$$\begin{aligned}
c(x) =& a(x)b(x) \\
\equiv& 3 + 14x + 16x^2 + 3x^3 + 6x^4 + 2x^5 + 15x^6 + 4x^7 \\
&+ 2x^8 + 12x^9 + 13x^{10} + 3x^{11} + 9x^{12} + 14x^{13} + 4x^{14} \mod q \\
\equiv& 3 + 14x + 16x^2 + 3x^3 + 6x^4 + 2x^5 + 15x^6 + 4x^7 \\
&+ 2 + 12x + 13x^2 + 3x^3 + 9x^4 + 14x^5 + 4x^6 \mod x^8 - 1 \\
\equiv& 5 + 9x + 12x^2 + 6x^3 + x15x^4 + 16x^5 + 2x^6 + 4x^7 \mod q
\end{aligned}$$

# B LBP on the XOR-AND example

We let LBP run for 5 iterations on the XOR-AND example, and find the following messages that are being sent:

| Message | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| $f_{l_1} \rightarrow x_1$ | [0.8 0.2] | [0.8 0.2] | [0.8 0.2] | [0.8 0.2] | [0.8 0.2] |
| $f_{l_2} \rightarrow x_2$ | [0.1 0.9] | [0.1 0.9] | [0.1 0.9] | [0.1 0.9] | [0.1 0.9] |
| $f_{l_3} \rightarrow x_3$ | [0.4 0.6] | [0.4 0.6] | [0.4 0.6] | [0.4 0.6] | [0.4 0.6] |
| $f_{l_4} \rightarrow x_4$ | [0.7 0.3] | [0.7 0.3] | [0.7 0.3] | [0.7 0.3] | [0.7 0.3] |
| $f_{XOR} \rightarrow x_1$ | [0.5 0.5] | [0.58 0.42] | [0.5777 0.4222] | [0.5782 0.4217] | [0.5782 0.4217] |
| $f_{XOR} \rightarrow x_2$ | [0.5 0.5] | [0.44 0.56] | [0.4216 0.5783] | [0.4212 0.5787] | [0.4210 0.5789] |
| $f_{XOR} \rightarrow x_3$ | [0.5 0.5] | [0.26 0.74] | [0.1956 0.8043] | [0.1915 0.8084] | [0.1911 0.8088] |
| $f_{AND} \rightarrow x_1$ | [0.5 0.5] | [0.6730 0.3269] | [0.6782 0.3217] | [0.6796 0.3203] | [0.6796 0.3203] |
| $f_{AND} \rightarrow x_2$ | [0.5 0.5] | [0.5303 0.4696] | [0.5229 0.4770] | [0.5230 0.4769] | [0.5230 0.4769] |
| $f_{AND} \rightarrow x_4$ | [0.75 0.25] | [0.82 0.18] | [0.8590 0.1409] | [0.8570 0.1429] | [0.8573 0.1426] |
| $x_1 \rightarrow f_{l_1}$ | [0.5 0.5] | [0.7397 0.2602] | [0.7424 0.2575] | [0.7441 0.2558] | [0.7441 0.2558] |
| $x_1 \rightarrow f_{XOR}$ | [0.8 0.2] | [0.8917 0.1082] | [0.8939 0.1060] | [0.8945 0.1054] | [0.8945 0.1054] |
| $x_1 \rightarrow f_{AND}$ | [0.8 0.2] | [0.8467 0.1532] | [0.8454 0.1545] | [0.8458 0.1541 | [0.8457 0.1542] |
| $x_2 \rightarrow f_{l_2}$ | [0.5 0.5] | [0.4700 0.5299] | [0.4441 0.5558] | [0.4438 0.5561] | [0.4437 0.5562] |
| $x_2 \rightarrow f_{XOR}$ | [0.1 0.9] | [0.1114 0.8885] | [0.1085 0.8914] | [0.1086 0.8913] | [0.1086 0.8913] |
| $x_2 \rightarrow f_{AND}$ | [0.1 0.9] | [0.0802 0.9197] | [0.0749 0.9250] | [0.0748 0.9251] | [0.0747 0.9252] |
| $x_3 \rightarrow f_{l_3}$ | [0.5 0.5] | [0.26 0.74] | [0.1956 0.8043] | [0.1915 0.8084] | [0.1911 0.8088] |
| $x_3 \rightarrow f_{XOR}$ | [0.4 0.6] | [0.4 0.6] | [0.4 0.6] | [0.4 0.6] | [0.4 0.6] |
| $x_4 \rightarrow f_{l_4}$ | [0.75 0.25] | [0.82 0.18] | [0.8590 0.1409] | [0.8570 0.1429] | [0.8573 0.1426] |
| $x_4 \rightarrow f_{AND}$ | [0.7 0.3] | [0.7 0.3] | [0.7 0.3] | [0.7 0.3] | [0.7 0.3] |