

# Hints from Hertz: Dynamic Frequency Scaling Side-Channel Analysis of Number Theoretic Transform in Lattice-Based KEMs

Tianrun Yu<sup>1</sup>, Chi Cheng<sup>1,✉</sup>, Zilong Yang<sup>1</sup>, Yingchen Wang<sup>2</sup>, Yanbin Pan<sup>3</sup>  
and Jian Weng<sup>4</sup>

<sup>1</sup> School of Computer Science, China University of Geosciences, Wuhan, 430074, China

<sup>2</sup> University of Texas at Austin

<sup>3</sup> Key Laboratory of Mathematics Mechanization, Academy of Mathematics and Systems Science,  
Chinese Academy of Sciences

<sup>4</sup> College of Cyber Security, Jinan University.

[chengchi@cug.edu.cn](mailto:chengchi@cug.edu.cn)

**Abstract.** Number Theoretic Transform (NTT) has been widely used in accelerating computations in lattice-based cryptography. However, attackers can potentially launch power analysis targeting NTT because it is usually the most time-consuming part of the implementation. This extended time frame provides a natural window of opportunity for attackers. In this paper, we investigate the first CPU frequency leakage (Hertzbleed-like) attacks against NTT in lattice-based KEMs. Our key observation is that different inputs to NTT incur different Hamming weights in its output and intermediate layers. By measuring the CPU frequency during the execution of NTT, we propose a simple yet effective attack idea to find the input to NTT that triggers NTT processing data with significantly low Hamming weight. We further apply our attack idea to real-world applications that are built upon NTT: CPA-secure Kyber without Compression and Decompression functions, and CCA-secure NTTRU. This leads us to extract information or frequency Hints about the secret key. Integrating these Hints into the LWE-estimator framework, we estimate a minimum of 35% security loss caused by the leakage. The frequency and timing measurements on the Reference and AVX2 implementations of NTT in both Kyber and NTTRU align well with our theoretical analysis, confirming the existence of frequency side-channel leakage in NTT. It is important to emphasize that our observation is not limited to a specific implementation but rather the algorithm on which NTT is based. Therefore, our results call for more attention to the analysis of power leakage against NTT in lattice-based cryptography.

**Keywords:** Lattice-based cryptography · Side-channel attacks · Hertzbleed attack · Post-Quantum cryptography · Kyber · Number Theoretic Transform

## 1 Introduction

In the face of threats from quantum computers, lattice-based cryptography has emerged as a promising candidate for traditional cryptographic schemes such as RSA and ECC. Lattice-based schemes lead the way in the selection of quantum-safe public key encryption or key encapsulation mechanism (KEM) since Kyber has been selected as the KEM standard by NIST. Some other schemes, like NTRU and NTRU Prime, although not on the standards list of NIST, have been put into practice in applications like CyaSSL and OpenSSH.

As a variant of the Fast Fourier Transform, the Number-Theoretic Transform (NTT) that operates on modular arithmetic has been widely used in accelerating computations

in lattice-based cryptography. In the case of lattice-based KEMs, two design approaches can be distinguished. One is an NTT-friendly design, such as Kyber [ABD<sup>+</sup>19], where the underlying ring and parameters are carefully selected to leverage NTT for efficient polynomial multiplication and computation. In the other case, the original design may not be friendly to NTT, and examples of such KEMs include NTRU and Saber. But as shown by Lyubashevsky and Seilers in their design called NTTRU [LS19], NTT can still be used by choosing different underlying rings. Another elegant example is given in [CHK<sup>+</sup>21], which makes NTT applicable to NTRU and Saber, achieving better performance. Furthermore, NTT is extensively utilized in lattice-based fully homomorphic encryption (FHE) schemes, enabling secure computation over encrypted data [DÖSS15, PS22].

The security analysis of lattice-based cryptography schemes in real-world implications is a crucial aspect of their practical deployment. Dynamic voltage and frequency scaling (DVFS) [Int20] is a commonly used mechanism to save power consumption in modern CPUs, which adjusts the clock frequency of a processor to enable a dynamic voltage supply. However, this mechanism can also make the processor vulnerable to certain side-channel attacks such as Hertzbleed [WPH<sup>+</sup>22]. By measuring the precise timings of fluctuations in the power supply, a Hertzbleed attack is able to deduce secret keys from a now-retired cryptosystem Supersingular Isogeny Key Encapsulation (SIKE) [JAC<sup>+</sup>20]. A similar Frequency Throttling Side-Channel is given in [LCCR22], which explores different attack methodologies and threat models through experiments against AES. Further improvements include a software-based DVFS side channel attack in [DG22] and a framework capable of exposing arbitrary timing leakage to coarse-grained timing sources [PBPV23].

Therefore, an intriguing question arises whether similar Hertzbleed attacks can be applied to lattice-based schemes, such as KEMs. Since Hertzbleed attacks typically exploit time or computation-consuming operations, it is reasonable to investigate the possibility of targeting the most time or computation-intensive component, i.e. NTT, in lattice-based schemes.

In this paper, we investigate the potential leakages of NTT in lattice-based KEMs through DVFS-based side channels. Our key observation is that, when processing polynomial inputs with only one non-zero coefficient or more than one non-zero coefficient, NTT processes data with low and random Hamming weights, respectively. This causes the CPU power consumption or frequency to be input-dependent under DVFS. We can distinguish the two cases via the frequency side channel by transforming power leakages into frequency leakage, and data-dependent Hamming weight is known to be a source of power leakage.

To summarize, we list the main contributions of this paper as follows:

- We propose a simple yet effective attack idea to analyze the NTT frequency leakage: Measuring the frequencies during the execution of NTT to find the parameters that can incur the low Hamming weight case, which corresponds to a higher frequency. The frequency and timing measurements align well with our theoretical analysis.
- We analyze the frequency leakage of NTT in a simplified version of CPA-secure Kyber without Compression and Decompression functions, as well as in the CCA-secure NTTRU. To extract information or Hints about the secret key, we elaborate on the parameters that trigger the low Hamming weight case through DVFS-based Side Channel. Integrating these Hints into the framework proposed in [DSDGR20], we estimate a minimum of 35% security loss caused by the leakage.
- We extensively conduct experiments on the Reference and AVX2 implementations of NTT in both Kyber and NTTRU. The results confirm that we can observe the Hamming weight difference stated in our analysis.

## 2 Preliminaries

### 2.1 Hertzbleed

#### 2.1.1 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling is a feature deployed on most modern processors to balance performance and power consumption. On Intel processors, this is achieved by dynamically adjusting the processor frequency between discrete P-States with 100 MHz granularity<sup>1</sup>. Following the Linux naming convention, higher P-states correspond to higher frequencies, while lower P-states correspond to lower frequencies [Wys].

During low CPU workload, DVFS schedules the CPU frequency to low P-States, which reduces power consumption. However, during high CPU workload, the thermal design point (TDP) limits the CPU’s performance, as it is the maximum amount of power that the cooling system can dissipate without causing the CPU to overheat [Int22]. To stay within the safe thermal limits, DVFS adjusts the CPU frequency to oscillate between multiple P-States.

#### 2.1.2 Hertzbleed: Dynamic Frequency Scaling Side-Channel

Hertzbleed attack exploits the data-dependent DVFS-induced frequency variation during high CPU workload [WPH<sup>+</sup>22]. Under the TDP limit, DVFS adjusts the CPU frequency based on power consumption, which is well known to be data-dependent. Transitively, CPU frequency adjustments are also data-dependent, as they reflect differences in power consumption. Such feature forms a Dynamic Frequency Scaling Side-Channel, which is also referred to as Frequency Throttling Side-Channel by adding workloads which is “throttled” by the thermal limit [LCCR22].

Hertzbleed enables side-channel attacks without privileges by decoupling time from the cycle. This is because the difference in CPU frequency variations translates directly to the program execution time. For example, consider a function `func` that is designed to be constant-cycle such that the number of CPU cycles of `func` is independent of secret. If `func` consumes more power when operating on `input1` compared to `input2`, during high CPU workload, DVFS would schedule the CPU to oscillate between lower P-States when running `func` with `input1`. As a result, `func` executes slower with `input1`. If `input1` and `input2` correspond to secret information, such a behavior can be exploited as a timing side channel. Therefore, `func` is no longer constant-time despite the fact that it is constant-cycle.

In the following, we use the DVFS-based Side-Channel to denote the Dynamic Frequency Scaling Side-Channel for short and use them interchangeably.

### 2.2 Threat model

In our proof-of-concept experiments, we assume a chosen ciphertext attacker. The attacker submits malformed ciphertexts to a decryption oracle. The attacker’s goal is to demonstrate that the malformed ciphertext can induce secret-dependent CPU frequency variations, thereby deducing information about the secret key based on CPU frequency leakages.

To this end, the attacker either collects the execution time of the target decryption oracle or samples the CPU frequency while the CPU executes the decryption. As CPU frequency is inversely proportional to program execution time, our experiments below interchangeably depict CPU frequency sampling and the execution time of the decryption oracle.

---

<sup>1</sup>DVFS also dynamically adjusts the CPU supply voltage, which is out of the scope of this paper.

## 2.3 Number Theoretic Transform

### 2.3.1 Notations.

Let  $q \in \mathbb{Z}$  be a prime, and  $\mathbb{Z}_q$  be the ring of residue classes modulo  $q$ . The polynomial ring over  $\mathbb{Z}_q$  is denoted by  $\mathbb{Z}_q[x]$ . We further denote by  $R_q$  the quotient ring  $\mathbb{Z}_q[x]/(\phi(x))$ , where  $\phi(x)$  is some polynomial with degree  $n$ . We represent every element in  $R_q$  as a polynomial  $\mathbf{f}(x) = f_0 + f_1x + \dots + f_{n-1}x^{n-1}$ , where each coefficient  $f_i \in \mathbb{Z}_q$  ( $0 \leq i \leq n-1$ ). Hereafter, we use both  $\mathbf{f} \in R_q$  and its vector form  $(\mathbf{f}_0, \dots, \mathbf{f}_{n-1})$  interchangeably to refer to a polynomial. The function  $\lceil x \rceil$  denotes the rounding function, which returns the nearest integer to  $x$ .

### 2.3.2 NTT from the Perspective of Chinese Remainder Theorem.

From an algebraic perspective, NTT can be expressed using the following Chinese Remainder Theorem (CRT).

**Theorem 1** (CRT in the ring form [Ber01]). *If  $R$  is a commutative ring with multiplicative identity and  $I_1, I_2, \dots, I_m$  are its pair-wisely co-prime ideals. Set  $I$  the intersection of  $I_1, I_2, \dots, I_m$ , then there exists a ring isomorphism  $\Phi$  as follows:*

$$\Phi : R/I \cong R/I_1 \times R/I_2 \times \dots \times R/I_m. \quad (1)$$

Applying Theorem 1 to the ring  $R_q$ , if the modulus polynomial  $\phi(x)$  in  $R_q$  can be factored as  $\phi(x) = \prod_i (\phi_i(x))$ , where each  $\phi_i(x)$  is irreducible, then the following isomorphism holds:

$$\mathbb{Z}_q[x]/(\phi(x)) \cong \prod_i \mathbb{Z}_q[x]/(\phi_i(x)). \quad (2)$$

NTT refers to the process of computing the isomorphic image of polynomials in  $R_q = \mathbb{Z}_q[x]/(\phi(x))$ , whereas the inverse NTT transform computes the preimage of elements in  $\prod_i \mathbb{Z}_q[x]/(\phi_i(x))$ . Denote by  $\hat{f} = \text{NTT}(f)$  the forward NTT transform for polynomial  $f$  and by  $f = \text{invNTT}(\hat{f})$  the inverse transform.

**Full NTT.** The most commonly used modulus polynomial in lattice-based cryptography is  $\phi(x) = x^n + 1$ , where  $n$  is a power of 2. In addition, the modulus  $q$  satisfies that  $2n|q-1$ , which implies that there exists a primitive  $2n$ -th root  $\zeta$  of unity in  $R_q$ . This leads to a full factorization of  $\phi(x)$ , which means each  $\phi_i(x)$  in (2) becomes a one-degree polynomial.

Now, we can define the forward NTT transform  $\hat{f} = \text{NTT}(f)$  as  $\hat{f}_i = \sum_{j=0}^{n-1} \zeta^{(2i+1)*j} f_j \bmod q$ . For the inverse transform  $f = \text{invNTT}(\hat{f})$ , we have  $f_i = n^{-1} \sum_{j=0}^{n-1} \zeta^{-i(2j+1)} \hat{f}_j \bmod q$ .

We refer to this type of NTT as *full NTT*. Full NTT has been widely used in various lattice-based cryptographic schemes such as NewHope [ADPS16], Falcon [FHK<sup>+</sup>18], and Dilithium [DKL<sup>+</sup>18].

**Incomplete NTT.** The strict constraint on parameter choices imposed by the requirement  $2n|q-1$  for full NTT can lead to increased computation and communication costs for lattice-based cryptosystems. To reduce these costs, an extension of NTT can be used when  $n = d * 2^k$  where  $d$  and  $k$  are some positive integers. In this case, the modulus polynomial  $\phi(x)$  cannot be completely factored into linear polynomials but into degree- $d$  irreducible polynomials. This type of NTT is referred to as *incomplete NTT*.

A typical example of incomplete NTT can be found in Kyber [ABD<sup>+</sup>19]. The security of Kyber relies on the Module Learning with Errors (MLWE) problem in dimension  $k$  over  $R_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$ . Since  $256|q-1$  but  $(2 * 256) \nmid q-1$ , Kyber can only guarantee

the existence of a primitive 256-th root of unity  $\zeta$  in  $R_q$ . To implement the incomplete NTT, Kyber utilizes the following isomorphism:

$$\mathbb{Z}_{3329}[x]/(x^{256} + 1) \cong \prod_{i=1}^{127} (\mathbb{Z}_{3329}[x]/(x^2 - \zeta^{2i+1})). \quad (3)$$

Recent research has shown that even schemes designed with NTT-unfriendly rings, such as Saber, NTRU [CHK<sup>+</sup>21], NTRU Prime [ACC<sup>+</sup>20], can also benefit from NTT. NTTTRU [LS19] is such an example, which is based on the NTRU cryptosystem but with NTT accelerations. NTTTRU adopts  $\mathbb{Z}_{7681}[x]/(x^{768} - x^{384} + 1)$  with  $n = 768 = 2^8 * 3$ . Thus, there exists a primitive 768-th root  $\eta$  of unity, and the following isomorphism

$$\mathbb{Z}_{7681}[x]/(x^{768} - x^{384} + 1) \cong \prod_{(i,768)=1} (\mathbb{Z}_{7681}[x]/(x^3 - \eta^i)), \quad (4)$$

is used to design the incomplete NTT.

### 2.3.3 Gentleman-Sande Butterfly.

Both NTT and invNTT can be efficiently implemented using a chain of butterflies. There are two kinds of butterflies used in NTT: The Cooley-Tukey (CT) butterfly and the Gentleman-Sande (GS) butterfly. Generally, we use the CT butterfly in forward NTT and the GS butterfly in invNTT.

The CT butterfly transforms  $f(x) \in \mathbb{Z}_q[x]/(x^n - c^2)$  into two polynomials in smaller ring for even  $n$ . To be specific,

$$\mathbb{Z}_q[x]/(x^n - c^2) \cong \mathbb{Z}_q[x]/(x^{n/2} - c) \times \mathbb{Z}_q[x]/(x^{n/2} + c) \quad (5)$$

$$CT: \quad f(x) = \sum_{i=0}^{n-1} f_i x^i \leftrightarrow \left( \sum_{i=0}^{n/2-1} (f_i + c f_{i+n/2}) x^i, \sum_{i=0}^{n/2-1} (f_i - c f_{i+n/2}) x^i \right) \quad (6)$$

For  $n = 2^k * d$ , by recursively calculating the CRT mapping for  $k$  layers, the last layer

---

**Algorithm 1** Inverse NTT based on the GS butterfly.

---

**Input:** Polynomial  $\hat{r} = (\hat{r}[0], \hat{r}[1], \dots, \hat{r}[n-1])$ ,  $\zeta$  as the corresponding primitive root

**Output:** Polynomial  $r = (r[0], r[1], \dots, r[n-1])$

```

1: for  $len = d \rightarrow d * 2^{k-1}$  do
2:   for  $i = 0 \rightarrow d * 2^k - 2 * 1en$  do
3:     for  $j = i \rightarrow i + len$  do                               /* Kyber: d=2; NTTTRU: d=3 */
4:       GS-Butterfly unit
5:          $j = j + 1$                                             $\triangleright$  GS-Butterfly unit
6:       end for                                               1:  $t = \hat{r}[j]$ 
7:          $i = j + len$                                          2:  $\hat{r}[j] = \hat{r}[j] + \hat{r}[j + len]$ 
8:       end for                                               3:  $\hat{r}[j + len] = \zeta^{-1} * (t - \hat{r}[j + len])$ 
9:      $len = len * 2$ 
10: end for
11: for  $i = 0 \rightarrow n - 1$  do
12:    $r[i] = \hat{r}[i]/2^k$ 
13: end for

```

---

usually corresponds to some ring  $\mathbb{Z}_q[X]/(x^d - \omega_i)$ . For example, there are 7 layers of butterflies in Kyber's NTT with  $k = 7$ ,  $d = 2$ , and the last layer works on  $\mathbb{Z}_{3329}[x]/(x^2 - \zeta)$ . In NTTTRU, a nice trick is that the modulus polynomial can be factored as  $(x^{768} - x^{384} + 1) =$

$(x^{384} + 684)(x^{384} - 685)$  since  $-684 = \eta^{\frac{768}{6}}$  and  $685 = \eta^{5 \cdot \frac{768}{6}}$ . Similarly, there are 7 layers of butterflies and the last layer represents polynomials of the form  $\mathbb{Z}_{7681}[x]/(x^3 - \eta)$ .

Similarly, the following GS butterfly is usually employed in invNTT:

$$GS : f(x) = \sum_{i=0}^{n-1} f_i x^i \leftrightarrow \left( \sum_{i=0}^{n/2-1} c^{-1}(f_i + f_{i+n/2}) x^i, \sum_{i=0}^{n/2-1} c^{-1}(f_i - f_{i+n/2}) x^i \right) \quad (7)$$

Since we mainly deal with invNTT in this paper, we summarize the GS butterfly used in Kyber and NTTTRU in Algorithm 1.

---

**Algorithm 2** Basemul between  $\hat{\mathbf{f}}_i$  and  $\hat{\mathbf{g}}_i$

---

**Input:**  $\hat{\mathbf{f}}_i = [\hat{\mathbf{f}}_i[0], \hat{\mathbf{f}}_i[1], \dots, \hat{\mathbf{f}}_i[d-1]]$  and  $\hat{\mathbf{g}}_i = [\hat{\mathbf{g}}_i[0], \hat{\mathbf{g}}_i[1], \dots, \hat{\mathbf{g}}_i[d-1]]$ ,  $\omega_i$

**Output:**  $\hat{\mathbf{r}}_i = [\hat{\mathbf{r}}_i[0], \hat{\mathbf{r}}_i[1], \dots, \hat{\mathbf{r}}_i[d-1]]$ , the  $i$ -th component of  $\hat{\mathbf{r}}$ .

1: **for**  $j = 0 \rightarrow d-1$  **do**

2:      $\hat{\mathbf{r}}_i[j] = \left( \sum_{\mathbf{u}=0}^j \hat{\mathbf{f}}_i[\mathbf{u}] \hat{\mathbf{g}}_i[j-\mathbf{u}] + \sum_{\mathbf{u}=j+1}^{d-1} \hat{\mathbf{f}}_i[\mathbf{u}] \hat{\mathbf{g}}_i[j+\mathbf{d}-\mathbf{u}] \omega_i \right) \bmod \mathbf{q}$

3: **end for**

---

### 2.3.4 Point-Wise Multiplication

In fact, NTT maps the coefficient representation of polynomials in  $R_q$  to the *NTT domain*, in which the multiplication between  $\hat{f}$  and  $\hat{g}$ , denoted by  $\hat{f} \circ \hat{g}$ , is done point-wisely.

Moreover, when the symbol  $\circ$  is used with matrices or vectors, it denotes the usual matrix multiplication, with the individual products of entries computed accordingly. Notice that in an incomplete NTT, the basic components of  $\hat{f}$  and  $\hat{g}$  are usually over  $\mathbb{Z}_q[x]/(x^d - \omega_i)$  for some  $d \geq 2$ . Hence, we need an algorithm to multiply  $\hat{\mathbf{f}}_i$  and  $\hat{\mathbf{g}}_i$ , the  $i$ -th component of  $\hat{f}$  and  $\hat{g}$ , respectively, in  $\mathbb{Z}_q[x]/(x^d - \omega_i)$ . In the implementation of Kyber and NTTTRU, a schoolbook multiplication is employed, which is described as Basemul in Algorithm 2. Thus, NTT-based multiplication of  $f$  and  $g$  can be represented as:  $fg = \text{invNTT}(\hat{f} \circ \hat{g}) = \text{invNTT}(\text{NTT}(f) \circ \text{NTT}(g))$ .

## 3 NTT Frequency Leakage Analysis

### 3.1 Attack idea

Our attack approach is based on the observation that when various crafted inputs are applied to the NTT function, they yield outputs with different Hamming weights. Our objective is to leak the Hamming weight of  $\hat{s} = \text{NTT}(s)$  via the frequency side-channel.

Figure 1 provides a visual representation of our idea when  $d = 2$ . A key observation is that when the input to invNTT contains either one non-zero value or two non-zero values, both the output of invNTT and the intermediate layer results involved during its execution demonstrate distinct Hamming weights. Here  $\hat{s}$  is a secret key in the NTT domain. We configure the function  $f$  in such a way that, when transformed into the NTT domain, its NTT representation contains only the last two coefficients as non-zero values:  $\hat{f} = (\hat{f}_0, \hat{f}_1, 0, \dots, 0)$ . By feeding different controllable  $\hat{f}$  into the multiplication  $\hat{f} \circ \hat{s}$ , which is then sent to invNTT as input, we observe two cases. In the first case, when the last two coefficients of  $\hat{f} \circ \hat{s}$  consist of a non-zero constant  $C$  followed by a 0, at least half of the coefficients in the output of invNTT are 0. Consequently, both the resulting

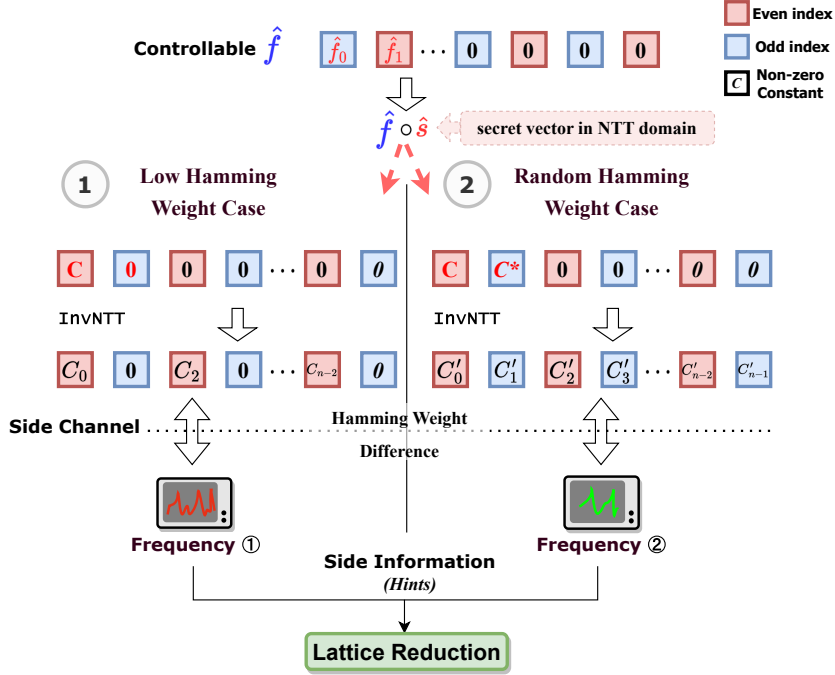


Figure 1: Basic attack idea

output of  $\text{invNTT}$  and the intermediate layer results involved during its execution have low Hamming weight. In the second case, when the last two coefficients are two non-zero constants  $C^*$  and  $C$ , most coefficients in the output of  $\text{invNTT}$  are non-zero. As a result, both the output of  $\text{invNTT}$  and the intermediate layer result involved during its execution have random Hamming weight. To summarize,  $\text{invNTT}$  amplifies even a minor difference in Hamming weights within its input, resulting in a significant difference in Hamming weights in both its output and the intermediate layer result.

We can distinguish the above two cases via the frequency side-channel because Hertzbleed transforms power leakages into a frequency leakage, and data-dependent Hamming weight is known to be a source of power leakage [WPH<sup>+</sup>22]. Specifically, when the input  $\hat{f} \circ \hat{s}$  has only one non-zero coefficient, it would trigger  $\text{invNTT}$  to process data with a lower Hamming weight, resulting in reduced CPU power consumption and an increase in CPU frequency. On the other hand, when the input  $\hat{f} \circ \hat{s}$  has two non-zero coefficients, the CPU runs at a lower frequency. When running  $\text{invNTT}$  with an unknown input which is either case one or case two, we can collect its frequency profile and compare the mean value of frequency to conclude either the unknown input contains only one non-zero coefficient or two non-zero coefficients. We refer to the collected frequency profile as “side information” or “Hints”. Lattice reduction tools such as the one from Dachman-Soled et al., can integrate these hints into the LWE problem and estimate security reduction [DSDGR20].

### 3.2 Frequency and power measurements of $\text{invNTT}$

In this subsection, we present the frequency and power measurements of  $\text{invNTT}$  processing data with low and random hamming weights under the DVFS side-channel. We show that  $\text{invNTT}$  exhibits a power leakage which translates to a frequency leakage under Hertzbleed.



### 3.2.1 Experiment Setup

We run our experiments on an Intel i7-9700 processor with a 3.0 GHz base frequency and 4.7 Turbo boost frequency. Our machine runs Ubuntu 20.04 with Linux kernel 5.15. To measure the CPU frequency and power consumption, we follow the same methodology as Wang et al. [WPH<sup>+</sup>22]. We use the default system setup for CPU frequency measurement. When measuring the CPU power consumption, we fix the CPU frequency to the base frequency to monitor power consumption. During each experiment, we sample the CPU frequency or power consumption every 1 ms.

When targeting the implementation of a specific cryptographic primitive, we configure the target implementation in a multi-threaded setting. We allocate a number of threads equivalent to the number of logical cores. For each thread, we run the target implementation in an infinite loop. When plotting data distributions or reporting the mean of sampled data points, we exclude outliers due to system noise from the unfiltered data. We remove data points that deviate by more than four or two standard deviations from the mean, respectively.

### 3.2.2 Measurement Results.

Our experiment uses the Kyber Reference implementation [ABD<sup>+</sup>b] and Avx2 implementation [ABD<sup>+</sup>a]. The Avx2 implementation leverages the Advanced Vector Extensions (AVX2) instructions to accelerate both NTT and invNTT. The input to invNTT is denoted as  $\hat{r}$ . To trigger the two cases in Section 3.1 that result in invNTT processing data with different Hamming weight, we generate two polynomials,  $\hat{r}_0$  and  $\hat{r}_1$  with  $\hat{r}_0 = [C_0, C_1, 0, \dots, 0]$  and  $\hat{r}_1 = [C_2, 0, 0, \dots, 0]$ . We randomly sample  $C_0, C_1, C_2$  from  $[1, q - 1]$ .

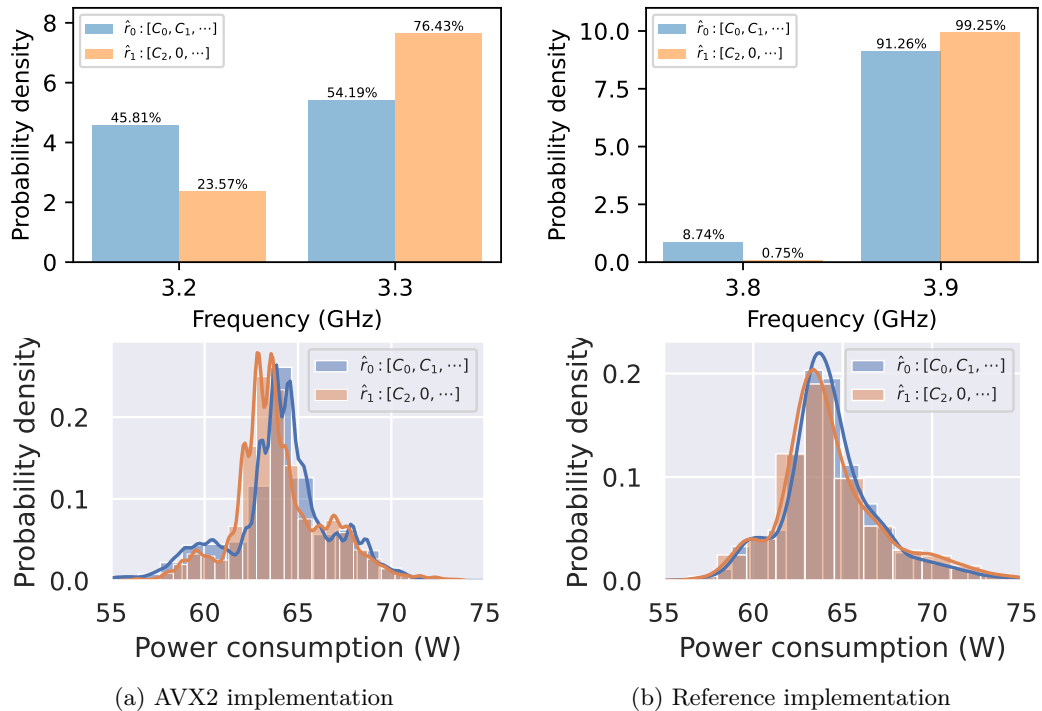


Figure 2: Distribution of CPU frequency and power consumption of executing invNTT followed by NTT in a loop. The results are measured over 10 pairs of randomly generated  $\hat{r}_0$  and  $\hat{r}_1$  on an Intel i7-9600k CPU. For each pair of  $\hat{r}_0$  and  $\hat{r}_1$ , we collect 200,000 data points.



To measure the CPU frequency and power consumption of `invNTT`, we execute `invNTT` and `NTT` back to back in a loop. In our target implementations, `invNTT` directly overwrites the input variable  $\hat{r}$  with the output, which means that we need to re-assign  $\hat{r}$  with  $\hat{r}_0$  or  $\hat{r}_1$  repeatedly to execute `invNTT` ( $\hat{r}$ ) in a loop. To eliminate the side effect due to repeatedly overwriting  $\hat{r}$ , we run `NTT` after `invNTT` to restore the `invNTT` ( $\hat{r}$ ) back to its original value  $\hat{r}$ . To ensure reliability, we randomly select 10 pairs of  $\hat{r}_0$  and  $\hat{r}_1$ . While running the target program, `invNTT` followed by `NTT`, in a while loop, we collect 200,000 data points for each pair of  $\hat{r}_0$  and  $\hat{r}_1$ .

Figure 2 illustrates the results obtained from our target `NTT` implementations. Both the AVX2 and Reference implementations display noticeable and consistent frequency differences. Compared to the Reference implementation, the AVX2 implementation exhibits a more significant frequency difference because the AVX2 workload is heavier, making it more vulnerable to frequency side-channels. The frequency difference is around 22% for the AVX2 implementation and around 8% for the Reference implementation. Specifically, in the random Hamming weight case (blue bar) of the AVX2 implementation shown in Figure 2a, the probability density of higher frequency (3.3 GHz) is 54.19%, which is lower than the 76.44% observed in the low Hamming weight case (yellow bar).

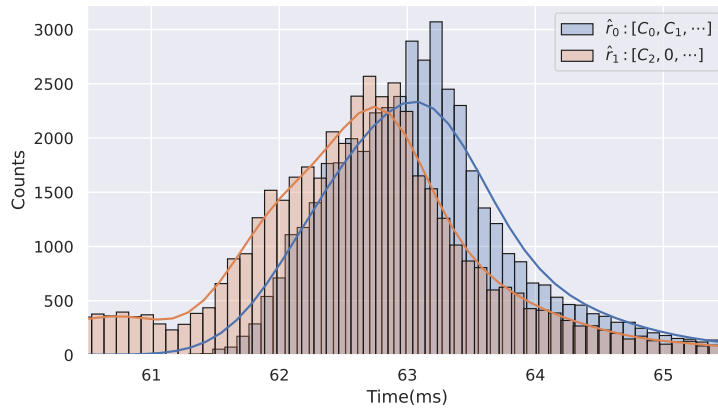


Figure 3: Histograms of timing measurements of 300 concurrent threads with each running a `For` loops of 10,000 iterations of `invNTT` followed by `NTT`. The target implementation is the AVX2 `NTT` implementation. The inputs to `invNTT` are 10 pairs of randomly generated  $\hat{r}_0$  and  $\hat{r}_1$ . We collect 50,000 data points for each experiment. The curves are the Kernel Density Estimate over the raw measurements.

We also perform CPU power consumption measurements and present the results in Figure 2. The power consumption measurements align with our frequency ones, providing further support to our observations. To be specific, in the AVX2 and Reference implementations, the low Hamming weight case (yellow curve) consumes less power than the random Hamming weight case (blue curve).

Due to the significant frequency difference observed in the AVX2 implementation, we can convert the frequency side-channel into a timing side-channel. To measure the timing signals, we modify the `while(1)` loop in the previous experiments into a `For` loop with 10,000 iterations. We start 300 concurrent threads and measure the total running time for 300 concurrent threads, each completing 10,000 iterations of `invNTT` followed by `NTT`. We randomly generate 10 pairs of  $\hat{r}_0$  and  $\hat{r}_1$ , and collect 50,000 data points for each experiment. The results are presented in Figure 3. It takes 62.4 ms on average to process  $\hat{r}_1$ , and 63.9 ms to process  $\hat{r}_0$ . The distinct timing distributions demonstrate that the timing side-channel leaks the number of non-zero coefficients in the `invNTT` input.

In conclusion, our experiment highlights that NTT is vulnerable to the frequency side-channel. It demonstrates that when processing polynomial inputs with non-zero coefficients, `invNTT` processes data with distinct Hamming weights, which causes the CPU power consumption to be input-dependent and the CPU frequency to be input-dependent too under DVFS. For lattice-based constructions that rely on NTT, our observation implies the need for these constructions to examine whether the input to `invNTT` ever depends on the secret key. If it does, the frequency side-channel can potentially leak the secret key. We target two different implementations but our observation is not limited to a specific implementation but rather the algorithm on which NTT is based.

## 4 Leakage Analysis against CPA-secure Kyber

In the previous section, we demonstrate that the number of non-zero coefficients in NTT input leaks via the frequency side-channel. In this section, we discuss the NTT frequency leakage in a broader setting, where we attempt to study the frequency leakage in real world applications that utilizes NTT. We focus on the CPA-secure decryption function in Kyber as a case study.

### 4.1 Kyber KEM

---

**Algorithm 3** CPA-secure Kyber without **Comp** and **Decomp**

---

|  |  |
|--|--|
| <p style="text-align: center;">◇ <b>CPAKEM.Keygen</b></p> <p><b>Output:</b> Public Key <math>\hat{\mathbf{r}}</math>, secret key <math>\hat{\mathbf{s}}</math></p> <p>1: <math>\hat{\mathbf{A}} \xleftarrow{\\$} \mathcal{R}_q^{\alpha \times \alpha}</math>, <math>\mathbf{s}, \mathbf{e} \xleftarrow{\\$} \mathcal{B}_{\eta_1}^{\alpha}</math></p> <p>2: <math>\hat{\mathbf{s}} = \text{NTT}(\mathbf{s})</math>, <math>\hat{\mathbf{e}} = \text{NTT}(\mathbf{e})</math></p> <p>3: <math>\hat{\mathbf{r}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}</math></p> <p style="text-align: center;">return <math>(\hat{\mathbf{r}}, \hat{\mathbf{s}})</math></p> <p style="text-align: center;">◇ <b>CPA.Encryption</b></p> <p><b>Input:</b> Public Key <math>\hat{\mathbf{p}}</math></p> <p><b>Output:</b> <math>\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)</math></p> <p>1: <math>\mathbf{m} \xleftarrow{\\$} \{0, 1\}^{256}</math>, <math>\mathbf{A} \xleftarrow{\\$} \mathcal{R}_q^{k \times k}</math></p> <p>2: <math>\mathbf{r} \xleftarrow{\\$} \mathcal{B}_{\eta_1}^{\alpha}</math>, <math>\mathbf{e}_1, \mathbf{e}_2 \xleftarrow{\\$} \mathcal{B}_{\eta_2}</math>, <math>\hat{\mathbf{r}} = \text{NTT}(\mathbf{r})</math></p> | <p>3: <math>\mathbf{u} = \text{invNTT}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1</math></p> <p>4: <math>\mathbf{v} = \text{invNTT}(\hat{\mathbf{p}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_2 + \mathbf{m}</math></p> <p>5: <math>\mathbf{c}_1 = \mathbf{u}</math>, <math>\mathbf{c}_2 = \mathbf{v}</math></p> <p style="text-align: center;">return <math>\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)</math></p> <p style="text-align: center;">◇ <b>CPA.Decryption</b></p> <p><b>Input:</b> <math>\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)</math>, secret key <math>\hat{\mathbf{s}}</math></p> <p><b>Output:</b> Message <math>\mathbf{m}'</math></p> <p>1: <math>\mathbf{u}' = \mathbf{c}_1, \mathbf{v}' = \mathbf{c}_2</math></p> <p>2: <math>\hat{\mathbf{u}}' = \text{NTT}(\mathbf{u}')</math></p> <p>3: <math>\hat{\mathbf{r}} = \hat{\mathbf{s}} \circ \hat{\mathbf{u}}'</math></p> <p>4: <math>\mathbf{m}' = \mathbf{v}' - \text{invNTT}(\hat{\mathbf{r}})</math></p> <p style="text-align: center;">return <math>\mathbf{m}'</math></p> |
|--|--|

---

Generally, a KEM consists of three parts: a key generation function **Keygen**, a key encapsulation function **Encaps**, and a key decapsulation function **Decaps**. The CCA-secure Kyber uses a CPA-secure encryption algorithm **CPA.Encryption** in both the encapsulation and decapsulation processes. To achieve CCA security, Kyber adopts the Fujisaki-Okamoto (FO) transform [FO99], which involves re-encrypting the decrypted message to verify the validity of the encapsulation process.

Our analysis against Kyber specifically targets the CPA-secure decryption function in Kyber decapsulation without the **Comp** and **Decomp** functions. The two functions, designed to reduce the size of ciphertext and message, present a substantial obstacle in executing our attack. The primary challenge arises from the fact that the **Decomp** function is not the exact inverse of the **Comp** function, making it difficult to find suitable inputs for our attack.

Algorithm 3 provides the details. In **Keygen**, the public key  $\hat{\mathbf{p}}$  is calculated as  $\hat{\mathbf{p}} = \hat{\mathbf{A}} \cdot \hat{\mathbf{s}} + \hat{\mathbf{e}}$ , where  $\hat{\mathbf{s}}$  and  $\hat{\mathbf{e}}$  are generated from a centered binomial distribution  $\mathcal{B}_{\eta}$  with parameter  $\eta$ . We use  $\xleftarrow{\$}$  to represent a random selection and T to represent the transpose of a matrix.

In the **Encryption** process of Kyber, a message  $\mathbf{m}$  is randomly sampled from  $\{0, 1\}^{256}$ , and a matrix  $\mathbf{A}$  is sampled from  $\mathcal{R}_q^{k \times k}$ . The process then involves sampling  $\mathbf{r}, \mathbf{e}_1$ , and  $\mathbf{e}_2$  from corresponding binomial distributions, followed by calculations of  $\mathbf{u}$  and  $\mathbf{v}$  using the input  $\hat{\mathbf{p}}$ . Eventually, the **Encryption** returns the ciphertext  $\mathbf{ct} = (\mathbf{u}, \mathbf{v})$ . In the **Decryption**, Kyber unpack the input ciphertext,  $\mathbf{ct}$ , as  $(\mathbf{u}', \mathbf{v}')$ , then it calculates  $\mathbf{m}' = \mathbf{v}' - \text{invNTT}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}})$  and returns  $\mathbf{m}'$  as the decrypted message.

## 4.2 The analysis against CPA-secure Kyber

---

**Algorithm 4** Analysis against Kyber

---

|   |   |
|---|---|
| <p>◇ DVFS-based Side-Channel Oracle <math>\mathcal{O}</math></p> <p><b>Input:</b> Polynomial <math>\mathbf{u}</math></p> <p><b>Output:</b> FrequencyHints</p> <p>1: Query <b>CPA.Decryption</b> with <math>\mathbf{c}_1 = \mathbf{u}, \mathbf{c}_2 = \mathbf{0}</math>.</p> <p>2: <b>CPA.Decryption</b> computes <math>\text{invNTT}(\text{NTT}(\mathbf{u}) \circ \text{NTT}(\mathbf{s}))</math>, with <math>\mathbf{s}</math> being the secret key.</p> <p>3: The Oracle <math>\mathcal{O}</math> monitors the CPU frequency during <b>CPA.Decryption</b> execution and stores the CPU frequency as FrequencyHints.</p> <p>    <b>return</b> FrequencyHints</p> <p>◇ Our analysis</p> <p><b>Output:</b> Hints about <math>\mathbf{s}_0</math></p> <p>1: <b>for</b> <math>z</math> in <math>[1, q - 1]</math> <b>do</b></p> | <p>2: Choose <math>\mathbf{u}</math> with <math>\hat{\mathbf{u}}'_0 = (1, z, 0, \dots, 0)</math>.</p> <p>3: FrequencyHints = <math>\mathcal{O}(u')</math></p> <p>4: Append the mean of FrequencyHints to MeasuredMeans.</p> <p>5: <b>end for</b></p> <p>6: Select one largest means from the MeasuredMeans, and find the corresponding <math>z_0</math>, which must satisfy <math>\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}z_0\zeta = 0</math> or <math>\hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z_0 = 0</math>.</p> <p>7: <b>for</b> <math>i</math> in <math>[1, n/2 - 1]</math> <b>do</b></p> <p>8: Repeat process 1 to 6 to obtain <math>z_i</math> except that for process 2 choose <math>\mathbf{u}</math> with <math>\hat{\mathbf{u}}'_0 = (1, z_0, 0, \dots, 0, 1, z, 0, \dots, 0)</math> with the additional 1 and <math>z</math> at index <math>2i</math> and <math>2i + 1</math> respectively.</p> <p>9: <b>end for</b></p> <p>10: <b>return</b> Hints = <math>\{z_0, z_1, \dots, z_{n/2-1}\}</math>.</p> <hr/> |
|---|---|

There are three variants of Kyber, namely Kyber-512, Kyber-768, and Kyber-1024, each offering different levels of security. Taking Kyber-1024 as an example, note that the secret key of Kyber-1024 consists of 4 polynomials:  $\mathbf{s} = [\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3]$ . We proceed to extract hints for the secret polynomials individually. Let us illustrate the process by considering the extraction of hints for  $\mathbf{s}_0$  as an example. It is worth noting that similar steps can be followed to recover hints for the remaining polynomials.

We summarize our analysis against the CPA-secure Kyber in Algorithm 4, in which we learn hints about  $\mathbf{s}_0$  via a DVFS-based Side-Channel Oracle  $\mathcal{O}$ . There are mainly two steps in Algorithm 4:

1. Find  $z_0 \in \mathbb{Z}_q$  such that

$$\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}z_0\zeta = 0 \pmod{q} \text{ or } \hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z_0 = 0 \pmod{q};$$

2. Find  $z_1, z_2, \dots, z_{n/2-1} \in \mathbb{Z}_q$  such that it holds that

$$\hat{\mathbf{s}}_{0,2i+1} + \hat{\mathbf{s}}_{0,2i}z_i = 0 \pmod{q} \text{ for } i = 0, 1, \dots, n/2 - 1,$$

or

$$\hat{\mathbf{s}}_{0,2i} + \hat{\mathbf{s}}_{0,2i+1}z_i\zeta = 0 \pmod{q} \text{ for } i = 0, 1, \dots, n/2 - 1.$$

Although we can not exactly tell which case happens in Step 2 by the side channel information, it is enough for us to employ the lattice reduction tools, such as [DSDGR20], to estimate how much the security is affected.

### 4.2.1 How to find $z_0$

Remind that the attacker targets the first polynomial  $\mathbf{s}_0$ . He sets the input to **CPA.Decryption** as  $\mathbf{c}_1 = \mathbf{u}' = (\mathbf{u}'_0, \mathbf{0}, \mathbf{0}, \mathbf{0})$  to target  $\mathbf{s}_0$ , because when **CPA.Decryption** multiplies  $\mathbf{s}$  with  $\mathbf{u}'$  (Line 3 in Algorithm 3), the result would involve  $\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0$ . More precisely, we have  $\hat{\mathbf{r}} = (\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0, 0, \dots, 0)$ .

The attacker next sets  $\mathbf{u}'_0$  such that  $\hat{\mathbf{u}}'_0$  is  $(1, z, 0, 0, \dots, 0)$  with  $z \in [1, q-1]$ . Note that the multiplication  $\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0$  is a point-wise multiplication that parses  $\hat{\mathbf{s}}_0$  and  $\hat{\mathbf{u}}'_0$  at block level with block size  $d = 2$ , and performs a Basemul operation between each pair of blocks. Since  $\hat{\mathbf{u}}'_0$  only has the first two elements as non-zero,  $\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0$  essentially involves with only one Basemul:

$$(\hat{\mathbf{r}}_{0,0} + \hat{\mathbf{r}}_{0,1}x) = (\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}x) \cdot (1 + zx) \bmod (x^2 - \zeta) \quad (8)$$

$$= (\hat{\mathbf{s}}_{0,0} + (\hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z)x + \hat{\mathbf{s}}_{0,1}zx^2) \bmod (x^2 - \zeta) \quad (9)$$

$$= (\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}z\zeta) + (\hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z)x. \quad (10)$$

Hence we have:

$$\hat{\mathbf{r}} = (\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0, 0, \dots, 0) = (\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}z\zeta, \hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z, 0, \dots, 0). \quad (11)$$

**CPA.Decryption** then computes invNTT on  $\hat{\mathbf{r}}$ . If  $\hat{\mathbf{r}}$  contains only one non-zero coefficient ( $\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}z\zeta = 0$  or  $\hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z = 0$ ), **CPA.Decryption** would process data with low Hamming weight, and as a result, the CPU runs faster. Otherwise, if  $\hat{\mathbf{r}}$  contains more than one non-zero coefficients, **CPA.Decryption** would process data with random Hamming weight, and as a result, the CPU runs relatively slower.

By this observation, we can enumerate  $z$  from 1 to  $q-1$  as in the attack and obtain FrequencyHints returned by oracle  $\mathcal{O}$  for each  $z$ . We know there exists exactly two values of  $z$ ,  $-(\hat{\mathbf{s}}_{0,1}\zeta)^{-1}\hat{\mathbf{s}}_{0,0}$  and  $-\hat{\mathbf{s}}_{0,0}^{-1}\hat{\mathbf{s}}_{0,1}$ , such that they trigger  $\hat{\mathbf{r}}$  to contain only one non-zero coefficient. Therefore, we sort all collected FrequencyHints and conclude that the two values correspond to the highest two CPU frequencies. Picking any one of the two values as  $z_0$ , we know it is either  $-(\hat{\mathbf{s}}_{0,1}\zeta)^{-1}\hat{\mathbf{s}}_{0,0}$  or  $-\hat{\mathbf{s}}_{0,0}^{-1}\hat{\mathbf{s}}_{0,1}$ . Or equivalently, it holds either

$$\hat{\mathbf{s}}_{0,0} + \hat{\mathbf{s}}_{0,1}z_0\zeta = 0 \pmod{q}, \quad (12)$$

which implies the resulting  $\hat{\mathbf{r}}$  equals to  $(0, C_1, 0, \dots, 0)$ , or

$$\hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z_0 = 0 \pmod{q}, \quad (13)$$

which implies the resulting  $\hat{\mathbf{r}}$  equals to  $(C_0, 0, 0, \dots, 0)$ .

We explain this observation in Lemma 1.

**Lemma 1.** *In the calculation of  $\mathbf{r} = \text{invNTT}(\hat{\mathbf{r}}) = \text{invNTT}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}}')$  in Kyber with Algorithm 1, if  $\hat{\mathbf{r}} = (C, 0, \dots, 0)$  or  $(0, C, \dots, 0, 0)$ , where  $C$  is a non-zero constant, then for the  $m$ -th layer (or equivalently, at the end of the  $m$ -th loop between Step 2 and Step 9) where  $1 \leq m \leq k-1$ , there are  $2^m$  non-zero components in the intermediate variable  $\hat{\mathbf{r}}$ . Specially, in the final output, there are only  $2^k$  non-zero components  $\mathbf{r}$ , and  $\mathbf{r}$  contains only even or odd coefficients, i.e.  $\mathbf{r}$  is of the form  $C_0 + C_2x^2 + C_4x^4 + \dots + C_{n-2}x^{n-2}$  or  $x(C_0 + C_2x^2 + C_4x^4 + \dots + C_{n-2}x^{n-2})$ .*

*If  $\hat{\mathbf{r}} = (C + C'x, 0, \dots, 0)$ , where  $C$  and  $C'$  are non-zero constants, then for the  $m$ -th layer, there are only  $2^{m+1}$  non-zero components in the intermediate variable  $\hat{\mathbf{r}}$ . Specially, there are  $2^{k+1}$  non-zero components in the final output  $\mathbf{r}$ .*

It can be easily seen that for every loop between Step 2 and Step 9 in Algorithm 1, the current  $\hat{r}[j]$  and  $\hat{r}[j + \text{len}]$  will only be used to update the new  $\hat{r}[j]$  and  $\hat{r}[j + \text{len}]$  and conversely the new  $\hat{r}[j]$  and  $\hat{r}[j + \text{len}]$  will be determined by the current  $\hat{r}[j]$  and  $\hat{r}[j + \text{len}]$

completely. Then Lemma 1 follows immediately by induction, that is, by the GS-Butterfly unit, in the first loop, a non-zero constant will yield 2 non-zero constants, and in the second loop, 2 non-zero constants will yield 4 non-zero constants and so on. Figure 4 provides an illustrative example of Lemma 1, demonstrating a simplified case with 8 coefficients.

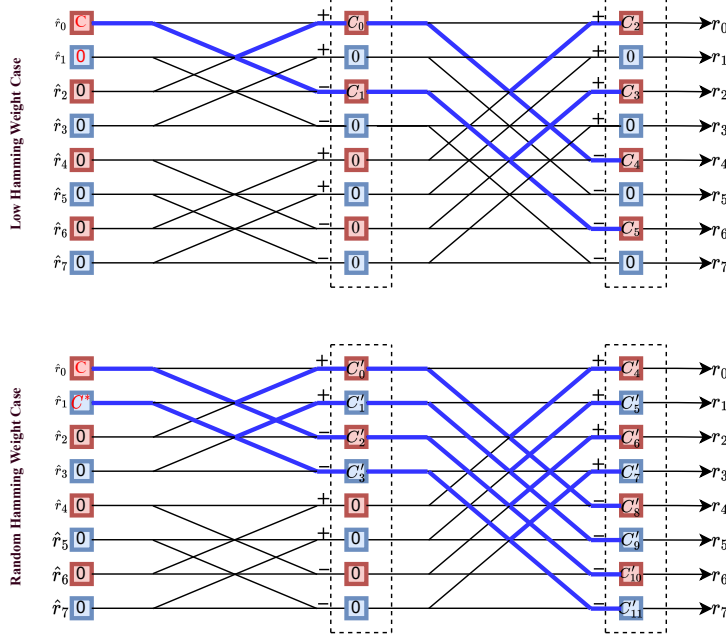


Figure 4: An example for Lemma 1 (A Simplified version with 8 coefficients. )

#### 4.2.2 How to find $z_1, z_2, \dots, z_{n/2-1}$

A direct attempt to find  $z_1$  is to set  $\mathbf{u}'_0$  such that  $\hat{\mathbf{u}}'_0$  is  $(0, 0, 1, z, \dots, 0)$  with  $z \in [1, q-1]$ , and then employ a similar process as in Section 4.2.1. However, note that we always have two possible equations similar to Equations (12) and (13) for  $z_1$ , and we cannot tell which one holds exactly. Hence, for  $z_0$  and  $z_1$ , there are a total of four candidate systems of equations they may satisfy. For  $z_0, z_1, z_2, \dots, z_{n/2-1}$ , it becomes much worse since the number of candidates increases to  $2^{n/2}$ . Luckily, we have a neat observation to solve this dilemma.

To elaborate, suppose the Basemul between the first block of  $\hat{\mathbf{s}}_0$  and  $\hat{\mathbf{u}}'_0$  is  $(C_0, 0)$ , that is,  $\hat{\mathbf{s}}_{0,1} + \hat{\mathbf{s}}_{0,0}z_0 = 0$ . When we target block  $i$ , we set  $\mathbf{u}'_0$  such that  $\hat{\mathbf{u}}'_0$  is  $(1, z_0, 0, 0, \dots, 1, z, \dots, 0)$  with the additional 1 and  $z$  at index  $2i$  and  $2i+1$  respectively. Similar to Equation (11), we have that

$$\hat{\mathbf{r}} = (\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0, 0, \dots, 0) = (C_0, 0, \dots, 0, \hat{\mathbf{s}}_{0,2i} + \hat{\mathbf{s}}_{0,2i+1}z_i\zeta, \hat{\mathbf{s}}_{0,2i+1} + \hat{\mathbf{s}}_{0,2i}z_i, 0, \dots, 0). \quad (14)$$

The  $\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0$  would trigger the oracle  $\mathcal{O}$  to return high FrequencyHints if and only if the Basemul between the  $i$ -th block of  $\hat{\mathbf{s}}_0$  and  $\hat{\mathbf{u}}'_0$  results in  $(C_{2i}, 0)$  but not  $(0, C_{2i+1})$  or  $(C_{2i}, C_{2i+1})$ . We explain this observation in Lemma 2:

**Lemma 2.** *In the calculation of  $\mathbf{r} = \text{invNTT}(\hat{\mathbf{r}}) = \text{invNTT}(\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}')$  in Kyber with Algorithm 1, if  $\hat{\mathbf{r}} = (C_0, 0, \dots, 0, C_{2i}, 0, \dots, 0)$  where  $C_0$  and  $C_{2i}$  are non-zero constants and  $1 \leq i < n/2$ , then there are at least  $2^k$  non-zero components in the final output of  $\mathbf{r}$ . However, if  $\hat{\mathbf{r}} = (C_0, 0, \dots, 0, C_{2i}, C_{2i+1}, 0, \dots, 0)$  or  $\hat{\mathbf{r}} = (C_0, 0, \dots, 0, C_{2i+1}, 0, \dots, 0)$  where  $C_0$*

and  $C_{2i}, C_{2i+1}$  are non-zero constants and  $1 \leq i < n/2$ , then there are  $2^{k+1}$  non-zero components in the final output  $\mathbf{r}$ .

Similarly, if  $\hat{\mathbf{r}} = (0, C_1, 0, \dots, 0, C_{2i+1}, 0, \dots, 0)$  where  $C_1$  and  $C_{2i+1}$  are non-zero constants and  $1 \leq i < n/2$ , there are  $2^k$  non-zero components in the final output  $\mathbf{r}$ . However, if  $\hat{\mathbf{r}} = (0, C_1, 0, \dots, 0, C_{2i}, 0, \dots, 0) = (0, C_1, \dots, 0, C_{2i}, C_{2i+1}, 0, \dots, 0)$  where  $C_1, C_{2i}$  and  $C_{2i+1}$  are non-zero constants and  $1 \leq i < n/2$ , then there are  $2^{k+1}$  non-zero components in the final output  $\mathbf{r}$ .

As a result, when  $\mathcal{O}$  return a  $z_i$  that corresponds to a `FrequencyHints`,  $z_i$  must be  $-\hat{\mathbf{s}}_{\mathbf{0},2i}^{-1}\hat{\mathbf{s}}_{\mathbf{0},2i+1}$ . In other words,  $z_i$  is enforced to have the same style of linear relationship as  $z_0$ :  $\hat{\mathbf{s}}_{\mathbf{0},1} + \hat{\mathbf{s}}_{\mathbf{0},0}z_0 = 0$  and  $\hat{\mathbf{s}}_{\mathbf{0},2i+1} + \hat{\mathbf{s}}_{\mathbf{0},2i}z_i = 0$ .

Applying the same methodology to all the other blocks, we obtain a set  $\{z_0, z_1, \dots, z_{n/2-1}\}$  which yields the linear relationships of the secret key for every block  $i$ :

$$\hat{\mathbf{s}}_{\mathbf{0},2i+1} + \hat{\mathbf{s}}_{\mathbf{0},2i}z_i = 0 \pmod{q}. \quad (15)$$

In the case that the `Basemul` between the first block of  $\hat{\mathbf{s}}_{\mathbf{0}}$  and  $\hat{\mathbf{u}}'_0$  is  $(0, C_1)$ , that is,  $\hat{\mathbf{s}}_{\mathbf{0},0} + \hat{\mathbf{s}}_{\mathbf{0},1}z_0\zeta = 0$ , a similar analysis shows that the set  $\{z_0, z_1, \dots, z_{n/2-1}\}$  returned by the attack yields the linear relationships of the secret key for every block  $i$ :

$$\hat{\mathbf{s}}_{\mathbf{0},2i} + \hat{\mathbf{s}}_{\mathbf{0},2i+1}z_i\zeta = 0 \pmod{q}. \quad (16)$$

### 4.3 Frequency Measurement in Kyber CPA.Decryption

To validate the effectiveness of our analysis, we select the `CPA.Decryption` in the Kyber AVX2 implementation [ABD<sup>+</sup>a] as our target. Following the setup described in Section 3.2.1, we execute a multi-threaded Kyber `CPA.Decryption` process on an Intel i7-9600 CPU with the default system configuration. Each thread in the process performs decryption of a ciphertext  $c = (\mathbf{u}, \mathbf{v})$  in a loop until the maximum time limit is reached.

#### 4.3.1 Searching for $z_0$ that triggers a low Hamming weight case

We target Kyber-1024 with  $\mathbf{s} = [\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3]$ . We demonstrate Lemma 1 in Section 4.2 by targeting the first NTT block of the first secret key polynomial  $\mathbf{s}_0$ . As discussed above, we set  $\mathbf{u}' = (\mathbf{u}'_0, \mathbf{0}, \mathbf{0}, \mathbf{0})$  with the NTT representation of  $\mathbf{u}'_0$  as  $\hat{\mathbf{u}}'_0 = (1, z_0, 0, 0, \dots, 0)$ . We show that while enumerating all  $z_0 \in [1, q-1]$ , it is possible to identify the two  $z_0$  that trigger the low Hamming weight case in `CPA.Decryption` via the DVFS-based oracle  $\mathcal{O}$ .

In Figure 5, we present the CPU frequency means of Kyber `CPA.Decryption` with different  $z_0$  as input. We prepare 100 different  $z_0$  including two special  $z_0$  that trigger the DVFS-based oracle  $\mathcal{O}$  to return high `FrequencyHints`. For each  $z_0$ , we collect 100,000 frequency data points and calculate the mean of the measured frequency. Among them, we can identify two points with higher frequency (the red points). To be specific, the mean of the red points is 3.379 GHz, while the mean of the blue points is 3.373 GHz. From Lemma 1, the two red  $z_0$  correspond to  $-(\hat{\mathbf{s}}_{\mathbf{0},1}\zeta)^{-1}\hat{\mathbf{s}}_{\mathbf{0},0}$  and  $-\hat{\mathbf{s}}_{\mathbf{0},0}^{-1}\hat{\mathbf{s}}_{\mathbf{0},1}$ . They trigger  $\hat{\mathbf{r}} = \hat{\mathbf{s}}_{\mathbf{0}} \circ \hat{\mathbf{u}}'_0 = (C_2, 0, 0, \dots, 0)$  or  $(0, C_3, 0, \dots, 0)$ . As a result, when computing `invNTT`( $\hat{\mathbf{r}}$ ), `CPA.Decryption` processes data with a lower Hamming weight and the CPU runs faster.

To further verify our result, we randomly select 10 secret keys. In Figure 6, we use a violin plot to depict the CPU frequency means distribution of  $z_0$  triggering the low Hamming weight case versus  $z_0$  triggering the random Hamming weight case. For each secret key, we choose two  $z_0$  that trigger lower Hamming weight and two  $z_0$  that trigger random Hamming weight. Each point in Figure 6 is a CPU frequency mean. Inside each violin plot, we give an additional box plot to provide details of the two distributions. The edge of the box represents the lower and upper quartiles of the distribution, while the distribution median and mean are marked by a solid line and a dotted line inside the box, respectively. We can clearly distinguish the two distributions.

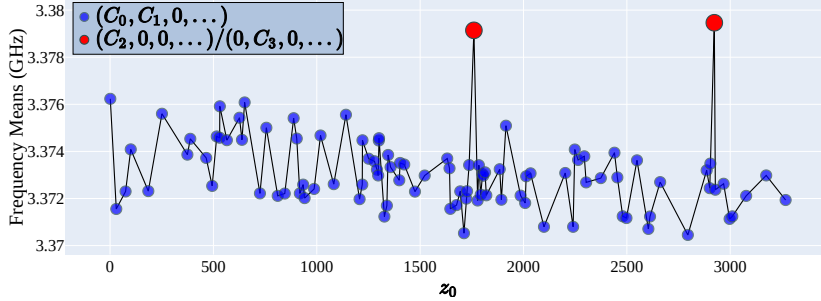


Figure 5: Frequency means of the AVX2 Kyber **CPA.Decryption**. We fix one randomly generated  $sk$  and sampled 100  $z_0$  in total. We sample 98 random  $z_0$  that trigger **CPA.Decryption** processing data with random Hamming weight ( $\hat{\mathbf{r}}$  to be the form of  $(C_0, C_1, 0, \dots, 0)$ ) and compute two special  $z_0$  that trigger **CPA.Decryption** processing data with low Hamming weight ( $\hat{\mathbf{r}}$  to be the form of  $(C_2, 0, 0, \dots, 0)$  or  $(0, C_3, 0, \dots, 0)$ ). The two special  $z_0$  are marked in red to reflect the fact that the DVFS-based oracle  $\mathcal{O}$  returns high FrequencyHints with them as inputs.

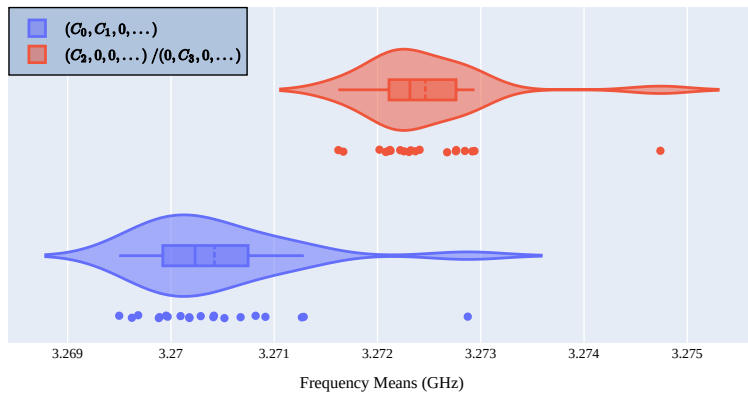


Figure 6: Distribution of measured CPU frequency means with 10 different secret keys. For each secret key, we choose two  $z_0$  that trigger **CPA.Decryption** processing data with lower Hamming weight ( $\hat{\mathbf{r}}$  to be the form of  $(C_2, 0, 0, \dots, 0)$  or  $(0, C_3, 0, \dots, 0)$ ) and two  $z_0$  that trigger **CPA.Decryption** processing data with random Hamming weight ( $\hat{\mathbf{r}}$  to be the form of  $(C_0, C_1, 0, \dots, 0)$ )



### 4.3.2 Searching for $z_1, z_2, \dots, z_{n/2-1}$

We go on to demonstrate Lemma 2 in Section 4.2. We randomly assign  $z_0$  to be one of the two red points, and  $\hat{\mathbf{u}}'_0$  is  $(1, z_0, 0, 0, \dots)$ .  $\hat{\mathbf{r}} = \hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0$  can be either  $(C_0, 0, 0, \dots, 0)$  or  $(0, C'_0, 0, \dots, 0)$ . When targeting the  $i$ -th NTT block of  $\hat{\mathbf{s}}_0$ , we set  $\hat{\mathbf{u}}'_0$  as  $(1, z_0, 0, 0, \dots, 1, z_i, 0, \dots)$  with the index of  $z_i$  to be  $2i + 1$  and enumerating  $z_i \in [1, q - 1]$ . The DVFS-based oracle  $\mathcal{O}$  would return high FrequencyHints if and only if the Basemul between  $(1, z_i)$  and the  $i$ -th block of  $\hat{\mathbf{s}}_0$  results in a block that echos the first block of  $\hat{\mathbf{r}}$ . If the first block is  $(C_0, 0)$ , the  $i$ -th block has to be  $(C_i, 0)$ . If the first block is  $(0, C'_0)$ , the  $i$ -th block has to be  $(0, C'_i)$ . We provide experimental results to support this Lemma.

Suppose the first block of  $\hat{\mathbf{r}}$  is  $(C_0, 0)$ . For a block at index  $i$ , we set the  $2i + 1$  index of  $\hat{\mathbf{u}}'_0$  as  $z_i$  or  $z'_i$ . The Basemul between  $(1, z_i)$  and the  $i$ -th block of  $\hat{\mathbf{s}}_0$  results in  $(C_i, 0)$  ( $z_i = -\hat{\mathbf{s}}_{1,0}^{-1} \hat{\mathbf{s}}_{i,1}$ ) and the Basemul between  $(1, z'_i)$  and the  $i$ -th block of  $\hat{\mathbf{s}}_0$  results in  $(0, C'_i)$  ( $z'_i = -(\hat{\mathbf{s}}_{i,1} \zeta)^{-1} \hat{\mathbf{s}}_{i,0}$ ). When  $\hat{\mathbf{u}}'_0$  is set with  $z_i$ , we trigger the  $i$ -th block of  $\hat{\mathbf{r}}$  to be  $(C_i, 0)$ . **CPA.Decryption** computing  $\text{invNTT}(\hat{\mathbf{r}})$  with  $\hat{\mathbf{r}} = (C_0, 0, \dots, C_i, 0, \dots)$  would process data with lower Hamming weight and the CPU runs faster. On the other hand, When  $\hat{\mathbf{u}}'_0$  is set with  $z'_i$ , we trigger the  $i$ -th block of  $\hat{\mathbf{r}}$  to be  $(0, C'_i)$ . **CPA.Decryption** computing  $\text{invNTT}(\hat{\mathbf{r}})$  with  $\hat{\mathbf{r}} = (C_0, 0, \dots, 0, C'_i, \dots, 0)$  would process data with random Hamming weight and the CPU runs slower.

We target 10 different blocks of the secret key. For each  $z_i$ , we collect 500,000 CPU frequency data points and calculate the mean of the measured frequency. We group the 10 CPU frequency means of  $z_i$  that triggers  $(C_i, 0)$  and the 10 CPU frequency means of  $z'_i$  that triggers  $(0, C'_i)$ . In Figure 7, we plot a violin plot that encloses a box plot for CPU frequency means that correspond to  $z_i$  and  $z'_i$ . As we can observe in the figure, the CPU frequency means of the low Hamming weight case caused by  $(C_0, 0, \dots, C_i, 0, \dots, 0)$  are distinctively higher than the CPU frequency means of the random Hamming weight case caused by  $(C_0, 0, \dots, 0, C'_i, \dots, 0)$ . The experimental result confirms that we can observe the Hamming weight difference stated in Lemma 2 through the DVFS-based oracle  $\mathcal{O}$ .

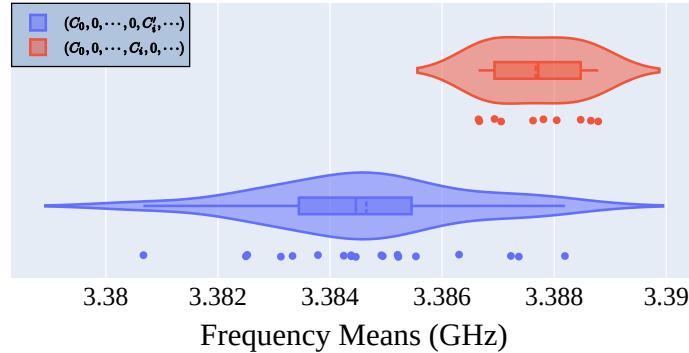


Figure 7: Distribution of measured CPU frequency means targeting 10 random blocks of the secret key. For each block  $i$ , we configure  $\hat{\mathbf{u}}'_0$  with  $z_i$  and  $z'_i$  such that  $z_i$  triggers **CPA.Decryption** to compute on  $(C_0, 0, \dots, C_i, 0, \dots, 0)$  and  $z'_i$  triggers **CPA.Decryption** to compute on  $(C_0, 0, \dots, 0, C'_i, \dots, 0)$ .

### 4.3.3 Comparison of number of queries

There has been a long line of research against the **CPA.decryption** of Kyber, which is called the key mismatch attack or plaintext checking (PC) oracle-based side channel attack [RRCB20, QCZ+21, UXT+22]. In fact, in the work of [QCZ+21], only a few thousand queries are needed. Moreover, with the help of information from electromagnetic (EM) or frequency leakages, side-channel attacks (SCAs) can also be launched against

**CCA.decaps** of Kyber [RRCB20, SCZ+23]. The reported queries needed in the work of [RRD+23] and [TUX+23] can be lowered to less than a thousand queries.

To compare the needed number of queries with the state-of-the-art, we define one query as access to the DVFS-based Side-Channel oracle in Algorithm 4. Therefore, for each  $z_i \in [1, q]$ , we need  $q$  queries to find  $z_i$  where  $i \in [0, n/2 - 1]$ . In total, we need  $qn/2$  queries, that is 425,984 for Kyber-512, 851,968 for Kyber-768, and 1,277,952 queries for Kyber-1024. We need to emphasize that, in our oracle, one query may correspond to a certain number of **CPA.decryption**, while in previous PC oracle, generally one query corresponds to one **CPA.decryption**. The reason is that in a DVFS-based side-channel attack, we have to repeatedly execute **CPA.decryption** in a fixed time interval to trigger the leakage. In our experiment, the time interval is set as 100 s, which collects 100,000 frequency points. In total, we can find  $z_0$  in approximately 138 hours. In terms of query numbers, our attack performs worse than the PC oracle-based ones. However, compared to traditional EM-based side-channel attacks, which demand fine-grained traces, our attack exploits the CPU’s dynamic frequency feature and relies only on coarse-grained CPU frequency leakages. A potential advantage of our approach is the new attacking surface, which may spark remote attacks without requiring physical access.

## 4.4 Reevaluating the security of Kyber with hints

### 4.4.1 Converting the relationship in the NTT domain to polynomial domain

In this Subsection, we show how to leverage the above hints. Taking  $\hat{\mathbf{s}}_0$  for example, after getting the corresponding hints  $\{z_0, z_1, \dots, z_{n/2-1}\}$ , we know that either Equation 15 or Equation 16 holds. Without loss of generality, we assume that Equation 15 holds, that is, for  $i = 0, 1, \dots, n/2 - 1$ , we have

$$\hat{\mathbf{s}}_{0,2i+1} + \hat{\mathbf{s}}_{0,2i}z_i = 0 \pmod{q}. \quad (17)$$

We next convert the linear relationship in the NTT domain to the linear combination of the coefficients of  $\mathbf{s}_0$ , which facilitates the lattice analysis. By the NTT transformation, it is well known that there exists a matrix  $\mathbf{N}$  such that

$$\hat{\mathbf{s}}_0 = \mathbf{N} \cdot \mathbf{s}_0 \pmod{q}, \quad (18)$$

which means that every  $\hat{\mathbf{s}}_{0,i}$  can be written as the linear combination of the coefficients of  $\mathbf{s}_0$ . Substituting  $\hat{\mathbf{s}}_{0,2j+1}$  and  $\hat{\mathbf{s}}_{0,2j}$  in Equation 17 with the corresponding linear combinations, we can easily find linear relations for the coefficients of  $\mathbf{s}_0$ . As a result, we can obtain  $n/2$  linear equations for each polynomial in total.

An interesting problem is whether we can solve the corresponding LWE problem to recover  $\mathbf{s}$  by using the primal attack with these new simultaneous equations. However, it is a challenging task since the dimension of the lattice is still too high to solve its SVP. Fortunately, [DSDGR20] presents a framework to evaluate the concrete hardness in this case.

### 4.4.2 Integrating Hints to Framework in [DSDGR20] and [MN23]

In [DSDGR20], the authors present a lattice framework that quantifies the security loss of lattice-based schemes, specifically the LWE problem, when there is side information leakage. They introduce the concept of “Hints” to capture this leakage and analyze the impact on the security of the schemes. The work of [MN23] extends the work of [DSDGR20] by using the Lattice LLL algorithm to estimate the security loss and secret recovery in lattice-based schemes with side information leakage. However, their approach requires a large number of perfect hints, nearly  $n/2$  for LWE dimension  $n$ , which may not apply to our work.

In [DSDGR20], the authors defines *modular Hints* as the inner product of  $\mathbf{v}$  and  $\mathbf{u}$  modular  $q$ , satisfying:  $\langle \mathbf{v}, \mathbf{s} \rangle = 0 \pmod{q}$ . We can observe that the hints we have in our analysis can indeed be considered as *modular Hints*. We execute the script provided by [DSDGR20] in Sage 9.6 and incorporate the linear equations as “modular hints” into the framework. The resulting estimation is summarized in Table 1. The security loss is estimated to be 49 bits, 65 bits, and 87 bits, respectively. The corresponding percentages of loss are 41.88%, 35.71%, and 35.80%.

Table 1: Estimation of Security Loss in our analysis against modified implementations of Kyber

|            | #Hints | Original Security | with hints | Security Loss |
|------------|--------|-------------------|------------|---------------|
| Kyber-512  | 256    | 117               | 68         | 49 (-41.88%)  |
| Kyber-768  | 384    | 182               | 117        | 65 (-35.71%)  |
| Kyber-1024 | 512    | 243               | 156        | 87 (-35.80%)  |

An interesting problem here is that if we have more possible partial leakage information, can we extract the secret key directly? In [MN23], the authors propose a new approach to determine the number of hints that are sufficient to efficiently break LWE-based lattice schemes. The approach in [MN23] demands a large number of hints to run lattice reduction, while in our analysis, the number of hints is enough. To fill such a gap, we can combine our analysis with possible partial leakage to achieve an end-to-end key recovery attack. We simulate the partial leakage from the other helpful side-channel techniques. We then integrate the partial leakage as hints and execute the script provided by [MN23] in Sage 9.6. In our experiments, if we can have 150 more hints for Kyber-512, 230 for Kyber-768, and 300 for Kyber-1024, respectively, then together with our existing frequency hints, we are able to achieve a full key recovery.

## 5 Security analysis against NTTRU

In this section, we extend our analysis to NTTRU, an NTRU-based KEM that employs NTT. One significant difference in NTTRU is the absence of Compression and Decompression functions, which simplifies the process of selecting specific ciphertexts for our analysis. As a result, we can more easily target and examine the desired ciphertexts in NTTRU.

### 5.1 NTTRU

First, we introduce NTTRU in Algorithm 5. Recall that NTTRU adopts  $\mathbb{Z}_{7681}[x] / (x^{768} - x^{384} + 1)$  with  $n = 768 = 2^8 * 3$  and  $q = 7681$ .

In the **NTTRU.Keygen**, the secret key  $f'$  (or  $g$ ) is sampled from the binomial distribution  $\mathcal{B}_2$ , which produces coefficients from the set  $\{-1, 0, 1\}$ . The key generation process then computes the secret key  $f$  as  $f = 3f' + 1$ . Additionally, it computes the element-wise multiplication of the NTT( $3g$ ) with the inverse of NTT( $f$ ), denoted as  $\hat{h} = \hat{3}g \circ \hat{f}^{-1}$ . These resulting values,  $\hat{f}$  and  $\hat{h}$ , are then used as the secret and public keys, respectively.

### 5.2 Analysis against NTTRU

Similar to our analysis of Kyber, we perform a DVFS-based side-channel analysis against NTTRU. Due to the simple design of NTTRU, we directly analyze the **CCAKEM.Decaps** scheme to identify potential side-channel vulnerabilities.

---

**Algorithm 5** CCA-secure NTTRU
 

---

|  |  |
|--|--|
| <p style="margin: 0;"> <math>\diamond</math> <b>NTTRU.Keygen</b><br/> <b>Output:</b> Public Key <math>\hat{h}</math>, secret key <math>\hat{f}</math><br/>           1: <math>f' \xleftarrow{\\$} \mathcal{B}_2</math><br/>           2: <math>f = 3f' + 1</math><br/>           3: <math>\hat{f} = \text{NTT}(f)</math><br/>           4: <math>g \xleftarrow{\\$} \mathcal{B}_2</math><br/>           5: <math>\hat{3}g = \text{NTT}(3g)</math><br/>           6: <b>If</b> <math>f</math> is not invertible : Restart<br/>           7: <math>\hat{h} = \hat{3}g \circ \hat{f}^{-1}</math><br/>           8: <b>return</b> <math>\hat{h}, \hat{f}</math><br/> <math>\diamond</math> <b>CCAKEM.Encaps</b><br/> <b>Input:</b> Public Key <math>\hat{h}</math><br/> <b>Output:</b> Ciphertext <math>\text{ct}</math>, Shared Key <math>K</math><br/>           1: <math>m \leftarrow \mathcal{R}</math><br/>           2: <math>r \leftarrow \text{Sampler}(H(m))</math><br/>           3: <math>\triangleright \text{CPA.Enc}(\hat{h}, m, r)</math><br/>           4: <math>\hat{r} = \text{NTT}(r)</math> </p> | <p style="margin: 0;">           5: <math>\hat{m} = \text{NTT}(m)</math><br/>           6: <math>\hat{v} = \hat{r} \circ \hat{h}</math><br/>           7: <math>\hat{c} = \hat{v} + \hat{m}</math><br/>           8: <b>return</b> <math>\hat{c}</math><br/> <math>\diamond</math> <b>CCAKEM.Decaps</b><br/> <b>Input:</b> Ciphertext <math>\hat{c}</math>, Secret Key <math>\hat{f}</math><br/>                     Public Key <math>\hat{h}</math><br/> <b>Output:</b> Shared Key <math>K</math><br/>           1: <math>\triangleright \text{CPA.Dec}(\text{s}, \text{ct})</math><br/>           2: <math>m = \text{invNTT}(\hat{m}) = \text{invNTT}(\hat{c} \circ \hat{f})</math><br/>           3: <math>r \leftarrow \text{Sampler}(H(m))</math><br/>           4: <math>c' = \text{CPA.Enc}(\hat{h}, m, r)</math><br/>           5: <b>if</b> <math>c' = c</math> <b>then</b><br/>           6:     <math>K = H(m)</math><br/>           7: <b>else</b><br/>           8:     <math>K = 0</math><br/>           9: <b>end if</b> </p> |
|--|--|

---

**Attack Methods.** To infer information about  $f$ , we set the ciphertext polynomial  $\hat{c}$  as  $(1 + x + zx^2, 0, 0 \dots, 0)$  with  $z \in [1, q - 1]$  and send it to **CCAKEM.Decaps** in Algorithm 5. With  $\hat{c}$ , **CCAKEM.Decaps** first calculates the Basemul with  $d = 3$  (Line 2 in Algorithm 5) over the NTT domain. That is, the first component of  $\hat{m}$  is:

$$\hat{m}[0] = (1 + x + zx^2) \cdot (\hat{f}_0 + \hat{f}_1x + \hat{f}_2x^2) \bmod (x^3 - \zeta) \quad (19)$$

$$= (\hat{f}_0 + \zeta(z\hat{f}_1 + \hat{f}_2) + (\hat{f}_0 + \hat{f}_1 + z\hat{f}_2\zeta)x + (z\hat{f}_0 + \hat{f}_1 + \hat{f}_2)x^2. \quad (20)$$

Then, we have:

- If  $\hat{m}[0] = (C_0, C_1, 0)/(C_2, 0, C_3)/(0, C_4, C_5)$ , which further lead to at least 1/3 of the coefficients in  $m$  as zero, we say this triggers a *low hamming weight case*.
- Otherwise, we say a *random hamming weight case* is triggered.

Therefore, similar to our analysis against Kyber, we need to: First enumerate  $z \in [1, q - 1]$ , measure the frequency, and select one  $z_0$  with higher frequency through the DVFS-based side channel. Next, set  $\hat{c} = (1 + x + z_0x^2, 0, \dots, 0, 1 + x + zx^2, 0, \dots, 0)$ , and enumerate  $z \in [1, q - 1]$  to further find  $z_j$  with higher frequency.

**Frequency Measurement.** We choose the AVX2-implementation of NTTRU from [LS]. In NTTRU,  $q = 7681$ , which makes the measurement on frequencies of  $z \in [1, q - 1]$  a hard task. To simplify the analysis, we select a specific  $z_0$  that triggers the low Hamming weight case and compare its frequency with frequencies corresponding to randomly selected  $z$  values from the range  $[1, q - 1]$ .

We show the frequency measurements in Figure 8 based on the two hamming weight cases. In Figure 8a, we can see that the Frequency density difference between the two cases reaches about 8%. The frequency means of low and random hamming weight cases are 2.9345 and 2.9269 GHz, respectively. In Figure 8b, we present the average time required to complete 300 concurrent threads of the process shown in Figure 8a, considering two different cases. In the low Hamming weight case, the average time is 3597.58  $\mu\text{s}$ , while in the random Hamming weight case, it is 3620.09  $\mu\text{s}$ . This results in a timing difference of 17.78  $\mu\text{s}$  between the two cases.

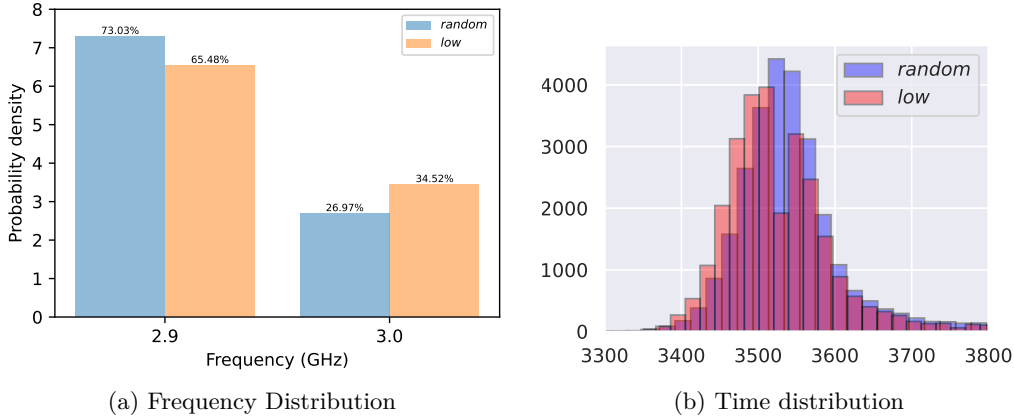


Figure 8: Distribution of CPU frequency and execution time over **CCA.Decaps** in NTTRU for the low and random hamming weight cases. In Figure 8a, we collect 800,000 frequency points running **CCA.Decaps** in NTTRU for all CPU logical cores. In Figure 8b, we operate 300 current threads **CCA.Decaps** with each thread running a single decapsulation and recording the total running time for 30,000 points.

**Security Loss.** Similarly, by performing a matrix-vector multiplication using the appropriate NTT matrix in NTTRU, we can derive the linear relationship, or “*Hints*” concerning the secret key  $f$ . We then integrate the *Hints* about  $f$  in the framework provided by [DSDGR20] and estimate the security loss.

Table 2: Estimation of Security Loss against NTTRU

|       | #Hints | Original Security | with hints | Security Loss |
|-------|--------|-------------------|------------|---------------|
| NTTRU | 256    | 154               | 114        | 40 (-35.08%)  |

We estimate the hardness of NTTRU by regarding it as an LWE instance  $(f, \hat{f}\hat{h} - \hat{3}g)$ . Then we initiate the LWE instance with  $n = 768$  and  $q = 7681$  and treat our *Hints* as the *Modular Hint* in the [DSDGR20] framework. With one hint in each Basemul block, in total, we obtain  $n/3 = 256$  hints. The estimation result is summarized in Table 2. With 256 hints, the estimated bit security drops from 154 bit to 114 bit, with a security loss of 35.08%.

### 5.3 Countermeasures

To defend against the frequency side-channel attacks presented in this paper, we recommend two different approaches. The first is to cut off the frequency channel as discussed by Wang et al. [WPH<sup>+</sup>22]. Disable Turbo Boost from the BIOS can fix the CPU frequency to the base CPU frequency. In such a way, any data-dependent power consumption of a given cryptography implementation will not be observable via the frequency side-channel. However, this approach reduces the overall system performance.

The second approach is to examine the application of NTT in a cryptography implementation and ensure that potential security risks associated with malicious user inputs are mitigated. Akin to Kyber’s approach, a cryptography implementation utilizing NTT can reduce its input space, ensuring that any malicious user input cannot trigger the abnormal low-hamming weight computation as discussed in Section 3.

## 6 Conclusion

In this paper, we have highlighted the importance of addressing leakages in DVFS-based side channels when using NTT in lattice-based KEMs. Through our analysis and experiments, we have provided substantial evidence of leakages in pure NTT and NTT-based lattice-based KEMs such as Kyber and NTTRU. We have identified that the distinct Hamming weights of the polynomial inputs processed by `invNTT` result in input-dependent CPU power consumption or frequency under DVFS. It is important to emphasize that our analysis only works against a simplified CPA version of Kyber, which removes Compression and Decompression functions. So, whether this kind of attack applies to the complete CPA version and CCA-secure Kyber is challenging work.

## References

- [ABD<sup>+</sup>a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. pq-crystals/kyber avx2 implementation. <https://github.com/pq-crystals/kyber/tree/main/avx2>. Accessed: 2023-05-06.
- [ABD<sup>+</sup>b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. pq-crystals/kyber reference implementation. <https://github.com/pq-crystals/kyber/tree/main/ref>. Accessed: 2023-05-06.
- [ABD<sup>+</sup>19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: Algorithm specification and supporting documentation (version 2.0). In *Submission to the NIST post-quantum project (2019)*, 2019. <https://pq-crystals.org/kyber>.
- [ACC<sup>+</sup>20] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, et al. Polynomial multiplication in ntru prime: Comparison of optimization strategies on cortex-m4. *Cryptology ePrint Archive*, 2020.
- [ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange—a new hope. In *USENIX security symposium*, volume 2016, 2016.
- [Ber01] Daniel J Bernstein. Multidigit multiplication for mathematicians. *Advances in Applied Mathematics*, pages 1–19, 2001.
- [CHK<sup>+</sup>21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. Ntt multiplication for ntt-unfriendly rings: New speed records for saber and ntru on cortex-m4 and avx2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, Feb. 2021.
- [DG22] Debopriya Roy Dipta and Berk Gulmezoglu. Df-sca: Dynamic frequency side channel attacks are practical. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 841–853, 2022.
- [DKL<sup>+</sup>18] 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.
- [DÖSS15] Yarkin Doröz, Erdinç Öztürk, Erkay Savaş, and Berk Sunar. Accelerating ltv based homomorphic encryption in reconfigurable hardware. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 185–204, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [DSDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In *Annual International Cryptology Conference*, pages 329–358. Springer, 2020.



- [FHK<sup>+</sup>18] 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), 2018.
- [FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.
- [Int20] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>, 2020. Accessed: 2023-03-15.
- [Int22] Intel. Thermal Design Power (TDP) in Intel® Processors. <https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html>, 2022. Accessed: 2023-03-15.
- [JAC<sup>+</sup>20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2020.
- [LCCR22] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1977–1991, 2022.
- [LS] Vadim Lyubashevsky and Gregor Seiler. NTTRU Avx2 Implementation. <https://github.com/gregorseiler/NTTRU/tree/master/avx2>. Accessed: 2023-07-06.
- [LS19] Vadim Lyubashevsky and Gregor Seiler. Ntru: Truly fast ntru using ntt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 180–201, 2019.
- [MN23] Alexander May and Julian Nowakowski. Too Many Hints-When LLL Breaks LWE. In *Advances in Cryptology-ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2023.
- [PBPV23] Antoon Purnal, Marton Bogнар, Frank Piessens, and Ingrid Verbauwhede. Showtime: Amplifying arbitrary cpu timing side channels. In *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*, 2023.
- [PS22] Rogério Paludo and Leonel Sousa. Ntt architecture for a linux-ready risc-v fully-homomorphic encryption accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2669–2682, 2022.
- [QCZ<sup>+</sup>21] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based nist candidate kems. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 92–121. Springer, 2021.

- [RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020.
- [RRD<sup>+</sup>23] Gokulnath Rajendran, Ravi Ravi, Jan-Pieter D’Anvers, Shivam Bhasin, and Anupam Chattopadhyay. Pushing the limits of generic side-channel attacks on lwe-based kems-parallel pc oracle attacks on kyber kem and beyond. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):418–446, 2023.
- [SCZ<sup>+</sup>23] Muyan Shen, Chi Cheng, Xiaohan Zhang, Qian Guo, and Tao Jiang. Find the bad apples: An efficient method for perfect key recovery under imperfect sca oracles—a case study of kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 89–112, 2023.
- [TUX<sup>+</sup>23] Yutaro Tanaka, Rei Ueno, Keita Xagawa, Akira Ito, Junko Takahashi, and Naofumi Homma. Multiple-valued plaintext-checking side-channel attacks on post-quantum kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 473–503, 2023.
- [UXT<sup>+</sup>22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/em analysis on post-quantum kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 296–322, 2022.
- [WPH<sup>+</sup>22] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning power Side-Channel attacks into remote timing attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 679–697, Boston, MA, August 2022. USENIX Association.
- [Wys] Rafael J. Wysocki. intel\_pstate CPU performance scaling driver. [https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel\\_pstate.html](https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel_pstate.html). Accessed on Jun 7, 2022.