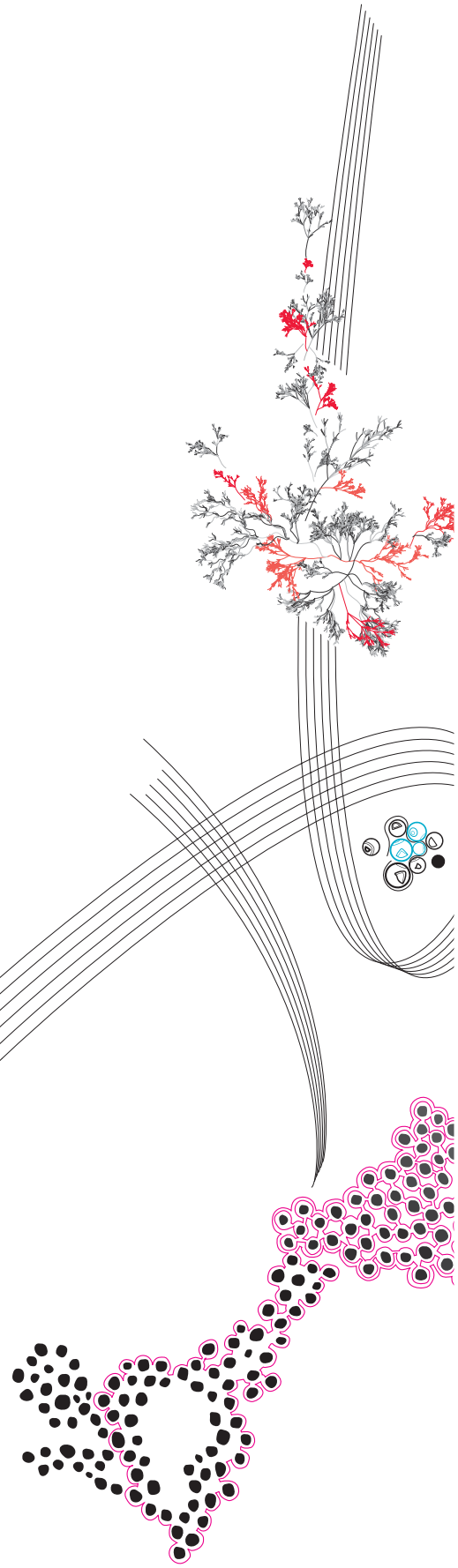

MSc Computer Science
Final Project

Test Vector Leakage Assessment on Number Theoretic Transform

Yuezhou Lyu

Supervisor: dr.ing. Florian Hahn

August, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

We propose algorithms for generating test vectors in power-based side-channel Test Vector Leakage Assessment (TVLA) of the Number Theoretic Transform (NTT) algorithm in CRYSTALS-Dilithium, a post-quantum signature algorithm selected by NIST for standardization. In particular, we focus on two implementations: plain NTT and `GKS20` NTT. Our algorithms encompass all three power models: Hamming Weight (HW), Hamming Distance (HD), and Identity (ID). We validate our test vectors on a Piñata board with ARM Cortex-M4F core. We are able to detect various leakages from the Piñata board with t-values ranging from 6.57 to 3174.96. In addition, we characterize leakages from NTT butterflies with correlation analysis, and conclude that most of the leakages come from memory operations. We therefore close the gap for a lack of practical leakage assessment for Dilithium NTT.

We also investigate how much such leakages can be utilized for launching a template attack against the Dilithium secret key $\mathbf{s}_1$. Our result shows a 68.45% reduction of entropy in the best case scenario for $\mathbf{s}_1$, even with a single-trace attack.

Keywords: post-quantum cryptography, side-channel analysis, leakage assessment, number theoretic transform, embedded systems security

Acknowledgements

First I would like to thank my academic advisors Dr. Florian Hahn and Dr. Tihanyi Norbert, and industrial advisors Dr. Barış Ege and Mikheil Kushashvili, for their guidance, encouragement, sharing of knowledge and meticulous feedback. I would also thank Dr. Roland van Rijswijk-Deij for being members of my graduation committee. Furthermore I would thank EIT Digital Master School for providing me with the opportunity and support for finishing the program. Last but not least I would thank my parents, grandparents, and friends, for their support and love ad infinitum.

List of Notations

$\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$	the set of integers, rational numbers, real numbers, complex numbers
$[a, b]$	the set of integers $\{x \in \mathbb{Z} \vert a \leq x \leq b\}$
q	the prime modulus of Dilithium, set at $q = 8380417$
$\mathbb{Z}_q$	the ring of integers modulo q, also a finite field since q is prime
$\mathbf{s}_1$	the $\mathbf{s}_1$ secret key of Dilithium
ψ	a 512-th root of unity in the field $\mathbb{Z}_q$, set at $\psi = 1753$
α, β	inputs of an NTT butterfly
ζ	coefficient multiplied to β in a butterfly, aka "twiddle constant"
$\mathbb{P}$	probability measure
$\mathcal{L}, \mathcal{B}$	a lattice and basis of a lattice
$\mathcal{F}, \mathcal{F}^{-1}$	Fourier transform and inverse Fourier transform on $\mathbb{R}$
$\mathcal{DFT}, \mathcal{DFT}^{-1}$	discrete Fourier transform and inverse discrete Fourier transform
$f * g$	convolution of functions f and g
$\mathfrak{T}$	t-value as result of Welch's t-test
Gen	key generation algorithm for Dilithium
Sign	signing algorithm for Dilithium
$\mathsf{NTT}, \mathsf{NTT}^{-1}$	number theoretic transformation and inverse number theoretic transform for Dilithium
$S_0, S_1, \ldots, S_8$	input and 8 stages of plain NTT
HW	Hamming weight of a number
HD	Hamming distance of a number
$\oplus$	XOR operation

Chapter 1

Introduction

Side-channel analysis exploits information leakage from a cryptographic system, usually implemented on a physical device [25]. While cryptanalysis attacks a cipher by inspecting the algorithm itself, side-channel analysis utilizes auxiliary information such as power consumption, electromagnetic emanations, timing, and acoustic data during the execution of the algorithm [38]. Leakage assessment determines if a specific implementation of a cipher is prone to side-channel analysis [44]. In other words, leakage assessment detects the cases when the device is leaking sensitive cryptographic information.
Classical cryptographic hard problems such as factorization and discrete logarithm are vulnerable to quantum computers because of the Shor's algorithm. Post-Quantum Cryptography (PQC) aims to provide security when quantum computers become a threat to classical cryptographic algorithms [7]. PQC is an umbrella term for many ciphers based on different types of cryptographic primitives, with lattice-based cryptography one of the most popular one currently. During execution, lattice-based ciphers are required to perform a high volume of polynomial multiplication, potentially being computationally expensive if implemented naively. Number Theoretic Transform (NTT) is an algorithm for fast computation of polynomial multiplication. Most lattice-based ciphers include NTT as a crucial component for efficiency [43].
Several side-channel attacks have been proposed to target the NTT [36][35][11]. However, no prior research has been done on practically assessing the leakage of NTT. Our research will fill this gap and investigate how far the Test Vector Leakage Assessment (TVLA) methodology could be used in leakage assessment for NTT. TVLA is usually the first step assessing a novel cryptographic device, by comparing the power consumption of many carefully crafted inputs (test vectors) fed into the device. Our work will focus two variations of NTT implementation, propose test vector generation algorithms for both variations, and assess the effectiveness of test vectors on real-world devices. Our main contributions are the algorithms proposed in Chapter 3, 4, and 5, and the experiment done in Chapter 4, 5, and 6. Intuitively, our algorithms generate randomized intermediate states and utilize the bijectivity of plain NTT to find the test vectors. After conducting experiment on the Piñata board, which is known to be leaky, we are able to confirm its leakages with our test vectors. With a threshold of $|\mathfrak{T}| > 4.5$, our TVLA t-values ranging from $|\mathfrak{T}| = 6.57$ to $|\mathfrak{T}| = 3174.96$, depending on which part of the NTT is tested against. Correlation analysis shows that most of the leakages come from memory operations. Template attack built on such leakages show on average a 1.15% to 68.45% reduction of entropy for $\mathbf{s}_1$.
An outline of our work is given here

Chapter 2 contains background information about side-channel analysis, post-quantum

cryptography, number theoretic transform, and a summary of prior research done on these topics.

Chapter 3 contains a purely theoretical treatment of the "textbook" version of NTT, namely plain NTT. Plain NTT is not implemented in real-world devices, but serves as a foundation for our further discussion of real-world implementations.

Chapter 4 investigates the leakage of a popular implementation of NTT on ARM Cortex-M4 microcontrollers, namely `GKS20` NTT. We will identify the differences between `GKS20` NTT and the plain NTT discussed in Chapter 3. Algorithms proposed in Chapter 3 are then adapted for this particular implementation. We then conduct real-world experiment, showing per-stage leakages of the Piñata board.

Chapter 5 goes one step further from the experiment results of Chapter 4 by generating test vectors not for a whole stage of NTT, but for a few NTT butterflies. We then conduct experiment for these test vectors, showing per-butterfly leakages of the Piñata board. Test vectors in this chapter can be combined with the test vectors in Chapter 4, to reveal a more delicate and comprehensive leakage profile of a certain device. Correlation analysis is then performed to match the leakages with processor instructions and power models.

Chapter 6 investigates how much of the leakages identified in Chapter 4 and 5 can be used to launch a template attack.

Riscure is a leading security lab specialized in side-channel analysis and fault injection. Riscure designs Inspector, a commercial-grade current probe for power-based side-channel analysis. Piñata board is a tutorial board with ARM Cortex-M4F processor, produced by Riscure for side-channel analysis training. All of our research is conducted at Riscure during the author's stay as a thesis-based intern. We utilize Riscure devices (current probe, Piñata, *etc.*) and Riscure software (Inspector SCA) for hardware testing.

Chapter 2

Background

2.1 Side-Channel Analysis

Most information of side-channel analysis, especially power-based side-channel analysis, is from "The Blue Book" [25].
A cryptographic device is a physical hardware with some cryptographic functionality implemented in it and usually stores a cryptographic key. Typical cryptographic devices include USB keys, smart cards, hardware security modules, and various Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) implementations of cryptographic protocols. Cryptographic devices are built with logic cells, where logic cells are the smallest logic building blocks of a circuit.
One goal of attacking a cryptographic device is to retrieve the key stored in the device. Attacks can be classified in two different ways: as passive or active, and as invasive, semi-invasive, or non-invasive. Passive attacks are performed when the cryptographic device is operated as normal, within its specification. Active attacks are performed when the device is forced to behave abnormally by manipulating the device itself, the input of the device, or the environment. Invasive attack sets no limits to what can be done to the device. In invasive attack, we start with depackaging and then access different electronic components using a probing station. In semi-invasive attack, the device is depackaged, but no direct electrical contact to a chip is ever made. In non-invasive attack, the device is attacked "as is", without leaving evidence for attacking.
The two ways of classification thus lead to six types of actual attacks. In invasive passive attacks, there is a probing station connected to the electronic components directly, but the probing station is only used to observe signals. For example, we can probe the signals fired on a bus. In invasive active attacks, signals or even physical components in the device are manually changed. In semi-invasive passive attacks, the attacker reads the content of memory cells without probing the circuit. Semi-invasive active attacks are essentially fault injection with depackaging, using X-rays, electromagnetic fields, lasers, *etc.* for example. Non-invasive passive attacks are also known as side-channel analysis, including timing-based, power-based, acoustic-based, *etc.* Non-invasive active attacks are fault injection without depackaging, for example, changing temperature of the environment, or injecting clock or power glitches.
Non-invasive attack is relatively easy and stealthy to setup because no depackaging or physical modification is required. Passive attack is relatively cheap to launch compared to invasive attacks because current probe is generally cheaper than probing station or fault injection tools. Thus the combination of the two requires the attacker to attack a device with the least amount of efforts and maximum stealthiness. Therefore we will

focus on non-invasive passive attack, and specifically power-based side-channel analysis.

2.1.1 Power-Based Side-Channel Analysis

As described above, power-based side-channel analysis is a non-invasive passive attack. It utilizes the information leaked from power consumption to infer knowledge about the cryptographic device that is otherwise unavailable. Power-based side-channel analysis contains

Simple Power Analysis (SPA) Interpreting power consumption directly to infer the key.

Differential Power Analysis (DPA) Statistically analyzing large amount of power consumption to infer the key.

Test Vector Leakage Assessment (TVLA) Instead of inferring the key, assess if the device leaks any side-channel information.

Recall that cryptographic devices are built from logic cells, we may analyze its power consumption in terms of these cells. Within a device, power consumption can be classified into two parts: static and dynamic. Usually static power consumption is negligible compared to dynamic power consumption.

- Static: power consumption when there is no switching activity in a cell.
- Dynamic: power consumption when an internal or output signal switches. Usually, power consumption from internal signal switch is negligible compared to output signal switch. Moreover, consumption is data-dependent: for one bit, switching for $0 \to 0$ and $1 \to 1$ are very low, essentially consuming only static power. Consumption for $0 \to 1$ and $1 \to 0$ are much larger.

During actual practice, we usually only care about the power consumption caused by flipping of the output signals because it is usually the dominant factor.
A power model is a mathematical abstraction of a device's power consumption. Based on the above discussion about power consumption, we introduce three power models. The Hamming Distance (HD) model conjectures that the power consumption is correlated to the Hamming distance of a register (or memory location) before and after computing a certain intermediate value. For example, for a 32-bit register that changed from `0xdeadbeef` to `0xfeedf00d`, the Hamming distance between the two states is precisely how many bits are flipped. In this case, the Hamming distance is 10. Hamming distance model is likely to perform better with hardware implementation of a cryptographic algorithm.
Another power model is the Hamming Weight (HW) model, which conjectures that power consumption is correlated to the Hamming weight of an intermediate value. For example, if a 32-bit register that changed from `0xdeadbeef` to `0xfeedf00d`, the Hamming weight model would predict a power consumption change that is proportional to $24 \to 20$. The Hamming weight model is cruder than the Hamming distance model, but is a good choice when we do not have white-box knowledge of a specific hardware implementation. Hamming weight model is likely to perform better with software implementation of a cryptographic algorithm.
Apart from HD and HW models, there is another popular model: the Identity (ID) model. The ID model essentially says, only the same intermediate value can produce the

same power consumption. In other words, each unique intermediate value corresponds to a unique power consumption. For example, two intermediate values `0xdeadbeee` and `0xdeadbeef`, although very similar in terms of HW and HD, still produce different power traces under ID model. Identity model is likely to happen with both hardware and software implementations, and shows a serious lack of side-channel protection in the implementations.
Various equipment is required for measuring power consumption

- Cryptographic device: the Device Under Test (DUT). It usually has an interface to connect to a PC for receiving commands of execution of cryptographic algorithms.
- Clock generator: some cryptographic devices need to be supplied with an external clock signal. For example, smart cards are supplied with a clock signal of up to 4MHz.
- Power supply: cryptographic devices also need external power supply. For example, power supply for smart cards are provided by the reader. Ideally, the power supply should provide highly stable voltage for measurement quality.
- Circuit probe or Electromagnetic (EM) probe: the actual probe for measuring power consumption. They generate voltage signals that are proportional to power consumption. Circuit probe is placed between power supply and the DUT, while EM probe can measure consumption indirectly to the DUT.
- Digital sampling oscilloscope: for recording the voltage signals provided by the circuit or EM probe. Modern setup uses digital sampling oscilloscope controlled by a PC.
- PC: controls the whole setup and stores data collected through the measurement. Data can be later analyzed either on the same PC controlling data collection, or on another PC specialized in data processing.

Collected power consumption data are called power traces, or simply traces. Power traces are then collected with the following steps

1. Supply the cryptographic device with power and clock signal; so the device is operational and ready to interact.
2. PC configures and starts the oscilloscope.
3. PC sends commands to cryptographic device that starts execution of an algorithm.
4. Power consumption measured by circuit probe or EM probe.
5. PC receives the result of execution from cryptographic device, and power traces from oscilloscope.

2.1.2 Simple Power Analysis

Simple Power Analysis (SPA) reveals the key stored in DUT by directly interpreting power traces. The assumption for a successful SPA attack is that the key has a significant impact on power consumption. SPA usually only requires a few traces. When the attack uses only one trace, it is called a single-trace or one-shot SPA. Attack using a few traces is called a multiple-trace or multiple-shot SPA. In multiple-trace SPA, either we have the traces of encrypting the same plaintext for multiple times, or we have the traces of encrypting different plaintexts.

Visual Inspection

Visual inspection of power traces is usually the first step of SPA. Visual inspection often helps identify the locations on power traces that correspond to stages of cryptographic algorithms. In extreme cases, the power traces can reveal the secret key directly. For example, as in Figure 2.1 if the exponential step in RSA is implemented using naive double-and-add algorithm, then the steps when "add" is performed would cost more power [38]. This potentially leaks the secret exponent by leaking one bit at a time on the power traces.

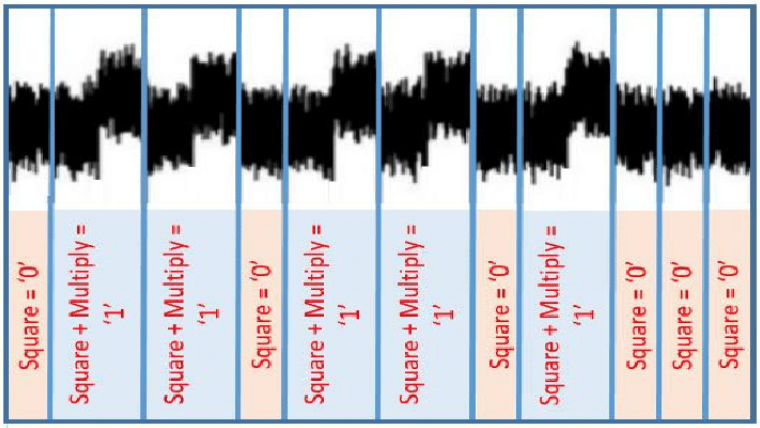

FIGURE 2.1: Power trace during RSA double-and-add from [38].

It is also possible to visually identify the ten rounds in an encryption of AES-128 on power traces [38] as in Figure 2.2.

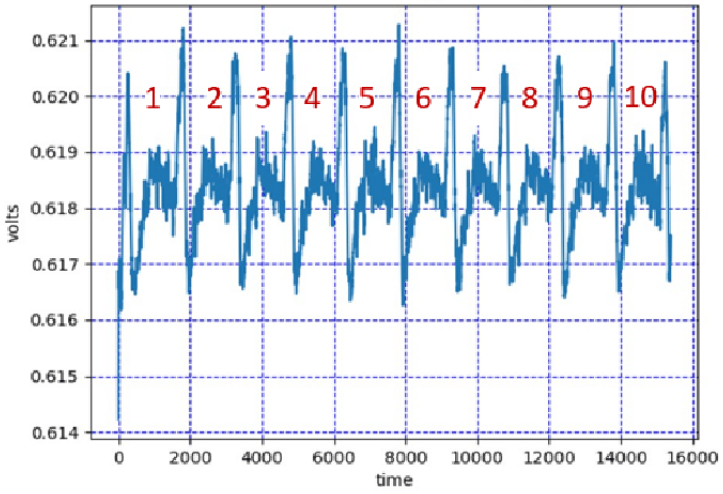

FIGURE 2.2: Power trace during ten rounds of AES-128 encryption from [38].

Template Attack

While visual inspection most likely only gives an impression about how the power traces look like, template attack will actually reveal the key.

In template attack, two devices are usually involved. Device A ("real" device) contains the key that the attacker is trying to obtain. Device B ("copy" device) has the exactly same configuration as Device A, without containing Device A's key. The attacker has full access to Device B, but only limited access to Device A. In particular, we assume the attacker can only obtain one or only a few power traces from Device A, without knowing the input or key for Device A. We assume the attacker can choose arbitrary inputs or keys and can freely obtain power traces from Device B. The attacker's task is to retrieve the key stored in Device A.
In a real-life scenario, consider the case where Device A is a malicious credit card reader, placed secretly by the attacker on a public vending machine. The attacker can record power traces from the malicious card reader, but can only obtain one trace from a certain customer (assuming the customer's card is only swiped once). Assume the attacker has no direct access to the customer's card. To launch a template attack and obtain the keys stored in customers' cards, the attacker uses another card reader, Device B, that is exactly the same as the malicious card reader. With one customer's trace obtained from Device A, and a lot of traces collected from Device B, a successful template attack would reveal the customer's key in the card.
Template attack has two phases: template building when the attacker collects traces from Device B, and template matching when the attacker makes inference with a few traces collected from Device A. Denote the key on the device A as k_{ck}, and the trace collected would be encrypting plaintext d with k_{ck}. Note that both k_{ck} and d are unknown to the attacker. While at the same time, the attacker can perform encryption of any pair (d_i, k_j) on device B and collect the traces when performing any encryption on the device B. Then the attacker needs to decide what the template is built upon. For example, the template may be built on the input (d_i, k_j), but most likely the template is built on some intermediate values $f(d_i, k_j)$. Then, for each unique value the template is built upon, the attacker runs N experiments repetitively, collects N traces, and find N_{IP} power consumption samples at different "interesting" moments, also called Points Of Interest (POI). The attacker finally builds (estimates) an N_{IP}-dimensional multivariate Gaussian distribution for each unique value the template is built upon.
The POI are usually selected with statistical methods, such as signal-to-noise ratio (SNR), sum of squared pairwise t-differences (SOST), and sum of squared pairwise differences of the average (SOSD) [15]. Usually, we first set manually how many POI we are going to use in our templates. Then, we compute for each time point

- the SNR, computed with dividing the variance of exploitable power consumption by the variance of noisy power consumption.
- the SOST, computed with first performing all pairwise t-tests and then the squared sum of pairwise t-values.
- the SOSD, similar to SOST, but computed with pairwise differences of the mean instead of t-values.

Finally we pick the top N_{IP} time points of SNR/SOST/SOSD results as our POI.
For a concrete example, the attacker decides to build a template on the first S-box of AES. S-boxes of AES have an 8-bit output, thus a total of $2^8 = 256$ outputs. For each output, the attacker builds an N_{IP}-dimensional multivariate Gaussian distribution. For example, the attacker is looking at the output `0b10000000`. Then the attacker tries to identify the (d_i, k_j) such that the first S-box produces this output. With the desired (d_i, k_j), the attacker performs N encryption experiments repetitively, and collect N

traces. After determining the POI, each trace obtained by the attacker contains N_{IP} points. Using the $N_{IP} \times N$ matrix of data, the attacker constructs an N_{IP}-dimensional multivariate Gaussian distribution by estimating the parameters (μ, Σ). The attacker repeats the above process for every output from `0b00000000` to `0b11111111`. In the end, the attacker finishes the template building phase after gathering 256 multivariate Gaussian distributions and starts the template matching phase. Denote the trace collected from device A as t. The attacker is then able to compute

$$\mathbb{P}(t|\mathcal{N}(\mu, \Sigma) \text{ related to } f(d_i, k_j)) = \frac{e^{-\frac{1}{2}(t-\mu)^{\mathrm{T}}\Sigma^{-1}(t-\mu)}}{\sqrt{(2\pi)^{N_{IP}} \det \Sigma}} \tag{2.1}$$

for each intermediate value, from `0b00000000` to `0b11111111`. Following maximum-likelihood principle, the attacker infers that the highest probability corresponds to the actual S-box output in device A.
Note that what the template is built upon highly depends on what the device leaks. If the device leaks all output of the first S-box with ID model, the attacker should build the template as in the last concrete example. However, a lot of devices leak the Hamming weight of the output of the S-boxes. In this case, the attacker only needs to build 9 templates, with each template representing the case of having output Hamming weight from 0 to 8.

Collision Attack

We say a collision occurs if there are different $d_i \neq d_i^*$ such that for some intermediate value, $f(d_i, k_j) = f(d_i^*, k_j)$. We observe that a collision cannot occur for all key values, but only a subset of keys. Thus collision helps us to reduce the search space for keys. To determine a collision in practice, first decide the part in power trace where the intermediate value is processed. Then we determine if two power traces are the same or not. In terms of templates:

- We can build the templates for the part of trace where the intermediate value collides.
- Alternatively, if we are certain where the collision will happen, we can simply compare the two traces.

Collision attack, however, is irrelevant to attack for plain NTT because plain NTT is fully deterministic, with each step equivalent to a bijective linear transformation. The kernels are thus all trivial and no collision is ever possible in plain NTT.

2.1.3 Differential Power Analysis

In Differential Power Analysis (DPA), we control the actual device instead of just a copy. The key k_{ck} of the DUT is unknown and we collect a lot of traces to reveal the key using statistical methods. Suppose we want to attack the first S-box of AES. We are again given some input (d_i, k_j) and an intermediate value $f(d_i, k_j)$. In particular, there are D data input and K key input, so $i = 1, \ldots, D$ and $j = 1, \ldots, K$. Since we have control over the device, for each data d_i, we can collect one trace with N_{IP} points. Stacking all the collected traces together, we obtain a $D \times N_{IP}$ matrix T such that each row corresponds to a unique data input d_i, and each column is the consumption at specific time. Then compute offline a $D \times K$ matrix of intermediate values V, such that $V_{i,j} = f(d_i, k_j)$. Then we pick a power model based on what the device leaks. With the power model,

each entry of V is applied to a function h that converts the actual intermediate value into its corresponding power consumption. The result power consumption model is a $D \times K$ matrix H such that $H_{i,j} = h(V_{i,j})$. In the end we construct a $N_{IP} \times K$ matrix R such that $R_{i,j}$ is the correlation of i-th column of H and j-th column of T. R basically finds: which intermediate value (and thus which key) could align with actual consumption at some points in time. Following maximum-likelihood principle, the largest entry of R, say $R_{ck,ct}$, has its row index corresponds to the actual key k_{ck}.

2.1.4 Test Vector Leakage Assessment

The aim of Test Vector Leakage Assessment (TVLA) is different from SPA or DPA. Instead of retrieving the key, TVLA assesses if the DUT is leaking any information [3]. Recall that in SPA and DPA, the choice of power model depends highly on what the DUT leaks. Device manufacturers may deploy countermeasures, namely hiding and masking, that potentially mitigate some leaks. TVLA can then assess the effectiveness of these countermeasures. For example, after proposing a new masking procedure for polynomial inversion, TVLA is performed to evaluate if the masking is effective against side-channel analysis [23]. Hardware implementation of masking in S-boxes of SKINNY block cipher is also tested and validated with TVLA [22]. Although assessments from TVLA are usually not comprehensive (not feasible to test all inputs and keys for an algorithm), TVLA serves as the first step to approach hardware devices or cipher implementations that we have no prior knowledge about them.
Essentially, TVLA compares two distributions of power traces. Each distribution can be one of the following three types: fixed, random, semi-fixed. A fixed distribution contains power traces obtained from executing a fixed input to the cryptographic algorithm. Similarly a random distribution contains power traces when executing random inputs. A semi-fixed distribution contains power traces of semi-fixed inputs. Only a specific part of semi-fixed inputs is fixed across all traces, and the rest part being random.
For example, suppose we want to test if a device is leaking the Hamming weight of the first S-box of AES with fixed-vs-random TVLA. We prepare one set of plaintexts such that the Hamming weight of the first S-box is low; these plaintexts are called fixed test vectors and they are said to be "biased" (because we picked them to have low Hamming weights in the first S-box). Another set of plaintexts are chosen randomly, and they are called random test vectors, also said to be "unbiased". After collecting traces from fixed-vs-random test vectors, we use Welch's t-test to compare if the two distributions are equal. If they are equal, it means our device is not leaking the Hamming weight of the first S-box because there is no difference in power traces between encryption of biased and unbiased data. The search for test vectors then becomes the main task for developing TVLA. The device may leak different things and we need test vectors for almost all of these situations.

2.2 Post-Quantum Cryptography

Post-quantum cryptography (PQC) aims to provide security when quantum computers become a threat to classical cryptographic algorithms. NIST has standardized some PQC algorithms after a multi-round competition. The winner for key encapsulation mechanism (KEM) is CRYSTALS-Kyber [9], and the winners for signature are CRYSTALS-Dilithium [12], FALCON [13], and SPHINCS+ [6]. In particular, CRYSTALS-Kyber, CRYSTALS-Dilithium and FALCON are lattice-based protocols. In

our research, we primarily focus on the CRYSTALS-Dilithium (or simply, Dilithium), and retain the possibility of generalizing our methods to other lattice-based protocols.

2.2.1 Quantum Computers

Quantum computers are computers utilizing quantum mechanical effects [32]. Analogous to a classical computer using traditional binary bits taking values of 0 and 1, quantum computers use quantum bits, or qubits, taking values as linear combinations of basis $|0\rangle := \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle := \begin{pmatrix}0\\1\end{pmatrix}$. Such structure exploits the power of linear algebra enabled by quantum effects, and allows quantum computers to perform Quantum Fourier Transform (QFT), an analogy to classical Fast Fourier Transform (FFT), in polynomial time. Enabled by QFT, Shor's algorithm [45] could solve a class of problems, namely the hidden subgroup problems for abelian groups, in polynomial time. Hidden subgroup problems for abelian groups contain a wide range of classical hard problems, including integer factorization and discrete logarithm [24]. As a result, we need hard problems that are not examples of the hidden subgroup problems for abelian groups, when quantum computers are strong enough to break classical public-key cryptography. To evaluate the hardness of these problems, we distinguish classical hardness based on classical security models such as random oracle model, and quantum hardness based on quantum security models such as quantum random oracle model.

2.2.2 Lattices

Most content in this and the next session is based on [34]. There are many lattice problems and we will only describe a few of them. A lattice is a free $\mathbb{Z}$-module embedded in $\mathbb{R}^n$. Concretely

Definition 2.2.1 (Lattice)**.** *Given $\mathcal{B} = \{b_1, \ldots, b_k\}$ a set of linearly independent vectors in $\mathbb{R}^n$. A lattice $\mathcal{L}$ is the $\mathbb{Z}$-linear combination of these vectors*

$$\mathcal{L} := \left\{ \sum_{i=1}^{k} a_i b_i \,\middle|\, a_i \in \mathbb{Z} \right\}.$$

In particular, k is called rank of a lattice. When $k = n$, the lattice is called a full lattice; we usually only consider full lattices. The set $\mathcal{B}$ is called a basis of a lattice. Notice that for the same lattice, there are many different choice of bases. In fact, the choice of basis plays a key role in lattice-based cryptography. We say a basis is "good" if there are only short vectors in the basis and the vectors are close to orthogonal. A basis is "bad" otherwise. Transforming a bad basis into a good one is called lattice reduction. While lattice reduction is trivial for lattices of rank $k = 2$, reduction is believed to be hard for lattices of higher ranks. The idea of lattice-based public-key cryptography is to make the bad basis a public key, and the good basis a private key.
There exists a variety of lattice problems that are easy to solve with a good basis, and hard to solve with a bad basis. One example being the Decisional Approximate SVP Problem, denoted as GapSVP.

2.2.3 Lattice-Based Cryptography

Lattice-based cryptography is built upon problems that are proved or believed to be (at least) as hard as the lattice problems introduced in the previous section. We briefly

introduce the hard problems that Dilithium is based upon, focusing more on the "rough idea", with simplifications omitting technical details like security parameters and hardness assumptions. The rigorous reader should consult references such as [34] for a rigorous description.

Definition 2.2.2 (Short Integer Solution)**.** *The Short Integer Solution* SIS *problem is formulated as following: given positive integers $n, q > 0$, and a matrix $A \in \mathbb{Z}_q^{n \times m}$ whose entries are uniformly sampled from $\mathbb{Z}_q$, find $z \in \mathbb{Z}_q^m$ such that $\|z\|$ is less than some positive threshold, and $Az = 0 \in \mathbb{Z}_q^m$, for some norm $\|\cdot\|$.*

Ajtai first proved that SIS is at least as hard as GapSVP in [2]. Stronger results are obtained subsequently [28][29].

Definition 2.2.3 (Learning With Errors)**.** *Given positive integers $n, q > 0$. Let $s \in \mathbb{Z}_q^n$ be an unknown vector. Define a distribution $A_{s,\chi}$ as: sample $a \in \mathbb{Z}_q^n$ uniformly, and sample $e \in \mathbb{Z}_q$ according to distribution χ. χ is usually taken as a discrete Gaussian distribution. Then sampling from $A_{s,\chi}$ gives an element of $\mathbb{Z}_q^n \times \mathbb{Z}_q$, specifically $(a, \langle a, s\rangle + e)$.*
Then the Search version of the Learning with Errors SearchLWE *is defined as: given m samples from $A_{s,\chi}$, determine s.*
The Decision version of the Learning with Errors DecisionLWE *is defined as: given m samples from the set $\mathbb{Z}_q^n \times \mathbb{Z}_q$, decide if they are drawn from a fixed $A_{s,\chi}$, or drawn uniformly randomly from $\mathbb{Z}_q^n \times \mathbb{Z}_q$.*

Blum *et al.* proved that the Search and Decision versions of Learning With Errors are equivalent [8], and Regev proved that they are at least as hard as GapSVP [40].

2.2.4 Dilithium

Dilithium is a digital signature scheme, based on three hard problems: Module Learning With Errors MLWE, Module Short Integer Solution MSIS, and SelfTargetMSIS [12]. In particular, MSIS and MLWE are the special version of SIS and LWE, where the ring $\mathbb{Z}_q$ is replaced by some polynomial ring. Although there is no proof that MSIS and MLWE are as hard as SIS and LWE respectively, it is widely believed that the attacker cannot gain advantages utilizing the additional module structure. The problem SelfTargetMSIS is proved to be as hard as MSIS classically, with recent proof showing reduction from MLWE in quantum random oracle model [19]. Dilithium is generally believed to be secure; it is not only selected by NIST as a standard post-quantum signature algorithm, but also recommended as the primary algorithm among other two winners (Falcon and SPHINCS+).
Dilithium comes with 3 levels of security: Level 2, Level 3, and Level 5, each with its own set of security parameters. In particular, we focus on the Level 3 version, because "according to a very conservative analysis, (Level 3 Dilithium) achieves more than 128 bits of security against all known classical and quantum attacks" claimed and recommended by the authors of Dilithium.
Throughout our whole thesis, q always denotes the prime number $q = 8380417$, and Dilithium works in the ring $\mathbb{Z}_q[x]/(x^{256} + 1)$. From an attacker's point of view, the most valuable information of Dilithium is the secret key vector $\mathbf{s}_1$, containing 5 polynomials from the aforementioned ring. Each polynomial of $\mathbf{s}_1$ has coefficients of range $[-4, 4]$. Each time the algorithms signs a new message, it pre-computes $\mathsf{NTT}(\mathbf{s}_1)$. $\mathbf{s}_1$ is subsequently used in computing the signature z (subject to rejection) with $z := y + c\mathbf{s}_1$, where y is a nonce, and c is a challenge. The challenge c is a polynomial in the

aforementioned ring with $\tau = 49$ nonzero coefficients, and an entropy of 225 bits. Note that the nonce y has the same dimension as $\mathbf{s}_1$, and c is just one polynomial. The multiplication $c\mathbf{s}_1$ is a scalar multiplication in a module, and the multiplication is performed elementwise in the NTT domain (we will discuss this in the next section). We are interested in the Sign algorithm because of the following reason. Usually, when attacking a real-life device, we have no access to key generation, because the key is already generated and stored in the device. The cryptographic device is usually designed to perform signatures. An attacker usually could only collect side-channel traces during the signing process of the device. Each time signing a new message, the secret key $\mathbf{s}_1$ is transformed into the NTT domain. Therefore attacking NTT could potentially reveal $\mathbf{s}_1$ directly.

2.3 Number Theoretic Transform

Polynomial multiplication is ubiquitous in lattice-based cryptography. Notice that in MSIS and MLWE, evaluating expressions such as Az and $\langle a, s \rangle$ all require polynomial multiplication. If implemented naively, polynomial multiplication can be time-consuming. Given two polynomials of degree n, a naive schoolbook multiplication has complexity $\mathcal{O}(n^2)$. Number Theoretic Transform (NTT) is a fast implementation of polynomial multiplication. NTT reduces the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. The following text regarding the continuous and discrete Fourier transform is mainly based on [47]. The content and figures regarding NTT are from [43].

2.3.1 Fourier Transform

Definition 2.3.1 (Fourier Transform)**.** *Suppose $f(x)$ is a sufficiently nice function defined on $\mathbb{R}$. Its Fourier transform $\mathcal{F}(f)$ is defined as*

$$[\mathcal{F}(f)](\xi) = \hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} \, \mathrm{d}x. \tag{2.2}$$

Definition 2.3.2 (Inverse Fourier Transform)**.** *The inverse Fourier transform $\mathcal{F}^{-1}$ on $\hat{f}(\xi)$ is defined as*

$$[\mathcal{F}^{-1}(\hat{f})](x) = f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i x \xi} \, \mathrm{d}\xi. \tag{2.3}$$

We usually say a function $f(x)$ is living in the time domain, and its Fourier transform $\hat{f}(\xi)$ is living in the frequency domain.

Definition 2.3.3 (Convolution)**.** *Given two sufficiently nice functions $f(x)$ and $g(x)$, the convolution of them is defined as*

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t) g(x - t) \, \mathrm{d}t. \tag{2.4}$$

Fourier transform has a nice property about convolution, namely the convolution theorem

Theorem 2.3.4 (Convolution Theorem)**.**

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)). \tag{2.5}$$

The theorem basically tells that convolution in time domain is multiplication in frequency domain.

2.3.2 Discrete Fourier Transform

Fourier transform deals with functions defined on $\mathbb{R}$. However, in computer science, numbers are represented and stored discretely. Instead of working with functions defined on $\mathbb{R}$, we need to deal with sequences of N data points. Sequences can be considered as a function defined on a finite set $[0, N-1]$. In this case, we will have a discrete transformation, namely the Discrete Fourier Transform (DFT).

Definition 2.3.5 (Discrete Fourier Transform)**.** *Given sequence* $\{x_0, \ldots, x_{N-1}\}$*, its Discrete Fourier Transform (DFT) is* $\mathcal{DFT}(\{x_0, \ldots, x_{N-1}\}) = \{X_0, \ldots, X_{N-1}\}$ *such that*

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{k}{N} n}. \tag{2.6}$$

Similarly for the inverse transformation

Definition 2.3.6 (Inverse Discrete Fourier Transform)**.** *Given sequence* $\{X_0, \ldots, X_{N-1}\}$*, its Inverse Discrete Fourier Transform (IDFT) is* $\mathcal{DFT}^{-1}(\{X_0, \ldots, X_{N-1}\}) = \{x_0, \ldots, x_{N-1}\}$ *such that*

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{2\pi i \frac{k}{N} n}. \tag{2.7}$$

There is also a discrete version of convolution

Definition 2.3.7 (Discrete Linear Convolution)**.** *The discrete linear convolution of two sequences* $\{x_n\}_{n=0}^{N-1}$ *and* $\{y_n\}_{n=0}^{N-1}$ *is defined as*

$$x_n * y_n = \sum_{m=0}^{N-1} x_m y_{n-m}. \tag{2.8}$$

Then the convolution theorem also holds for discrete linear convolution, namely

$$x_n * y_n = \mathcal{DFT}^{-1}(X_n \cdot Y_n) \tag{2.9}$$

where $\{X_n\} = \mathcal{DFT}(\{x_n\})$ and $\{Y_n\} = \mathcal{DFT}(\{y_n\})$.

2.3.3 Fast Fourier Transform

Fast Fourier Transform (FFT) is a fast implementation of the DFT algorithm [10]. If we consider $\{x_0, \ldots, x_{N-1}\}$ as a vector, the DFT is then realized as a full-rank linear transformation on N-dimensional vector space. The matrix representation of such linear transformation is called the DFT matrix. The naive matrix-vector multiplication has a complexity of $\mathcal{O}(n^2)$ for $n \times n$ matrix. Due to the innate symmetry within the DFT matrix, with divide and conquer, it is possible to break the huge matrix down into smaller block matrices. Such modification reduces the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, and is called the Fast Fourier Transform (FFT). There are multiple ways to divide the matrix, with Cooley-Tucky the most classical one. The computation is represented as the Cooley-Tucky butterflies, which we will describe in later sections.

2.3.4 Polynomial Multiplication and Number Theoretic Transform

Polynomial multiplication is itself a convolution. Given a commutative ring R and polynomials $f, g \in R[x]$ with degree less than N. If we consider the polynomial $f = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1}$ as a sequence $\{a_0, a_1, \ldots, a_{N-1}\}$, then multiplying two polynomial is the same as a linear discrete convolution on the two sequences. We can then naturally use DFT and FFT for fast computation of polynomial multiplication. Since the commutative ring R is usually taken as a finite field $\mathbb{Z}_q$, DFT on coefficients of polynomials is called Number Theoretic Transform (NTT). The inverse transformation is called Inverse NTT (INTT).
In lattice-based cryptography, we usually use quotient polynomial rings like $R[x]/(x^n + 1)$ or $R[x]/(x^n - 1)$ instead of $R[x]$. Polynomial multiplications in these quotient rings are similar, while convolutions in these rings are called cyclic convolutions, slightly different from the linear discrete convolutions. In particular, convolutions of coefficients of polynomials in $R[x]/(x^n + 1)$ are called negative wrapped convolution (because x^n is replaced by -1), and convolutions of coefficients of polynomials in $R[x]/(x^n - 1)$ are called positive wrapped convolution. In our research, since we focus on the Dilithium protocol, we only investigate the case of negative wrapped convolution as used in Dilithium in the ring $\mathbb{Z}_q[x]/(x^{256} + 1)$.

Definition 2.3.8 (NTT with Negative Wrapped Convolution)**.** *Given n a power of 2. Suppose $\mathbb{Z}_q$ with q prime is a finite field with a $2n$-th root of unity denoted ψ. Define*

$$\mathcal{NTT}^{\psi} : \mathbb{Z}_q^n \to \mathbb{Z}_q^n \tag{2.10}$$

$$(a_0, a_1, \ldots, a_{n-1}) \mapsto (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}), \text{where} \tag{2.11}$$

$$\hat{a}_j := \sum_{i=0}^{n-1} \psi^{2ij+i} a_i \tag{2.12}$$

Denote ψ a 512-th root of unity in the field $\mathbb{Z}_q$. In particular, Dilithium uses $\psi = 1753$. Then the fast NTT with negative wrapped convolution is denoted as NTT.
The algorithm can be summarized in butterfly diagrams, and there are two types of NTT used in Dilithium: Cooley-Tucky (CT) and Gentleman-Sande (GS). In particular, CT is used in NTT and GS is used in INTT. The ways of connecting butterflies are summarized in Figure 2.3 for CT butterflies. In Figure 2.3, we are using a smaller degree 7 polynomial and a 3-stage NTT to give an idea of how the CT butterflies look like. In particular, we transform a polynomial g into NTT domain $\hat{g}$ using CT butterflies. The coefficients of g forming the input of CT butterflies are denoted as $g[0], g[1], \ldots, g[7]$. The coefficients in the NTT domain are denoted as $\hat{g}[0], \hat{g}[1], \ldots, \hat{g}[7]$. To read such butterfly diagrams, we use the CT butterfly for example and walk through the computation of $\hat{g}[0]$ to $\hat{g}[7]$ from $g[0]$ to $g[7]$. First trace the red line starting from input $g[0]$ up to the vertical blue line marking the end of Stage 1, and realize the line stops at a level the same as $g[4]$. If tracing the red line starting from input $g[4]$, the line will terminates at the same level as $g[0]$ at the vertical blue line of Stage 1. This implies the 0-th input and the 4-th input are combined to form a butterfly (marked in red). Also notice that a factor $\zeta = \psi^4$ presents under the line of $g[4]$. These information combined tells us that, the 0-th output of Stage 1 is computed as $stage1[0] := g[0] + \psi^4 \cdot g[4]$, and the 4-th output of Stage 1 is computed as $stage1[4] := g[0] - \psi^4 \cdot g[4]$. Similarly, tracing the line of $g[1]$ leads to the same level as $g[5]$, which tells us that the the 1-st output of Stage 1 is computed as $stage1[1] := g[1] + \psi^4 \cdot g[5]$, and the 5-th output of Stage 1 is computed as $stage1[5] := g[1] - \psi^4 \cdot g[5]$. Now we proceed to Stage 1. By tracing yellow arrows

starting from the 0-th output of Stage 1, we realize that it stops at the same level as $stage1[2]$ at the blue vertical line of Stage 2. This tells us that the 0-th output of Stage 2 is computed as $stage2[0] := stage1[0] + \psi^2 \cdot stage1[2]$, and the 2-nd output of Stage 2 is computed as $stage2[2] := stage1[0] - \psi^2 \cdot stage1[2]$, forming the yellow butterfly. Similarly, tracing the green 4-th output of Stage 1, we realize that it stops at the same level as $stage1[6]$. Therefore the 4-th and 6-th outputs of Stage 2 are computed as $stage2[4] := stage1[4] + \psi^6 \cdot stage1[6]$ and $stage2[6] := stage1[4] - \psi^6 \cdot stage1[6]$, forming the green butterfly. Everything remains the same until the output of Stage 3. In particular, the indices for $stage3$ and $\hat{g}$ do not coincide. Instead, the relationship between the indices is bit reversal defined later.
In essence, the reader should realize that each time, tracing one line tells the reader how to compute a butterfly, where each butterfly contains two inputs and two outputs. The two outputs are linear combination of the form $\alpha \pm \zeta\beta$, and the first output is always $\alpha + \zeta\beta$, and the second output is always $\alpha - \zeta\beta$.

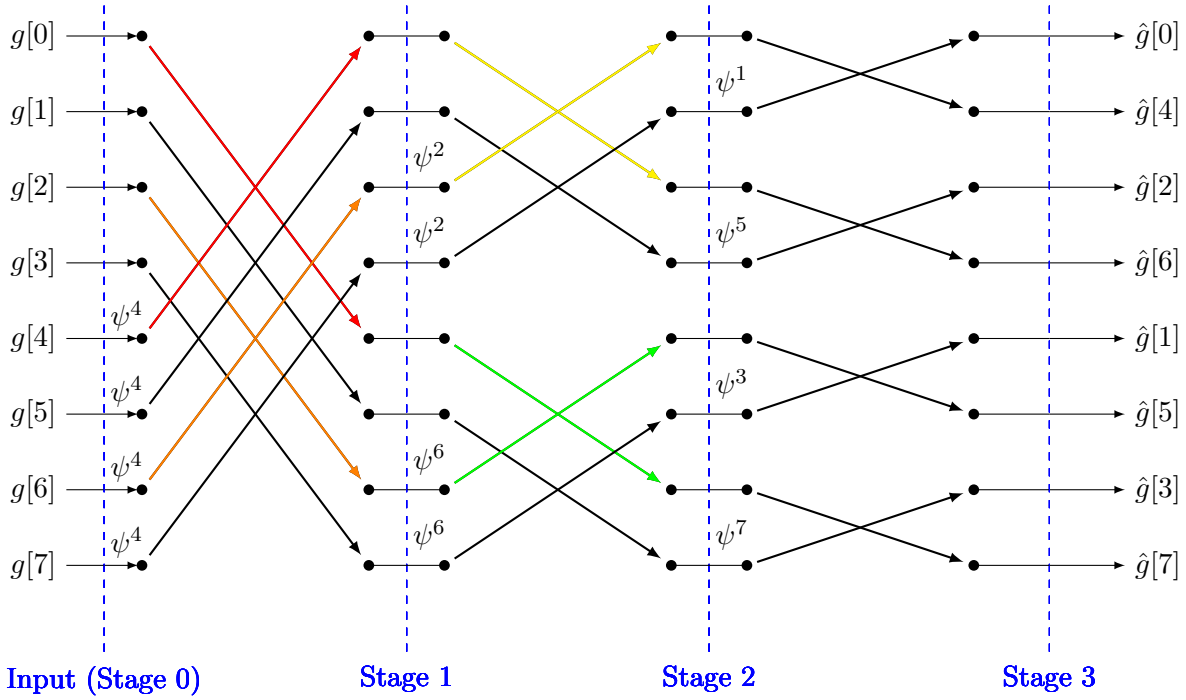

FIGURE 2.3: CT butterflies used in Dilithium with reduced number of coefficients.

Recall that NTT is for computing polynomial multiplications. For example, if we want to multiply two polynomials f and g, that is, we want to compute $h = f \cdot g$. We would first transform $f[0], \ldots, f[7]$ into $\hat{f}[0], \ldots, \hat{f}[7]$, and then transform $g[0], \ldots, g[7]$ into $\hat{g}[0], \ldots, \hat{g}[7]$, all using CT butterflies. Then we compute an elementwise multiplication, namely $\hat{h}[0] := \hat{f}[0] \cdot \hat{g}[0]$, $\hat{h}[1] := \hat{f}[1] \cdot \hat{g}[1]$, and so on. After obtaining $\hat{h}[0], \ldots, \hat{h}[7]$, we transform them back into the normal domain using GS butterflies and obtain $h[0], \ldots, h[7]$, coefficients of the polynomial $h = f \cdot g$.
The number of stages for NTT is $\log_2 N$. Note that for example, from Input to Stage 1, the output is essentially the repetition of a bijective linear transformation $\mathbb{Z}_q^2 \to \mathbb{Z}_q^2$ with matrix representation

$$\begin{pmatrix} 1 & \psi^4 \\ 1 & -\psi^4 \end{pmatrix}$$

where the constant, $\zeta = \psi^4$ in this case, is called the twiddle constant. If we take a look at all the powers of ψ, we realize that the CT butterfly has twiddle factors $\psi^4, \psi^2, \psi^6, \psi^1, \psi^5, \psi^3, \psi^7$. In fact, these numbers $4, 2, 6, 1, 5, 3, 7$ are actually the bit-reversal of numbers $1, 2, 3, 4, 5, 6, 7$. Bit reversal of an integer is defined as the reverse of binary representation of a number. For example, given a 3-bit number 1, represented as `0b001`. Then its bit reversal would be `0b100`, which is 4. More notably, the output of a CT butterfly also arranges in bit-reversal orders (BO). However, Dilithium would not change the bit-reversal ordered output into normal order (NO) so we can simply ignore it. The only pitfall is, we need to use bit-reversal order for GS input when computing INTT. GS will then output coefficients in normal order.
All the butterflies we have seen so far consists of 2 inputs, and 2 outputs. NTT with these kinds of butterflies are called radix-2. If number of coefficients of the polynomial is also a power of 4, we can make a butterfly consists of 4 inputs and 4 outputs. NTT with such kinds of butterflies are called radix-4. Notice that if we build a radix-4 NTT, the twiddle factors will not be the same as radix-2 NTT.

2.3.5 Real-World Implementation

On real-world devices, NTT is implemented with optimizations that deviate from textbook NTT. As an example, we discuss the `GKS20` NTT implementation proposed in [16], also implemented on the Piñata board.
In this particular implementation, integers are signed. Elements in the ring $\mathbb{Z}_q$ are 32-bit integers represented in range $[-\frac{q-1}{2}, \frac{q-1}{2}]$ instead of $[0, q-1]$. Representation of numbers significantly affects the Hamming weight of a number.
Moreover, each NTT butterfly contains one execution of Montgomery multiplication, for computing the multiplication with the twiddle factor. Montgomery multiplication is a constant-time fast algorithm for computing modular multiplication of two integers. When computing $a \cdot b \pmod q$, multiplication takes place in the ring $\mathbb{Z}_{2^k}$ instead of the ring $\mathbb{Z}_q$ because modular reduction in $\mathbb{Z}_{2^k}$ is computationally cheaper and constant time. $\mathbb{Z}_{2^k}$ is called the Montgomery domain. The result is then transformed back from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_q$. In this particular implementation, Greconici *et al.* include the feature "lazy reduction" where a penultimate intermediate result (in a range larger than $[-\frac{q-1}{2}, \frac{q-1}{2}]$) is kept for computing NTT; the final reduction to ring $\mathbb{Z}_q$ is not performed throughout the entire NTT.
On real-world devices, NTT is usually optimized for reducing certain computational overhead, improving security, or saving energy [27]. Greconici, Kannwischer, and Sprenkels have provided a balanced design `GKS20` for ARM Cortex processors by merging two stages of NTT into one loop of iteration. Thus, the execution order of `GKS20` NTT butterflies differs from the textbook NTT design. For example in `GKS20`, butterflies of the first and second stages are computed together, called level merging. An example of 2-level merging, using again Figure 2.3, would be: first compute the red butterfly $g[0]$ and $g[4]$. Next, instead of computing $g[1]$ and $g[5]$, we compute orange butterfly of $g[2]$ and $g[6]$. Then, instead of computing other butterflies in Stage 1, we compute butterflies in Stage 2, specifically, the yellow butterfly $stage1[0]$ and $stage1[2]$, and the green butterfly $stage1[4]$ and $stage1[6]$. We store the computation results, namely $stage2[0], stage2[2], stage2[4], stage2[6]$, and go back to the Input, repeat the same thing with the remaining butterflies, until obtaining the remaining outputs of Stage 2. This example is said to be a 2-level merging that merges the Stage 0 and 1.

2.4 Related Work

2.4.1 Side-Channel Attacks on NTT

While we focus on leakage assessment instead of specific attacks, we are still interested in specific leakages and their countermeasures because they hint us on designing test vectors. Primas *et al.* first demonstrate a single-trace template attacks on INTT, targeting the step where modular reduction is happening [36]. They utilize soft-analytical side-channel analysis [51] and belief propagation for attacking. However, the number of templates required is too large, rendering the attack less practical. The same authors later demonstrate a practical single-trace attack against NTT [35], based on their previous work attacking INTT. With improvement on number of templates needed, they are able to attack a real-world Kyber implementation with 213 templates. Hamburg *et al.* further improve the two previous attack for Kyber [18]. Custers presents a detailed treatment on soft-analytical side-channel analysis against NTT in a master's thesis [11]. Mujdei *et al.* attacks the Toom-Cook and NTT components for KEM algorithms such as Saber and NTRU instead of signature algorithm [30]. Qiao *et al.* present an SIS-assisted attack on NTT [37]. Steffen *et al.* investigate hardware implementation of NTT and propose countermeasures [46].
For countermeasures, Reparaz *et al.* have proposed masking by splitting a polynomial into two parts and then performing NTT on each part independently [41]. Oder *et al.* have proposed hiding on top of masking. Hiding is another countermeasure for SCA against NTT that randomizes the execution order of multiplications in NTT domain [33]. Another way of masking is to mask the twiddle constant [42][39]. Shuffling is another countermeasure that shuffles butterfly execution in NTT [52].
Other non-power-based side-channel attacks are proposed to attack the NTT. For example, Yu *et al.* investigates a CPU frequency-based side-channel attack of NTT [54].

2.4.2 Implementations of NTT

There are various software and hardware implementations of NTT. Our research mainly focuses on the textbook implementation and the implementation optimized on the ARM Cortex-M4 processor [16]. Mert *et al.* present an up-to-date comprehensive survey on software implementation of NTT [27]. For hardware, a list of implementations is provided in [43]. FPGA [26] and ASIC [50] implementations of NTT are also surveyed extensively. One noticeable recent hardware design is specified in [31].

2.4.3 Side-Channel Analysis and TVLA

Although its content mostly covered by [25], Randolph *et al.* provide a concise introduction to power-based SCA in layman's term. Guo *et al.* investigates soft analytical SCA from a coding theory point of view [17].
TVLA methodology is widely applied to both software [48] and hardware [20] implementations of classical public-key algorithms. TVLA is also applied to classical symmetric algorithms such as AES [3][49].

Chapter 3

Plain NTT

We start with a rough characterization of different implementations. We then define plain NTT and focus on algorithms for TVLA on plain NTT. Since plain NTT is never recommended to use in real hardware devices, this chapter focuses on theoretical discussion. Theories discussed in this chapter will be applied to practical cases in the subsequent chapters.

3.1 Characterization of Dilithium NTT Implementation

Numerous variations could present in real-world implementations of NTT. With different optimization targets in mind, one implementation could potentially differ from another in many aspects. To limit our scope, we only discuss the NTT used in Dilithium. We try to characterize Dilithium NTT according to the following aspects that could potentially impact power traces.

Integer representation The input of NTT is a set of coefficients of polynomials in $\mathbb{Z}_q[x]/(x^{256}+1)$. Each coefficient of a polynomial is mathematically an element of $\mathbb{Z}_q$. Usually in real-world implementations, such a number is represented as a signed 32-bit integer x within range $-\frac{q-1}{2} \leq x \leq \frac{q-1}{2}$. However, it is also possible to represent the number as a positive integer within range $0 \leq x \leq q-1$. Note that these ranges are for the input of NTT; the intermediate values of NTT could potentially be in a larger range depending on what kind of modular reduction is used.

Multiplication Within a butterfly of form $\alpha \pm \zeta\beta$, the modular multiplication of a coefficient to a twiddle factor $\zeta\beta \in \mathbb{Z}_q$ is usually implemented with Montgomery multiplication [16]. However, some hardware implementations use Barrett reduction [5] or NewHope-flavored reduction [53] instead.

Twiddle factors storage Twiddle factors can be pre-computed and stored on a device, or computed on-the-fly. In addition, when using Montgomery multiplication, twiddle factors are commonly stored in Montgomery domain for immediate use.

Modular reduction If placed at a wrong place, modular reduction can lead to side-channel vulnerabilities. Lazy reduction is implemented in most real-world cases for avoiding branching during reduction. With lazy reduction, no exact modular reduction is performed throughout the whole NTT process. Some implementation with unsigned integer representation could perform an incomplete reduction by

replacing $\alpha - \zeta\beta$ with $\alpha + (2q - \zeta\beta)$. Note that such incomplete reduction is still constant-time, and is not an exact reduction modulo q.

Level merging Level merging, or stage merging, is an optional optimization strategy by computing multiple NTT stages together, instead of computing one stage after another. The most common way is 2-level merging for Dilithium NTT as in [16]. For Cortex-M4 microcontrollers, implementation with 3-level merging is also proposed [1]. Level merging will change the order of butterflies being computed.

Radix Similar to FFT, Dilithium NTT with $256 = 2^8 = 4^4 = 16^2$ coefficients could potentially be implemented in both radix-2 and radix-4 fashions. While most software implementations on microcontrollers such as [16][1] stick to the standard radix-2 NTT, some hardware implementations include both radix-2 and radix-4 designs [31]. The trade-off is a more complicated computation for twiddle factors because twiddle factors will change in each iteration for high-radix implementations.

Intermediate value handling During NTT execution, intermediate values are read multiple times from the memory, and are written multiple times to the memory. In software implementations, how and when these values are handled could differ by programming languages and compiler settings.

SCA mitigation Side-channel mitigations have been proposed to Dilithium NTT. Shuffling will change the execution order of butterflies. Masking will change the twiddle factors.

Table 3.1 summarizes the characterizations of Dilithium NTT implementations related to our research.

NTT Implementation	Plain NTT	Reference C	`GKS20` ARM Assembly
Integer representation	Unsigned 32-bit integer in range $[0, q-1]$	Signed 32-bit integer in range $[-\frac{q-1}{2}, \frac{q-1}{2}]$	Signed 32-bit integer in range $[-\frac{q-1}{2}, \frac{q-1}{2}]$
Multiplication	Integer multiplication	Montgomery multiplication	Montgomery multiplication
Twiddle factor storage	Pre-computed, in $\mathbb{Z}_q$	Pre-computed, in Montgomery domain	Pre-computed, in Montgomery domain
Modular reduction	Performed after computing each butterfly	Lazy reduction	Lazy reduction
Level merging	No merging	No merging	2-level merging
Radix	Radix-2	Radix-2	Radix-2
Intermediate value handling	Values are read and stored per stage	Values are read and stored in-place per butterfly	Values are read and stored in-place per butterfly
SCA mitigation	None	None	None

TABLE 3.1: Characterizations of implementations related to our research.

3.2 Plain NTT

We define the plain NTT as NTT in its "textbook" form, as described in Section IV.A of [43], with all values represented by unsigned integers. Plain NTT is essentially what we have described in the CT butterflies of Section 2.3.4 but with a larger dimension (256 coefficients instead of 8 coefficients). Modular reduction is performed after computing each butterfly. We assume the modular reduction is time-constant here, for convenience of theoretical discussion and avoidance of timing attack. The plain NTT is rarely implemented in real-world devices, but serves as a basis for our discussion for other real-world variations.

Plain NTT in Dilithium, denoted $S : \mathbb{Z}_q^{256} \to \mathbb{Z}_q^{256}$, is essentially a bijective linear operator on vector space $\mathbb{Z}_q^{256}$ over the finite field $\mathbb{Z}_q$. For the sake of convenience, we say the 0-th stage of NTT is the input of NTT. Define $S_i : \mathbb{Z}_q^{256} \to \mathbb{Z}_q^{256}$ as a map from stage $i-1$ to i. For the sake of convenience, let $S_0 : \mathbb{Z}_q^{256} \to \mathbb{Z}_q^{256}$ be the identity map. Then plain NTT is essentially a composition $S_8 \circ S_7 \circ \cdots \circ S_1 \circ S_0$. Notice that each S_i is a bijective linear transformation, and S_i^{-1} corresponds exactly to a stage in INTT (they differ by when to multiply each $2^{-1} \in \mathbb{Z}_q$). Moreover, each butterfly B_ζ is a bijective linear transformation $B_\zeta : \mathbb{Z}_q^2 \to \mathbb{Z}_q^2$ with standard matrix representation

$$B_\zeta = \begin{pmatrix} 1 & \zeta \\ 1 & -\zeta \end{pmatrix}. \tag{3.1}$$

Butterflies that share the same ζ are said to be "in the same block". Each S_i can be decomposed into 2^{i-1} blocks.

3.3 HW Model

Suppose we want to know if a device running plain NTT leaks the Hamming weight of (at least) one NTT stage. Recall that our plain NTT represents integers in range $[0, q-1]$, where $\lceil \log_2(q-1) \rceil = 23$. Thus for each value, it can take Hamming weight from 0 to 23. Notice that with 256 coefficients, the total Hamming weight of a stage can range from 0 to $256 \times 23 = 5888$.

3.3.1 Test Vector Generation

Our test vectors for both fixed-vs-fixed and fixed-vs-random TVLA should then bias the total Hamming weight of a certain stage. More precisely, we want to find input $x \in \mathbb{Z}_q^{256}$ of NTT such that at stage $i \in [1, 8]$ with Hamming weight $h \in [0, 5888]$ and threshold $k \in [0, 5888]$, the total Hamming weight at stage i differs the Hamming weight at stage $i-1$ by at least k.

Since the plain NTT is fully deterministic, the idea is to generate a stage with a desired Hamming weight, and evolve backwards (by applying S_i^{-1}) to compute the Hamming weight of the previous stage. If the difference in Hamming weight is greater than the threshold, evolve further back to get the input. If the difference in Hamming weight is not large enough, repeat the process until we can find one input that satisfies our needs.

To generate a stage with a desired Hamming weight, we first find a (not necessarily uniformly) random integer partition of h with $(h_0, \ldots, h_{255}) \leftarrow \textbf{Partition}(h)$, that is, $h_0, \ldots, h_{255} \in [0, 23]$ such that $h = h_0 + h_1 + \cdots + h_{255}$. We apply a random permutation σ on indices of $(h_0, h_1, \ldots, h_{255})$ as part of $\textbf{Partition}(h)$. Then, for each h_j, find a random integer $y_j \in [0, q-1]$ such that $\mathsf{HW}(y_j) = h_j$. This is realized by **ReverseHW**

algorithm. Let $y = (y_0, y_1, \ldots, y_{255})$ be the candidate for i-th stage. If $\left|\mathsf{HW}(y) - \mathsf{HW}(S_i^{-1}(y))\right| \geq k$, we output $x = S_1^{-1} \circ \cdots S_{i-1}^{-1} \circ S_i^{-1}(y)$ as the NTT input; if not, we repeat with a new integer partition of h.
Although theoretically we have $h \in [0, 5888]$ and $k \in [0, 5888]$, practically since we want to bias an intermediate stage with small Hamming weight, we are only interested in a restricted range $h \in [0, 200]$ and thus $k \in [0, 5888]$. Note that the Hamming weight of a random vector $x \xleftarrow{\$} [0, q-1]^{256}$ is essentially "independently placing binary `1`'s to 5888 slots with probability roughly equal to $\frac{1}{2}$ for each slot". Thus, the Hamming weight of a random vector $x \xleftarrow{\$} [0, q-1]^{256}$ follows a binomial distribution $\mathsf{HW}(x) \sim B(5888, \frac{1}{2})$. Thus, the expectation of Hamming weight for a random vector is $\mathbb{E}(\mathsf{HW}(x)) = 5888 \cdot \frac{1}{2} = 2944$. By setting $h \leq 200$, we are biasing an intermediate stage such that, a random input only has $\mathbb{P}(\mathsf{HW}(x) \leq 200) = \frac{1}{2^{5888}} \sum_{k=0}^{200} \binom{5888}{k} < 2^{-4633}$ probability of having a comparable Hamming weight. Therefore, when comparing a random vector and a biased vector with Hamming weight as low as 200, we are confident about the effectiveness of biasing.
The threshold k is mainly for "making the biased stage stands out more than other unbiased stages". From a leakage detection point of view, k is not of crucial importance and is usually set with $k = 0$ because in TVLA we are comparing the biased stage with a random stage, not other stages of the test vector itself. By setting a large k we might be able to observe a cleaner signal, but the computational cost for test vector searching could become unnecessarily higher. In our experiment in later chapters, we usually set $k = 0$.
The test vector generation algorithm for plain NTT with HW model **TVGenPlainHW** is summarized in Algorithm 1.

Algorithm 1 TVGenPlainHW

Input: $(i, h, k) \in [1, 8] \times [0, 200] \times [0, 5688]$
Output: $x \in [0, q-1]^{256}$

$x := \perp$
while $x = \perp$ **do**
  $(h_0, h_1, \ldots, h_{255}) \leftarrow$ **Partition**(h)
  $\sigma \xleftarrow{\$} \mathrm{Sym}([0, 255])$
  for $j = 0, 1, \ldots, 255$ **do**
    $y_{\sigma(j)} \leftarrow$ **ReverseHW**(h_j)
  end for
  $y := (y_0, y_1, \ldots, y_{255})$
  if $\left|\mathsf{HW}(y) - \mathsf{HW}(S_i^{-1}(y))\right| \geq k$ **then**
    $x := S_1^{-1} \circ \cdots S_{i-1}^{-1} \circ S_i^{-1}(y)$
  end if
end while

There are multiple ways to implement the random integer partition **Partition**. One naive way is to sample recursively

$$h_0 \xleftarrow{\$} [0, \max(\{h, 23\})] \tag{3.2}$$

$$h_1 \xleftarrow{\$} [0, \max(\{h - h_0, 23\})] \tag{3.3}$$

and so on, until $h = h_0 + h_1 + \cdots + h_j$ and we set $h_{j+1} = h_{j+2} = \cdots = h_{255} = 0$. This would indeed give us a legit random partition but it would generate a vector y with a lot of entries with 0, and high hamming weight concentrated at a single entry of y.

Since we are mostly dealing with small $h \leq 200$, we propose an easy empirical solution for partitioning small h by simply sample each h_j uniformly from the set $\{0, 1, 2\}$ as in Algorithm 2. The problem of random integer partition is complicated and out of scope for our research. For larger h, we could sample each h_j from a Gaussian distribution, or use Fristedt's method [14].

Algorithm 2 Partition

Input: $h \in [0, 200]$
Output: $(h_0, h_1, \ldots, h_{255}) \in \{0, 1, 2\}^{256}$

for $j = 0, 1, \ldots, 255$ **do**
  if $h > 2$ **then**
    $h_j \xleftarrow{\$} \{0, 1, 2\}$
    $h := h - h_j$
  else if $0 \leq h \leq 2$ **then**
    $h_j \leftarrow h$
    $h := 0$
  end if
end for
$h := (h_0, h_1, \ldots, h_{255})$

Algorithm 3, **ReverseHW**, by taking a Hamming weight h, randomly samples a number y within range of Hamming weight h. More precisely, we want $y \xleftarrow{\$} \{z \in \mathbb{Z} | 0 \leq z \leq q-1, \mathsf{HW}(z) = h\}$. Note that this set roughly has size $\binom{23}{h}$ and can be easily computed if h is not too close to $\frac{23}{2}$.

Algorithm 3 ReverseHW

Input: $h \in [0, 23]$
Output: $y \in [0, q-1]$

$Y := \{z \in \mathbb{Z} | 0 \leq z \leq q-1, \mathsf{HW}(z) = h\}$
$y \xleftarrow{\$} Y$

3.3.2 TVLA for HW model

For TVLA, we usually compare two groups of test vectors with Welch's t-test. First we need to determine how many test vectors are placed in each group. Denote N_A the number of test vectors in group A and N_B the number of test vectors in group B. Then we need to determine which stage to bias and set parameters (i, h, k). By performing a fixed-vs-random TVLA in Algorithm 4, we acquire our traces $T_0, T_1, \ldots, T_{N_A+N_B-1}$, where **TraceAcquisition** abstracts the collection of NTT traces.

Algorithm 4 TVLAFvRAcquisition

Input: $N_A, N_B, (i, h, k)$
Output: $T_0, T_1, \ldots, T_{N_A+N_B-1}$
1: $j_A := 0, j_B := 0$
2: $n = 0$
3: $x :=$ **TVGenPlainHW**(i, h, k)
4: **while** $j_A + j_B < N_A + N_B$ **do**
5: $\quad r_n \xleftarrow{\$} \{0, 1\}$
6: $\quad$ **if** $r_n = 0$ and $j_A < N_A$ **then**
7: $\quad\quad T_n \leftarrow$ **TraceAcquisition**$(\mathrm{NTT}(x))$
8: $\quad\quad j_A := j_A + 1$
9: $\quad\quad n := n + 1$
10: $\quad$ **else if** $r_n = 1$ and $j_B < N_B$ **then**
11: $\quad\quad x' \xleftarrow{\$} [0, q-1]^{256}$
12: $\quad\quad T_n \leftarrow$ **TraceAcquisition**$(\mathrm{NTT}(x'))$
13: $\quad\quad j_B := j_B + 1$
14: $\quad\quad n := n + 1$
15: $\quad$ **end if**
16: **end while**

Then our fixed group, group A, would be $A := \{T_n | r_n = 0\}$. Our random group, group B, would be $B := \{T_n | r_n = 1\}$. Note that each T_n is a collection of measurement points $\{(0, v_0), (1, v_1), \ldots, (\tau, v_\tau)\}$ where τ is the total number of time points collected for NTT. v_0 is the voltage signal from the current probe at $t = 0$, and v_τ is the voltage signal from the current probe at $t = \tau$. For convenience of notation, we treat each T_n as a function $T_n : t \mapsto v$, with $T_n(0) = v_0, T_n(1) = v_1, \ldots, T_n(\tau) = v_\tau$, where $t = 0$ is the beginning of NTT, and $t = \tau$ is the end of NTT. Then for each time point, we perform a Welch's t-test among these two groups. To reduce ambiguity, denote t-values with the Gothic $\mathfrak{T}$. We then have, for each time t, a t-value $\mathfrak{T}_t$ computed by

$$\mathfrak{T}_t = \frac{\overline{A_t} - \overline{B_t}}{\sqrt{\frac{s_{A_t}^2}{N_A} + \frac{s_{B_t}^2}{N_B}}} \tag{3.4}$$

where s is the corrected sample standard deviation, $A_t = \{T(t) | T \in A\}$ and $B_t = \{T(t) | T \in B\}$. For all $t = 0, 1, \ldots, \tau$, if there exists one t such that $|\mathfrak{T}_t| > 4.5$, we will have a $> 99.999\%$ confidence that the device leaks side-channel information. However, to confirm that the device is leaky, we usually repeat TVLA at least twice to confirm that the same t point gives us $|\mathfrak{T}_t| > 4.5$.
For fixed-vs-fixed TVLA in Algorithm 5, everything is similar to Algorithm 4, with one difference that x' is generated after line 3 (highlighted in red), so x' is still random, but stays the same during the TVLA.

Algorithm 5 TVLAFvFAcquisition

Input: $N_A, N_B, (i, h, k)$
Output: $T_0, T_1, \ldots, T_{N_A+N_B-1}$

$j_A := 0, j_B := 0$
$n = 0$
$x := \textbf{TVGenPlainHW}(i, h, k)$
$x' \xleftarrow{\$} [0, q-1]^{256}$
while $j_A + j_B < N_A + N_B$ **do**
    $r_n \xleftarrow{\$} \{0, 1\}$
    if $r_n = 0$ and $j_A < N_A$ **then**
        $T_n \leftarrow \textbf{TraceAcquisition}(\text{NTT}(x))$
        $j_A := j_A + 1$
        $n := n + 1$
    else if $r_n = 1$ and $j_B < N_B$ **then**
        $T_n \leftarrow \textbf{TraceAcquisition}(\text{NTT}(x'))$
        $j_B := j_B + 1$
        $n := n + 1$
    end if
end while

The computation for $\mathfrak{T}_t$ and the choice of threshold are all the same with the fixed-vs-random case.

3.4 HD Model

TVLA with HD model reveals if a device is leaking the Hamming distance when computing a certain NTT stage. More precisely, if y is the output of the $(i-1)$-th stage, we would like to know if the device leaks $\mathsf{HD}(y, S_i(y))$. Note that $\mathsf{HD}(y, S_i(y)) = \mathsf{HW}(y \oplus S_i(y))$ where $\oplus$ is the XOR operation.

3.4.1 Test Vector Generation

Our test vectors intend to create an intermediate stage with low Hamming distance between its input and output. These test vectors are suitable for both fixed-vs-fixed and fixed-vs-random TVLA. More precisely we want to find input $x \in \mathbb{Z}_q^{256}$ of NTT with threshold $k \in [0, 5888]$ such that at stage $i \in [1, 8]$, the Hamming distance $\mathsf{HD}(y, S_i(y)) \leq k$ where $y = S_{i-1} \circ S_{i-2} \circ \cdots \circ S_1(x)$. Although ideally, adjacent stages should have larger HD compared to the biased stage $\mathsf{HD}(S_i^{-1}(y), y) \geq \mathsf{HD}(y, S_i(y))$ and $\mathsf{HD}(S_i(y), S_{i+1}S_i(y)) \geq \mathsf{HD}(y, S_i(y))$ for a clear signal, we usually do not enforce this as it could introduce unnecessarily high computational overhead.
Biasing a HD model is harder than biasing the HW model, because it essentially asks "how many bits are flipped after applying a linear transformation", which appears random and lacks proper mathematical tools to tackle the problem. In this case, we can either exploit the "zero trick" (Algorithm 6) or brute-force (Algorithm 8). Instead of considering all 256 coefficients, we break one stage into blocks, and only consider one NTT butterfly per block. This would make the "zero trick" clearer, and the brute-forcing easier.
Recall that butterflies in a block are essentially bijective linear transformations $B_\zeta : \mathbb{Z}_q^2 \to \mathbb{Z}_q^2$ of form $B_\zeta = \begin{pmatrix} 1 & \zeta \\ 1 & -\zeta \end{pmatrix}$. Given an input (α, β), the output is

$(\alpha + \zeta\beta, \alpha - \zeta\beta)$. If $\beta = 0$, we have input $(\alpha, 0)$ and output (α, α), so the Hamming distance is $\mathsf{HW}(\alpha)$. In this case, between stages, we are able to achieve a Hamming distance from 0 to $128 \times 23 = 2944$. Again, for convenience we only consider threshold $k \in [0, 200]$ so we can reuse the **Partition** function in Algorithm 2. It is also possible to play the "zero trick" on the first entry with inputs like $(0, \beta)$ but the idea is similar.

Algorithm 6 TVGenPlainHDZero

Input: $(i, k) \in [1, 8] \times [0, 200]$
Output: $x \in [0, q-1]^{256}$

$(k_0, k_1, \ldots, k_{255}) \leftarrow$ **Partition**(k)
$(\mathrm{idx}_0, \mathrm{idx}_1, \ldots, \mathrm{idx}_{255}) :=$ **Index**(i)
for $j = 0, 1, \ldots, 127$ **do**
    $y_{\mathrm{idx}_{2j}} \leftarrow$ **ReverseHW**$(k_{2j} + k_{2j+1})$
    $y_{\mathrm{idx}_{2j+1}} := 0$
end for
$x := S_1^{-1} \circ \cdots S_{i-2}^{-1} \circ S_{i-1}^{-1}(y)$

We need to figure out the indices for each butterfly in each stage, by Algorithm 7.

Algorithm 7 Index

Input: $i \in [1, 8]$
Output: $(\mathrm{idx}_0, \mathrm{idx}_1, \ldots, \mathrm{idx}_{255}) \in [0, 255]^{256}$

for $j = 0, 1, \ldots, 2^{i-1} - 1$ **do**
    for $k = 0, 1, \ldots, 2^{8-i} - 1$ **do**
        $\mathrm{idx}_{j \cdot 2^{9-i} + 2k} := j \cdot 2^{9-i} + k$
        $\mathrm{idx}_{j \cdot 2^{9-i} + 2k + 1} := j \cdot 2^{9-i} + k + 2^{8-i}$
    end for
end for

Another way of generating test vectors is by a brute-force searching of all possible inputs $(\alpha, \beta) \in [0, q-1]^2$ of a butterfly (note that α and β start from 1 because there is no need to brute-force anything that can be found with "zero trick"). In this case, it is better to first determine the threshold per butterfly k' empirically from the total threshold. For example, we want to bias stage $i = 2$ with total threshold $k = 500$. There are 2 blocks in stage 2, and in our case it is better to set $k' = 10$ and gather one data (α, β) by brute-forcing one of the two blocks ($\zeta = \psi^{64}$ or $\zeta = \psi^{192}$). Then we duplicate (α, β) for $m = 50$ times, and leave all other butterflies $(0, 0)$.

Algorithm 8 ButterflyBruteForcing

Input: $(\zeta, k') \in \{\psi^n \in \mathbb{Z}_q | n \in [0, 255]\} \times [1, 23]$
Output: $(\alpha, \beta) \in [0, q-1]^2$

$i := 1, j := 1$
$(\alpha, \beta) := \perp$
while $(\alpha, \beta) = \perp$ **do**
    if $\mathsf{HD}(i, i + \zeta j) + \mathsf{HD}(j, i - \zeta j) \leq k'$ **then**
        $(\alpha, \beta) := (i, j)$
    else if $j = q - 1$ **then**
        $j := 1$
        $i := i + 1$
    else
        $j := j + 1$
    end if
end while

Then we can build test vectors by stacking **ButterflyBruteForcing**(ζ, k') for m times, and fill 0 with other butterflies in the same block.

3.4.2 TVLA for HD Model

The fixed-vs-random and fixed-vs-fixed TVLA for HD model are exactly the same with TVLA for HW model. The only difference is to replace the **TVGenPlainHW** function with **TVGenPlainHDZero** or results from **ButterflyBruteForcing**, in both Algorithm 4 and 5.

3.4.3 Performance Evaluation

We implemented the brute-forcing algorithm in C language, with maximal parallelism and compiler optimization. Then we run the code on a Windows 10 desktop, with an Intel i7-7700K CPU at 4.2 GHz and 32 GB RAM. For benchmarking, we brute-force the first butterfly of NTT, namely with $\zeta = \psi^{128}$. We record the approximate time T elapsed for generating 1000 test vectors satisfying a threshold k'. The time for generating 1 test vector is then $\frac{T}{1000}$. The measurement is summarized in Table 3.2.

HD Threshold	Time Elapsed for Generating 1 TV
10	35 μs
9	160 μs
8	600 μs
7	3.5 ms
6	20 ms
5	200 ms

TABLE 3.2: Time for brute-forcing one test vector given HD threshold.

We have also brute-forced all possible input for the first butterfly, which took less than 3 days on the desktop. Out of $(q-1)^2$ cases, there are 4 inputs with Hamming distance 1, 286 cases with Hamming distance 2, and 7482 cases with Hamming distance 3. Interestingly there is 1 input $(4553774, 3115892)$ with Hamming distance 46, so all bits are flipped after linear transformation.

3.5 ID Model

Recall the discussion about ID model in Section 2.1.1, the model assumes that only the same intermediate value can produce the same power consumption. Therefore, test vector generation and TVLA for ID model of plain NTT is trivial, because we expect to find leakage between any two test vectors that are different. For fixed-vs-random TVLA, we fix one random vector $x \xleftarrow{\$} [0, q-1]^{256}$ and test against other varying random vectors. For fixed-vs-fixed TVLA, we fix two random vectors $x, x' \xleftarrow{\$} [0, q-1]^{256}$ and test against each other.

Chapter 4

GKS20 NTT

The GKS20 implementation [21][16] is a popular optimized implementation for ARM Cortex-M4 microcontroller. It is also the Dilithium NTT implemented in Piñata.

4.1 Differences between GKS20 and Plain NTT

The GKS20 implementation differs from the plain NTT in the following aspects:

- Integers in GKS20 are signed 32-bit. This means negative values have the highest Hamming weights.
- GKS20 implementation uses Montgomery multiplication with lazy reduction. ζ's are pre-computed and stored in Montgomery domain.
- GKS20 implements 2-level merging.

Each aspect could affect both test vector construction and power traces.

Integer Representation

Since integers have been stored as 32-bit 2-complement signed numbers, the computation for HW and HD and function such as **ReverseHW** should adapt to this construction. In particular, HW should compute a number using its signed 32-bit representation, similar to HD. We will adapt **ReverseHW** into **ReverseSignedHW** for giving a result within the correct range.

Montgomery Multiplication and Lazy Reduction

For convenience of discussion in this section, we use a slightly larger range $[-\frac{q}{2}, \frac{q}{2}]$ instead of $[-\frac{q-1}{2}, \frac{q-1}{2}]$, and $[-q, q]$ instead of $[-(q-1), q-1]$, and so on. In our previous discussion about plain NTT without Montgomery, each stage S_i is a bijective linear transformation $[0, q]^{256} \to [0, q]^{256}$, or effectively with a signed range $[-\frac{q}{2}, \frac{q}{2}] \to [-\frac{q}{2}, \frac{q}{2}]$. Montgomery multiplication has one property: the result of Montgomery multiplication is in a larger range $[-q, q]$. With Montgomery multiplication, in the first stage, the result of $\zeta\beta$ is extended to $[-q, q]$, and thus result of a butterfly $\alpha \pm \zeta\beta$ is in range $[-\frac{3q}{2}, \frac{3q}{2}]$. Note that the result of the first stage is input of the second stage, thus when computing $\zeta\beta$ in the second stage, β is in range $[-\frac{3q}{2}, \frac{3q}{2}]$, but $\zeta\beta$ is in range $[-q, q]$ again, because it is the result of a Montgomery multiplication. Since α is in range $[-\frac{3q}{2}, \frac{3q}{2}]$, the results of second stage will then be in range $[-\frac{5q}{2}, \frac{5q}{2}]$. In the end, the largest range of intermediate value

in the whole GKS20 NTT is $[-\frac{17q}{2}, \frac{17q}{2}]$. Large values can only be attained by coefficients that do not undergo Montgomery multiplication, for example the 0-th coefficient of the whole NTT input. From numerical simulation, we realized that almost all intermediate values lie in range $[-4q, 4q]$.
Montgomery multiplication with lazy reduction not only makes our bruteforcing for a deeper stage harder, but also ruins the one-to-one correspondence between (α, β) and $(\alpha + \zeta\beta, \alpha - \zeta\beta)$. In other words, a butterfly is not a bijective linear transformation anymore, because the result of Montgomery multiplication $\zeta\beta$ will randomly fall either in range $[-\frac{q}{2}, \frac{q}{2}]$, or in range $[-q, -\frac{q}{2}] \cup [\frac{q}{2}, q]$, and accumulated through stages due to lazy reduction.
However, Montgomery multiplication is not utterly disastrous. Fix ζ and denote

$$M_\zeta : \left[-\frac{3q}{2}, \frac{3q}{2}\right] \to [-q, q] \tag{4.1}$$

$$\beta \mapsto \zeta\beta \tag{4.2}$$

This M_ζ will mimic the Montgomery multiplication in most cases. If we set $\zeta = 25847$, which is the ζ used in the entire first stage, a preliminary computation reveals $M_\zeta(-1) = M_\zeta(-1 + q)$, and $M_\zeta(1) = M_\zeta(1 - q)$. This hints that the range of M_ζ does not fill all of $[-q, q]$ but is restricted to a smaller range. With numerical simulation, we discovered that that range of M_ζ is always only slightly larger than $[-\frac{q}{2}, \frac{q}{2}]$, and the mapping M_ζ is almost 3-to-1. (Similarly, we also simulated the case where $\beta \in [-q, q]$ and $\beta \in [-2q, 2q]$, the range of $\zeta\beta$ is similar and the mappings are almost 2-to-1 and almost 4-to-1 correspondingly.) The range of M_ζ associated to the first 30 ζ's are computed in Table 4.1. For each M_ζ, we also find $\varepsilon > 0$ such that its range is within $[-\frac{q}{2} - \varepsilon, \frac{q}{2} + \varepsilon]$.

ζ	$\max(M_\zeta(\beta))$	$\min(M_\zeta(\beta))$	ε
25847	4190274	−4190274	66
−2608894	4197730	−4197730	7522
−518909	4191704	−4191704	1496
237124	4190893	−4190893	685
1826347	4195472	−4195472	5264
2353451	4196981	−4196981	6773
−777960	4192477	−4192477	2269
−359251	4191234	−4191234	1026
−2091905	4196223	−4196223	6015
−876248	4192747	−4192747	2539
3119733	4199217	−4199217	9009
−2884855	4198452	−4198452	8244
466468	4191550	−4191550	1342
3111497	4199262	−4199262	9054
2680103	4197869	−4197869	7661
2725464	4197878	−4197878	7670
2706023	4197967	−4197967	7759
95776	4190474	−4190474	266
1024112	4193173	−4193173	2965
3077325	4199121	−4199121	8913
3530437	4200448	−4200448	10240
−1079900	4193335	−4193335	3127
−1661693	4194970	−4194970	4762
−3592148	4200600	−4200600	10392
3585928	4200517	−4200517	10309
−2537516	4197421	−4197421	7213
3915439	4201560	−4201560	11352
−549488	419178	−4191789	1581
−3861115	4201506	−4201506	11298
−3043716	4199060	−4199060	8852

TABLE 4.1: Ranges of Montgomery multiplication output.

This implies that even if our β's are from a larger range, if $\beta_1 \equiv \beta_2 \pmod{q}$, we have very high chance that $\zeta\beta_1 = \zeta\beta_2 \in [-\frac{q}{2}, \frac{q}{2}]$,

Level Merging

In plain NTT or the Dilithium reference C implementation, NTT is performed stage by stage. Butterflies in the second stage are not executed unless the first stage is finished. In `GKS20`, the first and second stages are computed together, exactly as in Figure 2.3 first computing two butterflies of the first stage (red then orange), then two butterflies of the second stage (yellow then green). In this case, intermediate values of the first stage are never stored in memory. Similarly, the third and fourth stages are computed together, and intermediate values of the third stage are never stored in memory. This hints us that for a software implementation like `GKS20`, per-stage leakages could be too crude, especially for odd-numbered stages. For a more refined characterization of leakage, we need to look into the leakage per butterfly in the next chapter.

4.2 Measurement Setup

We use Piñata board as our target device. Piñata is a development board modified and programmed for side-channel analysis and fault injection training, with its firmware developed by Riscure. It has a STM32F417IG processor with 32-bit ARM Cortex-M4F core, operating at a clock frequency of 168 MHz. We use the UART Interface for communication between our PC and Piñata board, through a FT232RL USB-TTL adaptor. The baudrate is set at 115200, with a timeout of 0.01 second.
Power for Piñata board is supplied with 2 AA alkaline batteries. Between the power supply and Piñata, we place a Riscure Inspector current probe for power consumption signals.
We use a modified custom firmware for executing standalone NTT, made available at `https://github.com/sqmshossifrage/Pinata-dev`.
We use PicoScope 5203 for triggering and power trace acquisition. The oscilloscope has a vertical resolution of 8 bits, and a max sampling rate of 500 MSa/s when two channels are used simultaneously. Table 4.2 summarizes our setup parameters.

Parameters	Channel A	Channel B
Signal Source	Riscure Inspector current probe	Trigger signal from Piñata PC2 pin
Channel Range	200 mV	5 V
Trigger Threshold	N/A	1 V
Trigger Direction	N/A	Rising
Trigger Delay	N/A	5750
Coupling Type	DC	DC
Timebase	1	
Sampling Rate	500 MSa/s	500 MSa/s
Pre-trigger Samples	0	
Post-trigger Samples	34000	
Oversample	0	
Acquisition Mode	Rapid Block Mode	

TABLE 4.2: PicoScope setup parameters.

The PC used for trace acquisition has Intel i7-7600U processor, with 16 GB RAM and Windows 11 Business (version 23H2) operating system. The acquisition scripts are running in a Linux virtual machine. Unless otherwise specified, all TVLAs are performed with collecting $N_A = N_B = 1000$ traces. Our acquisition scripts are running with Python (version 3.11.8), PySerial (version 3.5), and the Python wrapper for PicoSDK. Our acquisition rate for standalone NTT is about 4 traces per second. We also transform our traces into `.trs` files with Riscure's `python-trsfile` tool (version 2.2.2) and then use Riscure's Inspector SCA software (version 2024.1) for correlation analysis.
Figure 4.1 summarizes our hardware setup for trace acquisition.

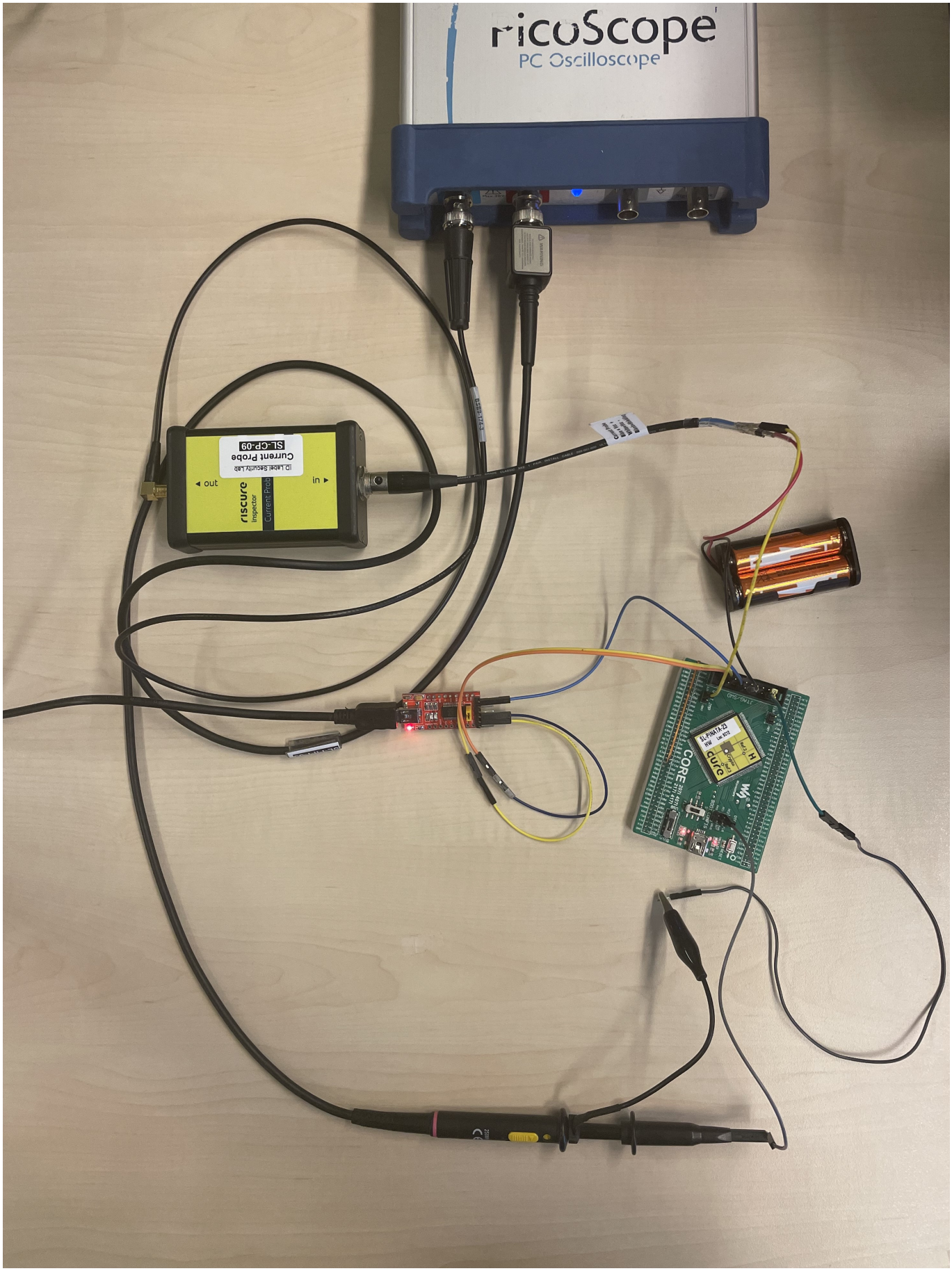

FIGURE 4.1: Hardware setup for experiment.

4.3 Trace Example

We acquire two traces, and plot them as examples of `GKS20` NTT traces in Figure 4.2. The first input `x_A` has all its entries 0, and the second input `x_B` is a random input. Detailed specification of these inputs are presented in Appendix A.1.

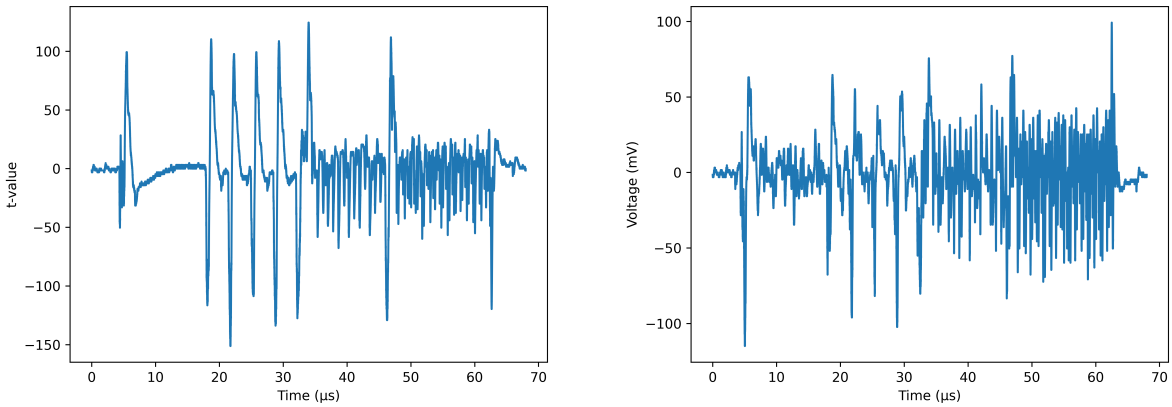

FIGURE 4.2: Example NTT traces with input `x_A` (left) and input `x_B` (right).

4.4 HW Model

4.4.1 Test Vector Generation

We need modifications to plain NTT test vector generation (Algorithm 1) in order to generate better test vectors for `GKS20`. Specifically, we need to modify Algorithm 3 into Algorithm 9, and HW here will compute the Hamming weight of a 32-bit signed integer.

Algorithm 9 ReverseSignedHW

Input: $h \in [0, 32]$
Output: $y \in [-\frac{q-1}{2}, \frac{q-1}{2}]$
1: $Y := \{z \in \mathbb{Z} | -\frac{q-1}{2} \leq z \leq \frac{q-1}{2}, \mathsf{HW}(z) = h\}$
2: $y \xleftarrow{\$} Y$

We define **GKS20NTT**(x, i) an algorithm that takes $x \in [-\frac{q-1}{2}, \frac{q-1}{2}]^{256}$ and $i \in [0, 8]$, that outputs $y \in \mathbb{Z}^{256}$ the intermediate values of i-th stage. Note that due to Montgomery multiplication, y sits in a bigger set compared to x.
When generating test vectors, the strategy is to first generate a test vector similar to Algorithm 1, and then evolve it to stage i using `GKS20` NTT instead of Plain NTT. We introduce a new threshold h' regarding the HW of `GKS20` NTT at the biased stage. Then, we check both thresholds and see if the test vector satisfies them. The whole process is summarized in Algorithm 10.

Algorithm 10 TVGenGKS20HW

Input: $(i, h, h', k) \in [1, 8] \times [0, 200]^2 \times [0, 5688]$
Output: $x \in [-\frac{q-1}{2}, \frac{q-1}{2}]^{256}$

$x := \perp$
while $x = \perp$ **do**
  $(h_0, h_1, \ldots, h_{255}) \leftarrow$ **Partition**(h)
  $\sigma \xleftarrow{\$} \mathrm{Sym}([0, 255])$
  for $j = 0, 1, \ldots, 255$ **do**
    $y'_{\sigma(j)} \leftarrow$ **ReverseSignedHW**(h_j)
  end for
  $y' := (y'_0, y'_1, \ldots, y'_{255})$
  $x' := S_1^{-1} \circ \cdots S_{i-1}^{-1} \circ S_i^{-1}(y')$
  $y :=$ **GKS20NTT**(x', i)
  if $|\mathsf{HW}(y) - \mathsf{HW}($**GKS20NTT**$(x', i-1))| \geq k$ and $\mathsf{HW}(y) \leq h'$ **then**
    $x := x'$
  end if
end while

In particular, usually we have $y' > y$ (which is acceptable if the threshold h' is satisfied), but we could find one case of $y' = y$ with less than 20 trials for smaller i and h, for example, $i = 3$ and $h \leq 20$.
The discussion for k is similar to the discussion in Section 3.3.1 of plain NTT. Usually we set $k = 0$ to avoid the computational cost. Moreover, with Montgomery multiplication, we usually observe a larger gap between Hamming weight of the biased stage and unbiased stage, so usually we are granted to satisfy a large k for free.

4.4.2 TVLA for HW Model

For `GKS20` NTT, there is not much difference in TVLA trace acquisition compared to plain NTT. The only difference being replace **TVGenPlainHW** (Algorithm 1) with **TVGenGKS20HW** (Algorithm 10) in **TVLAFvRAcquisition** (Algorithm 4) and **TVLAFvRAcquisition** (Algorithm 5). The computation for Welch's t-test remains the same as in Section 3.3.2.

4.4.3 Experiment Result

First we bias the second stage and generate a fixed test vector `x_A` with parameters $i = 2, h = 50, h' = 100, k = 0$. The test vector is included in the Appendix A.2.1. In this case, we have biased the second stage of `x_A` to have Hamming weight 94. The evolution for Hamming weights for each stage (with Stage 0 the input vector) of `x_A` is summarized in Table 4.3. We also summarize the evolution for Hamming weights of `x_B`, where `x_B` is the other fixed vector in fixed-vs-fixed TVLA, specified in Appendix A.2.2.

Stage	0	1	2	3	4	5	6	7	8
HW of `x_A`	1037	595	94	555	1626	2754	4084	4209	3989
HW of `x_B`	3912	3875	3907	3899	3981	4043	3854	4014	3746

TABLE 4.3: `GKS20` Test Vector HW Evolution.

We then perform fixed-vs-random TVLA, with $N_A = N_B = 1000$ and we plot the first 5 fixed traces, and the first 5 random traces.

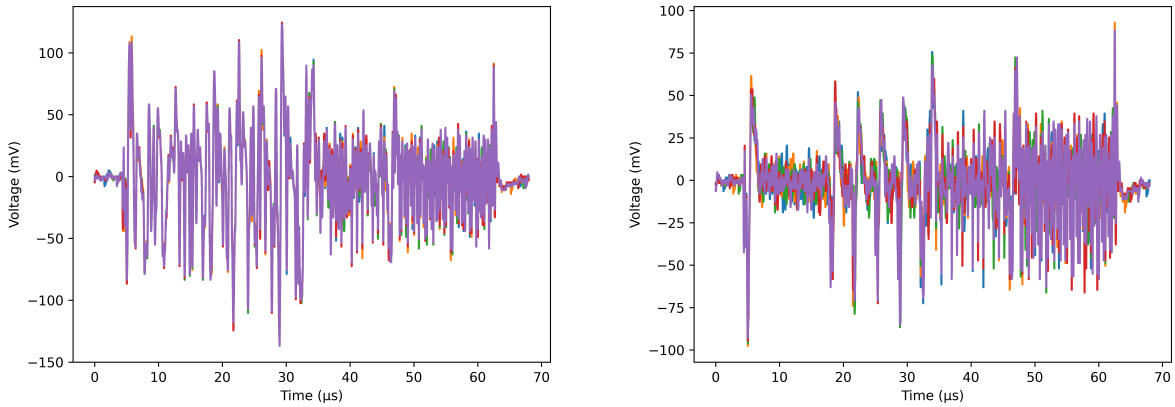

FIGURE 4.3: Example plots of 5 fixed (left) and 5 random (right) traces during fixed-vs-random TVLA for HW model.

We also perform fixed-vs-fixed TVLA, with $N_A = N_B = 1000$, and test vectors specified in the Appendix A.2.2.

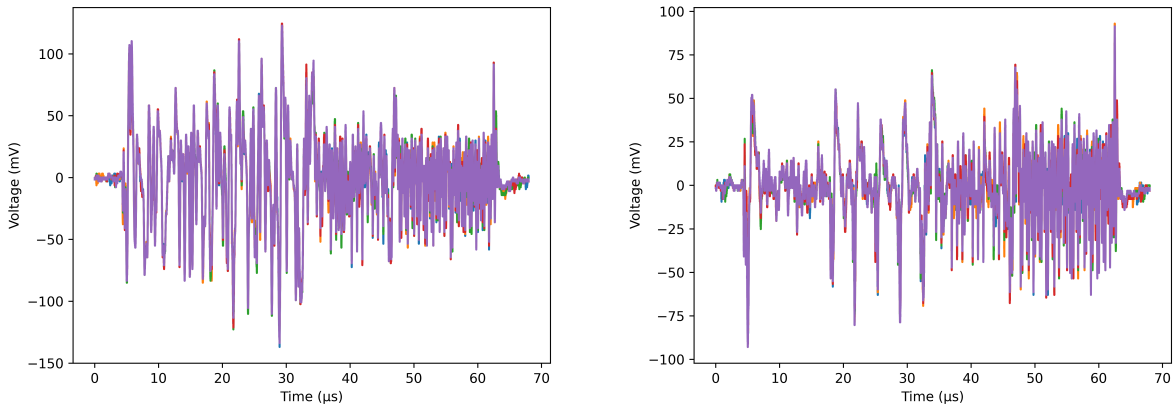

FIGURE 4.4: Example plots of 5 fixed `x_A` (left) and 5 fixed `x_B` (right) traces during fixed-vs-fixed TVLA for HW model.

The t-values versus time plots are shown in Figure 4.5.

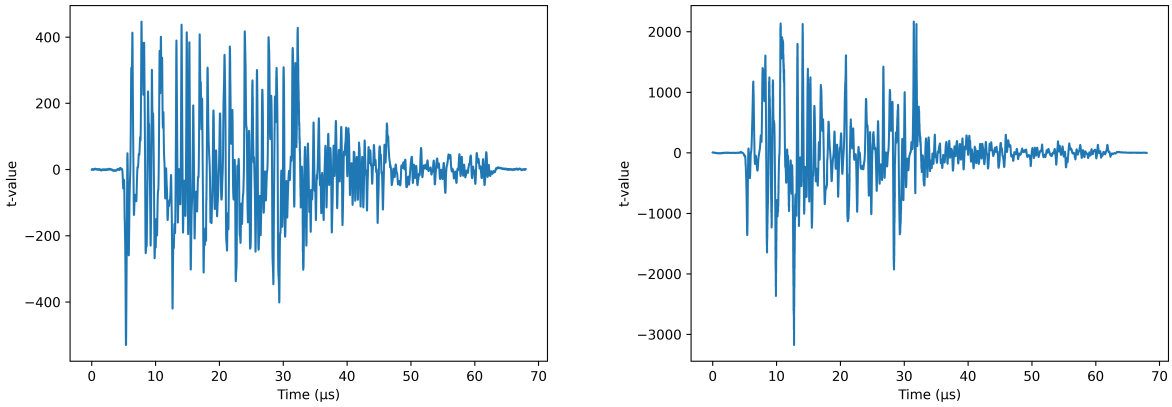

FIGURE 4.5: Example t-values in fixed-vs-random (left) and fixed-vs-fixed (right) TVLA for HW model.

Results for both experiment are summarized in Table 4.4.

	Fixed-vs-Random	Fixed-vs-Fixed
Max t-value	446.576	2167.06
Time offset for max t-value	3904	15755
Min t-value	−530.07	−3174.96
Time offset for min t-value	2683	6372
Leakage detected	Yes	Yes

TABLE 4.4: `GKS20` NTT TVLA for HW model.

4.5 HD Model

4.5.1 Test Vector Generation

Note that even with Montgomery multiplication, the "zero trick" and brute force still works. The only difference is that we need to compute the signed Hamming weight of 32-bit numbers, instead of the unsigned counterpart. Moreover, we need to evolve the test vector to the biased stage and inspect if the Hamming distance is still biased, up to a tolerance of k', just as the h' in **TVGenGKS20HW** (Algorithm 10).
A numerical simulation of averaging the Hamming distance of 1000000 trials of two random numbers shows that $\mathsf{HD}(x, x')$ features a double peaked distribution with an average of 16 for $x, x' \xleftarrow{\$} [-\frac{q-1}{2}, \frac{q-1}{2}]$. The double peaked shape is a result of the 2-complement representation of negative numbers. The histogram of $\mathsf{HD}(x, x')$ is plotted in Figure 4.6.

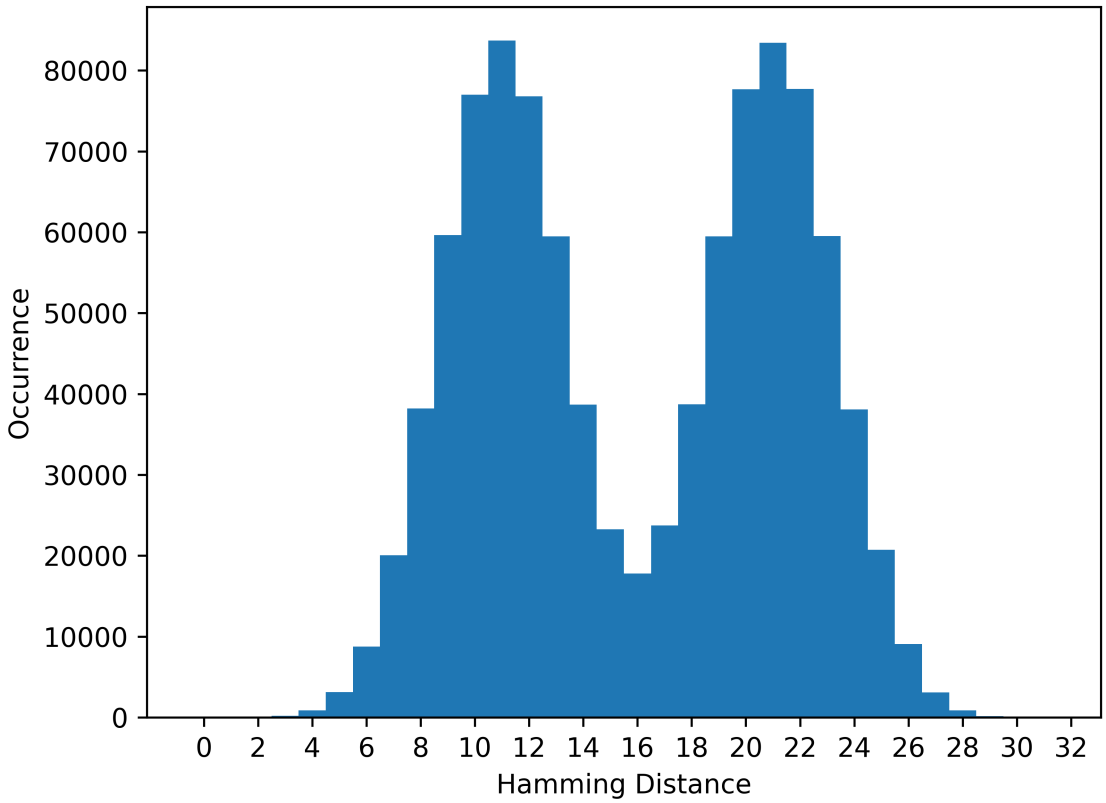

FIGURE 4.6: Histogram of Hamming distance for two random numbers.

Notice that a butterfly is a 2-by-2 transformation involving the Hamming distance change of 2 coefficients. If we assume a linear transformation changes the bits in the input vector randomly, a total Hamming distance of a butterfly is then the sum of two random variables following the above distribution. After randomly sampling 1000000 pairs of random inputs and outputs, we have obtained the Hamming distance distribution of a butterfly plotted in Figure 4.7.

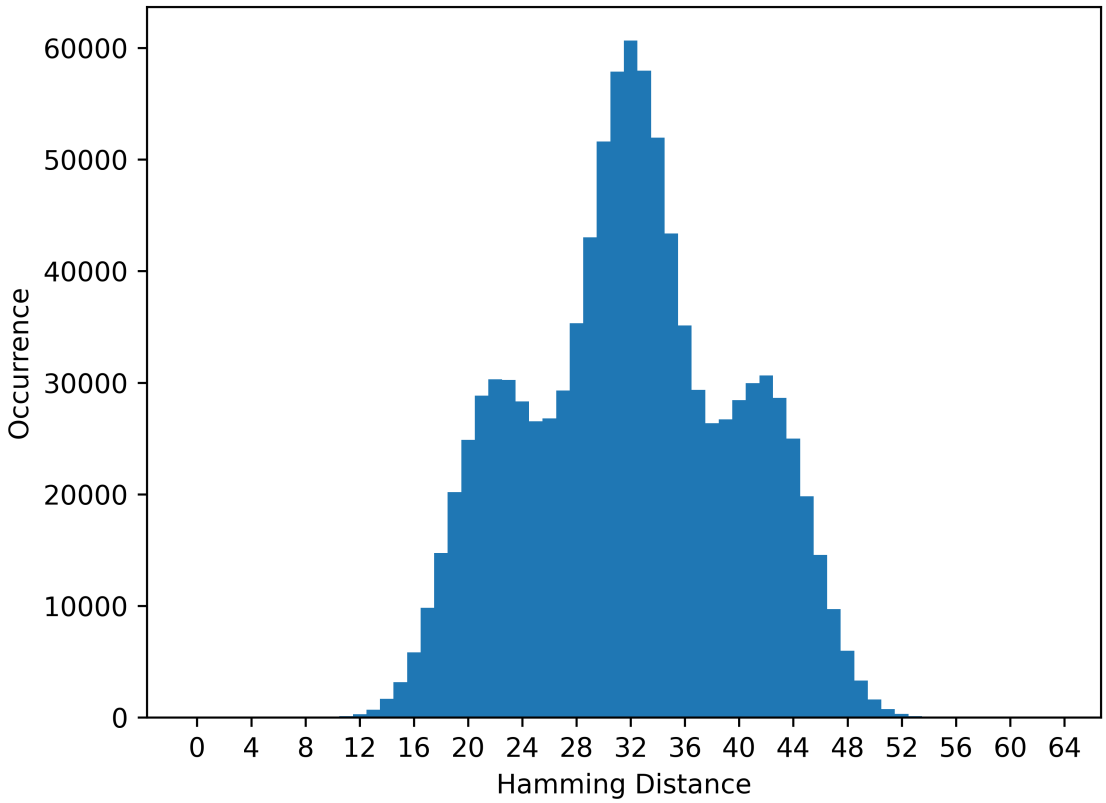

FIGURE 4.7: Histogram of Hamming distance for butterfly.

We have only obtained 41 occurrences with Hamming distance of a butterfly less than 10. An empirical estimation shows that, biasing each butterfly with Hamming distance less than 10 gives us a bias such that a random butterfly only has about 0.04% chance of attaining a comparable Hamming distance.
This suggests that setting a $k' \leq 128 * 10 = 1280$ is usually sufficient for TVLA.
Algorithm 11 summarizes the modified "zero trick" specifically for `GKS20` NTT.

Algorithm 11 TVGenGKS20HDZero

Input: $(i, k, k') \in [1, 8] \times [0, 1280]^2$
Output: $x \in [-\frac{q-1}{2}, \frac{q-1}{2}]^{256}$

$x := \perp$
while $x = \perp$ **do**
  $(k_0, k_1, \ldots, k_{255}) \leftarrow$ **Partition**(k)
  for $j = 0, 1, \ldots, 127$ **do**
    $y_{\text{idx}_{2j}} \leftarrow$ **ReverseSignedHW**$(k_{2j} + k_{2j+1})$
    $y_{\text{idx}_{2j+1}} := 0$
  end for
  $x' := S_1^{-1} \circ \cdots S_{i-2}^{-1} \circ S_{i-1}^{-1}(y)$
  if HD(**GKS20NTT**$(x', i-1)$, **GKS20NTT**$(x', i)) \leq k'$ **then**
    $x := x'$
  end if
end while

4.5.2 TVLA for HD Model

There is not much difference in TVLA trace acquisition compared to plain NTT. The only difference is to replace **TVGenPlainHW** with **TVGenGKS20HDZero** in **TVLAFvRAcquisition** and **TVLAFvRAcquisition**. The computation for Welch's t-test remains the same as in Section 3.3.2.

4.5.3 Experiment Result

We use **TVGenGKS20HDZero** with parameter $i = 2, k = 250, k' = 500$ and generate a fixed test vector. The test vector is included in Appendix A.3.1. In this case, we have biased the second stage of `x_A` to have Hamming distance 307. The evolution for Hamming distances for each stage (with Stage 0 the input vector) of `x_A` is summarized below. We also summarize the evolution for Hamming distances of `x_B`, where `x_B` is the other fixed vector in fixed-vs-fixed TVLA, specified in Appendix A.3.2. Note that Stage i here denotes "the Hamming distance between the input and the output of Stage i".

Stage	1	2	3	4	5	6	7	8
HD of `x_A`	1055	307	3603	3971	3850	3906	3664	3799
HD of `x_B`	3845	3682	3670	3614	3718	3667	3648	3692

TABLE 4.5: `GKS20` Test Vector HD Evolution.

We then perform fixed-vs-random TVLA, with $N_A = N_B = 1000$ and we plot the first 5 fixed traces, and the first 5 random traces.

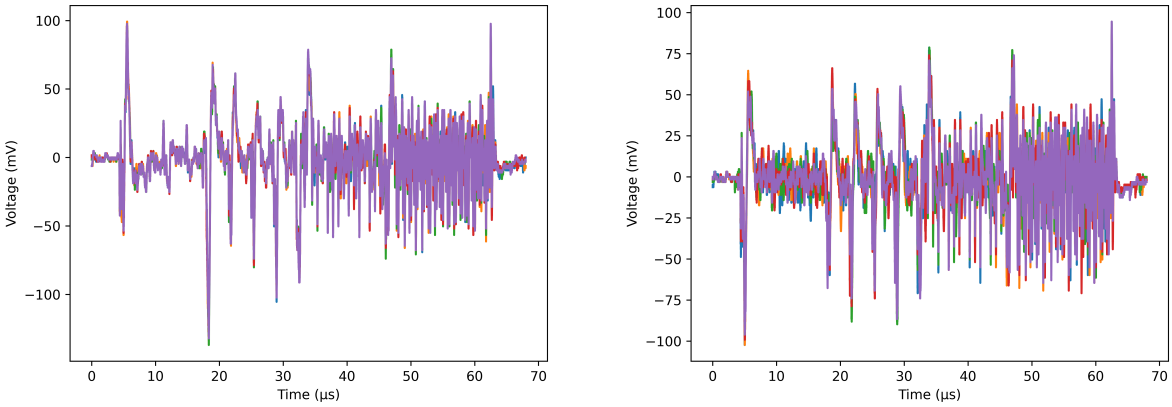

FIGURE 4.8: Example plot of 5 fixed (left) and 5 random (right) traces during fixed-vs-random TVLA for HD model.

We also perform fixed-vs-fixed TVLA, with $N_A = N_B = 1000$, and test vectors specified in Appendix A.3.2.

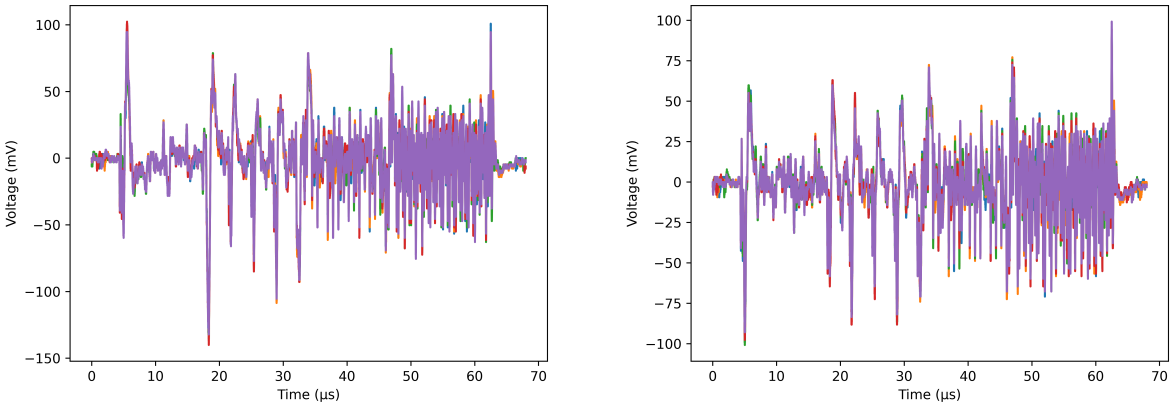

FIGURE 4.9: Example plot of 5 fixed `x_A` (left) and 5 fixed `x_B` (right) traces during fixed-vs-fixed TVLA for HD model.

The t-values versus time plots are shown in Figure 4.10.

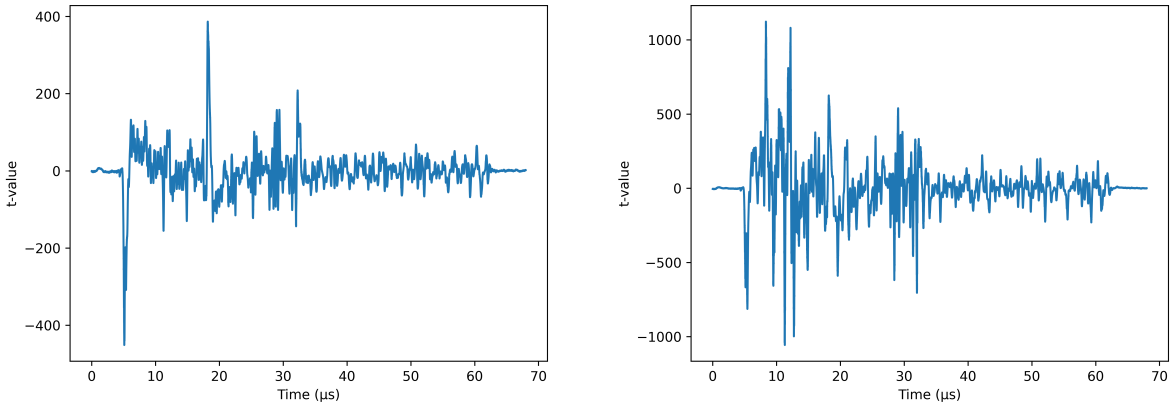

FIGURE 4.10: Example t-values in fixed-vs-random (left) and fixed-vs-fixed (right) TVLA for HD model.

Results for both experiment are summarized in Table 4.6.

	Fixed-vs-Random	Fixed-vs-Fixed
Max t-value	386.69	1123.18
Time offset for max t-value	9094	4169
Min t-value	−451.44	−1056.28
Time offset for min t-value	2552	5650
Leakage detected	Yes	Yes

TABLE 4.6: `GKS20` NTT TVLA for HD model.

4.6 ID Model

ID model is similar to plain NTT, except we need a range $[-\frac{q-1}{2}, \frac{q-1}{2}]$ instead of $[0, q-1]$. Test vector generation and TVLA are also trivial. Note that this does not necessarily imply that the result of TVLA based on ID model is trivial. Instead, devices showing ID leakage should be assessed in a more detailed sense, as what we present in the next chapter.

4.6.1 Experiment Result

Our test vectors `x_A` and `x_B` are specified in the Appendix A.4. For both TVLA, we use $N_A = N_B = 1000$.
For fixed-vs-random TVLA, we plot the first 5 fixed traces, and the first 5 random traces.

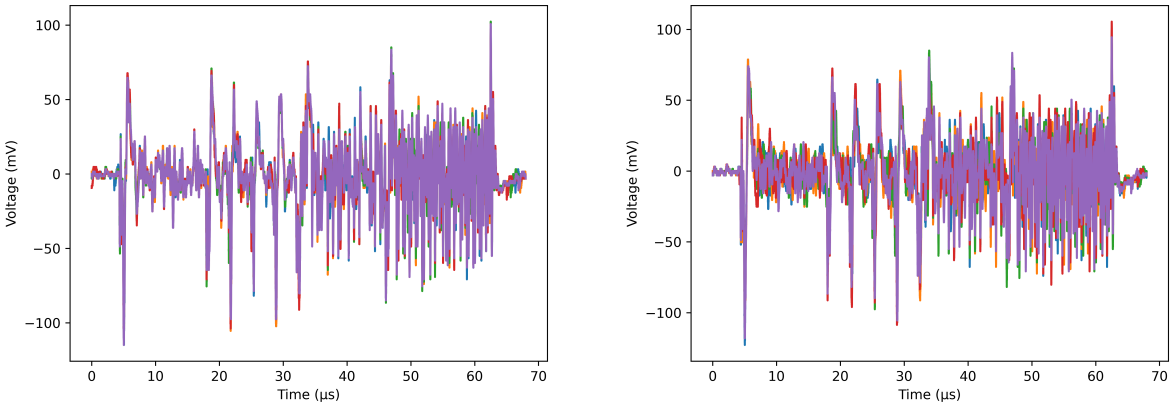

FIGURE 4.11: Example plot of 5 fixed (left) and 5 random (right) traces during fixed-vs-random TVLA for ID model.

For fixed-vs-fixed TVLA, we plot the first 5 fixed traces with input `x_A`, and the first 5 fixed traces with input `x_B`.

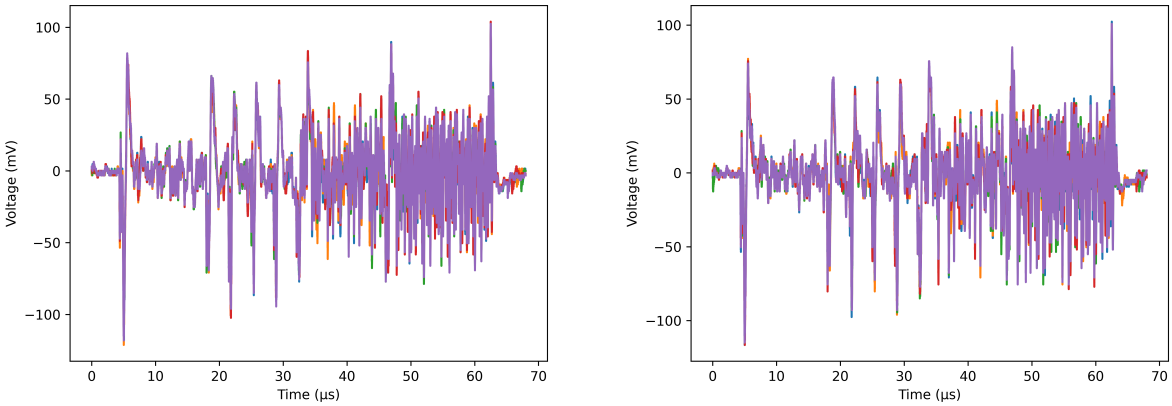

FIGURE 4.12: Example plot of 5 fixed `x_A` (left) and 5 fixed `x_B` (right) traces during fixed-vs-fixed TVLA for ID model.

The t-values versus time plots are shown in Figure 4.13.

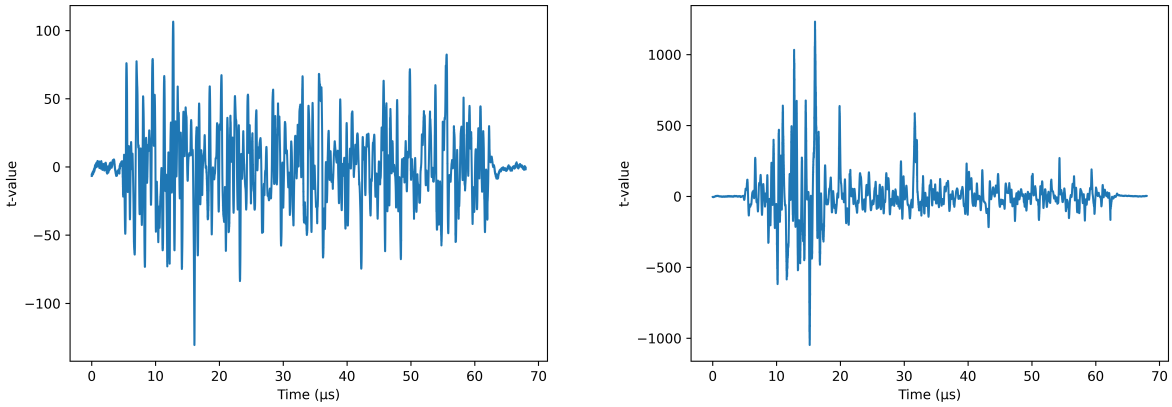

FIGURE 4.13: Example t-values in fixed-vs-random (left) and fixed-vs-fixed (right) TVLA for ID model.

Results for both experiment are summarized in Table 4.7.

	Fixed-vs-Random	Fixed-vs-Fixed
Max t-value	105.24	1231.99
Time offset for max t-value	6384	8015
Min t-value	−130.57	−1047.66
Time offset for min t-value	8048	7599
Leakage detected	Yes	Yes

TABLE 4.7: `GKS20` NTT TVLA for ID model.

Chapter 5

GKS20 NTT Butterfly

From the previous chapter, we realized that Piñata leaks even with ID model. This implies we could potentially characterize the leakage from Piñata in a greater depth. We bias just a few butterflies of Piñata instead of biasing a whole stage. We will launch semi-fixed-vs-random, and semi-fixed-vs-semi-fixed TVLA against the board.

5.1 Structure of a GKS20 NTT Butterfly

One GKS20 NTT Butterfly includes 5 instructions of ARM Assembly code.

```
.macro ct_butterfly_montg pol0, pol1, zeta, q, qinv, th, tl
  smull \tl, \th, \pol1, \zeta
  mul \pol1, \tl, \qinv  // q is -qinv
  smlal \tl, \th, \pol1, \q
  sub \pol1, \pol0, \th
  add.w \pol0, \pol0, \th
.endm
```

In particular, the first three instructions are Montgomery multiplication computing $\zeta\beta$. The last two instructions are linear combinations computing $\alpha \mp \zeta\beta$.
In addition, 4 butterflies are always grouped together to achieve a 2-level merging, In each group, 4 `ldr` operations are performed before 4 butterfly operations. Then 4 `str` operations are performed after butterfly operations to put the computation results back, into the same memory places where they come from. Example assembly code of such grouping is included in Appendix B.1.

5.2 ID Model

Since our target is just a few butterflies, and we already know that Piñata leaks with ID model, it is natural for us to also assume that the butterflies would leak with ID model as well.
To better understand leakage per butterfly, we only focus on the first two stages, thus we will fix a few coefficients of input $x = (x_0, x_1, \ldots, x_{255})$. For example, if we want to fix the first butterfly ever computed in the first stage, we fix input coefficients $\{x_0, x_{128}\}$. If we want to fix the first 4 butterflies computed, with 2 of them in the first stage and 2 of them in the second stage, we fix input coefficients $\{x_0, x_{128}, x_{64}, x_{192}\}$.

5.2.1 Test Vector Generation

When generating test vectors for ID model of butterflies, first we need to determine constrains on coefficients by specifying $C \subseteq [-\frac{q-1}{2}, \frac{q-1}{2}]$. For example, if we are interested in leakage associated to secret key $\mathbf{s}_1$, we need to impose constraint $x_j \in [-4, 4]$ for all j, and set $C := [-4, 4]$. Such C allows us to detect leakages that can potentially be exploited later. Since we mainly use semi-fixed test vectors for `GKS20` butterflies, we then need to determine which coefficients to fix. The fixed coefficients, together with their values, are denoted as a function $F : [0, 255] \to C \cup \{\perp\}$. For example, if we want to fix the first 4 butterflies ever computed, with $x_0 = 1, x_{128} = -2, x_{64} = 3, x_{192} = -4$, we just let

$$F : x \mapsto \begin{cases} 1 & \text{if } x = 0 \\ -2 & \text{if } x = 128 \\ 3 & \text{if } x = 64 \\ -4 & \text{if } x = 192 \\ \perp & \text{otherwise} \end{cases}$$

Test vectors are then generated by Algorithm 12.

Algorithm 12 TVGenGKS20BflyID

Input: $C \subseteq [-\frac{q-1}{2}, \frac{q-1}{2}], F : [0, 255] \to C \cup \{\perp\}$
Output: $x \in C^{256}$

1: **for** $j = 0, 1, \ldots, 255$ **do**
2: **if** $F(j) \neq \perp$ **then**
3: $x_j := F(j)$
4: **else**
5: $x_j \xleftarrow{\$} C$
6: **end if**
7: **end for**
8: $x := (x_0, x_1, \ldots, x_{255})$

5.2.2 TVLA for ID Model

For semi-fixed-vs-semi-fixed TVLA (Algorithm 13), we compare two groups with the same constraint, but different fixed coefficients F_A, F_B. Usually, we perform experiments with F_A and F_B fixing different values for coefficients at the same location. For example, we have F_A fixing $x_0 = 1$ and $x_{128} = -2$; and F_B fixing $x_0 = 0$ and $x_{128} = 1$. Occasionally, we fix coefficients at different locations, for example, F_A fixing $x_0 = 1$ and $x_{128} = -2$; and F_B fixing $x_0 = 1$ and $x_1 = -2$. The choice of F_A, F_B depends on the aim of the experiments.

Algorithm 13 TVLASFvSFAcquisition

Input: N_A, N_B, C, F_A, F_B
Output: $T_0, T_1, \ldots, T_{N_A+N_B-1}$

$j_A := 0, j_B := 0$
$n = 0$
while $j_A + j_B < N_A + N_B$ **do**
    $r_n \xleftarrow{\$} \{0, 1\}$
    if $r_n = 0$ and $j_A < N_A$ **then**
        $x \leftarrow$ **TVGenGKS20BflyID**(C, F_A)
        $T_n \leftarrow$ **TraceAcquisition**$(\mathrm{NTT}(x))$
        $j_A := j_A + 1$
        $n := n + 1$
    else if $r_n = 1$ and $j_B < N_B$ **then**
        $x \leftarrow$ **TVGenGKS20BflyID**(C, F_B)
        $T_n \leftarrow$ **TraceAcquisition**$(\mathrm{NTT}(x'))$
        $j_B := j_B + 1$
        $n := n + 1$
    end if
end while

For semi-fixed-vs-random TVLA, we use Algorithm 13 with a fixed $F_B(x) = \perp$ for all $x \in [0, 255]$.
Then the Welch's t-test remains the same as in Section 3.3.2.

5.2.3 Experiment Result

We have conducted multiple experiments to reveal the leakage gradually. In Experiment 1, we confirm that leakage of a single butterfly is not only possible but shows up clearly. In Experiment 2 we present a stronger leakage from 4 butterflies executed consecutively. Experiment 3 shows the leakage from a single coefficient, and hints that leakages are mostly from memory operations. Experiment 4 is a preliminary exploration of how much of the leakage detected can be used in a template attack against secret key $\mathbf{s}_1$.
Experiment 5 shows that more leakage can be detected if we double the number of TVLA traces.

Experiment 1

First we want to conduct a semi-fixed-vs-random TVLA, and a semi-fixed-vs-semi-fixed TVLA, both with $N_A = N_B = 1000$, to confirm that leakage of a single butterfly actually happens. We fix 2 input coefficients that belong to the same butterfly. We want to choose a good butterfly for a clear signal, that is, the butterfly should not be too close to the beginning or end of a stage (otherwise power consumption would mix with the constant loading at beginning of this or next stage). Thus we choose $C = [-\frac{q-1}{2}, \frac{q-1}{2}]$ and F, F_A, F_B all with x_{31}, x_{159} fixed. The values of the fixed coefficients are specified in Appendix B.2.1.
For semi-fixed-vs-random TVLA, we plot the first 5 semi-fixed traces, and the first 5 random traces.

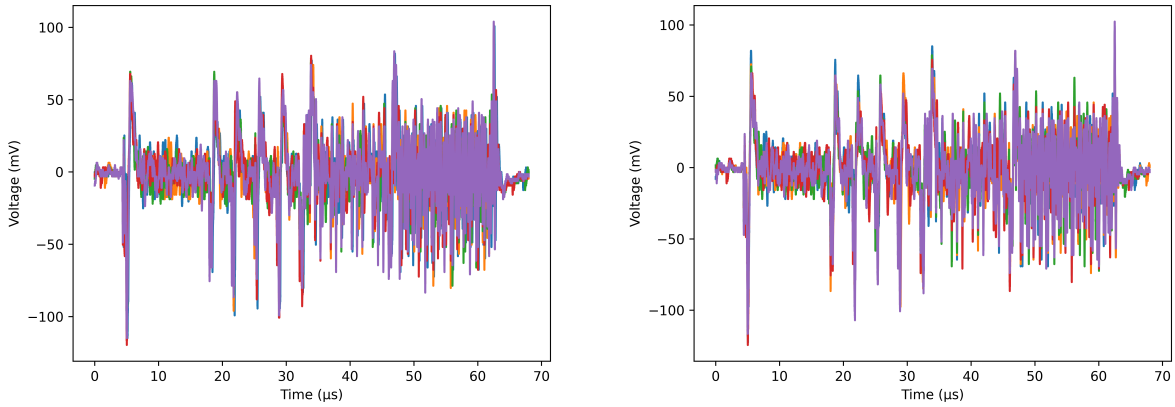

FIGURE 5.1: Example plot of 5 semi-fixed (left) and 5 random (right) traces during fixed-vs-random TVLA for Butterflies in ID model.

Since one butterfly is too fast (only 5 clock cycles), visual inspection does not show a big difference. However, the highest t-value is attained at offset 5795 with value 18.16. We plot 20 semi-fixed traces and 20 random traces between offset 5500 and 6000, to show zoomed-in version of traces. The red vertical line marks the place where the largest t-value is attained.

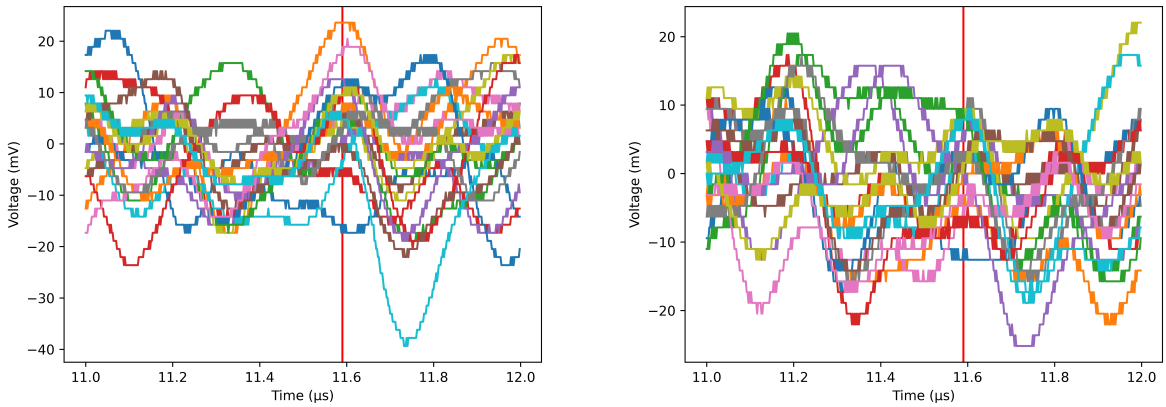

FIGURE 5.2: Zoomed in plot of 20 semi-fixed (left) and 20 random (right) traces during semi-fixed-vs-random TVLA for butterflies in ID model.

For semi-fixed-vs-semi-fixed TVLA, we plot the first 5 semi-fixed traces with input `x_A`, and the first 5 random traces with input `x_B`.

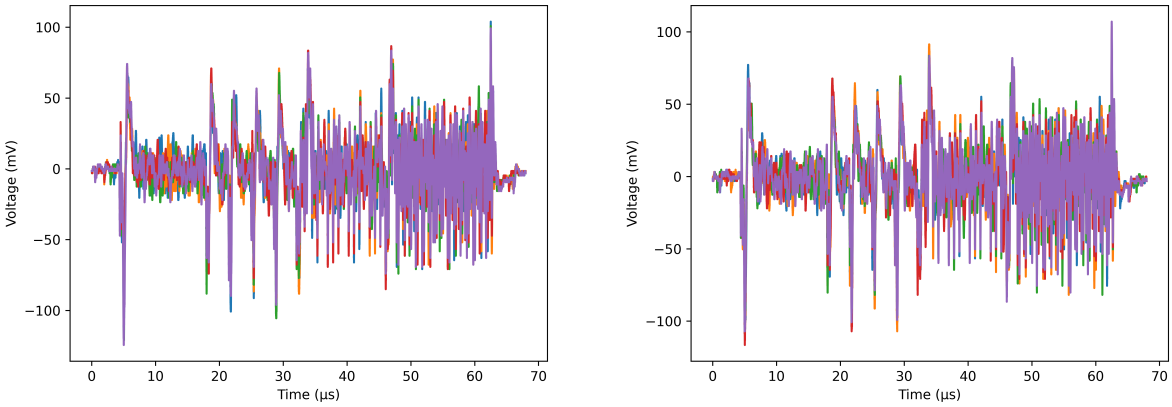

FIGURE 5.3: Example plot of 5 fixed `x_A` (left) and 5 fixed `x_B` (right) traces during semi-fixed-vs-semi-fixed TVLA for butterflies ID model.

We also plot the zoomed-in version of traces, from the same 5500 to 6000 offset, with the red vertical line indicates the place where the largest t-value is attained.

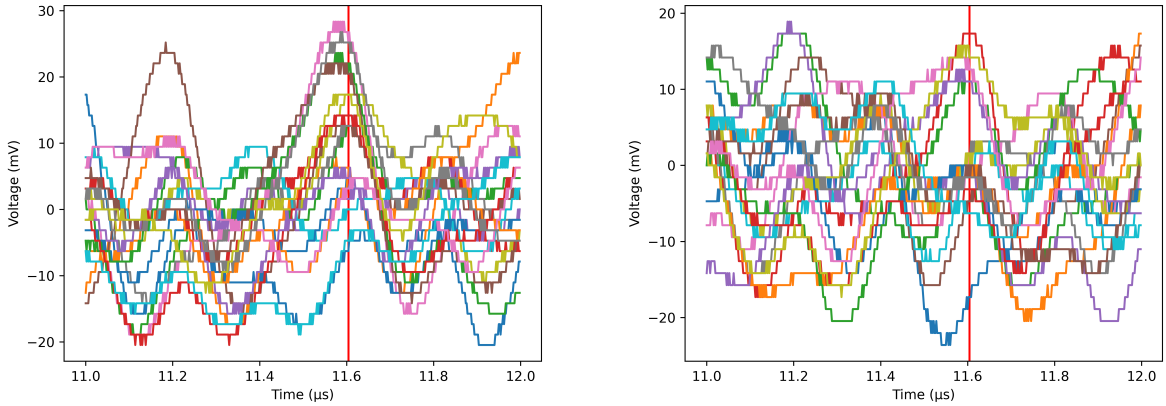

FIGURE 5.4: Zoomed in plot of 20 semi-fixed `x_A` (left) and 20 semi-fixed `x_B` (right) traces during semi-fixed-vs-semi-fixed TVLA for butterflies in ID model.

The t-values versus time plots are shown in Figure 5.5.

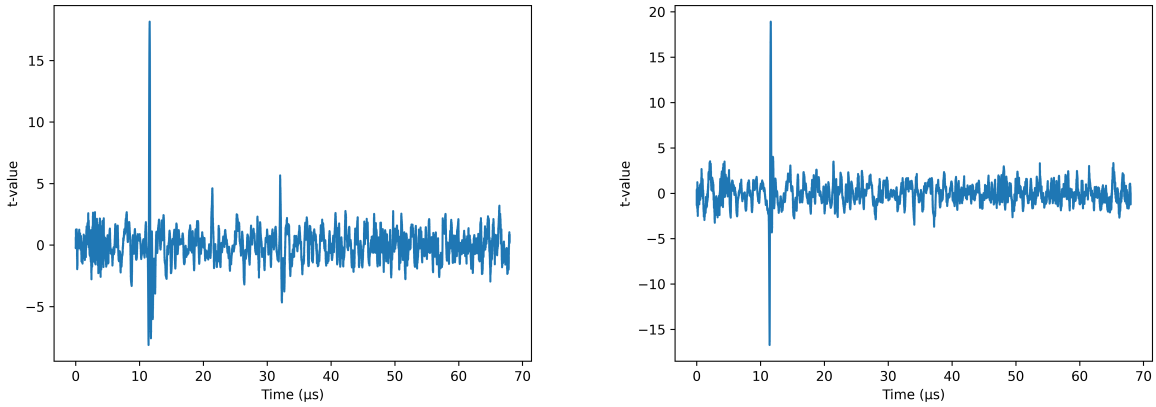

FIGURE 5.5: Example t-values in fixed-vs-random (left) and fixed-vs-fixed (right) TVLA for ID model.

Results for both experiment are summarized in Table 5.1.

	Semi-Fixed-vs-Random	Semi-Fixed-vs-Semi-Fixed
Max t-value	18.16	18.92
Time offset for max t-value	5795	5802
Min t-value	−8.12	−16.72
Time offset for min t-value	5707	5713
Leakage detected	Yes	Yes

TABLE 5.1: `GKS20` NTT Butterfly Experiment 1.

This experiment concludes that even a single butterfly would leak side-channel information.

Experiment 2

We perform a similar experiment to Experiment 1, but with 4 consecutive butterflies fixed. Two of the butterflies are in Stage 1, and two of the butterflies are in Stage 2, because of the 2-level merging. Therefore, we are going to fix $x_{31}, x_{95}, x_{159}, x_{223}$ in all F, F_A, F_B. The test vectors are specified in Appendix B.2.2. Since we are biasing more butterflies, we expect to see a stronger and clearer t-value result.
Since visual inspection does not help much in these semi-fixed TVLA, we would not show the plots for traces. The t-values are plotted in Figure 5.6.

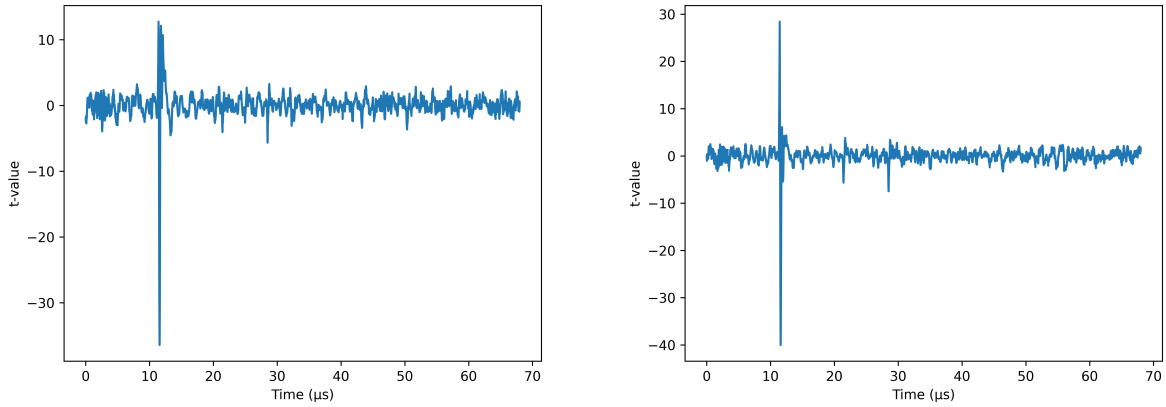

FIGURE 5.6: Example t-values in fixed-vs-random (left) and fixed-vs-fixed (right) TVLA for 4 butterflies in ID model.

Results for both experiment are summarized in Table 5.2.

	Semi-Fixed-vs-Random	Semi-Fixed-vs-Semi-Fixed
Max t-value	12.73	28.42
Time offset for max t-value	5795	5719
Min t-value	−36.43	−40.02
Time offset for min t-value	5787	5796
Leakage detected	Yes	Yes

TABLE 5.2: `GKS20` NTT Butterfly Experiment 2.

This experiment concludes that when fixing 4 butterflies, it would show a stronger leakage of side-channel information.

Experiment 3

Now we have identified leakages within a single butterfly. However, we are not sure if the leakage is from the memory operations (`ldr` and `str`), or the Montgomery multiplication and linear combination. The following experiment hints us that at least part of our leakage is from the `ldr` operation.
We perform a similar experiment to the semi-fixed-vs-random version of Experiment 1, but with only 1 input coefficient fixed. In particular, we use F with x_{31} fixed. The test vector is specified in Appendix B.2.3. Notice that in this case, the only difference between a semi-fixed test vector and a random test vector is the loading part with `ldr`. The semi-fixed test vector will load exactly the same coefficient x_{31} for every run, while the random test vector will load different values for x_{31}. Since the other coefficient of the same butterfly, x_{159}, is random per every run for semi-fixed test vectors, the butterfly operations and `str` operations are essentially processing random inputs. Thus the only difference between the semi-fixed and the random test vectors is the `ldr` operation.

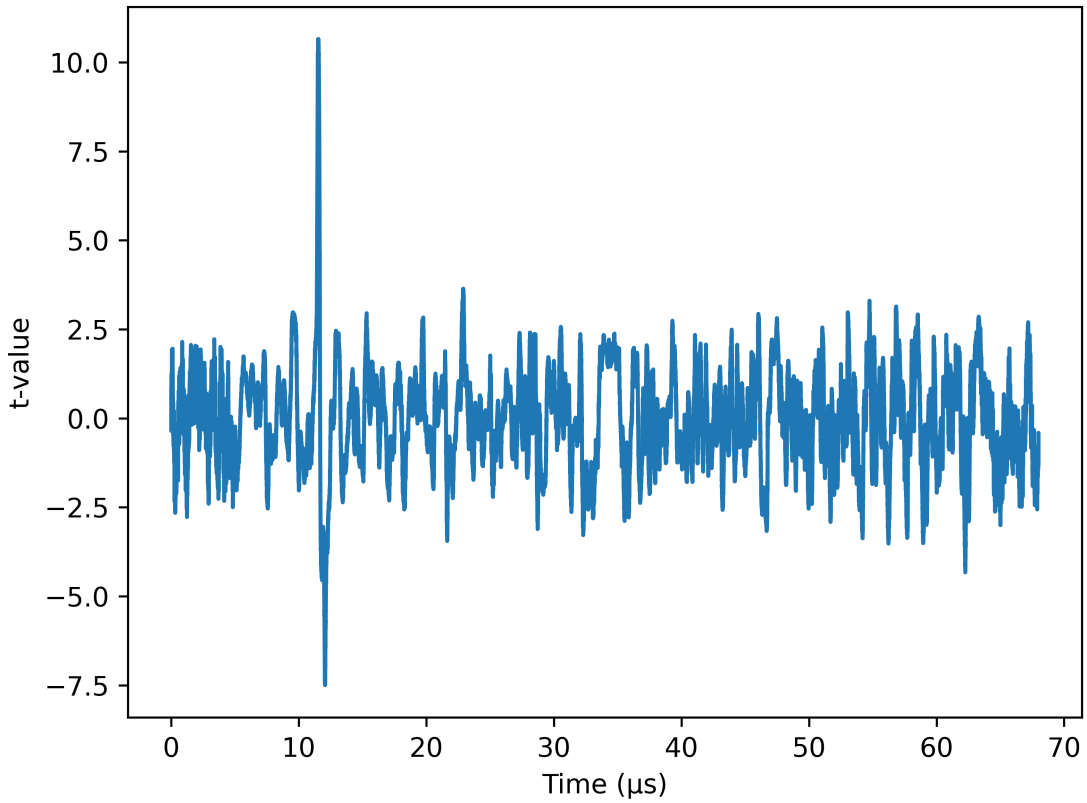

FIGURE 5.7: Example t-values in semi-fixed-vs-random TVLA for a single coefficient in ID model.

Results for the experiment are summarized in Table 5.3.

	Semi-Fixed-vs-Random
Max t-value	10.65
Time offset for max t-value	5768
Min t-value	−7.50
Time offset for min t-value	6030
Leakage detected	Yes

TABLE 5.3: `GKS20` NTT Butterfly Experiment 3.

This experiment concludes that `ldr` does leak side-channel information.

Experiment 4

In this experiment, we try to use an input that mimics the structure of the secret key $\mathbf{s}_1$ of Dilithium. That is, we use restriction $C = [-4, 4]$ to make sure coefficients are within this range. We would like to perform 2 semi-fixed-vs-semi-fixed experiments, by fixing the same places, again x_{31}, x_{159} with $F_A = F_B$. This experiment will show how much the leakages are actually exploitable. If we can observe strong leakage by comparing two different input, we could use the leakage to reduce the entropy for the secret key. In

particular, this time we use hand-crafted test vectors with specific features. In the first experiment, we compare $(x_{31}, x_{159}) = (0, 0)$ versus $(x_{31}, x_{159}) = (1, 0)$. Note that these butterflies have a special property that, the output is the same as the input. Moreover, both Hamming weight differences and Hamming distances between $(0, 0)$ and $(1, 0)$ are quite small. Had we observed any leakage in this experiment, we could conclude that the butterflies are leaking with ID model. In the second experiment, we compare $(x_{31}, x_{159}) = (0, 0)$ versus $(x_{31}, x_{159}) = (0, 1)$. Notice that for this example, the output for a butterfly with input $(0, 1)$ would be $(\zeta, -\zeta)$, and ζ usually has larger Hamming weights. Test vectors are specified formally in Appendix B.2.4.

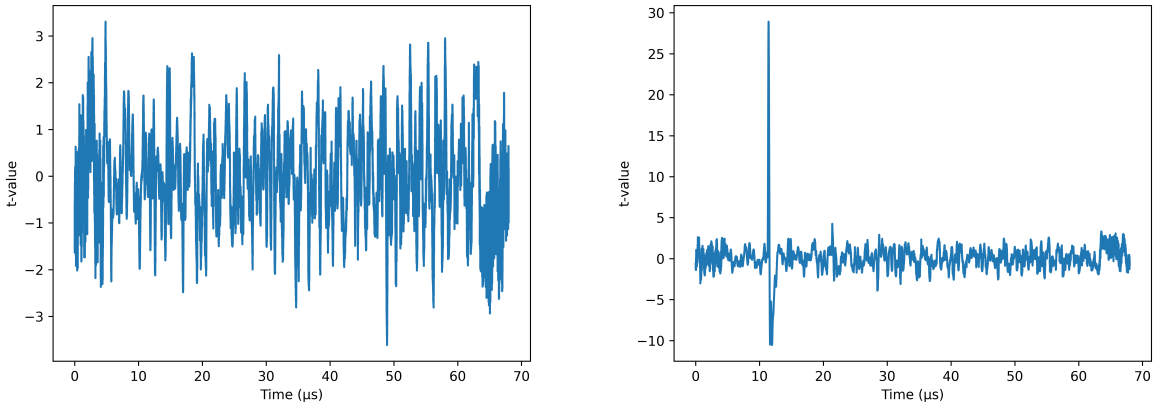

FIGURE 5.8: Example t-values in semi-fixed-vs-semi-fixed TVLA for first (left) and second (right) experiments.

Results for the experiment are summarized in Table 5.4.

	Semi-Fixed-vs-Semi-Fixed First	Semi-Fixed-vs-Semi-Fixed Second
Max t-value	3.31	28.92
Time offset for max t-value	2420	5711
Min t-value	−3.61	−10.54
Time offset for min t-value	24471	5973
Leakage detected	No	Yes

TABLE 5.4: `GKS20` NTT Butterfly Experiment 4.

This experiment concludes that butterflies are probably not leaking according to ID model, because the first experiment does not show leakage. It also tells us that some butterfly inputs are very hard to distinguish, thus less exploitable; while some other butterfly inputs are easy to distinguish, thus more exploitable in a template attack.

Experiment 5

We conduct a further experiment to investigate the number of traces needed to detect a leakage. We again perform two experiments, both with the same coefficients X_{31}, x_{159} fixed with the same values: $(0, 2)$ versus $(0, 4)$. In the first experiment, we use the ordinary $N_A = N_B = 1000$ number of traces throughout our whole thesis. In the second experiment, we double the number of traces collected for both Group A and Group B

$N_A = N_B = 2000$. The experimental result shows that while increasing the number of traces, some inputs could show leakage that is not otherwise observed with less traces. The test vectors are specified in Appendix B.2.5.

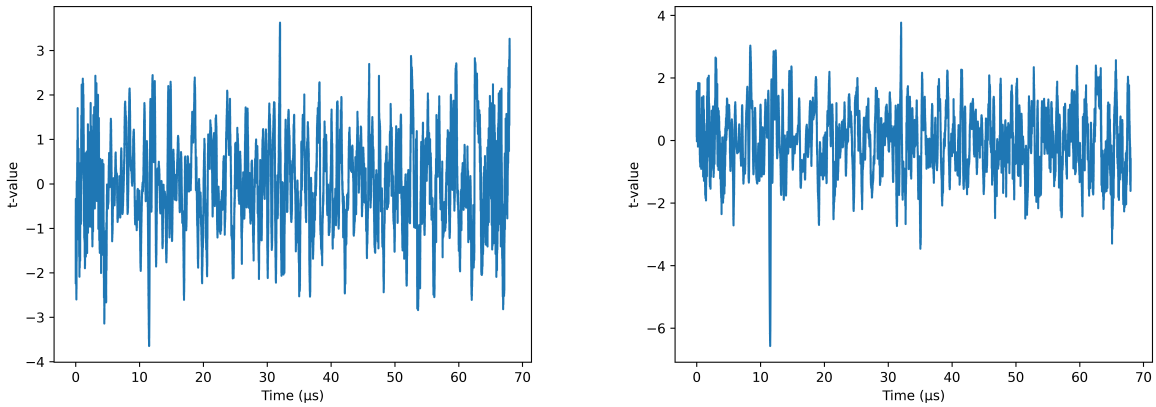

FIGURE 5.9: Example t-values in semi-fixed-vs-semi-fixed TVLA for first (left) and second (right) experiments.

Results for the experiment are summarized in Table 5.5.

	Semi-Fixed-vs-Semi-Fixed First	Semi-Fixed-vs-Semi-Fixed Second
Max t-value	3.63	3.77
Time offset for max t-value	16007	16015
Min t-value	−3.65	−6.57
Time offset for min t-value	5749	5754
Leakage detected	No	Yes

TABLE 5.5: `GKS20` NTT Butterfly Experiment 5.

5.3 Correlation Analysis

We already know from the assembly code that with each iteration, 4 butterflies are enclosed by 4 memory reading and 4 memory writing operations. We are also hinted that the `ldr` operation would leak side-channel information. However, we are not sure if there is any other part that leaks more side-channel information. Also we have used the ID model for most of the discussion, but we also realize that the ID model does not best describe the leakage presented in Piñata. Thus we perform a correlation analysis and figure out which factor contributes the most to the traces.
Correlation analysis is similar to DPA, but without the attack part. We first collect N traces with known input x. Each trace is collected with N_{IP} time points. For each trace, we select and compute intermediate values of our interest. Then for a specific type of intermediate value, we compute the correlation of that value and the trace voltage for every time point.
In our experiment, we collect $N = 10000$ traces, with known but random input $x \xleftarrow{\$} [-\frac{q-1}{2}, \frac{q-1}{2}]$. For each trace, we collect with $N_{IP} = 34000$ time points. We choose our

interesting intermediate values by grouping 4 input coefficients, where these 4 coefficients are computed in the same iteration of Stage 1 and 2 (2 butterflies in Stage 1 and 2 butterflies in Stage 2). Since the number of input coefficients is 256, we have $\frac{256}{4} = 64$ such groupings. For each grouping, the interesting intermediate values of our choice are

- Hamming weights of 4 input coefficients. This indicates the correlation between `ldr` and power consumption.
- Hamming weights of 4 Stage 2 output. Notice that the output coefficients after Stage 2 stay at the same place as the input coefficients. This indicates the correlation between `str` and power consumption.
- Hamming weights of 4 input and 4 Stage 2 output combined. Note that this combines the `ldr` and `str`, thus indicates the correlation between memory operations and power consumption.
- Hamming weights of 4 Stage 1 output. Note that the results of Stage 1 are never stored in the memory. This reveals the effectiveness of biasing the Hamming weight of an odd-numbered stage in this 2-level merged `GKS20` implementation.
- Hamming distance of 4 input coefficients and 4 Stage 1 output. This reveals the effectiveness of biasing the Hamming distance of a single stage in `GKS20` implementation.
- Hamming distance of 4 input coefficients and 4 Stage 2 output. This reveals the effectiveness of biasing the Hamming distance of two stages in `GKS20` implementation.

5.3.1 Experiment Result

After computing the correlation for all the aforementioned interesting intermediate values, we conclude that the `str`, and the `ldr` and `str` combined, have the best correlation with our traces. This implies that the most leakage of NTT comes from memory operation during execution of NTT.
We first plot the correlation of Hamming weight of input and Stage 2 output with different grouping in Figure 5.10. Note that the 0-th grouping contains the very first 4 butterflies being computed, and the 63-th grouping contains the very last 4 butterflies being computed during Stage 1 and 2. Moreover, the 31-th grouping contains the 4 butterflies that we have conducted the most of our TVLA experiments on. Notice that correlation is a dimensionless value from -1 to 1. When reading the plots, the y-axis always has range from -1 to 1. The `*100m` label indicates that the numbers on y-axis should be read as multiplied with $100 \times 10^{-3} = 0.1$. For example, the 5 labeled on the y-axis actually represents 0.5, and the 10 labeled on the y-axis actually represents 1.
We realize that peaks of correlation are moving to the right when grouping indices increase, because these peaks are roughly located at the place where these groupings are executed, thus confirming the execution order of the groupings. For groupings with smaller indices $(0, 7, 15)$, correlations also propagate to deeper stages, especially in Stage 3 and 4. For groupings with larger indices $(31, 63)$ correlations propagated to deeper stages are negligible.

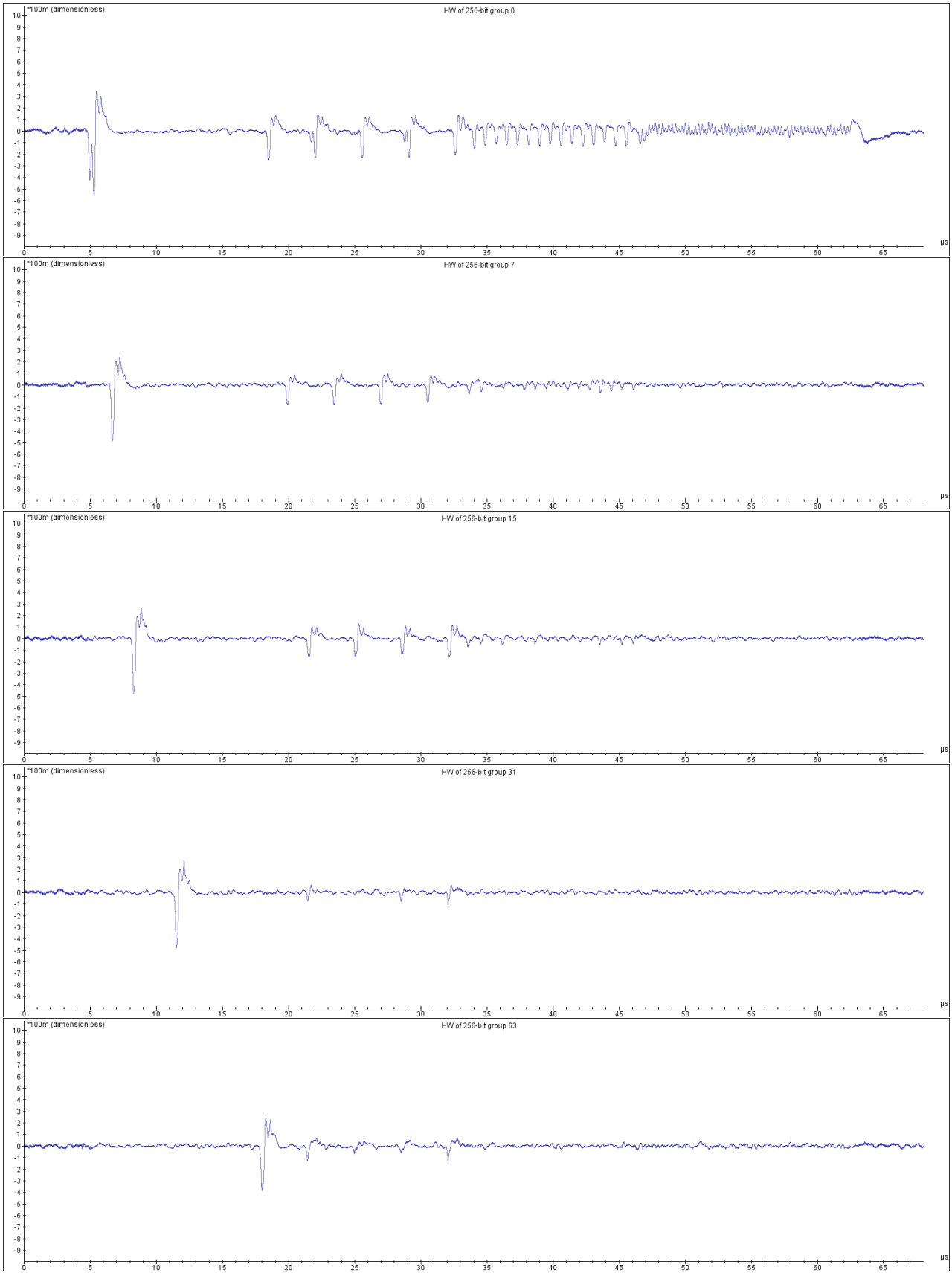

FIGURE 5.10: Correlation of Hamming weight of input and Stage 2 output with $0, 7, 15, 31, 63$-rd grouping.

We also plot 5 traces on top of each other to illustrate the order of butterfly executions in Figure 5.11. Here we stack plots from the 31-st grouping to the 35-th grouping and we can clearly see the time progression of these butterfly groupings, although execution of a grouping is very fast and the correlations are not very well separated among groupings.

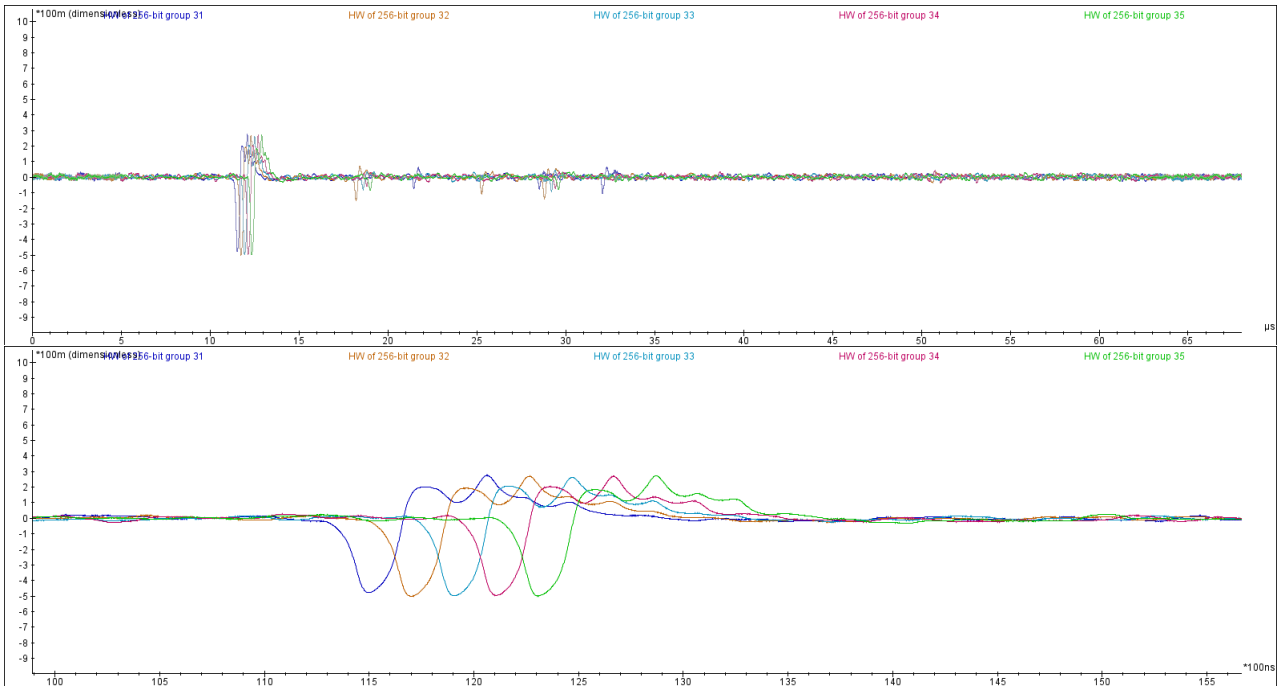

FIGURE 5.11: Correlation of Hamming weight of input and Stage 2 output with 31-st to 35-th grouping stacking.

We would also include the correlation of just the input in Figure 5.12, and of just the Stage 2 output in Figure 5.13. From visual inspection, we compare them with Figure 5.10 and conclude that the benefit of including the HW of the input in addition to the HW of Stage 2 output is negligible. Therefore we believe Stage 2 output is likely to be the main contributor for leakages, corresponding to the 4 memory writing operations within a grouping.
Moreover, we realize that Stage 2 output is also Stage 3 input. Thus when computing Stage 3 and Stage 4, these values are again loaded into memory with `ldr`. That is probably the reason why we can observe correlations for Stage 2 output even in Stage 3 and Stage 4.

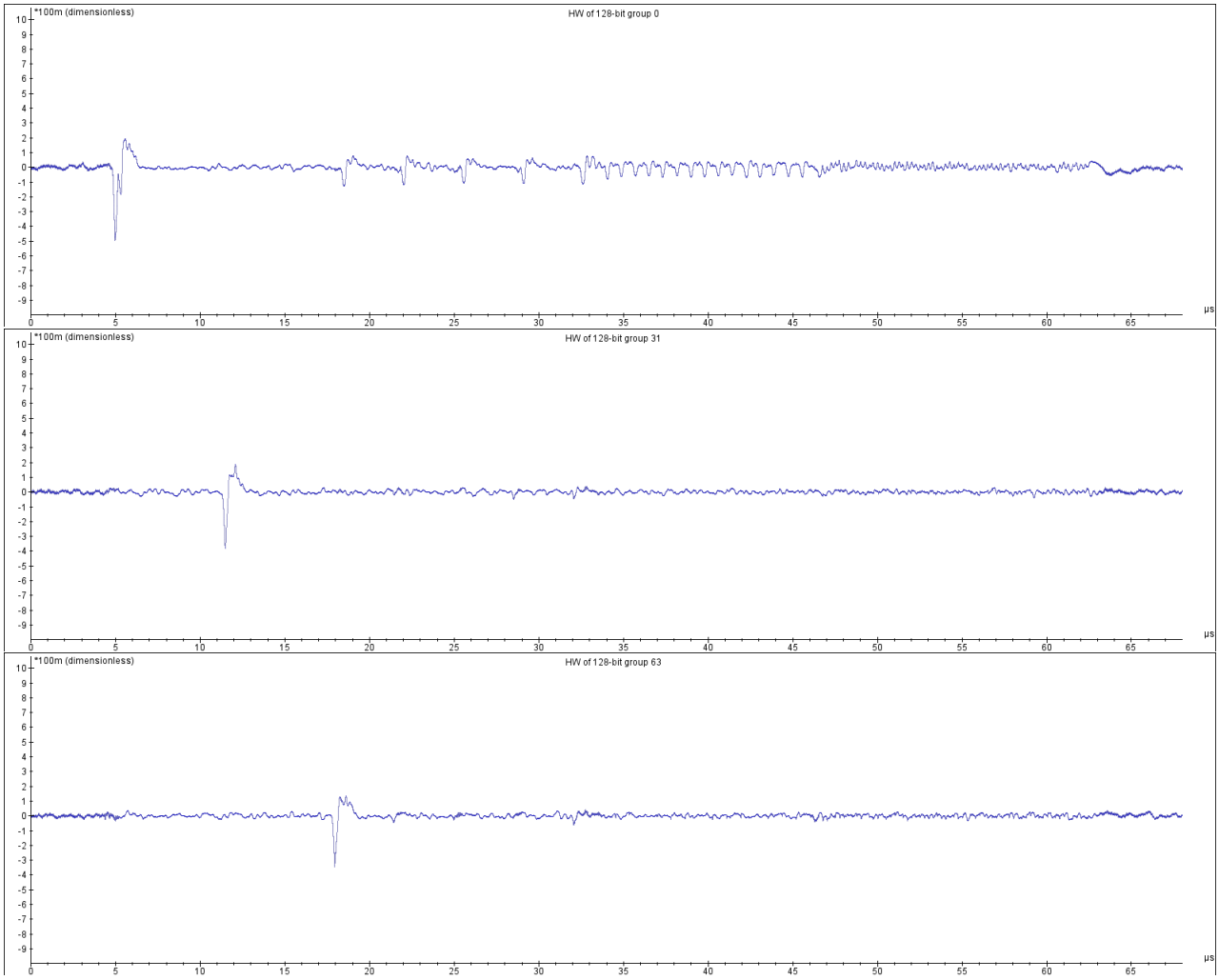

FIGURE 5.12: Correlation of Hamming weight of input alone, with $0, 31, 63$-rd groupings.

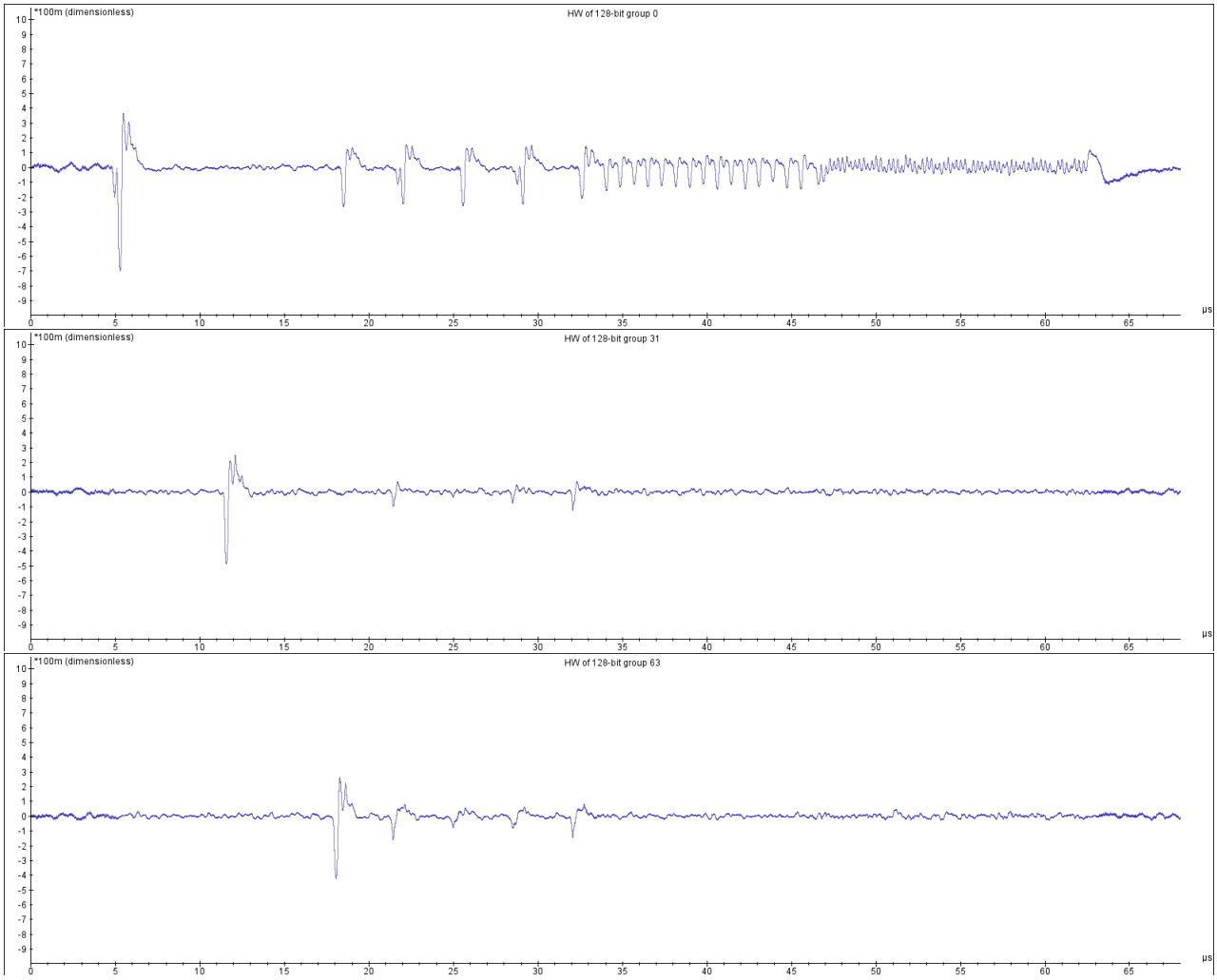

FIGURE 5.13: Correlation of Hamming weight of Stage 2 output alone, with $0, 31, 63$-rd groupings.

We also test the HW of Stage 1, and discover that the HW of the Stage 1 output has a noticeable correlation with our traces. We believe such correlation partly comes from the parity of the input. However, the correlation of Stage 1 output is weaker than the correlation of Stage 2 output. Therefore the correlation of Stage 1 is probably due to an arithmetic relationship between the HW of intermediate values of Stage 1, and the HW of intermediate values of Stage 2.

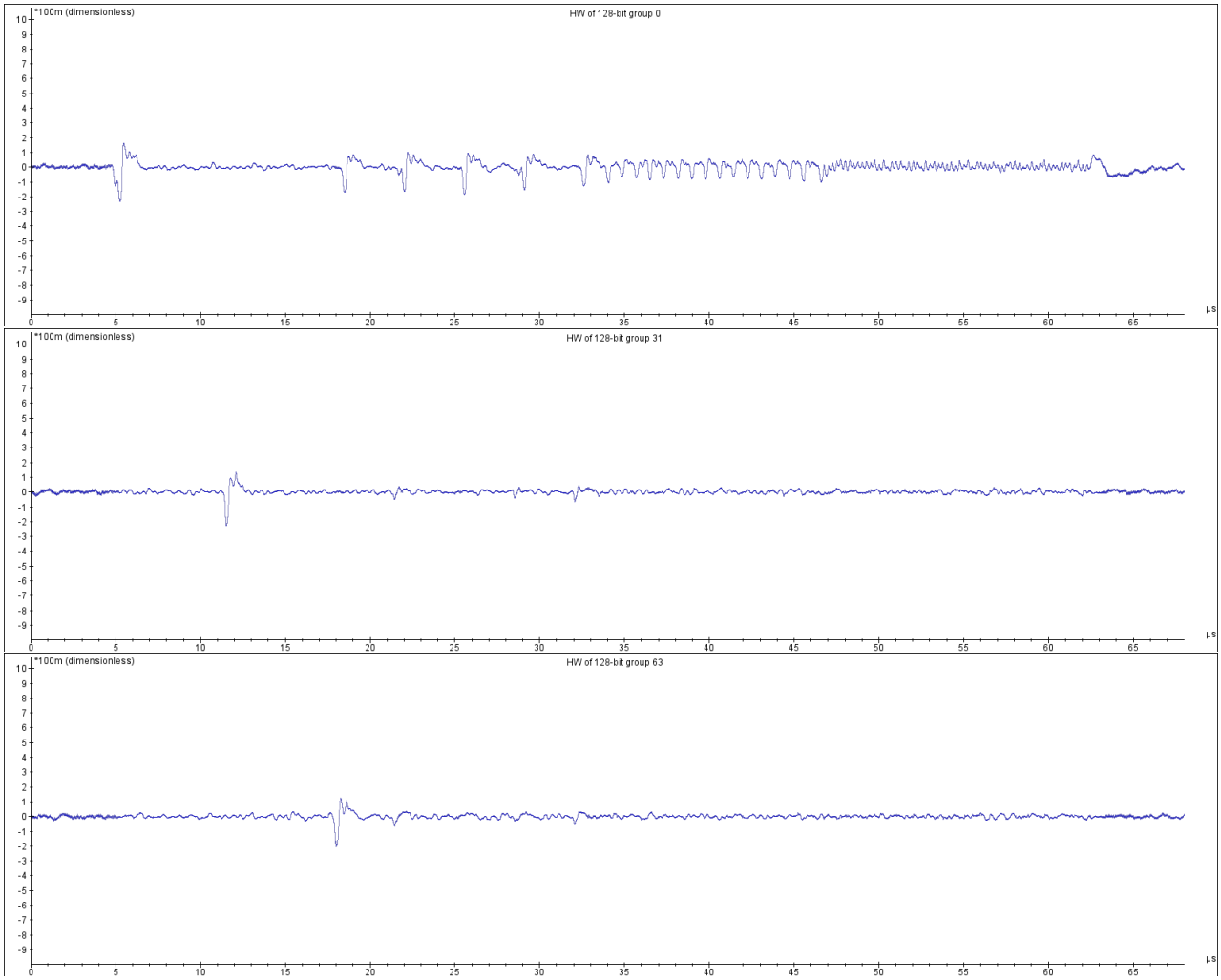

FIGURE 5.14: Correlation of Hamming weight of Stage 1 output alone, with $0, 31, 63$-rd groupings.

We also realize that the Hamming distance models are not ideal in terms of correlating with our traces. No noticeable correlation appears with the HD of either the first layer, or the whole merged two layers. This suggests that for this software implementation, HD model is not as realistic as HW model.

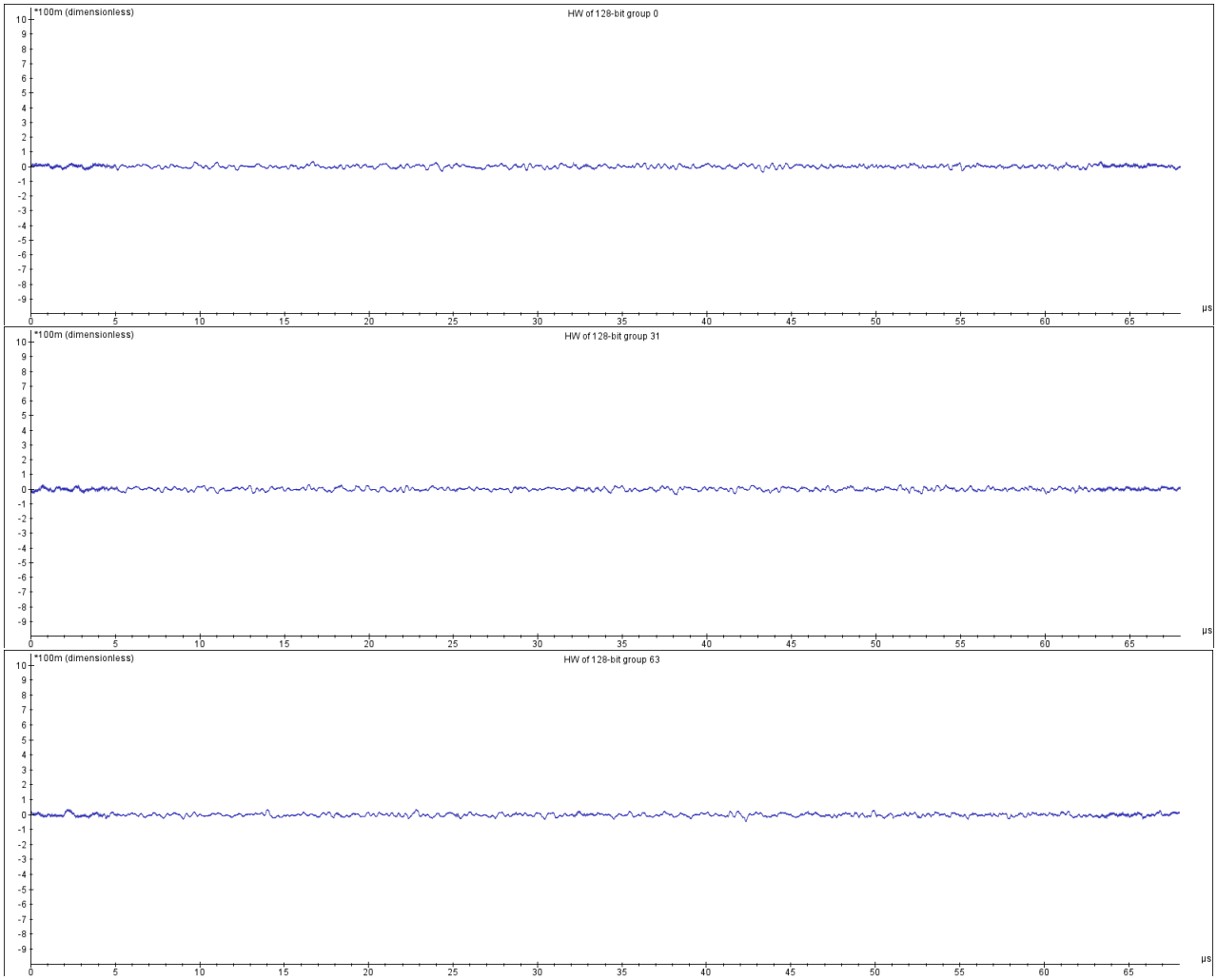

FIGURE 5.15: Correlation of Hamming distance of input and Stage 1 output, with $0, 31, 63$-rd groupings.

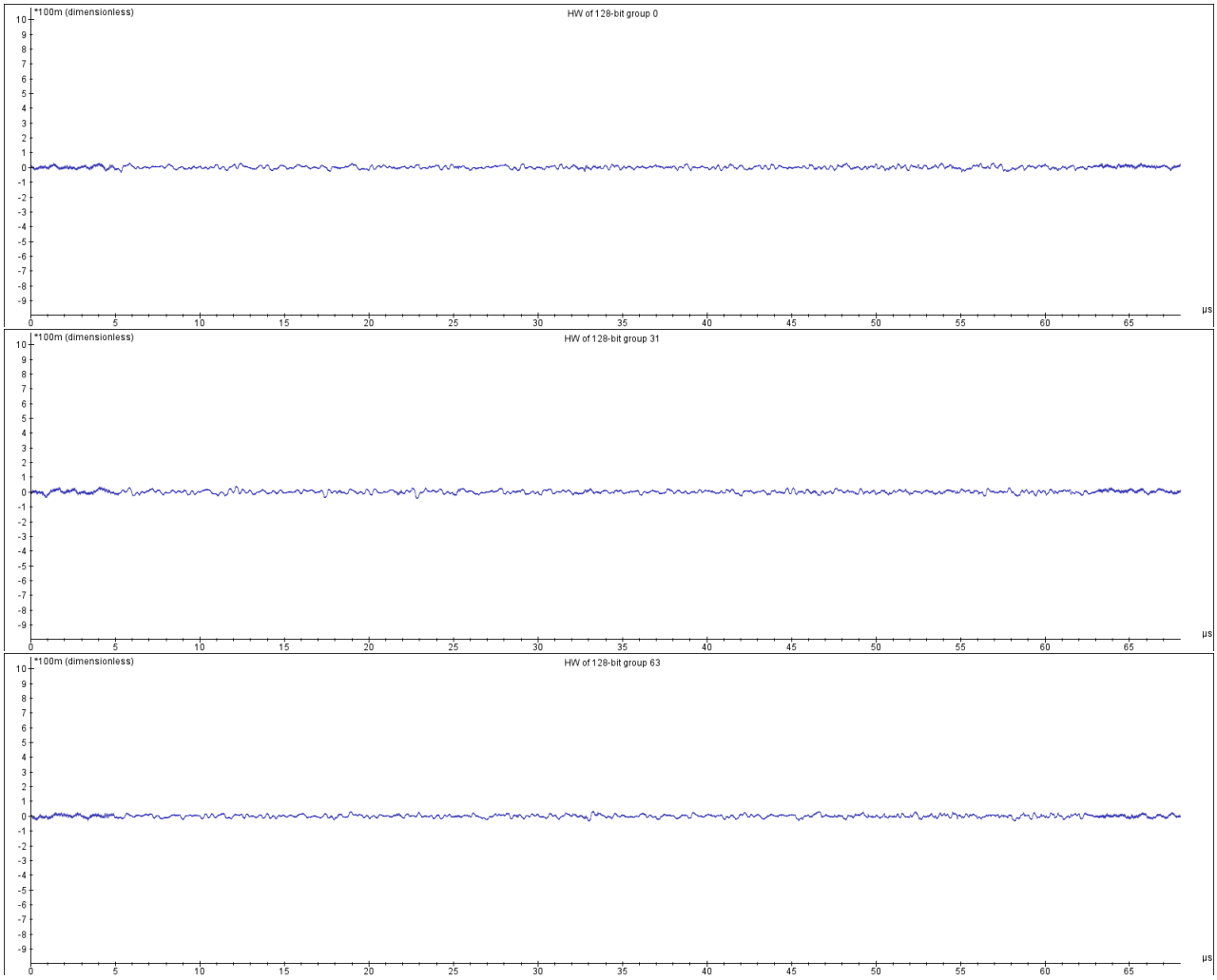

FIGURE 5.16: Correlation of Hamming distance of input and Stage 2 output, with $0, 31, 63$-rd groupings.

Recall that we can observe leakage by biasing Hamming distance per-stage in the experiment summarized in Table 4.6. This does not contradict with our observation that correlation of HD model is very low because the leakage in Table 4.6 does not come from biasing the Hamming distance, but ID or HW leakages by fixing the butterflies.
Note that although the Hamming distance models do not work very well with Piñata or the `GKS20` implementation, they may work well with other hardware implementations.

Chapter 6

Template Attack

6.1 Secret Keys in Dilithium

Secret key of Dilithium contains two secret parts $\mathbf{s}_1$ and $\mathbf{s}_2$, both generated by the $\mathsf{ExpandS}(\rho')$ algorithm, where ρ' is a 512-bit seed. In particular $\mathbf{s}_1$ contains 5 polynomials, with each polynomial containing 256 coefficients, and each coefficient in range $[-4, 4]$. The total entropy for a single polynomial in $\mathbf{s}_1$ is then $256 \cdot \log_2(9) \approx 811.50$ bits. Knowing the full $\mathbf{s}_1$ is sufficient for the attacker to trivially recover the other secret part $\mathbf{s}_2$. Therefore we focus on attacking the $\mathbf{s}_1$ parameter in Dilithium.
From an attacker's point of view, usually the attacker does not have direct access to the key generation Gen part of Dilithium. Instead, the IoT device pre-stores the secrets $\mathbf{s}_1, \mathbf{s}_2$ and only implements the signing Sign part of Dilithium. Thus usually the attacker only has direct access to the signing part of Dilithium on a target, where NTT is performed once on $\mathbf{s}_1$ each time a new message is signed. We aim to build templates on $\mathbf{s}_1$ and compute the reduction in entropy for a single polynomial in $\mathbf{s}_1$.

6.2 Template Building

There are two different approaches to building the templates, each with its own trace-off.

1. Build templates on each input butterfly with ID model. In this case, we need to build templates on 128 input butterflies. Each butterfly contains 2 coefficients, and each coefficient contains 9 possibilities. Therefore for each butterfly we need to build $9^2 = 81$ templates. If we collect 2000 traces for each template, in total we need to collect $81 \times 2000 = 162000$ traces. Given an acquisition rate of 4 traces per second, it takes about 12 hours to collect all traces. The storage required for all the traces is 20.5 GB in a `.trs` file.

2. Build templates on each grouping of 2 input butterflies with ID model. In this case, we need to build templates on 64 input butterfly groupings. Each butterfly grouping contains 4 coefficients, and each coefficient contains 9 possibilities. Therefore for each butterfly grouping we need to build $9^4 = 6561$ templates. If we collect 2000 traces for each template, in total we need to collect $6561 \times 2000 = 13122000$ traces. Given an acquisition rate of 4 traces per second, it takes about 38 days to collect all traces. The storage required for all traces is estimated at 1.66 TB.

The acquisition time for the first approach is feasible, but the accuracy for each template could be lower than the second approach because the first approach does not combine the

leakages from the second layer. The second approach could provide better accuracy, but the acquisition time and storage overhead made it nearly impossible to execute. Therefore we follow the first approach and try to benchmark its performance.
For each possibility of the 0-th butterfly, we collect 2000 traces and their corresponding NTT inputs for template building. Specifically, we collect 2000 traces with the 0-th and 128-th coefficients fixed at $(\alpha, \beta) = (-4, -4)$ and all the rest coefficients uniformly random from $[-4, 4]$. We then collect 2000 similar traces with the 0-th and 128-th coefficients fixed at $(\alpha, \beta) = (-4, -3)$, and so on. Note that it is in fact unnecessary to force an even number of traces per each possibility of the 0-th butterfly. Alternatively we can just collect 162000 traces, all with uniformly random coefficients from $[-4, 4]$. All input coefficients should be stored together with the traces.
After collecting all traces, we start to build templates on different butterflies. Recall that there are 128 butterflies. For each butterfly, we compute the Points of Interest (POI) using Riscure Inspector SCA. Ideally we want to pick POI from a more spread and diverse distribution of time points, because leakages may propagate to a deeper stage, and POI from these deeper stages can carry more information for our templates. We realize that for some butterflies (such as the 0-th butterfly), SOSD gives a more spread selection of POI while for other butterflies (such as the 1-st butterfly), SOST gives a more diverse selection of POI. In the end we decide to combine both methods: we compute the first 200 POI with SOSD as well as the first 200 POI with SOST, and then take the union set of the two as our final POI. The computation for POI is done with Riscure Inspector SCA software.
With one POI set for each butterfly, we build templates based on full multivariate Gaussian distributions. By saying "full" we mean to use both mean and covariance in our template building. Using only the mean will significantly decrease the quality of our templates.
For performance evaluation, we in addition collect 10000 random evaluation traces with coefficients in $[-4, 4]$.

6.2.1 Experiment Result

We focus on templates built on the $0-7$-th, $64-67$-th, and $124-127$-th butterflies. In other words, these butterflies are from the $0-3$-rd, $32-33$-rd, and $62-63$-rd groupings. For each butterfly and each evaluation trace, we match the evaluation trace with all templates, and compute the probability of all templates. We then sort the probabilities in descending order, and figure out the ranking of the ground truth in this sorted list. For example, if the ranking of ground truth is 0, it means our templates give the correct prediction. If the ranking of ground truth is 1, it means the ground truth lies in the first two predictions of our templates. Recall that there are 81 templates per butterfly, so a ranking of 1 reduces the entropy of a butterfly from $\log_2(81) \approx 6.3$ bits to 1 bit. In the end, for each butterfly, we obtain 10000 rankings for every single evaluation trace, and a histogram plot of rankings. For example, ranking histograms for the 0-th and 1-st butterfly are shown in Figure 6.1.

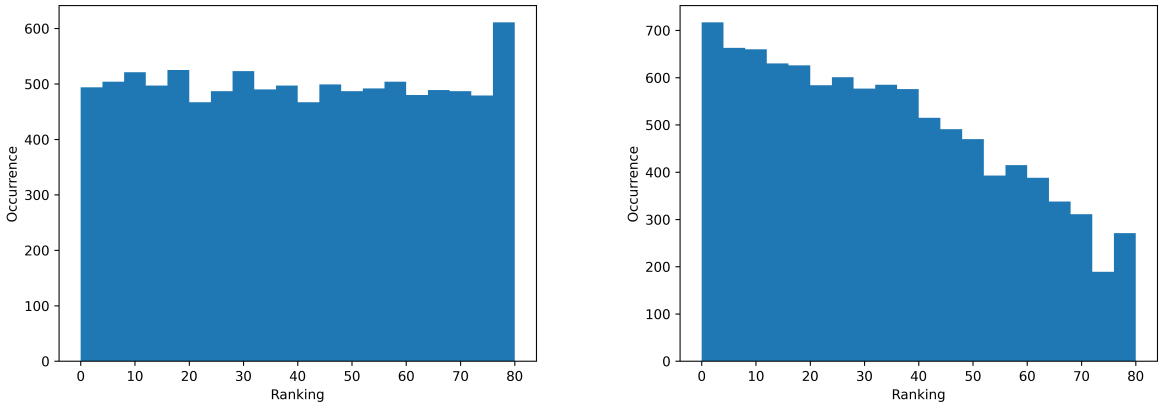

FIGURE 6.1: Example histograms of rankings for template attack on 0-th (left) and 1-st (right) butterflies.

Given a fixed butterfly, we can further break down the 10000 evaluation traces by ground truths, and plot the ranking distribution for each ground truth. For example, ranking histograms for the 64-th and 65-th butterflies on ground truths $(0, 4)$ and $(4, 0)$.

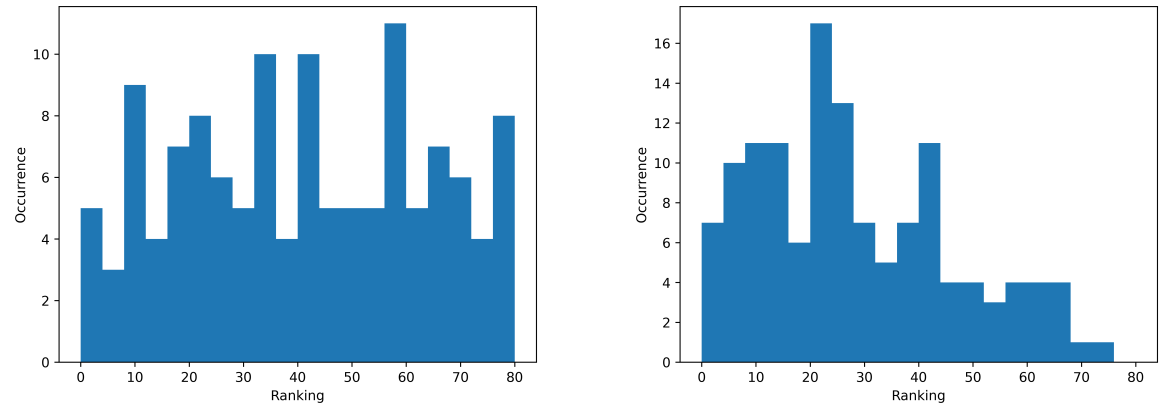

FIGURE 6.2: Example histograms of rankings for template attack on 64-th (left) and 65-th (right) butterflies with ground truth $(0, 4)$.

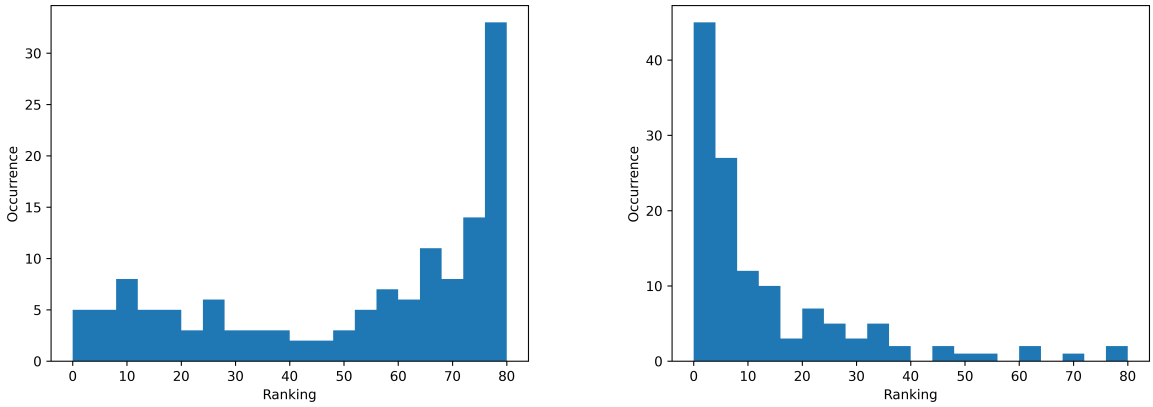

FIGURE 6.3: Example histograms of rankings for template attack on 64-th (left) and 65-th (right) butterflies with ground truth $(4, 0)$.

We can then compute the average best, median, and worst rankings for each butterfly, and estimate the reduction of entropy for $\mathbf{s}_1$.

Butterfly Index	Average Best Ranking	Average Median Ranking	Average Worst Ranking
0	26.88	39.89	54.12
1	2.51	33.44	66.48
2	0.81	40.04	79.25
3	0.51	31.63	76.94
4	0.53	39.33	79.22
5	0.30	31.86	77.23
6	0.40	39.77	79.44
7	0.19	32.78	78.07
64	0.22	39.70	79.41
65	0.19	30.00	77.80
66	0.32	39.67	79.32
67	0.23	30.17	77.64
124	0.78	39.09	78.95
125	0.52	32.38	76.75
126	2.95	39.81	75.21
127	2.72	34.48	73.51
averaged among butterflies	2.50	35.88	75.58

TABLE 6.1: Best, median, and worst rankings for template attack, averaged among all 81 possibilities.

In general we observe a better ranking with odd-indexed butterflies compared to even-indexed butterflies, probably because more leakages are propagated to a deeper stage with odd-indexed butterflies. To compute the reduction in entropy, we first increase the rankings by 1 and then round up. Using the total average among butterflies, we have

- Best case scenario, entropy of a single polynomial in $\mathbf{s}_1$ is reduced to $128 \cdot \log_2(\lceil 2.50 + 1 \rceil) = 256$ bits, a $1 - \frac{256}{811.5} \approx 68.45\%$ reduction.

- Median case scenario, entropy of a single polynomial in $\mathbf{s}_1$ is reduced to $128 \cdot \log_2(\lceil 35.88 + 1 \rceil) \approx 666.81$ bits, a $1 - \frac{666.81}{811.5} \approx 17.83\%$ reduction.
- Worst case scenario, entropy of a single polynomial in $\mathbf{s}_1$ is reduced to $128 \cdot \log_2(\lceil 75.58 + 1 \rceil) \approx 802.15$ bits, a $1 - \frac{802.15}{811.5} \approx 1.15\%$ reduction.

In conclusion, on average the leakage could at least reduce the entropy of $\mathbf{s}_1$ by 1.15%, and at most by 68.45% with a single-trace template attack. Such reduction is far from a realistic retrieval of secret key $\mathbf{s}_1$, but more serves as a benchmark for the advantage gained by an attacker, even with a cheap and crude attack.

Chapter 7

Conclusion

We aim to fill the research gap of practically assessing the leakage of Dilithium NTT. We first propose algorithms for generating per-stage biased test vectors for plain NTT in Section 3.3 to 3.5, for all three power models: HW, HD, and ID. Our algorithm mainly exploits the bijectivity of the NTT stages and evolves the desired stage back to the NTT input.
We then analyze the differences between real-world NTT and plain NTT, and proposed modified algorithms for generating per-stage biased test vectors for `GKS20` NTT in Section 4.4 to 4.6, again for all three power models: HW, HD, and ID. To figure out the effectiveness of our test vectors, we then conducted experiment and evaluated the leakage of Piñata when performing NTT. We have observed leakage from Piñata for all three power models, with highest t-value 3174.96. Due to time limitation, we only investigate one particular implementation of Dilithium NTT, namely `GKS20`. Future research regarding leakage assessment on NTT could fall in the following categories, ranked from more specific to more general.

- Design test vectors for other implementations of Dilithium NTT on ARM Cortex-M4. For example, we may analyze the level merging in [1] that is different from `GKS20`, and design optimized test vectors for this implementation.
- Investigate the implementations of Dilithium NTT on other architectures or platforms. For example, we may design test vectors for software implementation on Cortex A-72 and Apple M1 presented in [4]. In addition, leakages from hardware implementations may drastically differ from software implementations. Therefore, hardware implementation like [31] awaits further investigation as well.
- Evaluate the leakages from a SCA-protected implementation of Dilithium NTT. As shown in Table 3.1, all implementations included in our research have no side-channel mitigation at all. Thus we are curious about precisely how much leakage can be reduced in protected NTT designs such as [39].
- Investigate NTT components in other ciphers such as Kyber, Falcon, NewHope, *etc.* NTT components in other ciphers are likely to have different sizes with different set of parameters. We may need to develop new test vector generation strategies for these ciphers.

In addition, potential minor improvement to our test vector generation algorithms exits. For example, Algorithm 2 uses an ad hoc uniform sampling for generating random integer partition with constraint. A more elegant way is to implement the Fristedt's method [14].

Inspired by the per-stage biased test vectors, we also propose algorithms for generating per-butterfly biased test vectors for GKS20 NTT butterflies in Section 5.2 for ID model. We conduct experiment and evaluate the per-butterfly leakage of Piñata. We have observed leakage from Piñata for most cases with ID model, with highest t-value 40.02. We also use correlation analysis in Section 5.3 to profile leakages of a few butterflies, and conclude that most of the leakages come from memory operation, especially memory writing.

Furthermore, we conduct a template attack against the NTT operation on the Dilithium secret key $\mathbf{s}_1$. Results show at least a 1.15% reduction of entropy, and at most a 68.45% reduction of entropy. The large gap between the minimal and maximal reduction of entropy reveals that our attack robustness can be improved. In particular, if time permits, we may collect more traces and build templates on butterfly grouping (4 coefficients) instead of a single butterfly (2 coefficients).

In conclusion, our work propose algorithms for generating test vectors for Dilithium NTT, and effectively detect per-tage and per-butterfly leakages from the Piñata board. We further identify the main source of leakage with correlation analysis, and conduct a template attack to evaluate the reduction of entropy caused by the leakages.

Bibliography

[1] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-m4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security*, pages 853–871, Cham, 2022. Springer International Publishing.

[2] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 99–108, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237838.

[3] George Becker, Jim Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Timofei Kouzminov, Andrew Leiserson, Mark Marson, Pankaj Rohatgi, et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference*, volume 1001, page 13. sn, 2013.

[4] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022:221–244, 2021. URL: https://api.semanticscholar.org/CorpusID:236941216.

[5] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. High-performance hardware implementation of crystals-dilithium. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10, 2021. doi:10.1109/ICFPT52863.2021.9609917.

[6] 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.

[7] Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.

[8] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 278–291, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[9] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[10] 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.

[11] Frank Custers. Soft analytical side-channel attacks on the number theoretic transform for post-quantum cryptography. *Master's Thesis*, 2022. URL: https://research.tue.nl/en/studentTheses/soft-analytical-side-channel-attacks-on-the-number-theoretic-tran.

[12] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[13] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5):1–75, 2018.

[14] Bert Fristedt. The structure of random partitions of large integers. *Transactions of the American Mathematical Society*, 337(2):703–735, 1993. URL: http://www.jstor.org/stable/2154239.

[15] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 15–29, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[16] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, December 2020. Artifact available at https://artifacts.iacr.org/tches/2021/a1. doi:10.46586/tches.v2021.i1.1-24.

[17] Qian Guo, Vincent Grosso, François-Xavier Standaert, and Olivier Bronchain. Modeling soft analytical side-channel attacks from a coding theory viewpoint. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[18] 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*, 2021(4):88–113, Aug. 2021. URL: https://tches.iacr.org/index.php/TCHES/article/view/9061, doi:10.46586/tches.v2021.i4.88-113.

[19] Kelsey A. Jackson, Carl A. Miller, and Daochen Wang. Evaluating the security of crystals-dilithium in the quantum random oracle model. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 418–446, Cham, 2024. Springer Nature Switzerland.

[20] Aruna Jayasena, Emma Andrews, and Prabhat Mishra. Tvla*: Test vector leakage assessment on hardware implementations of asymmetric cryptography algorithms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[21] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Second NIST PQC Standardization Conference, 2019. URL: https://eprint.iacr.org/2019/844.

[22] Mustafa Khairallah and Shivam Bhasin. Hardware implementation of masked skinny sbox with application to aead. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 50–69, Cham, 2022. Springer Nature Switzerland.

[23] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography*, pages 309–326, Cham, 2022. Springer International Publishing.

[24] Chris Lomont. The hidden subgroup problem-review and open problems. *arXiv preprint quant-ph/0411037*, 2004.

[25] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[26] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, and Aydin Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, 71(11):2829–2843, 2020.

[27] Ahmet Can Mert, Ferhat Yaman, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, and Aydin Aysu. A survey of software implementations for the number theoretic transform. In Cristina Silvano, Christian Pilato, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 328–344, Cham, 2023. Springer Nature Switzerland.

[28] D. Micciancio and O. Regev. Worst-case to average-case reductions based on gaussian measures. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 372–381, 2004. doi:10.1109/FOCS.2004.72.

[29] Daniele Micciancio and Chris Peikert. Hardness of sis and lwe with small parameters. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 21–39, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] Catinca Mujdei, Lennert Wouters, Angshuman Karmakar, Arthur Beckers, Jose Maria Bermudo Mera, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. *ACM Transactions on Embedded Computing Systems*, 2022.

[31] Trong-Hung Nguyen, Binh Kieu-Do-Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang. High-speed ntt accelerator for crystal-kyber and crystal-dilithium. *IEEE Access*, 12:34918–34930, 2024. doi:10.1109/ACCESS.2024.3371581.

[32] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[33] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical cca2-secure and masked ring-lwe implementation. *Cryptology ePrint Archive*, 2016.

[34] Chris Peikert et al. A decade of lattice cryptography. *Foundations and trends® in theoretical computer science*, 10(4):283–424, 2016.

[35] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *Progress in Cryptology–LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2–4, 2019, Proceedings 6*, pages 130–149. Springer, 2019.

[36] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 513–533. Springer, 2017.

[37] Zehua Qiao, Yuejun Liu, Yongbin Zhou, Mingyao Shao, and Shuo Sun. When NTT meets SIS: Efficient side-channel attacks on dilithium and kyber. Cryptology ePrint Archive, Paper 2023/1866, 2023. https://eprint.iacr.org/2023/1866. URL: https://eprint.iacr.org/2023/1866.

[38] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020.

[39] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable sca countermeasures against single trace attacks for the ntt. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 123–146, Cham, 2020. Springer International Publishing.

[40] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1060590.1060603.

[41] Oscar Reparaz, Sujoy Sinha Roy, Ruan De Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking ring-lwe. *Journal of Cryptographic Engineering*, 6(2):139–153, 2016.

[42] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 8(1):71–84, 2018. doi:10.1007/s13389-017-0149-6.

[43] Ardianto Satriawan, Infall Syafalni, Rella Mareta, Isa Anshori, Wervyan Shalannanda, and Aleams Barra. Conceptual review on number theoretic transform and comprehensive review on its implementations. *IEEE Access*, 2023.

[44] Tobias Schneider and Amir Moradi. Leakage assessment methodology. *Journal of Cryptographic Engineering*, 6(2):85–99, 2016. doi:10.1007/s13389-016-0120-y.

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

[46] Hauke Steffen, Georg Land, Lucie Kogelheide, and Tim Güneysu. Breaking and protecting the crystal: Side-channel analysis of dilithium in hardware. In Thomas Johansson and Daniel Smith-Tone, editors, *Post-Quantum Cryptography*, pages 688–711, Cham, 2023. Springer Nature Switzerland.

[47] Elias M Stein and Rami Shakarchi. *Fourier analysis: an introduction*, volume 1. Princeton University Press, 2011.

[48] Michael Tunstall and Gilbert Goodwill. Applying tvla to public key cryptographic algorithms. *Cryptology ePrint Archive*, 2016.

[49] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward more efficient dpa-resistant aes hardware architecture based on threshold implementation. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 50–64, Cham, 2017. Springer International Publishing.

[50] Felipe Valencia, Ayesha Khalid, Elizabeth O'Sullivan, and Francesco Regazzoni. The design space of the number theoretic transform: A survey. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 273–277. IEEE, 2017.

[51] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 282–296, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[52] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 740–757, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[53] Zixuan Wu, Rongmao Chen, Yi Wang, Qiong Wang, and Wei Peng. An efficient hardware implementation of crystal-dilithium on fpga. In Tianqing Zhu and Yannan Li, editors, *Information Security and Privacy*, pages 64–83, Singapore, 2024. Springer Nature Singapore.

[54] Tianrun Yu, Chi Cheng, Zilong Yang, Yingchen Wang, Yanbin Pan, and Jian Weng. Hints from hertz: Dynamic frequency scaling side-channel analysis of number theoretic transform in lattice-based kems. *Cryptology ePrint Archive*, 2024.

Appendix A

Test Vectors for GKS20 NTT

A.1 Trace Example

Our first example NTT input contains all zero, generated with the following Python code

```
x_A = [0 for _ in range(256)]
```

and our second example input is generated with Python's `random` module, with seed `0`

```
import random
random.seed(0)
x_B = [random.randint(-q//2,q//2) for _ in range(256)]
```

In particular, the values are

```
x_B = [2893314, -958538, 2167973, 3277698, -662199, -3850602, -2018258,
↪  3908697, 98674, -113953, -793376, 3512498, 2384770, 2771101, -1645838,
↪  3928285, -192224, -1186674, 703554, 3289582, 3427594, -2357779, 43411,
↪  -3021897, -1825982, -3017937, 2149936, -3394711, 997046, 2515665,
↪  -2088810, 4091521, 3441295, 277258, 4053993, 1725204, 2606385, 858911,
↪  3377707, -2957408, -1588503, -3361723, 1932063, -3571613, 3349793,
↪  2944082, 1547464, -1420314, -229589, 505848, -3345467, -1222393,
↪  -548112, -1537759, 934282, 1182051, 3468682, -2474926, 3918272, 444695,
↪  -188761, -476628, 3068536, 183222, -2005042, -3667770, 2562853,
↪  3514432, 412614, 3493709, -4072419, -3407788, 1847111, 2856986,
↪  -844635, 1767828, 2729092, 2395248, 1414666, 1054900, -4180623, 942755,
↪  -49778, 2756089, 3087770, -1395669, -2144137, 1935811, -1462198,
↪  1712372, 3111832, -3661859, -2587423, 3502303, 570416, -2330422,
↪  -2188554, 2548737, 3926106, -2994880, 2547348, 364720, -432368,
↪  -3425035, -3515381, 4141537, -1505397, 3150555, 70705, 4181383,
↪  3634968, -85774, -3275366, -1661491, 434155, -1748364, 1736857,
↪  -3143221, 402058, -1398896, 2642034, 3546182, 342018, -2485629,
↪  3895054, 2515871, 869678, 400289, 739080, -1777165, -457371, -3421544,
↪  811807, 2503355, -961425, -1530628, 638701, -2159267, -1754749,
↪  -2647807, -2601480, 2700757, -2623809, -3913617, 950104, 4041529,
↪  1318249, -2008700, -192778, -3610680, -3436708, 1503340, 2165033,
↪  -3097808, 3164673, -2935690, 3555672, -3866061, 2876859, -3517038,
↪  3344213, 1676135, 3553167, 2766974, 344671, 1543391, -907780, 2838313,
↪  1726136, 210236, -1878023, 186883, 2618226, -2214678, 2935795,
↪  -2384970, 3317922, 1509635, 757401, 2732601, 3776996, -671832, 672625,
↪  -1881574, -410612, -57517, 1348330, 1188539, 4167007, 1683708, 3501651,
↪  4051266, 2464684, -1192393, -3499215, -1469576, 950188, -3222586,
↪  -109620, 734530, 1095850, -1377757, 2898815, -2593196, -2151479,
↪  -4054238, 1945967, -1916578, -3207616, 1726554, -2340852, -1069142,
↪  2475236, -2760105, -1400694, -615562, 2654166, -3668501, -3346252,
↪  2379065, -2962466, 2984757, 1662276, -2355004, -3810759, 2664292,
↪  623728, 1129899, 3443558, 3665887, 290968, 861163, 1518986, -3569479,
↪  -3966261, -3146300, 1136556, -2608784, 896126, 2773058, 641234,
↪  -3186131, -908507, -3422385, -1085251, 2804585, 4034960, -3216744,
↪  -3884931, 889128, -4008709, -2557862, 3871505, 3952544, -2638552,
↪  1833762]
```

A.2 HW model

A.2.1 Fixed-vs-Random

Fixed test vector is obtained by running **TVGenGKS20HW**$(2, 50, 100, 0)$. The coefficients are

```
x_A = [128, 0, 266242, 0, 0, 4190211, 524288, 0, 0, 0, 0, 0, 0, 133376,
↪  512, 4255745, 0, 0, 2, 33024, 0, 263168, 768, 0, 0, 0, 0, 8196, 2,
↪  147456, 73728, 131072, 16388, 132, 128, 2064, 532480, 0, 0, 0, 131072,
↪  0, 0, 0, 16, 0, 0, 0, 65568, 0, 262146, 0, 0, 32, 0, 0, 0, 0, 131074,
↪  4224, 4, 0, 0, 0, 4590522, 0, 3324028, 0, 0, 3166843, 5502781, 0, 0, 0,
↪  0, 0, 0, 455365, 494874, 6006869, 0, 0, 7523026, 2715879, 0, 6616727,
↪  7638106, 0, 0, 0, 0, 1714782, 7531214, 1644275, 4218520, 981489,
↪  5708009, 2075113, 4066490, 3566289, 2398823, 0, 0, 0, 7661008, 0, 0, 0,
↪  6793624, 0, 0, 0, 7854539, 0, 7266642, 0, 0, 3173586, 0, 0, 0, 0,
↪  8242435, 107498, 1714782, 0, 0, 0, 4702026, 0, 6111657, 0, 0, 4740349,
↪  1300230, 0, 0, 0, 0, 0, 0, 5347564, 6333147, 146464, 0, 0, 7144446,
↪  6342660, 0, 4744655, 5309512, 0, 0, 0, 0, 5143127, 1235971, 6045962,
↪  6888132, 3865151, 4377779, 2230084, 3678391, 1698688, 534882, 0, 0, 0,
↪  4515266, 0, 0, 0, 1507351, 0, 0, 0, 757069, 0, 585856, 0, 0, 3014702,
↪  0, 0, 0, 0, 3279295, 4061065, 5908475, 0, 0, 0, 4066490, 0, 3831810, 0,
↪  0, 3156608, 4454461, 0, 0, 0, 0, 0, 0, 1245444, 6779163, 6139956, 0, 0,
↪  7531214, 1602295, 0, 2952726, 2401881, 0, 0, 0, 0, 1698406, 7523026,
↪  1939115, 4071100, 719409, 5757145, 2615521, 4590522, 3496690, 3463523,
↪  0, 0, 0, 7398928, 0, 0, 0, 6859128, 0, 0, 0, 7592491, 0, 7798990, 0, 0,
↪  3042578, 0, 0, 0, 0, 132286, 639720, 1698406, 0, 0, 0]
```

A.2.2 Fixed-vs-Fixed

Our first fixed test vector `x_A` is the same as the test vector `x_A` in the last section, generated with **TVGenGKS20HW**$(2, 50, 100, 0)$.
Our second fixed test vector `x_B` is the same as `x_B` in A.1.

A.3 HD model

A.3.1 Fixed-vs-Random

Fixed test vector is obtained by running **TVGenGKS20HDZero**$(2, 250, 500)$. The coefficients are

```
x_A = [9248, 1319468, 165896, 90368, 4190532, 129, 393769, 131493, 65792,
↪  73728, 163850, 16, 1067009, 590336, 32832, 1444352, 5796001, 25100,
↪  32776, 4845833, 66112, 264224, 81920, 4239369, 8264, 81921, 72704,
↪  917826, 69888, 33100, 1024, 827394, 133136, 4355593, 4723009, 1049664,
↪  552964, 1049648, 2052, 9477, 263170, 1118218, 4224, 524617, 1048577,
↪  1024, 146, 1062, 32768, 32, 1062976, 280614, 262658, 262406, 524800,
↪  1044, 400200, 1082881, 524435, 67653, 4453025, 266256, 524801, 528384,
↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6343894, 3538851,
↪  6397758, 7395193, 3442493, 106168, 2762029, 2538960, 3281268, 1492285,
↪  2254039, 1507351, 4833324, 4304903, 7670038, 323801, 1556071, 6133904,
↪  375141, 3206235, 6655916, 3856154, 726937, 7858812, 1827523, 7360552,
↪  2028244, 371948, 6763745, 651335, 0, 640424, 2816578, 6754585, 5243863,
↪  7714821, 3619964, 5543781, 191337, 6409793, 2399791, 5398005, 3678391,
↪  7035904, 6172683, 4094540, 4430646, 5873271, 0, 5365715, 1206132,
↪  706813, 6494331, 7517966, 747040, 5059131, 5391758, 5349898, 1677778,
↪  2620969, 4966259, 5840277, 2825183, 917556, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 0, 0, 0, 0]
```

A.3.2 Fixed-vs-Fixed

Our first fixed test vector `x_A` is the same as the test vector `x_A` in the last section, generated with **TVGenGKS20HDZero**$(2, 250, 500)$.
Our second fixed test vector `x_B` is the same as `x_B` in A.1.

A.4 ID model

A.4.1 Fixed-vs-Random

Our fixed test vector is the same as `x_B` in A.1.

A.4.2 Fixed-vs-Fixed

Our two fixed test vectors are generated with the following Python code.

```
import random
random.seed(1)
x_A = [random.randint(-q//2,q//2) for _ in range(256)]
x_B = [random.randint(-q//2,q//2) for _ in range(256)]
```

Appendix B

Test Vectors for GKS20 NTT Butterfly

B.1 Structure of GKS20 NTT Butterfly

The following block of assembly code shows a specific grouping of 4 memory reading, followed by 4 butterfly operations, and then 4 memory writing.

```
ldr.w pol0, [ptr_p]
ldr pol1, [ptr_p, #256]  //64*4
ldr pol2, [ptr_p, #512]  //128*4
ldr pol3, [ptr_p, #768]  //192*4
ct_butterfly_montg pol0, pol2, zeta0, q, qinv, temp_h, temp_l  //stage1
ct_butterfly_montg pol1, pol3, zeta0, q, qinv, temp_h, temp_l  //stage1
ct_butterfly_montg pol0, pol1, zeta1, q, qinv, temp_h, temp_l  //stage2
ct_butterfly_montg pol2, pol3, zeta2, q, qinv, temp_h, temp_l  //stage2
str pol1, [ptr_p, #256]
str pol2, [ptr_p, #512]
str pol3, [ptr_p, #768]
str pol0, [ptr_p], #4
```

B.2 ID Model

B.2.1 Experiment 1

The fixed values in the semi-fixed test vectors `x_A` and `x_B` are generated with

```
import random
random.seed(0)
fixed = [random.randint(-q//2,q//2) for _ in range(4)]
```

and concretely

```
fixed = [2893314, -958538, 2167973, 3277698]
```

Formally, for `x_A` we set $F_A(31) = 2893314$ and $F_A(159) = -958538$. For `x_B` we set $F_B(31) = 2167973$ and $F_B(159) = 3277698$.

B.2.2 Experiment 2

The fixed values in the semi-fixed test vectors `x_A` and `x_B` are generated with

```
import random
random.seed(1)
fixed = [random.randint(-q//2,q//2) for _ in range(8)]
```

and concretely

```
fixed = [-3063081, 584619, 2918580, 2539678, 2216796, -3660831, -2050535,
↪  -3201036]
```

Formally, for `x_A` we set $F_A(31) = -3063081$, $F_A(95) = 584619$, $F_A(159) = 2918580$, and $F_A(223) = 2539678$. For `x_B` we set $F_B(31) = 2216796$, $F_B(95) = -3660831$, $F_B(159) = -2050535$, and $F_B(223) = -3201036$.

B.2.3 Experiment 3

The fixed value in the semi-fixed test vector `x_A` uses the first random number of `fixed` generated in Experiment 1. Concretely, we set $F_A(31) = 2893314$.

B.2.4 Experiment 4

For the first experiment, we choose `x_A` with $F_A(31) = F_A(159) = 0$, and `x_B` with $F_B(31) = 1$ and $F_B(159) = 0$.
For the second experiment, we choose `x_A` with $F_A(31) = F_A(159) = 0$, and `x_B` with $F_B(31) = 0$ and $F_B(159) = 1$.

B.2.5 Experiment 5

For both the first and second experiment, we choose `x_A` with $F_A(31) = 0$ and $F_A(159) = 2$, and `x_B` with $F_B(31) = 0$ and $F_B(159) = 4$. For the first experiment, we collect $N_A = N_B = 1000$ traces; while for the second experiment, we collect $N_A = N_B = 2000$ traces.