IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 3, pp. 200–223. DOI:10.46586/tches.v2024.i3.200-223

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

Tianrun Yu[1,2], Chi Cheng[1,2]✉, Zilong Yang[1,2], Yingchen Wang[3], Yanbin Pan[4] and Jian Weng[5]

[1] Hubei Key Laboratory of Intelligent Geo-Information Processing, School of Computer Science, China University of Geosciences, Wuhan, China {chengchi,yutianrun}@cug.edu.cn,
[2] State Key Laboratory of Integrated Services Networks, Xidian University, Xian, China
[3] The University of Texas at Austin, Austin, TX, USA yingchen@cs.utexas.edu
[4] Key Laboratory of Mathematics Mechanization, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China panyanbin@amss.ac.cn
[5] College of Information Science and Technology, Jinan University, Guangzhou, China

Abstract. Number Theoretic Transform (NTT) has been widely used in accelerating computations in lattice-based cryptography. However, attackers can potentially launch power analysis targeting the NTT because it is one of the most time-consuming parts 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 alternative 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 wolfSSL [Wol] and [Ope22].

Received: 2024-01-15 Accepted: 2024-03-15 Published: 2024-07-18

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+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 Seiler in their design called NTTRU [LS19], NTT can still be used by choosing different underlying rings. Another elegant example is given in [CHK+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 cryptographic schemes in real-world applications 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+22]. By measuring the precise timings of fluctuations in the power supply, a Hertzbleed attack is able to deduce secret keys from a now-broken cryptosystem Supersingular Isogeny Key Encapsulation (SIKE) [JAC+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 a computation-intensive component, e.g., 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.
- We have open-sourced our code of all the experiments of this paper at https://github.com/Yutianrun/Hint_from_Hertz.

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[1]. 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+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 $\texttt{input}_1$ compared to $\texttt{input}_2$, during high CPU workload, DVFS would schedule the CPU to oscillate between lower P-States when running `func` with $\texttt{input}_1$. As a result, `func` executes slower with $\texttt{input}_1$. If $\texttt{input}_1$ and $\texttt{input}_2$ 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. Speficially, we use the `MSR_IA32_MPERF` and `MSR_IA32_APERF` registers in the Linux kernel to monitor CPU frequency.

[1] 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 $\mathcal{R}_q$ as a polynomial $\mathbf{f}(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$, where each coefficient $f_i \in \mathbb{Z}_q$ $(0 \le i \le n-1)$. Hereafter, we use both $\mathbf{f} \in \mathcal{R}_q$ and its vector form $(\mathbf{f}_0, \cdots, \mathbf{f}_{n-1})$ interchangeably to refer to a polynomial. The function $\lceil x \rfloor$ 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 $I_1, I_2, \cdots, I_m$ are its pair-wisely co-prime ideals. Set I the intersection of $I_1, I_2, \cdots, I_m$, then there exists a ring isomorphism Φ as follows:*

$$\Phi : R/I \cong R/I_1 \times R/I_2 \times \cdots \times R/I_m. \tag{1}$$

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:

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

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 $\hat{f} = \mathsf{NTT}(f)$ the forward NTT transform for polynomial f and by $f = \mathsf{invNTT}(\hat{f})$ 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 ζ of unity in R_q. This leads to a full factorization of $\phi(x)$, which means each $\phi_i(x)$ in (2) becomes a degree-one polynomial.

Now, we can define the forward NTT transform $\hat{f} = \mathsf{NTT}(f)$ as $\hat{f}_i = \sum_{j=0}^{n-1} \zeta^{(2i+1)*j} f_j \bmod q$. For the inverse transform $f = \mathsf{invNTT}(\hat{f})$, 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+18], and Dilithium [DKL+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+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 does not have

a 512-th primitive root of unity, only a 256-th root of unity in R_q. To implement the incomplete NTT, Kyber utilizes the following isomorphism:

$$\mathbb{Z}_{3329}[x]/(x^{256}+1) \cong \prod_{i=0}^{127} \left(\mathbb{Z}_{3329}[x]/(x^2-\zeta^{2i+1})\right). \tag{3}$$

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

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

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,

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

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

Algorithm 1 Inverse NTT based on the GS butterfly.

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

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**
4: **GS-Butterfly unit**
5: $j = j + 1$
6: **end for**
7: $i = j + len$
8: **end for**
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**

```
/*  Kyber:  d=2; NTTRU: d=3  */
```

▷ **GS-Butterfly unit**

1: $t = \hat{r}[j]$
2: $\hat{r}[j] = \hat{r}[j] \ + \ \hat{r}[j + len]$
3: $\hat{r}[j + len] = \zeta^{-1} * (t - \hat{r}[j + len])$

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

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

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

Algorithm 2 Basemul between $\hat{\mathbf{f}}_\mathbf{i}$ and $\hat{\mathbf{g}}_\mathbf{i}$

Input: $\hat{\mathbf{f}}_\mathbf{i}=[\hat{\mathbf{f}}_\mathbf{i}[\mathbf{0}],\hat{\mathbf{f}}_\mathbf{i}[\mathbf{1}],\cdots,\hat{\mathbf{f}}_\mathbf{i}[\mathbf{d-1}]]$ and $\hat{\mathbf{g}}_\mathbf{i}=[\hat{\mathbf{g}}_\mathbf{i}[\mathbf{0}],\hat{\mathbf{g}}_\mathbf{i}[\mathbf{1}],\cdots,\hat{\mathbf{g}}_\mathbf{i}[\mathbf{d-1}]]$, $\omega_\mathbf{i}$
Output: $\hat{\mathbf{r}}_\mathbf{i}=[\hat{\mathbf{r}}_\mathbf{i}[\mathbf{0}],\hat{\mathbf{r}}_\mathbf{i}[\mathbf{1}],\cdots,\hat{\mathbf{r}}_\mathbf{i}[\mathbf{d-1}]]$, the i-th component of $\hat{\mathbf{r}}$.
1: **for** $j=0\rightarrow d-1$ **do**
2: $\quad\hat{\mathbf{r}}_\mathbf{i}[\mathbf{j}]=(\sum_{\mathbf{u=0}}^{\mathbf{j}}\hat{\mathbf{f}}_\mathbf{i}[\mathbf{u}]\hat{\mathbf{g}}_\mathbf{i}[\mathbf{j-u}]+\sum_{\mathbf{u=j+1}}^{\mathbf{d-1}}\hat{\mathbf{f}}_\mathbf{i}[\mathbf{u}]\hat{\mathbf{g}}_\mathbf{i}[\mathbf{j+d-u}]\omega_\mathbf{i})\bmod\mathbf{q}$
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 $\hat{\mathbf{f}}_\mathbf{i}$ and $\hat{\mathbf{g}}_\mathbf{i}$, 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 NTTRU, 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: $fg=\mathsf{invNTT}(\hat{f}\circ\hat{g})=\mathsf{invNTT}(\mathsf{NTT}(f)\circ\mathsf{NTT}(g))$.

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 $\hat{s}=\mathsf{NTT}(s)$ 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: $\hat{f}=(\hat{f}_0,\hat{f}_1,0,\ldots,0)$. 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 output of 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 invNTT are non-zero. As a result, both the output of invNTT and the intermediate layer result involved during its execution have random Hamming weight. To summarize, 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+22]. Specifically, when the input $\hat{f} \circ \hat{s}$ has only one non-zero coefficient, it would trigger 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 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].

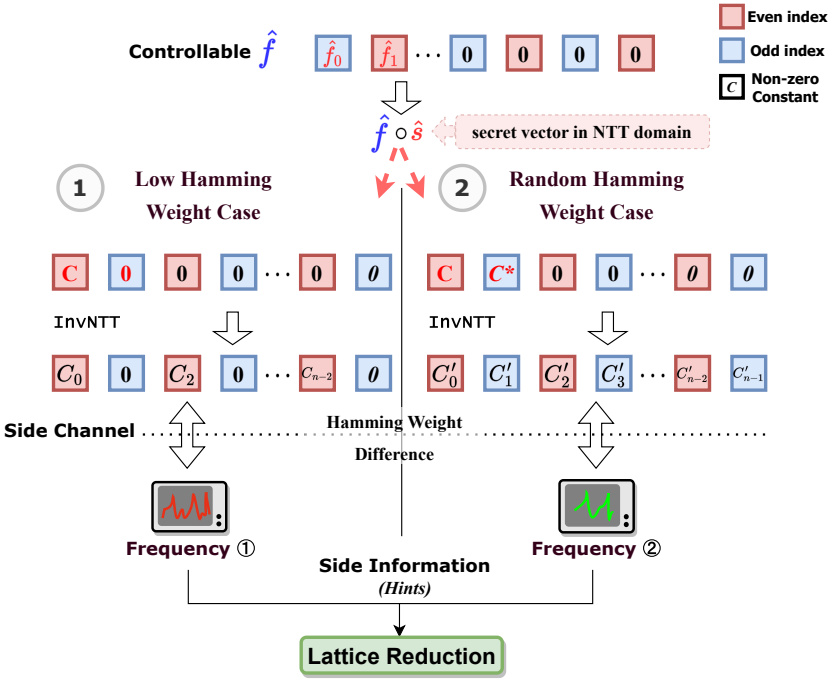


Figure 1: Basic attack idea

3.2 Frequency and power measurements of invNTT

In this subsection, we present the frequency and power measurements of invNTT processing data with low and random hamming weights under the DVFS side-channel. We show that 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 GHz Turbo boost frequency. Our machine runs Ubuntu 20.04 with Linux kernel 5.15.

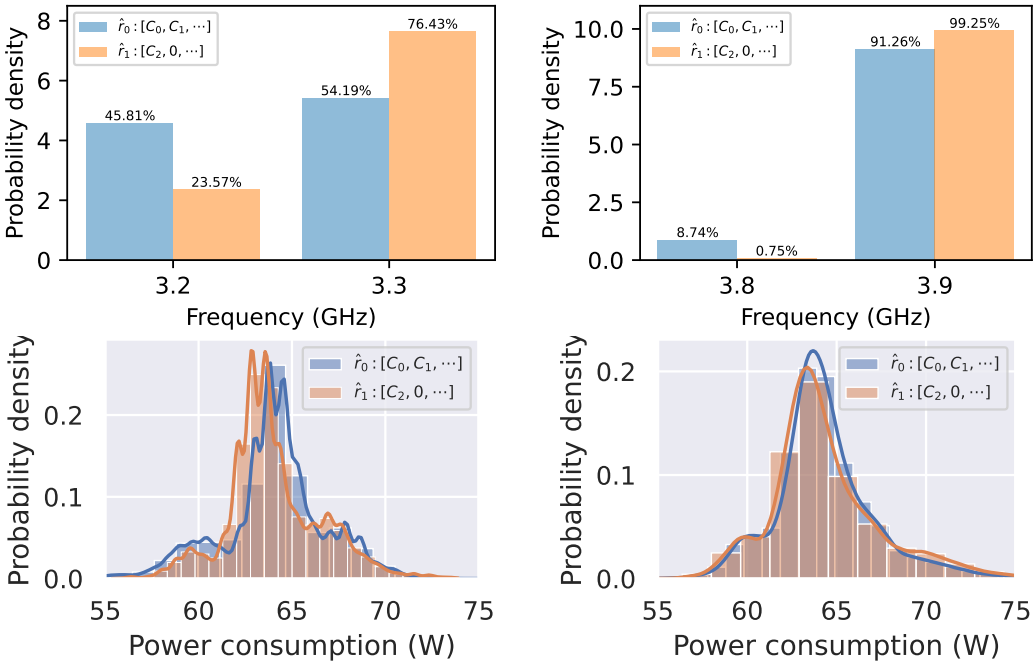
To measure the CPU frequency and power consumption, we follow the same methodology as Wang et al. [WPH+22]. We use the default system setup for CPU frequency measurement.

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+b] and AVX2 implementation [ABD+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 $\hat{r}_0 = [C_0, C_1, 0, \cdots, 0]$ and $\hat{r}_1 = [C_2, 0, 0, \cdots, 0]$. We randomly sample C_0, C_1, C_2 from $[1, q-1]$.



(a) AVX2 implementation **(b)** Reference implementation

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).

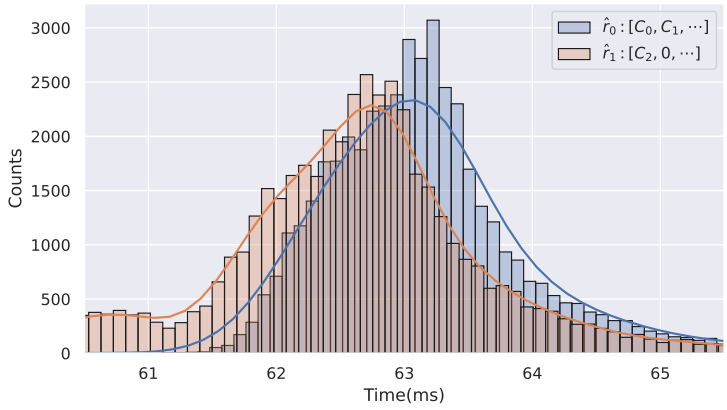


Figure 3: Histograms of timing measurements of 300 concurrent threads with each running a For loops of 10,000 iterations of invNTT followed by NTT. The target implementation is the AVX2 NTT implementation. The inputs to invNTT are 10 pairs of randomly generated $\hat{r}_0$ and $\hat{r}_1$. We collect 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**

◇ **CPAKEM.Keygen**

Output: Public Key pk, secret key sk.

1: $\rho, \sigma \leftarrow \{0,1\}^{256}$
2: $\hat{\mathbf{A}} \xleftarrow{\rho} \mathcal{R}_q^{\alpha\times\alpha}$, $\mathbf{s}, \mathbf{e} \xleftarrow{\sigma} \mathcal{B}_{\eta_1}^{\alpha}$
3: $\hat{\mathbf{s}} = \mathsf{NTT}(\mathbf{s})$, $\hat{\mathbf{e}} = \mathsf{NTT}(\mathbf{e})$
4: $\hat{\mathbf{p}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
return $(pk = (\hat{\mathbf{p}}, \rho), sk = \hat{\mathbf{s}})$

◇ **CPA.Encryption**

Input: Public Key $pk = (\hat{\mathbf{p}}, \rho)$
Output: $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$

1: $m, r \leftarrow \{0,1\}^{256}$, $\hat{\mathbf{A}} \xleftarrow{\rho} \mathcal{R}_q^{k\times k}$
2: $\mathbf{r} \xleftarrow{r} \mathcal{B}_{\eta_1}^{\alpha}$, $\mathbf{e}_1, \mathbf{e}_2 \xleftarrow{r} \mathcal{B}_{\eta_2}$, $\hat{\mathbf{r}} = \mathsf{NTT}(\mathbf{r})$
3: $\mathbf{u} = \mathsf{InvNTT}(\hat{\mathbf{A}}^{\mathbf{T}} \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
4: $v = \mathsf{InvNTT}(\hat{\mathbf{p}}^{\mathbf{T}} \circ \hat{\mathbf{r}}) + \mathbf{e}_2 + m$
5: $\mathbf{c}_1 = \mathbf{u}$, $\mathbf{c}_2 = v$
return $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$

◇ **CPA.Decryption**

Input: $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$, secret key $\hat{\mathbf{s}}$
Output: Message m'

1: $\mathbf{u}' = \mathbf{c}_1$,$v' = \mathbf{c}_2$
2: $\hat{\mathbf{u}}' = \mathsf{NTT}(\mathbf{u}')$
3: $\hat{\mathbf{r}} = \hat{\mathbf{s}} \circ \hat{\mathbf{u}}'$
4: $m' = v' - \mathsf{invNTT}(\hat{\mathbf{r}})$
return m'

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, where we use $\mathbf{x} \xleftarrow{\rho} \mathcal{D}$ to represent generating a random sample $\mathbf{x}$ according to distribution $\mathcal{D}$ with seed ρ and $\mathbf{T}$ to represent the transpose of a matrix or a vector.

In **Keygen**, $\hat{\mathbf{p}}$ 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 η.

In the **Encryption** process of Kyber, a message 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 v using

the input $\hat{\mathbf{p}}$. Eventually, the **Encryption** returns the ciphertext $\mathbf{ct} = (\mathbf{u}, v)$. In the **Decryption**, Kyber unpacks the input ciphertext, $\mathbf{ct}$, as $(\mathbf{u}', v')$, then it calculates $m' = v' - \mathsf{invNTT}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}})$ and returns m' as the decrypted message.

4.2 The analysis against CPA-secure Kyber

There are three parameter sets 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.

Algorithm 4 Analysis against Kyber

◇ **DVFS-based Side-Channel Oracle** $\mathcal{O}$
Input: Polynomial $\mathbf{u}$
Output: FrequencyHints
1: Query **CPA.Decryption** with $\mathbf{c}_1 = \mathbf{u}, \mathbf{c}_2 = \mathbf{0}$.
2: **CPA.Decryption** computes invNTT $(\mathsf{NTT}(\mathbf{u}) \circ \mathsf{NTT}(\mathbf{s}))$, with $\mathbf{s}$ being the secret key.
3: The Oracle $\mathcal{O}$ monitors the CPU frequency during **CPA.Decryption** execution and stores the CPU frequency as FrequencyHints.
return FrequencyHints

◇ **Our analysis**
Output: *Hints* about $\mathbf{s}_0$
1: **for** z in $[1, q-1]$ **do**
2: Choose $\mathbf{u}$ with $\hat{\mathbf{u}}'_{\mathbf{0}} = (1, z, 0, \cdots, 0)$.
3: FrequencyHints$= \mathcal{O}(u')$
4: Append the mean of FrequencyHints to MeasuredMeans.
5: **end for**
6: Select one largest means from the MeasuredMeans, and find the corresponding z_0, which must satisfy $\hat{\mathbf{s}}_{\mathbf{0,0}} + \hat{\mathbf{s}}_{\mathbf{0,1}} z_0 \zeta = 0$ or $\hat{\mathbf{s}}_{\mathbf{0,1}} + \hat{\mathbf{s}}_{\mathbf{0,0}} z_0 = 0$.
7: **for** i in $[1, n/2-1]$ **do**
8: Repeat process 1 to 6 to obtain z_i except that for process 2 choose $\mathbf{u}$ with $\hat{\mathbf{u}}'_{\mathbf{0}} = (1, z_0, 0, \cdots, 0, 1, z, 0 \cdots, 0)$ with the additional 1 and z at index $2i$ and $2i+1$ respectively.
9: **end for**
10: **return** *Hints*$=\{z_0, z_1, \cdots, z_{n/2-1}\}$.

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

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

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

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

or

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

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

Remember that the attacker targets the first polynomial $\mathbf{s_0}$. He sets the input to **CPA.Decryption** as $\mathbf{c}_1 = \mathbf{u}' = (\mathbf{u}'_\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0})$ 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 $\hat{\mathbf{r}} = (\hat{\mathbf{s}}_\mathbf{0} \circ \hat{\mathbf{u}}'_\mathbf{0}, 0, \cdots, 0)$.

The attacker next sets $\mathbf{u}'_\mathbf{0}$ such that $\hat{\mathbf{u}}'_\mathbf{0}$ is $(1, z, 0, 0, \cdots, 0)$ 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 $\hat{\mathbf{u}}'_\mathbf{0}$ only has the first two elements as non-zero, $\hat{\mathbf{s}}_0 \circ \hat{\mathbf{u}}'_0$ essentially involves with only one Basemul:

$$
\begin{aligned}
(\hat{\mathbf{r}}_{\mathbf{0},0} + \hat{\mathbf{r}}_{\mathbf{0},1}x) &= (\hat{\mathbf{s}}_{\mathbf{0},0} + \hat{\mathbf{s}}_{\mathbf{0},1}x) \cdot (1 + zx) \bmod (x^2 - \zeta) && (8)\\
&= (\hat{\mathbf{s}}_{\mathbf{0},0} + (\hat{\mathbf{s}}_{\mathbf{0},1} + \hat{\mathbf{s}}_{\mathbf{0},0}z)x + \hat{\mathbf{s}}_{\mathbf{0},1}zx^2) \bmod (x^2 - \zeta) && (9)\\
&= (\hat{\mathbf{s}}_{\mathbf{0},0} + \hat{\mathbf{s}}_{\mathbf{0},1}z\zeta) + (\hat{\mathbf{s}}_{\mathbf{0},1} + \hat{\mathbf{s}}_{\mathbf{0},0}z)x. && (10)
\end{aligned}
$$

Hence we have:

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

CPA.Decryption then computes invNTT on $\hat{\mathbf{r}}$. If $\hat{\mathbf{r}}$ contains only one non-zero coefficient ($\hat{\mathbf{s}}_{\mathbf{0},0} + \hat{\mathbf{s}}_{\mathbf{0},1}z\zeta = 0$ or $\hat{\mathbf{s}}_{\mathbf{0},1} + \hat{\mathbf{s}}_{\mathbf{0},0}z = 0$), **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 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, $-(\hat{\mathbf{s}}_{\mathbf{0},1}\zeta)^{-1}\hat{\mathbf{s}}_{\mathbf{0},0}$ and $-\hat{\mathbf{s}}_{\mathbf{0},0}^{-1}\hat{\mathbf{s}}_{\mathbf{0},1}$, 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 $-(\hat{\mathbf{s}}_{\mathbf{0},1}\zeta)^{-1}\hat{\mathbf{s}}_{\mathbf{0},0}$ or $-\hat{\mathbf{s}}_{\mathbf{0},0}^{-1}\hat{\mathbf{s}}_{\mathbf{0},1}$. Or equivalently, it holds either

$$
\hat{\mathbf{s}}_{\mathbf{0},0} + \hat{\mathbf{s}}_{\mathbf{0},1}z_0\zeta = 0 \mod q, \tag{12}
$$

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

$$
\hat{\mathbf{s}}_{\mathbf{0},1} + \hat{\mathbf{s}}_{\mathbf{0},0}z_0 = 0 \mod q, \tag{13}
$$

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

We explain this observation in Lemma 1.

Lemma 1. *In the calculation of* $\mathbf{r} = \mathsf{invNTT}(\hat{\mathbf{r}}) = \mathsf{invNTT}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}}')$ *in Kyber with Algorithm 1, if* $\hat{\mathbf{r}} = (C, 0, \ldots, 0)$ *or* $(0, C, \cdots, 0, 0)$*, 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* $1 \le m \le k-1$*, 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* $C_0 + C_2x^2 + C_4x^4 + \cdots + C_{n-2}x^{n-2}$ *or* $x(C_0 + C_2x^2 + C_4x^4 + \cdots + C_{n-2}x^{n-2})$*.*

If $\hat{\mathbf{r}} = (C + C'x, 0, \ldots, 0)$*, 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 $\hat{r}[j + len]$ will only used to update the new $\hat{r}[j]$ and $\hat{r}[j + len]$ and conversely the new $\hat{r}[j]$ and $\hat{r}[j + len]$ will be determined by the current $\hat{r}[j]$ and $\hat{r}[j + len]$

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.

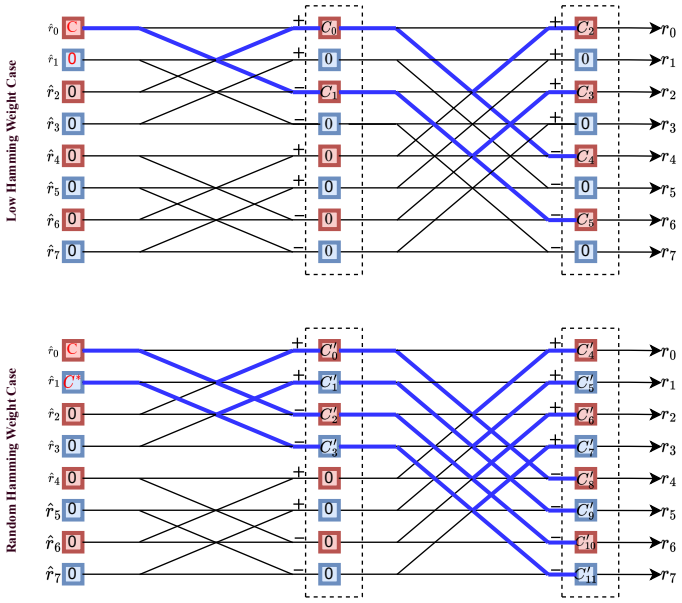


Figure 4: An example for Lemma 1 (A Simplified version with 8 coefficients.)

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

A direct attempt to find z_1 is to set $\mathbf{u}'_0$ such that $\hat{\mathbf{u}}'_0$ is $(0, 0, 1, z, \cdots, 0)$ 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 $z_0, z_1, z_2, \cdots, z_{n/2-1}$, 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 $\hat{\mathbf{s}}_\mathbf{0}$ and $\hat{\mathbf{u}}'_\mathbf{0}$ is $(C_0, 0)$, that is, $\hat{\mathbf{s}}_{\mathbf{0,1}} + \hat{\mathbf{s}}_{\mathbf{0,0}} z_0 = 0$. When we target block i, we set $\mathbf{u}'_0$ such that $\hat{\mathbf{u}}'_0$ is $(1, z_0, 0, 0, \cdots, 1, z, \cdots, 0)$ with the additional 1 and z at index $2i$ and $2i+1$ respectively. Similar to Equation (11), we have that

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

The $\hat{\mathbf{s}}_\mathbf{0} \circ \hat{\mathbf{u}}'_\mathbf{0}$ would trigger the oracle $\mathcal{O}$ to return high FrequencyHints if and only if the Basemul between the i-th block of $\hat{\mathbf{s}}_\mathbf{0}$ and $\hat{\mathbf{u}}'_\mathbf{0}$ 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* $\mathbf{r} = \mathsf{invNTT}(\hat{\mathbf{r}}) = \mathsf{invNTT}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}}')$ *in Kyber with Algorithm 1, if* $\hat{\mathbf{r}} = (C_0, 0, \ldots, 0, C_{2i}, 0, \ldots, 0)$ *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* $\hat{\mathbf{r}} = (C_0, 0, \ldots, 0, C_{2i}, C_{2i+1}, 0, \ldots, 0)$ *or* $\hat{\mathbf{r}} = (C_0, 0, \ldots, 0, C_{2i+1}, 0, \ldots, 0)$ *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 $\hat{\mathbf{r}} = (0, C_1, 0, \ldots, 0, C_{2i+1}, 0, \ldots, 0)$ *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* $\hat{\mathbf{r}} = (0, C_1, 0, \ldots, 0, C_{2i}, 0, \ldots, 0) = (0, C_1, \ldots, 0, C_{2i}, C_{2i+1}, 0, \ldots, 0)$ *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 $-\hat{\mathbf{s}}^{-1}_{\mathbf{0,2i}}\hat{\mathbf{s}}_{\mathbf{0,2i+1}}$. 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 $\{z_0, z_1, \cdots, z_{n/2-1}\}$ which yields the linear relationships of the secret key for every block i:

$$\hat{\mathbf{s}}_{\mathbf{0,2i+1}} + \hat{\mathbf{s}}_{\mathbf{0,2i}} z_i = 0 \mod q. \tag{15}$$

In the case that the Basemul between the first block of $\hat{\mathbf{s}}_{\mathbf{0}}$ and $\hat{\mathbf{u}}'_{\mathbf{0}}$ 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 $\{z_0, z_1, \cdots, z_{n/2-1}\}$ returned by the attack yields the linear relationships of the secret key for every block i:

$$\hat{\mathbf{s}}_{\mathbf{0,2i}} + \hat{\mathbf{s}}_{\mathbf{0,2i+1}} z_i \zeta = 0 \mod q. \tag{16}$$

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+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 $\mathbf{ct} = (\mathbf{u}, v)$ 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 $\mathbf{u}' = (\mathbf{u}'_{\mathbf{0}}, \mathbf{0}, \mathbf{0}, \mathbf{0})$ with the NTT representation of $\mathbf{u}'_{\mathbf{0}}$ as $\hat{\mathbf{u}}'_{\mathbf{0}} = (1, z_0, 0, 0, \cdots, 0)$. 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 $\mathcal{O}$.

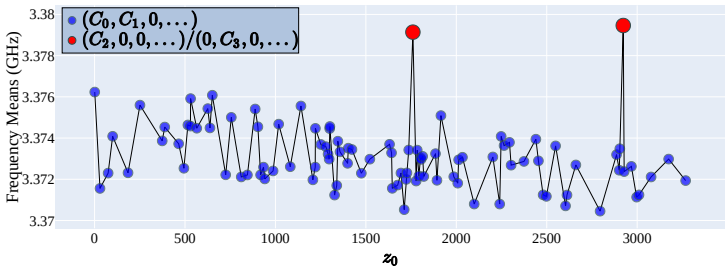


Figure 5: Frequency means of the AVX2 Kyber **CPA.Decryption**. We fix one randomly generated sk and sampled 100 z_0 in total. We sample 98 random z_0 that trigger **CPA.Decryption** processing data with random Hamming weight ($\hat{\mathbf{r}}$ to be the form of $(C_0, C_1, 0, \cdots, 0)$) and compute two special z_0 that trigger **CPA.Decryption** processing data with low Hamming weight ($\hat{\mathbf{r}}$ to be the form of $(C_2, 0, 0, \cdots, 0)$ or $(0, C_3, 0, \cdots, 0)$). 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.

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 $-(\hat{\mathbf{s}}_{\mathbf{0},\mathbf{1}}\zeta)^{-1}\hat{\mathbf{s}}_{\mathbf{0},\mathbf{0}}$ and $-\hat{\mathbf{s}}_{\mathbf{0},\mathbf{0}}^{-1}\hat{\mathbf{s}}_{\mathbf{0},\mathbf{1}}$. They trigger $\hat{\mathbf{r}} = \hat{\mathbf{s}}_{\mathbf{0}} \circ \hat{\mathbf{u}}'_{\mathbf{0}} = (C_2, 0, 0, \cdots, 0)$ or $(0, C_3, 0, \cdots, 0)$. As a result, when computing $\mathsf{invNTT}(\hat{\mathbf{r}}$, **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.

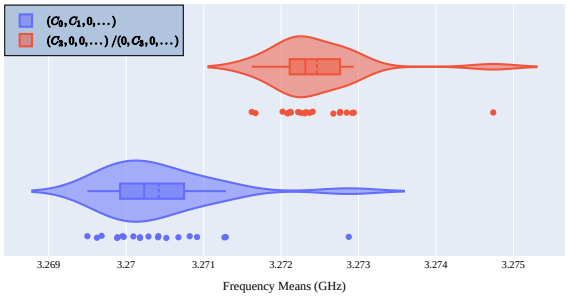


Figure 6: Distribution of measured CPU frequency means with 10 different secret keys. For each secret key, we choose two z_0 that trigger **CPA.Decryption** processing data with lower Hamming weight ($\hat{\mathbf{r}}$ to be the form of $(C_2, 0, 0, \cdots, 0)$ or $(0, C_3, 0, \cdots, 0)$) and two z_0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

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

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 $\hat{\mathbf{u}}'_{\mathbf{0}}$ is $(1, z_0, 0, 0, \cdots)$. $\hat{\mathbf{r}} = \hat{\mathbf{s}}_{\mathbf{0}} \circ \hat{\mathbf{u}}'_{\mathbf{0}}$ can be either $(C_0, 0, 0, \cdots, 0)$ or $(0, C'_0, 0, \cdots, 0)$. When targeting the i-th NTT block of $\hat{s}_0$, we set $\hat{\mathbf{u}}'_{\mathbf{0}}$ as $(1, z_0, 0, 0, \cdots, 1, z_i, 0, \cdots)$ 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 $\hat{s}_0$ 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 $\hat{\mathbf{u}}'_{\mathbf{0}}$ as z_i or z'_i. The Basemul between $(1, z_i)$ and the i-th block of $\hat{s}_0$ results in $(C_i, 0)$ $(z_i = -\hat{\mathbf{s}}_{\mathbf{i},\mathbf{0}}^{-1}\hat{\mathbf{s}}_{\mathbf{i},\mathbf{1}})$ and the Basemul between $(1, z'_i)$ and the i-th block of $\hat{s}_0$ results in $(0, C'_i)(z_i = -(\hat{\mathbf{s}}_{\mathbf{i},\mathbf{1}}\zeta)^{-1}\hat{\mathbf{s}}_{\mathbf{i},\mathbf{0}})$. When $\hat{\mathbf{u}}'_{\mathbf{0}}$ is set with z_i, we trigger the i-th block of $\hat{\mathbf{r}}$ to be $(C_i, 0)$. **CPA.Decryption** computing $\mathsf{invNTT}(\hat{\mathbf{r}})$ with $\hat{r} = (C_0, 0, \cdots C_i, 0, \cdots)$ would process data with lower Hamming weight and the CPU runs faster. On the other hand, When $\hat{\mathbf{u}}'_{\mathbf{0}}$ is set with z'_i, we trigger the i-th block of $\hat{\mathbf{r}}$ to be $(0, C'_i)$. **CPA.Decryption**

computing $\mathsf{invNTT}(\hat{\mathbf{r}})$ with $\hat{r} = (C_0, 0, \cdots 0, C_i', \cdots)$ 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 $(C_0, 0, \cdots C_i, 0, \cdots, 0)$ are distinctively higher than the CPU frequency means of the random Hamming weight case caused by $(C_0, 0, \cdots 0, C_i', \cdots, 0)$. The experimental result confirms that we can observe the Hamming weight difference stated in Lemma 2 through the DVFS-based oracle $\mathcal{O}$.

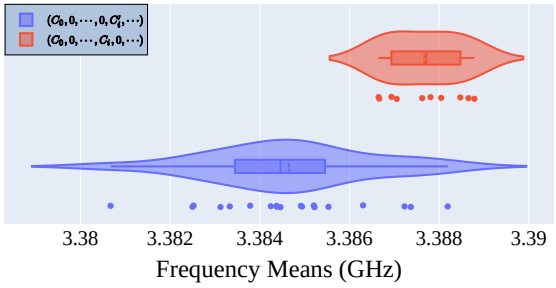


Figure 7: Distribution of measured CPU frequency means targeting 10 random blocks of the secret key. For each block i, we configure $\hat{\mathbf{u}}_0'$ with z_i and z_i' such that z_i triggers **CPA.Decryption** to compute on $(C_0, 0, \cdots C_i, 0, \cdots, 0)$ and z_i' triggers **CPA.Decryption**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

4.3.3 Comparison of number of queries

There has been a long line of research in attacks 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 that are against the full version of Kyber. 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 enable 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 $\hat{\mathbf{s}}_{\mathbf{0}}$ for example, after getting the corresponding hints $\{z_0, z_1, \cdots, z_{n/2-1}\}$, we know that either Equation 15 or Equation 16 holds. Without loss of generality, we assume that Equation 15 holds, that is, for $i = 0, 1, \cdots, n/2 - 1$, we have

$$\hat{\mathbf{s}}_{\mathbf{0},\mathbf{2i+1}} + \hat{\mathbf{s}}_{\mathbf{0},\mathbf{2i}} z_i = 0 \mod q. \tag{17}$$

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

$$\hat{\mathbf{s}}_{\mathbf{0}} = \mathbf{N} \cdot \mathbf{s}_0 \mod q, \tag{18}$$

which means that every $\hat{\mathbf{s}}_{\mathbf{0},\mathbf{i}}$ can be written as the linear combination of the coefficients of $\mathbf{s}_0$. Substituting $\hat{\mathbf{s}}_{\mathbf{0},\mathbf{2j+1}}$ and $\hat{\mathbf{s}}_{\mathbf{0},\mathbf{2j}}$ 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: $\langle \mathbf{v}, \mathbf{s} \rangle = 0 \bmod q$. 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 41.52%, 37.91%, and 36.30%.

Table 1: Estimation of security loss in our analysis against modified implementations of CPA-Kyber without the Compression and Decompression functions

	#Hints	original security	with hints	security loss
Kyber-512	256	118	69	49 (–41.52%)
Kyber-768	384	182	113	69 (–37.91%)
Kyber-1024	512	256	163	87 (–36.30%)

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.

4.4.3 End-to-End Attacks against a Toy Example

We demonstrate an end-to-end attack against a toy example of Kyber, namely Kyber-256, where R_q is set as $\mathbb{Z}_{3329}[x]/(x^{128}+1)$ and $k=2$. An incomplete NTT comprising 6 layers is employed. The isomorphism can be expressed as $\mathbb{Z}_{3329}[x]/(x^{128}+1) \cong \prod_{i=0}^{63}\left(\mathbb{Z}_{3329}[x]/(x^2-\zeta^{2i+1})\right)$ with $\zeta = 17^2$.

We conduct a similar experiment on our AVX2 implementation of Kyber-256 to measure the frequency leakage. The experimental results indicate a leakage pattern similar to that of Kyber1024. In total, we can extract 128 hints from these measurements. By simply regarding these hints as "mod-q hints", we employ the script provided in [MN23] and succeed in fully recovering the secret key. In the experiment, a (progressive) BKZ with block size 21 is used. More details can be found in Appendix A.

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

Algorithm 5 CCA-secure NTTRU

◇ **NTTRU.Keygen**

Output: Public Key $\hat{h}$, secret key $\hat{f}$

$f' \xleftarrow{\$} \mathcal{B}_2$
$f = 3f' + 1$
$\hat{f} = \mathsf{NTT}(f)$
$g \xleftarrow{\$} \mathcal{B}_2$
$\hat{3g} = \mathsf{NTT}(3g)$
If f is not invertible : Restart
$\hat{h} = \hat{3g} \circ \hat{f}^{-1}$
return $\hat{h}, \hat{f}$

◇ **CCAKEM.Encaps**

Input: Public Key $\hat{h}$

Output: *Ciphertext* **ct**, Shared Key K

$m \leftarrow \mathcal{R}$
$r \leftarrow Sampler(H(m))$
▷ CPA.Enc$(\hat{h}, m, r)$
  $\hat{r} = \mathsf{NTT}(r)$
  $\hat{m} = \mathsf{NTT}(m)$
  $\hat{v} = \hat{r} \circ \hat{h}$
  $\hat{c} = \hat{v} + \hat{m}$
return $\hat{c}$

◇ **CCAKEM.Decaps**

Input: Ciphertext $\hat{c}$, Secret Key $\hat{f}$
Public Key $\hat{h}$

Output: Shared Key K

▷ CPA.Dec$(\mathbf{s}, \mathbf{ct})$
  $m = \mathsf{invNTT}(\hat{m}) = \mathsf{invNTT}(\hat{c} \circ \hat{f})$
$r \leftarrow Sampler(H(m))$
$c' = \text{CPA.ENC}(\hat{h}, m, r)$
if $c' = c$ **then**
  $K = H(m))$
else
  $K = 0$
end if

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 $\hat{h}=\hat{3g}\circ\hat{f}^{-1}$. 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.

Attack Methods. To infer information about f, we set the ciphertext polynomial $\hat{c}$ as $(1+x+zx^2,0,0\cdots,0)$ 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:

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

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

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 first to enumerate $z\in[1,q-1]$, measure the frequency, and select three special z with higher frequency through the DVFS-based side channel. Next, set $\hat{c}=(1+x+zx^2,0,\cdots,0,1+x+z'x^2,0,\cdots,0)$, and enumerate $z'\in[1,q-1]$ to further find z' with higher frequency.

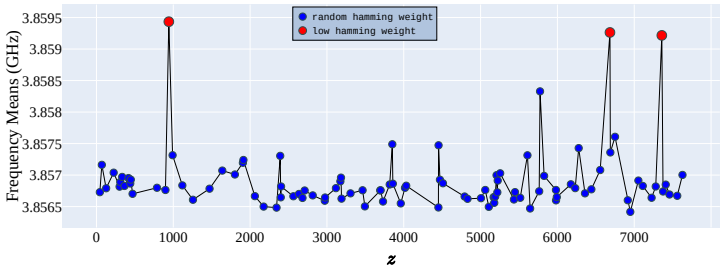


Figure 8: Frequency means of the AVX2 NTTRU **CCA.Decryption**. We fix one randomly generated sk and show 100 randomly selected z for simplicity. Specifically, we sample 97 random z that trigger **CCA.Decryption** processing data with random Hamming weight. We compute three special z that trigger **CCA.Decryption** processing data with low Hamming weight.

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 three specific z that triggers the low Hamming weight case and compare its frequency with frequencies corresponding to randomly selected z values from the range $[1, q-1]$.

In Figure 8, we present the CPU frequency means of NTTRU **CCA.Decryption** with different z as input. Similar to the analysis of Kyber, we prepared 100 different z, including three special z that trigger the low hamming weight. For each z, we collect $200,000$ frequency data points and calculate the mean of the measured frequency. we can identify three points with higher frequency (the red points). The result aligns well with our anlysis.

We show the frequency measurements in Figure 8 based on the two hamming weight cases. In Figure 8, we show 100 different z_0 including three special z_0 that trigger the data in NTTRU to return high Frequency.

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 $(f, \hat{f}\hat{h} - \hat{3g})$. 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%.

6 Countermeasures and Conclusion

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+22]. Disabling 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 cryptographic implementation and ensure that potential security risks associated with malicious user inputs are mitigated. That is, a cryptographic implementation utilizing NTT can check the input of invNTT, discarding abnormal ones with low-hamming weight as discussed in Section 3. Akin to Kyber's approach, a cryptographic implementation utilizing NTT can reduce its input space, ensuring that any malicious user cannot trigger the abnormal low-hamming weight computation as discussed in Section 3.

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, determing whether this kind of attack applies to the complete CPA version and CCA-secure Kyber requires further study.

Acknowledgment

We thank Dr. Patrick Longa for shepherding this paper. This work was partly funded by the National Natural Science Foundation of China under Grant No. 62172374. Yanbin Pan was supported in part by National Key Research and Development Project (No. 2018YFA0704705), National Natural Science Foundation of China (No. 62372445, 62032009, 12226006) and Innovation Program for Quantum Science and Technology under Grant 2021ZD0302902. Jian Weng is supported by Major Program of Guangdong Basic and Applied Research Project under Grant No. 2019B030302008, National Natural Science Foundation of China under Grant Nos. 62332007 and U22B2028, Science and Technology Major Project of Tibetan Autonomous Region of China under Grant No. XZ202201ZD0006G, Guangdong Provincial Science and Technology Project under Grant No. 2021A0505030033, National Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Key Laboratory of Data Security and Privacy Preserving, Guangdong Hong Kong Joint Laboratory for Data Security and Privacy Protection, and Engineering Research Center of Trustworthy AI, Ministry of Education.

References

[ABD+a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Kyber AVX2 implementation. https://github.com/pq-crystals/kyber/tree/main/avx2. Accessed: 2023-05-06.

[ABD+b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Kyber Reference implementation. https://github.com/pq-crystals/kyber/tree/main/ref. Accessed: 2023-05-06.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specification and supporting documentation (version 2.0). In *Submission to the NIST post-quantum project (2019)*, 2019. https://pq-crystals.org/kyber.

[ACC+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+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+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] Yarkın 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+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+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. NTTRU: 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.

[Ope22] OpenSSH 9.0. https://www.openssh.com/txt/release-9.0, 2022. Accessed: 2023-03-16.

[PBPV23] Antoon Purnal, Marton Bognar, 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+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+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+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+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+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.

[Wol] WolfSSL. Cyassl+ntru - high-performance ssl. https://www.wolfssl.com/documentation/flyers/cyassl_ntru.pdf. Accessed: 2023-03-16.

[WPH+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. Accessed on Jun 7, 2022.

A Frequency Measurement against Toy Example

Figure 9 depicts the experimental results in Section 4.4.3, where we extract hints by distinguishing between two different Hamming weight cases in frequency measurement.

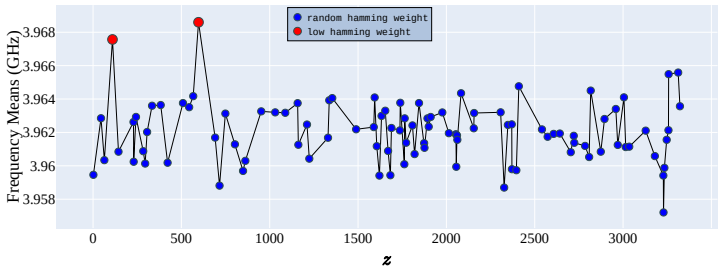


Figure 9: We fix one randomly generated sk and sample a total of 100 z. Similar to Figure 5, we sample 98 random z that trigger **CPA.Decryption** processing data with random Hamming weights and two special z that trigger **CPA.Decryption** processing data with low Hamming weights.

We adopt the same experimental setup as given in Section 3.2.1 for our i7-9700 CPU. We collect $200,000$ frequency data points and calculate the mean of the measured frequency for each z. We exhaustively enumerate all z values in $[1, 3328]$. For simplicity of illustration, we only show 100 randomly selected points. Among them, we can identify the low Hamming weight case with higher frequency (the red points). This indicates the frequency leakage in the implementation of the Toy Example, which enables us to extract hints.

Then, we run a script based on [MN23] in Sage 10.2, which constructs a 256-dimensional LWE instance. By integrating 128 hints as mod-q hints into this instance, we could observe a reduction of the 513-dimensional Distorted Bounded Distance Decoding (DBDD) problem to dimension 385. After running for approximately 2 hours, the script successfully solves the LWE instance and fully recovers the secret key with a blocksize of around 21.