A practical key-recovery attack on LWE-based key-encapsulation mechanism schemes using Rowhammer

Puja Mondal[1], Suparna Kundu[2],
Sarani Bhattacharya[3], Angshuman Karmakar[1,2], and Ingrid Verbauwhede[2]

[1] Department of Computer Science and Engineering, IIT Kanpur, India
{pujamondal,angshuman}@cse.iitk.ac.in
[2] COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
{suparna.kundu,ingrid.verbauwhede}@esat.kuleuven.be
[3] Department of Computer Science and Engineering, IIT Kharagpur, India
sarani@cse.iitkgp.ac.in

Abstract. Physical attacks are serious threats to cryptosystems deployed in the real world. In this work, we propose a microarchitectural end-to-end attack methodology on generic lattice-based post-quantum key encapsulation mechanisms to recover the long-term secret key. Our attack targets a critical component of a Fujisaki-Okamoto transform that is used in the construction of almost all lattice-based key encapsulation mechanisms. We demonstrate our attack model on practical schemes such as Kyber and Saber by using Rowhammer. We show that our attack is highly practical and imposes little preconditions on the attacker to succeed. As an additional contribution, we propose an improved version of the plaintext checking oracle, which is used by almost all physical attack strategies on lattice-based key-encapsulation mechanisms. Our improvement reduces the number of queries to the plaintext checking oracle by as much as 39% for Saber and approximately 23% for Kyber768. This can be of independent interest and can also be used to reduce the complexity of other attacks.

Keywords: Post-quantum cryptography · Key-encapsulation mechanism · micro-architecture attacks · Rowhammer · Saber · Kyber

1 Introduction

Post-quantum cryptography (PQC) refers to cryptographic protocols and algorithms designed to be secure against attacks by both classical and quantum computers. A large quantum computer can *easily* subvert the security assurance of our current widely used public-key cryptographic (PKC) schemes based on integer factorization [52] and elliptic curve cryptography [40] using Shor's [55] and Proos-Zalka's [46] algorithm respectively. Therefore, it is imperative that we replace our existing PKC cryptographic with PQC schemes. However, the transition to post-quantum cryptography is a complex process that involves careful evaluation, standardization, and implementation of new cryptographic algorithms. A watershed moment in this process is the recently concluded standardization procedure by the National Institute of Standards and Technology (NIST) [1]. NIST proposed the key-encapsulation mechanism

(KEM) Kyber [14] and digital signature schemes Dilithium [22], Falcon [25] and SPHINCS+ [6] as PQC standards.

Nevertheless, a pivotal step before a cryptosystem can be deployed for widespread public use is the assessment of its physical security. It is not a rare instance when the security of a mathematically secure cryptosystem is completely compromised by physical attacks [4,12,7]. During the standardization process, NIST also highlighted resilience against physical attacks as one of the important criteria in the selection of standards. For physical security assessments, usually, two primary types of attacks are considered. First, passive side-channel attacks (SCA), that work by exploiting flaws in the implementation and using leakage of secret information through physical channels such as power consumption, electromagnetic radiation, acoustic channels, etc. Second, active fault attacks (FA), that work by disrupting the normal execution of a cryptographic scheme through laser radiation, power glitches, etc., and then manipulate the result of the faulty execution to extract the secret key. There exists another type of physical attack known as microarchitectural (MA) attacks. This type of attack exploits the vulnerabilities or imperfections in the architecture of the platform where the cryptographic scheme executes. A strong motivation for studying MA attacks is that while traditional side-channel and fault attacks primarily target small, low-power devices such as microcontrollers e.g. Cortex-M devices and Internet of Things (IoT) devices, MA attacks can affect a much broader range of platforms such as enterprise servers, cloud platforms where multiple honest processes share the same hardware with a potentially hostile process. The two former physical attacks require the attacker to have physical access to the target device, but MA attacks can be performed remotely. Also, there are some relatively simpler methods like constant-time coding techniques that can help defend against some side-channel attack vectors like simple-power analysis, but for MA attacks e.g. Rowhammer-induced bit-flips [43] cannot be easily mitigated through coding practices alone. In the past, successful MA attacks on classical cryptographic schemes such as elliptic-curve discrete signature algorithms and symmetric schemes such as AES [20] have been demonstrated before [5,44,24,54].

Currently, there exist studies on physical attacks on PQC using SCA and FA [45,49,41,30] and some generic countermeasures such as masking and shuffling [15,58,35]. At this moment there exists only a handful of MA attacks on PQ schemes such as digital signature schemes Dilithium and LUOV [31,42] and key-encapsulation mechanism (KEM) Frodo [23]. Among these only Dilithium is a PQC standard. Therefore, we can safely admit that currently there is a huge gap in the literature regarding the assessment and countermeasures of MA attacks. As PQC seems to be prevalent in the near future it is crucial to study MA attacks in the context of PQC schemes. Hence, in this work, we focus on mounting efficient MA attacks on PQC schemes. We briefly summarize our contributions below.

– We study MA attacks or specifically Rowhammer attacks on KEMs based on hard lattice problem learning with errors (LWE). Most LWE-based KEMs share a generic framework Lyubashevsky et al. [38] to first create public-key encryption and then convert it to key-encapsulation mechanism using a version of Fujisaki-Okamoto transform [27]. We sketch an outline of how such generic constructions can be attacked using a Rowhammer-based MA attack. In Rowhammer attacks an attacker repeatedly accesses the memory rows adjacent to the victim process's memory row. Such repeated access can result in bit-flips in the victim process's memory row. Rowhammer can be single-sided when the attacker accesses memory rows only on one side of the target memory row or more aggressively double-sided, where the attacker accesses memory rows above and below the target memory row. This happens due to imperfections in the dynamic random-access memory (DRAM). For interested readers, we have provided more details of Rowhammer in Appendix A.

– Physical attacks on chosen-ciphertext attack (CCA) secure KEMs [30,45,41,49] work by running the decapsulation procedure or the plaintext checking oracle multiple times with different ciphertexts. At each run side-channel traces are captured or faults are induced which reveal some part of the key. Therefore, reducing the number of invoking the plaintext checking oracle can make the attack more practical. The work in [47] proposed a method to reduce the number of times the plaintext checking oracle is invoked. Here, we further reduced the number of times the oracle is invoked by as much as 39% for Saber and approximately 23% for Kyber768 compared to the previous work using some offline computations. The advantage of our method is not limited to this work only and can be of independent interest in the context of physical attacks on lattice-based KEMs.
– We choose two PQ KEMs Kyber [14] and Saber [21] to demonstrate the practicality of our attack. Kyber is a PQ KEM standard proposed by NIST and Saber was a finalist of the NIST PQC standardization procedure. We tailor our attack according to the design choices and parameters of Saber and Kyber.
– We show an end-to-end key-recovery method on Saber and Kyber based on remote software-induced faults only without using electromagnetic radiation, voltage glitch, laser radiation, etc. Our attack is very realistic as our conditions of attack are very relaxed compared to the previous works.
– Finally, we discuss the effect of existing physical attack countermeasures on our attack.

1.1 Paper Organization

The paper is organized as follows: Section 2 provides an overview of the necessary background information and introduces the notation and definitions used throughout the paper. Section 3 reviews the previous research conducted in the field. Section 4 presents the generic fault model of LPR schemes and explains its application to Kyber and Saber. Section 5 focuses on the practical realization of the fault model.

2 Preliminaries

Notations: We denote $\mathbb{Z}_q$ to represent the ring of integers modulo q. We use lowercase letters, lowercase letters with a bar, and uppercase letters to denote an element in $\mathbb{Z}_q$, vectors containing elements in $\mathbb{Z}_q$, and matrices with elements in $\mathbb{Z}_q$ respectively. Let $x \in \mathbb{Z}_q$, then x^i represents the i-th bit of x. Bold lowercase letters are used to denote elements in R_q where R_q is the polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$. For $i \in \{0, 1, ..., n-1\}$,

$\mathbf{x}[i]$ represents the i-th coefficient of the polynomial $\mathbf{x} \in R_q$. R_q^l represents the ring with vectors of l polynomials of R_q and the ring with matrices of $l \times k$ polynomials of R_q is presented by $R_q^{l\times k}$. We use bold lowercase with a bar and bold uppercase letters to denote elements in R_q^l and $R_q^{l\times k}$, respectively. For $\bar{\mathbf{x}} \in R_q^l$ and $\mathbf{X} \in R_q^{l\times k}$, $\bar{\mathbf{x}}_i$ denotes the i-th polynomial of the vector $\bar{\mathbf{x}}$ and $\mathbf{X}_{i,j}$ denotes the (i, j)-th polynomial of the matrix $\mathbf{X}$. The product of two polynomials $\mathbf{x}$ and $\mathbf{y}$ is denoted by $\mathbf{xy}$. The inner product of $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ in R_q^l is equal to $\sum_{i=0}^{l-1} \bar{\mathbf{x}}_i \bar{\mathbf{y}}_i$ is denoted by $\langle \bar{\mathbf{x}}, \bar{\mathbf{y}} \rangle$. If x is sampled from the set S according to the distribution $\mathcal{X}$, then we denote it as $x \leftarrow \mathcal{X}(S)$. We use $\mathcal{U}$ to represent uniform distribution and β_ν to indicate centered binomial distribution (CBD) with the standard deviation $\sqrt{\nu}/2$. $\lfloor x \rfloor$ outputs the largest integer, which is less than or equal to x. $\lfloor x \rceil$ represents the rounding of x to the nearest integer, which is equal to $\lfloor x + \frac{1}{2} \rfloor$. $r \gg x$ and $r \ll x$ denotes r shifted by x bit positions towards right and left respectively. All these operations can be extended to the polynomials, vectors, and matrices by applying them coefficient-wise. The cardinality of a set S is denoted by $|S|$.

2.1 Learning with errors (LWE) problem and its variants

LWE problem: Let us assume $A \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times k})$, error $\bar{e} \leftarrow \chi(\mathbb{Z}_q^l)$, secret $\bar{s} \leftarrow \chi(\mathbb{Z}_q^k)$, $\bar{b} = A\bar{s} + \bar{e} \in \mathbb{Z}_q^l$, and $\bar{b}' \leftarrow \mathcal{U}(\mathbb{Z}_q^l)$, where l, k, n are positive integers and χ is a distribution. Then, the decision version of the LWE problem states that distinguishing between $(A, \bar{b})$ and $(A, \bar{b}')$ is hard. This hardness depends on the parameter (n, l, k, q, χ) [51].
Ring-LWE (RLWE) problem: If we use the polynomial ring $R_q = \mathbb{Z}_q[X]/(x^n + 1)$ instead of $\mathbb{Z}_q$ and $l = k = 1$, then we call the problem as Ring learning with error problem (RLWE) [38]. So, in the RLWE problem, given $\mathbf{a} \leftarrow \mathcal{U}(R_q)$, $\mathbf{e}$, $\mathbf{s} \leftarrow \chi(R_q)$, $\mathbf{b} = \mathbf{as} + \mathbf{e} \in R_q$, and $\mathbf{b}' \leftarrow \mathcal{U}(R_q)$, it is hard to distinguish between $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{a}, \mathbf{b}')$. This hardness depends on the parameter (n, q, χ).
Module-LWE (MLWE) problem: In the MLWE problem [37], $\mathbf{A} \leftarrow \mathcal{U}(R_q^{l\times l})$ and $\bar{\mathbf{e}}$, $\bar{\mathbf{s}} \leftarrow \chi(R_q^l)$, $\bar{\mathbf{b}} = \mathbf{A}\bar{\mathbf{s}} + \bar{\mathbf{e}} \in R_q^l$, and $\bar{\mathbf{b}}' \leftarrow \mathcal{U}(R_q^l)$. The MLWE problem states that it is hard to distinguish between $(\mathbf{A}, \bar{\mathbf{b}})$ and $(\mathbf{A}, \bar{\mathbf{b}}')$. Here, the hardness depends on the parameter (n, l, q, χ).
Learning with Rounding (LWR) problem: In this problem, the error sampling is replaced by the rounding operation. Let us assume $\mathbf{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times k})$, $\mathbf{s} \leftarrow \chi(\mathbb{Z}_q^k)$, $\mathbf{b} = \lfloor \frac{\mathbf{p}}{\mathbf{q}}(\mathbf{As}) \rceil \in \mathbb{Z}_p^l$, and $\mathbf{b}' \leftarrow \mathcal{U}(\mathbb{Z}_p^l)$, where $q > p > 0$. Then the LWR problem states that distinguishing between $(\mathbf{A}, \mathbf{b})$ and $(\mathbf{A}, \mathbf{b}')$ is hard. This hardness depends on the parameter (n, l, k, q, χ) [10].

The ring-LWR (RLWR) problem and the module-LWR (MLWR) problem can be defined from the LWR problem in a similar way as the RLWE problem and the MLWE problem are defined from the LWE problem.

2.2 LPR public-key encryption

Lyubashevsky, Peikert, and Regev proposed the LPR public-key encryption (PKE) scheme based on the RLWE problem [38] as shown in Figure 1. Throughout this paper, we call this scheme as `LPR.PKE`. Here all the polynomials are elements of R_q, where q is a prime number and n is a power of two. In `LPR.PKE.KeyGen`, the secret $\mathbf{s} \leftarrow \chi(R_q)$ and the error $\mathbf{e} \leftarrow \chi(R_q)$. Here, χ is the Gaussian distribution, which is replaced by CBD in Kyber and Saber. $\mathbf{a} \leftarrow \mathcal{U}(\mathbb{Z}_q)$ and $\mathbf{b} = \mathbf{as} + \mathbf{e} \in R_q$. This algorithm declares $pk = (\mathbf{a}, \mathbf{b})$ as public key and saves $sk = (\mathbf{a}, \mathbf{s})$ as private key. In the `LPR.PKE.Enc` algorithm, a part of the ciphertext $\mathbf{u}$ is computed similarly to the public key $\mathbf{b}$. The other part of the ciphertext $\mathbf{v}$ contains message m and is computed as $\mathbf{v} = \mathbf{br} + \mathbf{e_2} + \texttt{Encode}(m)$. Here the `Encode` function is defined as $\texttt{Encode}(m) = m \cdot \lfloor \frac{q}{2} \rfloor$ *i.e.* multiplication of each message coefficient $m[i]$ with $\frac{q}{2}$. Then this algorithm outputs $c = (\mathbf{u}, \mathbf{v})$ as the ciphertext of the message m. The `LPR.PKE.Dec` algorithm takes the ciphertext c, and the secret key $\mathbf{s}$ as input, and then computes $\mathbf{m'} = \mathbf{v} - \mathbf{us}$. Now,

$$\begin{aligned}\mathbf{m'} &= \mathbf{v} - \mathbf{us} = (\mathbf{br} + \mathbf{e_2} + \texttt{Encode}(m) - (\mathbf{ar} + \mathbf{e_1})\mathbf{s} \\ &= (\mathbf{as} + \mathbf{e})\mathbf{r} + \mathbf{e_2} + m \cdot \lfloor q/2 \rfloor - (\mathbf{ar} + \mathbf{e_1})\mathbf{s} = m \cdot \lfloor q/2 \rfloor + \mathbf{er} + \mathbf{e_2} - \mathbf{e_1 s}\end{aligned}$$

Here, $\mathbf{d} = \mathbf{er} + \mathbf{e_2} - \mathbf{e_1 s}$ is known as decryption noise. The `LPR.PKE.Dec` algorithm uses the `Decode` function to remove the decryption noise from the message polynomial $\mathbf{m'}$ and recovers the message $m \in \{0, 1\}^n$.

Fujisaki-Okamoto (FO) transformation: The `LPR.PKE` scheme provides security against chosen-plaintext attacks (CPA) but does not offer protection against chosen-ciphertext attacks (CCA). FO transform is a generic transform to transform a CPA-secure PKE to CCA-secure KEM. Due to the presence of noise in the LPR-based scheme, a variant of FO transformation proposed by Jiang et al. [32] is generally used. The algorithms of this KEM are shown in Figure 2. A more detailed discussion regarding this FO transformation is provided in Appendix B.

2.3 Kyber

Kyber [14] is an LPR-based KEM with MLWE as its underlying hard problem. In the key generation algorithm of Kyber, the secret $\bar{\mathbf{s}} \leftarrow \beta_{\eta_1}(\mathbf{R_q^l})$ and error $\bar{\mathbf{e}} \leftarrow \beta_{\eta_1}(R_q^l)$. One part of the public key $\mathbf{A} \leftarrow \mathcal{U}(R_q^{l \times l})$ and the another part of the public key is $\bar{\mathbf{b}} = \mathbf{A}\bar{\mathbf{s}} + \bar{\mathbf{e}}$. The secret key is $sk = (\mathbf{A}, \bar{\mathbf{s}})$. In the encryption algorithm, the errors $\bar{\mathbf{r}} \leftarrow \beta_{\eta_1}(R_q^l)$ and the errors $\bar{\mathbf{e}}_1 \leftarrow \beta_{\eta_2}(R_q^l)$ and $\mathbf{e_2} \leftarrow \beta_{\eta_2}(R_q)$. A part of the ciphertext

```
LPR.PKE.KeyGen()
1. $\boldsymbol{a} \leftarrow \mathcal{U}(R_q)$
2. $\boldsymbol{s}, \boldsymbol{e} \leftarrow \chi(R_q)$
3. $\boldsymbol{b} = \boldsymbol{as} + \boldsymbol{e}$
4. return $(pk = (\boldsymbol{a}, \boldsymbol{b}), sk = (\boldsymbol{a}, \boldsymbol{s}))$

LPR.PKE.Dec($sk = (\boldsymbol{a}, \boldsymbol{s}), c = (\boldsymbol{u}, \boldsymbol{v})$)
1. $\boldsymbol{m'} = \boldsymbol{v} - \boldsymbol{us}$
2. $m = \texttt{Decode}(\boldsymbol{m'})$
3. return $m$

LPR.PKE.Enc($pk = (\boldsymbol{a}, \boldsymbol{b})$,
message $m \in \{0, 1\}^n$)
1. $\boldsymbol{r}, \boldsymbol{e_1}, \boldsymbol{e_2} \leftarrow \chi(R_q)$
2. $\boldsymbol{u} = \boldsymbol{ar} + \boldsymbol{e_1}$
3. $\boldsymbol{v} = \boldsymbol{br} + \boldsymbol{e_2} + \texttt{Encode}(m)$
4. return $c = (\boldsymbol{u}, \boldsymbol{v})$
```

Fig. 1: CPA secure LPR.PKE [38]

Table 1: Parameter set of Kyber and Saber corresponding to different security levels

Scheme Name		Parameters							Post-quantum Security	Failure Probability	NIST Security Level
		l	n	q	p	T	CBD parameters				
Kyber					$p=2^{d_u}$	$T=2^{d_v}$	η_1	η_2			
	Kyber512	2	256	3329	2^{10}	2^4	3	2	2^{107}	2^{-139}	1
	Kyber768	3			2^{10}	2^4	2	2	2^{166}	2^{-164}	3
	Kyber1024	4			2^{11}	2^5	2	2	2^{232}	2^{-174}	5
Saber				$q=2^{\epsilon_q}$	$p=2^{\epsilon_p}$	$T=2^{\epsilon_T}$	μ				
	LightSaber	2	256	2^{13}	2^{10}	2^3	5		2^{107}	2^{-120}	1
	Saber	3				2^4	4		2^{172}	2^{-136}	3
	FireSaber	4				2^6	3		2^{236}	2^{-165}	5

$\bar{\mathbf{u}}$ is computed similarly to the public key $\bar{\mathbf{b}}$ generation. Another part of the ciphertext $\mathbf{v} = \langle \bar{\mathbf{b}}, \bar{\mathbf{r}} \rangle + \mathbf{e_2} + \texttt{Encode}(m)$, where $\texttt{Encode}(m) = \lfloor m \cdot \frac{q}{2} \rfloor$. Then this algorithm uses $\texttt{compress}_\texttt{q}$ to compress each coefficient of $\bar{\mathbf{u}}$ to d_u bits and $\mathbf{v}$ to d_v bits. $c = (\bar{\mathbf{c}}_1, \mathbf{c_2}) = (\texttt{compress}_\texttt{q}(\bar{\mathbf{u}}), \texttt{compress}_\texttt{q}(\mathbf{v}, d_v))$ serves as the ciphertext associated with the message m. The decryption algorithm first decompresses both components $\bar{\mathbf{c}}_1$ and $\mathbf{c_2}$ of the ciphertext c with $\texttt{Decompress}_\texttt{q}$ function. Suppose $\bar{\mathbf{u}}' = \texttt{Decompress}_\texttt{q}(\bar{\mathbf{c}}_1, d_u)$ and $\mathbf{v}' = \texttt{Decompress}_\texttt{q}(\mathbf{c_2}, d_v)$. Then it computes $\texttt{Decode}(\mathbf{v}' - \langle \bar{\mathbf{s}}, \bar{\mathbf{u}}' \rangle) = \texttt{Compress}_\texttt{q}((\mathbf{v}' - \langle \bar{\mathbf{s}}, \bar{\mathbf{u}}' \rangle), 1)$ to recover the message m. There are three security versions of Kyber based on the parameter set, and we include them in Table 1. In this paper, unless otherwise specified we refer to the parameter set of Kyber768 with Kyber. For more details, we refer the interested reader kindly to the original paper [14] for further details.

2.4 Saber

Saber [21] is a KEM based that also follows the LPR model. Saber is based on the hard problem MLWR. Here q in R_q is power-of-two. In the key generation algorithm

```
KEM.KeyGen()
  1. $(pk, sk) = \texttt{PKE.KeyGen}()$
  2. $z \leftarrow \mathcal{U}(\{0, 1\}^n)$
  3. return $(pk, sk' = (sk||pk||\mathcal{H}(pk)||z)$

KEM.Encaps($pk$)
  1. $m \leftarrow \mathcal{U}(\{0, 1\}^n)$
  2. $(K', r) = \mathcal{G}(m, \mathcal{H}(pk))$
  3. $ct = \texttt{PKE.Enc}(pk, m, r)$
  4. $K = \mathcal{F}(K', \mathcal{H}(ct))$
  5. return $(ct, K)$

KEM.Decaps($sk' = (sk||pk||\mathcal{H}(pk)||z), ct$)
  1. $m' = \texttt{PKE.Dec}(sk, ct)$
  2. $(K'', r') = \mathcal{G}(m', \mathcal{H}(pk))$
  3. $c = \texttt{LPR.PKE.Enc}(pk, m', r')$
  4. if: $ct = c$ return $K = \mathcal{F}(K'', \mathcal{H}(ct))$
  5. else: return $K = \mathcal{F}(z, \mathcal{H}(ct))$
```

Fig. 2: CCA secure KEM based on LPR.PKE using FO transformation [32]

of Saber, the secret $\bar{\mathbf{s}} \leftarrow \beta_\mu(R_q^l)$. The public key here is $(\mathbf{A},\ \bar{\mathbf{b}})$ where $\mathbf{A} \in R$ is an element of $R_q^{l\times l}$ and is sampled uniformly and $\bar{\mathbf{b}} = (\mathbf{A}\bar{\mathbf{s}} + \bar{\mathbf{h}}) \gg (\epsilon_q - \epsilon_p) \in R_p^l$. The vector $\bar{\mathbf{h}}$ is needed for rounding, and it consists of constant polynomials with each coefficient equal to $2^{\epsilon_q-\epsilon_p-1}$. In the case of the encryption algorithm, $\bar{\mathbf{s}}'$ is also sampled from the CBD distribution β_μ. The key contained part of the ciphertext $\bar{\mathbf{u}}$ is computed similarly to the public key $\bar{\mathbf{b}}$. The message contained part of the ciphertext $\mathbf{v}$ is computed as $(\langle\bar{\mathbf{b}},\ \bar{\mathbf{s}}'\rangle + \mathbf{h_1} - \texttt{Encode}(m) \bmod p) \gg \epsilon_p - \epsilon_T \in R_T$. $\mathbf{h_1}$ is a constant polynomial with each coefficient equal to $2^{\epsilon_q-\epsilon_p-1}$. It is required for rounding. Let $c = (\bar{\mathbf{u}},\ \mathbf{v})$ is the ciphertext corresponding to the message m. Then, the decryption algorithm takes the ciphertext $c = (\bar{\mathbf{u}},\ \mathbf{v})$ and secret $\bar{\mathbf{s}}$ as inputs. It computes $(\langle\bar{\mathbf{u}},\ \bar{\mathbf{s}}\rangle \bmod p - 2^{\epsilon_p-\epsilon_T}\mathbf{v} + \mathbf{h_2}) \bmod p \gg (\epsilon_p - 1) \in R_2$ to find the decrypted message. $\mathbf{h_2}$ is also a constant polynomial with each coefficient equal to $2^{\epsilon_p-2} - 2^{\epsilon_p-\epsilon_T-1}$. Like Kyber, Saber also has three security versions depending on the parameter set, and we present them in Table 1. Similar to Kyber, in this paper, we refer to the parameter set of Saber with $l = 3$ with Saber, and we refer to the original paper [21] for further details.

2.5 Related works

Lattice-based post-quantum KEMs are vulnerable to side-channel attacks. A timing attack on the `KEM.Decaps` has been shown in [29], it targets the non-constant time implementation of the ciphertext equality checking (Line 4 in `KEM.Decap` algorithm of Figure 2). [49], proposed a generic and practical Electromagnetic (EM) power analysis assisted CCA on LWE-based KEMs. They also target the `KEM.Decaps` in their attack. They have constructed a plaintext-checking oracle $\mathcal{O}$ with the help of an EM power attack, which can distinguish two particular messages $m_1 = 00...0$ (all zeros) and $m_2 = 00...01$ (all zeros except the LSB). This oracle provides single-bit information related to one coefficient of the secret key. Continuing the same methods, the attacker can find the whole secret key. This paper has shown that 2000 to 4000 queries are required to retrieve the complete secret key for Kyber. [56] reduced the query requirements by creating a multiple-valued plaintext-checking oracle. Here, the attacker acquires information regarding multiple secret key coefficients from a single query. In [47], the authors further reduced the number of queries required to recover the whole secret keys by improving the model of plaintext-checking oracle. One significant area of research in this domain revolves around improving the efficiency of attacks by minimizing the number of required quires. This reduction enables a more precise evaluation of the cost of an optimal attack. Our attack contributes in this direction by improving the process of using the parallel plaintext checking oracle model of the paper [47].

Rowhammer has been used to successfully attack many cryptographic primitives. In the paper [50], researchers demonstrated a Rowhammer attack on RSA signatures. Additionally, in [36], the authors illustrate the direct reading of RSA key bits from the memory address. However, there is limited research on Rowhammer attacks targeting post-quantum schemes. The current state-of-the-art in this domain focuses on a single work involving Rowhammer attacks on the PQC KEM Frodo [13]. This research primarily targets the key-generation procedure, which is known to be relatively easy to protect. In this work, we will demonstrate an end-to-end Rowhammer attack on the decapsulation algorithm of the targeted schemes.

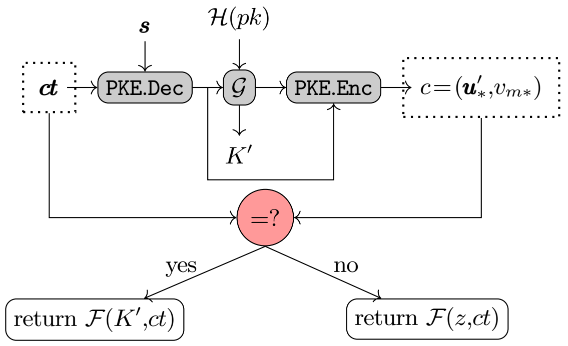

Fig. 3: Decapsulation algorithm of KEM based on `LPR.PKE` (Figure 2). Here z is a random number generated in the `KEM.KeyGen()` algorithm (Figure 2). The fault location is marked in red.

3 Our attack using binary decision tree on the LPR-based schemes

Attack Surface: The KEM based on `LPR.PKE` shown in Figure 2 is resistant to CCA. In such schemes, the secret key is generated using the `KEM.KeyGen` is *non-ephemeral i.e.* stored and used for the long term. The key generation and encapsulation processes are executed only once. Therefore, the attacker needs to recover the secret key or the shared key from a single execution. However, the secret key remains fixed in the decapsulation algorithm for a long time and is used to derive the shared secret key K from multiple users. This is done to remove the huge overhead of running the key generation process and distributing the public key each time two communicating parties want to establish the shared secret K. However, this convenience also helps an attacker. An attacker can now execute the decapsulation operation multiple times and collect multiple traces or induce faults at different locations. This helps the attacker to refine its attack strategy and increase the probability of success manifold. This is why attacking the decapsulation operation is mostly chosen by attackers to mount physical attacks [45,30,41,49]. So, we also choose the decapsulation method as our target. The structure of the `KEM.Decaps` given in Figure 2 is shown in Figure 3. Here we also assume the attacker can invoke the victim's decapsulation procedure by submitting any ciphertexts of its preference.

We assume the general Rowhammer threat model, where the attacker and victim use two different processes in the same operating system or two virtual machines on the same server [59]. This threat model is also used in most of the micro-architectural attacks work [60]. Here the attacker shares the same hardware responsible for performing the victim's decapsulation procedure of LPR-based KEM. The attacker can also invoke the victim's decapsulation procedure by submitting any ciphertexts of its preference.

3.1 Implementing a parallel plaintext checking (PC) oracle.

In the `KEM.Decaps` procedure in Figure 2, the decrypted message m undergoes a hashing operation $\mathcal{G}$ with the public key. The resulting hash, denoted as (K', r), where r is combined with the message m and is used as input for the subsequent re-encryption procedure using `LPR.PKE.Enc` algorithm. The generated key K' is employed to create a valid shared key K. It is crucial to note that the hash function $\mathcal{G}$ is deterministic and solely relies on the decrypted message m and public key pk. Considering 2^t messages where a fixed chunk of t bits are changed while keeping all other $n-t$ bits fixed, such as

$$m^{(0)} = \underbrace{000...0}_{t\ bits}\ \underbrace{000...0}_{(n-t)\ bits}$$

$$m^{(1)} = \underbrace{100...0}_{t\ bits}\ \underbrace{000...0}_{(n-t)\ bits}$$

$$m^{(2)} = \underbrace{010...0}_{t\ bits}\ \underbrace{000...0}_{(n-t)\ bits}$$

$$...$$

$$m^{(2^t-1)} = \underbrace{111...1}_{t\ bits}\ \underbrace{000...0}_{(n-t)\ bits}$$

A variation of t bits in these messages leads to substantial variations in the computations performed during the hash $\mathcal{G}$ operation. Consequently, the ciphertexts generated by the `LPR.PKE.Enc` algorithm will differ for each of the 2^t messages.

In our attack scenario, we require the output to be dependent on the decrypted message. However, if we use artificially constructed ciphertext ct (which is not generated from `LPR.PKE.Enc`), then with high probability, the re-encrypted ciphertext c and ct will be unequal. The current implementation always returns $\mathcal{F}(z, \mathcal{H}(ct))$ as the shared key, which is independent of the decrypted message. In order to distinguish the potential 2^t decrypted messages of the ciphertext ct, we need the output to be message-dependent. By omitting this equality checking condition, we ensure that the hash value $\mathcal{F}(K', \mathcal{H}(ct))$ is consistently returned, which is decrypted message dependent. That allows us to differentiate between the possible decrypted messages of ct. Our goal is to reliably acquire the shared key $\mathcal{F}(K', \mathcal{H}(ct))$ by employing a physical attack.

In the `KEM.Decaps`, both Saber and Kyber use a variable named "fail". Compare the ciphertexts ct and c by calling the function `verify(c, ct, BYTES_CCA_DEC)` and storing the return value of this function in the "fail" variable. If the value of fail is 0, then it returns the shared key $\mathcal{F}(K', \mathcal{H}(ct))$, which depends on the decrypted message. Otherwise, it returns the random shared key. Our aim is to flip the value of the variable "fail" by introducing fault even when the ciphertexts are not equal.

3.2 Generic attack model using PC oracle

The first stage of our attack is to carefully craft ciphertexts c to reduce the number of invocations of the `KEM.Decaps` procedure. Here, we target to recover t secret coefficients of the secret key $\mathbf{s}$ at a time. We introduce a notation $s_i^{(t)}$ to represent a block of consecutive t coefficients of $\mathbf{s}$, where $i \in \{0, 1, ..., \lfloor \frac{n}{t} \rfloor - 1\}$ and the last block $s^{(t')}_{\lfloor \frac{n}{t} \rfloor}$ consists $t' = n - (\lfloor \frac{n}{t} \rfloor \times t)$ secret coefficients. This ciphertext c is then transmitted to the oracle $\mathcal{O}_\mu$. Here the oracle $\mathcal{O}_\mu$ defined as follows:

$$\mathcal{O}_\mu(c; x^{(0)}, x^{(1)}, ..., x^{(\mu-1)}) = r, \text{ if } \texttt{PKE.Dec}(c) = x^{(r)}, 0 \leq r \leq \mu - 1$$

. This oracle $\mathcal{O}_\mu$ takes a ciphertext c and μ number of messages $x^{(i)}$ and returns the value r such that the decrypted message of c is $x^{(r)}$. Upon receiving the ciphertext, the oracle $\mathcal{O}_\mu$ processes c along with a set of potential messages $x^{(0)}, x^{(1)}, ..., x^{(\mu-1)}$. Then, the oracle provides a response r such that $\texttt{LPR.PKE.Dec}(sk, c) = x^{(r)}$. By analyzing the decrypted message $x^{(r)}$, we gain knowledge about the secret block $s^{(t)}_i$. As each secret coefficient is intricately tied to the decrypted message, this process gradually reduces the dimension of the secret coefficients within the targeted block. This reduction process involves considering the relationship between the decrypted message and the secret coefficients. After successfully reducing (not fully recovering) the dimension of the secret block, we construct another new ciphertext c_α that exploits the potential secret block $s^{(t)}_i$. Then, repeating the aforementioned process, we further reduce the cardinality of the secret set corresponding to each coefficient of the secret block to get our desired secret. The challenge lies in determining how many iterations of this process are necessary to effectively reduce the dimension of the secret block $s^{(t)}_i$. One possible approach is to repeat until the entire secret block $s^{(t)}_i$ is obtained. In the paper [47], the authors used this approach. In this method, we need to query the oracle $\mathcal{O}_\mu$ $\lceil \log |S_0| \rceil$ times to find each of the secret blocks $s^{(t)}_i$ and $s^{(t')}_{\lfloor \frac{n}{t} \rfloor}$, where $i \in \{0, 1, ..., \lfloor \frac{n}{t} \rfloor - 1\}$ and $t' = n - (\lfloor \frac{n}{t} \rfloor \times t)$. Here, S_0 represents the set of all possible values of a coefficient of the secret key. However, each iteration incurs a cost regarding the number of injected faults. Since each fault is resource-intensive, the objective is to find the secret with the minimum number of faults.

In our approach, we reduce the number of queries to the oracle $\mathcal{O}_\mu$ to find the all secret blocks $s^{(t)}_i$ and $s^{(t')}_{\lfloor \frac{n}{t} \rfloor}$, where $i \in \{0, 1, ..., \lfloor \frac{n}{t} \rfloor - 1\}$ and $t' = n - (\lfloor \frac{n}{t} \rfloor \times t)$. Here, the previous approach is repeated $\lfloor \log |S_0| \rfloor$ times to progressively reduce the cardinality of the secret set corresponding to each coefficient of each secret block. Since we query $\lfloor \log |S_0| \rfloor$ times to the oracle $\mathcal{O}_\mu$ for each block, there will be some secrets that have not been determined yet. So, after reducing the dimension of each secret block, an index set, denoted as `Index[]`, is created to track the indices of the secret coefficients that have not been determined yet. A new ciphertext c_α is then constructed based on the `Index[]` set, and the values of the secret coefficients corresponding to the indices in `Index[]` are updated accordingly. For simplicity, we describe this attack template step by step for a parallelization factor t, which is a divisor of n, to unveil the secret block gradually. The process will be similar for other parallelization factor t.

Constructing the ciphertext c Here, we present a method to construct a dummy ciphertext $ct = (\mathbf{u}, \mathbf{v}) \in R_q \times R_q$. This method helps to decrease the number of queries required to retrieve all the secrets of the block $s^{(t)}_0$, which contains $\{\mathbf{s}[0], \mathbf{s}[1], ..., \mathbf{s}[t-1]\}$ first t coefficients of the secret polynomial $\mathbf{s}$. To construct the ciphertext ct, first we set $\mathbf{u}[0] = k_u$ and $\mathbf{v}[j] = k_{v_j}$, $\forall 0 \le j \le t-1$ are non zero and others coefficients of $\mathbf{u}$ and $\mathbf{v}$ are zero. Then

$$(\mathbf{v} - \mathbf{u}\mathbf{s}) = \sum_{j=0}^{t-1} k_{v_j}.x^j - \sum_{j=0}^{n-1} k_u \mathbf{s}[j].x^j$$

$$\text{So } (\mathbf{v} - \mathbf{u}\mathbf{s})[j] = \begin{cases} (k_{v_j} - k_u \mathbf{s}[j]) & \text{if } 0 \le j \le t-1 \\ (-k_u \mathbf{s}[j]) & \text{Otherwise .} \end{cases}$$

Hence the coefficients of the decrypted message m will be

$$m^j = \begin{cases} \mathtt{Decode}(k_{v_j} - k_u \mathbf{s}[j]) & \text{if } 0 \le j \le t-1 \\ \mathtt{Decode}(-k_u \mathbf{s}[j]) & \text{Otherwise .} \end{cases}$$

We choose the value $(k_u, k_{v_0}, k_{v_1}, ..., k_{v_{t-1}})$ such that

$$m^j = \begin{cases} \text{Depends on } \mathbf{s}[j] & \text{if } 0 \le j \le t-1 \\ 0 & \text{Otherwise} \end{cases}$$

We construct a binary decision tree shown in Figure 4 to distinguish the secrets. We select each value k_{v_j} from the tree accordingly. Initially, all the values k_{v_j} will be the root value d_0. Then, depending on the decrypted message, we update the value k_{v_j} from the tree. Also, the value of k_u will be fixed in an iteration because we are constructing the dummy ciphertext to get t bits of information at a time.

To recover j'-th secret block, $\mathbf{s}_{j'}^{(t)}$ that contains the secret coefficients $\mathbf{s}[j']$, $\mathbf{s}[j'+1]$, ..., $\mathbf{s}[j'+t-1]$), where $j' > 0$ we have to construct the dummy ciphertext $ct = (\mathbf{u}, \mathbf{v}) \in R_q \times R_q$, where $\mathbf{u}[n-j'] = k_u$ and $\mathbf{v}[j] = k_{v_j}$, $\forall 0 \le j \le t-1$ are non zero and others coefficients of $\mathbf{u}$ and $\mathbf{v}$ are zero. Then

$$(\mathbf{v} - \mathbf{u}\mathbf{s}) = \sum_{j=0}^{t-1} k_{v_j}.x^j + \sum_{j=j'}^{n-1} k_u \mathbf{s}[j].x^{j-j'} - \sum_{j=0}^{j'-1} k_u \mathbf{s}[j].x^{n-j'+j}$$

Here, the decrypted message m will be

$$m^j = \begin{cases} \mathtt{Decode}(k_{v_j} + k_u \mathbf{s}[j'+j]) & \text{if } 0 \le j \le t-1 \\ \mathtt{Decode}(-k_u \mathbf{s}[j]) & \text{Otherwise} \end{cases} \quad (1)$$

Similarly, the value of k_{v_j} will be taken from the binary decision tree pictured in Figure 4. We also present the algorithm to create ciphertext in Algorithm 1.

Parallel PC oracle for $s_i^{(t)}$ by pruned binary decision tree: We construct a binary decision tree with two types of nodes; one is the (S_y, d_y) where secret set S_y with $|S_y| > 1$ and the constant values d_y which helps us to split the secret set S_y into two disjoint sets S_{2y+1} and S_{2y+2}. The other one is S_y with $|S_y| = 1$ as shown in Figure 4. We construct the tree such that the tree will be almost complete and the distance of

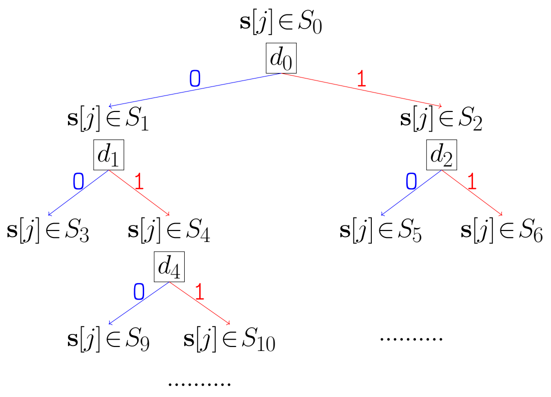

Fig. 4: Binary tree to select the value of $k_{v_j} = d_i$ for each $v[j]$

node (S_y, d_y) from the root node (S_0, d_0) will be longer if the set S_y contains the secret coefficients with comparatively lower probability. Let h be the maximum height of the node of format (S_y, d_y) from the root node (S_0, d_0). Without loss of generality, assume that this maximum height node is (S_w, d_w) i.e., the distance of the node (S_w, d_w) from the root node is h and the height of the tree is $h+1$ which is the distance from the node (S_0, d_0) to the node S_{2w+1}. We have distinguished the secrets from the tree as follows:

First, we will query to the oracle $\mathcal{O}_\mu$ with the constructed ciphertext ct and 2^t messages $m^{(0)}$, $m^{(1)}$,..., $m^{(2^t-1)}$ described before. Let $m^{(r)}$ be the decrypted message of the ciphertext ct, which is received from the oracle $\mathcal{O}_\mu$. If the j-th secret coefficient of the block $\mathbf{s}_i^{(t)}$, $\mathbf{s}[i+j]\in S_y$ and $|S_y|>1$, then we will distinguish $\mathbf{s}[i+j]\in S_{2y+1}$ or $\mathbf{s}[i+j]\in S_{2y+2}$ according to the value of the corresponding j-th message bit of $m^{(r)}$ which is $\texttt{Decode}(d_y - k_u\mathbf{s}[i+j]) = 0/1$ i.e., observing the current secret set S_y in which the secret coefficient belongs and the decrypted bit $\texttt{Decode}(d_y - k_u\mathbf{s}[i+j])$, we reduce the possible values of the secrets from S_y to S_{2y+1} or S_{2y+2}. In each iteration of the

Algorithm 1 Ciphertext creation I

Input: The index i of secret block $s_i^{(t)}$ and the current secret set S_{r_k} corresponding to the block.

Output: Ciphertext ct such that the decrypted message of c will be zero except the first t positions.

```
1: for $k=0;k<n;k++$ do
2:     $\mathbf{u}[k]=0$; $\mathbf{v}[k]=0$
3: end for
4: $\mathbf{u}[(n-i)\%n]=k_u$
5: for $k=0;k<t;k++$ do
6:     if $\mathbf{s}[i+k]\in S_{r_k}$ then
7:         $\mathbf{v}[k]=d_{r_k}$
8:     end if
9: end for
```

block $\mathbf{s}_i^{(t)}$, each value of k_{v_j} will traverse this tree from the root node (S_0, d_0) (with height 0). In our attack, we traverse each value k_{v_j} from the tree up to the height $h-1$, i.e., We pruned the highest heighted node (S_w, d_w) from this tree. In this way, we reduce the cardinality of the secret set corresponding to each secret. Since we ignore the highest height node (S_w, d_w), only secret coefficients that belong to the secret set S_w will still be undetected.

Construction of `Index[]` set As we discussed before, only secret coefficients belonging to the secret set S_w will still be undetected. Now, we will search the indexes of the secret coefficients that are still not decided and store them in a set named "`Index[]`". Then, we apply the parallel checking oracle $\mathcal{O}_\mu$ on this `Index[]` set. We describe the detailed process in the following section.

Construction of ciphertext c_α from `Index[]`: Before arriving at this stage, we found most secrets except the `Index[]` set secrets. Without loss of generality, assume that $\mathbf{s}[i] \in S_w\ \forall i \in$ `Index[]`, where S_w contains the values with a low probability occurrence and d_w is the corresponding value of ciphertext selection in the Figure 4.

Let `Index[]` $= \{\alpha_0, \alpha_1, ..., \alpha_r\}$. Construct the dummy ciphertext $ct = (\mathbf{u}, \mathbf{v}) \in R_q \times R_q$ to reduce the cardinality of the secret set corresponding to each coefficient of the secret coefficients $\mathbf{s}[\alpha_0], ..., \mathbf{s}[\alpha_{t-1}]$ (we called it secret block $\mathbf{s}^{(t)}_{\alpha_0,...,\alpha_{t-1}}$ of size t). We choose $\mathbf{u}[0] = k_u$ and $\mathbf{v}[\alpha_j] = d_w$, $\forall 0 \leq j \leq t-1$, as each $\mathbf{s}[\alpha_j] \in S_w$. All the remaining coefficients of $\mathbf{u}$ and $\mathbf{v}$ will be zero. Then the decrypted message will be

$$m^j = \begin{cases} \texttt{Decode}(d_w - k_u \mathbf{s}[j]) & \text{if } j = \alpha_0, ..., \alpha_{t-1} \\ \texttt{Decode}(-k_u \mathbf{s}[j]) & \text{Otherwise} \end{cases} \tag{2}$$

So, the message will depend on all the α_j-th secret coefficient $\mathbf{s}[\alpha_j]$, where $0 \leq j \leq t-1$, which is followed by the construction of our binary decision tree shown in the Figure 4.

We query the oracle $\mathcal{O}_\mu$ with the forged ciphertext c_α and the 2^t messages $m^{(0)'}$, $m^{(1)'}$,..., $m^{(2^t-1)'}$ to get t bits of information with location α_0,..., α_{t-1} simultaneously. Here, we take each message $m^{(i)'}$ such that α_j-th bit of the message $m^{(i)'}$

Algorithm 2 Cardinality reduction of the secret set of the block $s_i^{(t)}$

```
    Input: The decrypted message $m$ of the ciphertext $c$ such that $m$ is non-zero at most
in the first $t$ positions.
    Input: The value $r_k$ such that $\mathbf{s}[i+k] \in S_{r_k}$, $0 \leq k \leq t-1$.
    Output: Update $[i+k]$ where $0 \leq k \leq t-1$.
1: for $l=0; l<t; l++$ do
2:     if $m^l = 0$ then
3:         $\mathbf{s}[i+l] \in S_{2r_l+1}$
4:     else
5:         $\mathbf{s}[i+l] \in S_{2r_l+2}$
6:     end if
7: end for
```

Algorithm 3 Ciphertext creation II

Input: The index α_0,..., α_{t-1} of those we want to find actual secret.

Output: Ciphertext ct such that the decrypted message of c will be zero except the possitions α_0,..., α_{t-1}.

```
1: for $k=0;k<n;k{+}{+}$ do
2:     $\mathbf{u}[k]=0$; $\mathbf{v}[k]=0$
3: end for
4: $\mathbf{u}[0]=k_u$
5: for $k=0;k<t;k{+}{+}$ do
6:     $\mathbf{v}[\alpha_k]=d_w$
7: end for
```

Algorithm 4 Rotating secret coefficients

Input: The secret $\mathbf{s}$ is in the sequence $\mathbf{s}[0]$, ..., $\mathbf{s}[t-1]$, $-\mathbf{s}[n-t]$, $-\mathbf{s}[n-t+1]$, ..., $-\mathbf{s}[n-1]$, $-\mathbf{s}[n-2t]$, ..., $-\mathbf{s}[n-t-1]$, $\cdots=\mathbf{s1}$

Output: The secret $\mathbf{s}$ with actual order i.e.,($\mathbf{s}[0]$, $\mathbf{s}[1]$, ..., $\mathbf{s}[n-1]$)

```
1: for $j=0;j<t;j{+}{+}$ do
2:     $\mathbf{s}[j]=\mathbf{s1}[j]$;
3: end for
4: for $j=1;j<\lfloor\frac{n}{t}\rfloor;j{+}{+}$ do
5:     for $k=0;k<t;k{+}{+}$ do
6:         $\mathbf{s}[t*j+k]=-\mathbf{s1}[(n-t*j+k)\%n]$;
7:     end for
8: end for
9: Return $\mathbf{s}$
```

is the j-th bit of i and the others bits are zero. Here, we use Algorithm 3 to create forged ciphertexts.

Updating the secret coefficients whose index lies in `Index[]`: We divide the sampling set into two distinct parts: $S_{2w+1}=\{s:\texttt{Decode}(d_w-k_us)=0\}$ and $S_{2w+2}=\{s:\texttt{Decode}(d_w-k_us)=1\}$, where d_w is a predefined constant. Since S_w contains the values such that the highest distance from the root node with $|S_w|>1$, therefore $|S_{2w+1}|$ and $|S_{2w+2}|$ must be 1. Otherwise, it violates our assumption of the set S_w. So, querying the oracle $\mathcal{O}_\mu$ with one ciphertext c_α and the above messages $m^{(0)'}$, $m^{(1)'}$, ..., $m^{(2^t-1)'}$, we will get a decrypted message as a response. This decrypted message decides the t number of secret coefficients $\mathbf{s}[\alpha_0]$, $\mathbf{s}[\alpha_1]$ ..., $\mathbf{s}[\alpha_{t-1}]$ at a time. So, running the process $\lceil\frac{|\texttt{Index[]}|}{t}\rceil$ times, we will find the whole secret with mixed signs and in a different order. We described the process of finding the secret in actual order. Also, from Equation 1, we can see that for the secret block $s^{(t)}_{j'}$, each j-th message m^j will depend on the secret coefficient $-s[j'+j]$, $0\leq j\leq t-1$. So basically, we are decreasing the dimension secret coefficients $\mathbf{s}[0]$, ..., $\mathbf{s}[t-1]$, $-\mathbf{s}[n-t]$, $-\mathbf{s}[n-t+1]$..., $-\mathbf{s}[n-1]$, $-\mathbf{s}[n-2t]$,..., $-\mathbf{s}[n-t-1]$, $\cdots-\mathbf{s}[t]$, $-\mathbf{s}[t+1]$, ..., $-\mathbf{s}[2t-1]$. We transformed it into the actual secret block using the Algorithm 4.

Table 2: For Kyber768

	$u=38, v=14$			
S	$d_0=12$	$d_1=4$	$d_2=13$	$d_4=3$
-2	0	1	0	1
-1	0	1	0	0
0	0	0	0	0
1	1	0	0	0
2	1	0	1	0

Table 3: For Saber

S	$u=0x3c8$							$u=7$
	$d_0=4$	$d_2=2$	$d_5=3$	$d_6=1$	$d_1=6$	$d_3=7$	$d_4=5$	$d_7=12$
-4	0	0	0	0	0	0	0	0
-3	0	0	0	0	0	1	0	0
-2	0	0	0	0	1	1	0	0
-1	0	0	0	0	1	1	1	0
0	1	0	0	0	1	1	1	0
1	1	0	1	0	1	1	1	0
2	1	1	1	0	1	1	1	0
3	1	1	1	1	1	1	1	0
4	1	1	1	1	1	1	1	1

Number of queries: Here (S_w, d_w) is the most distanced node from the root node with $|S_w|>1$ and containing secrets occurring with comparatively lower probability.

1. Best case: If all the secret values lie in S_0-S_w, then the number of queries will be minimum because, in this case, we need $\lfloor \log|S_0| \rfloor$ queries to find each block of secrets $\mathbf{s}_i[j]$, $\mathbf{s}_i[j+1]$, ..., $\mathbf{s}_i[j+t-1]$ of blocksize t. The total number of queries will be: $\lceil \frac{n}{t} \rceil \times \lfloor \log|S_0| \rfloor$.
2. Average case: Let E_1 be the expected number the secret coefficients those belongs to S_w. Then the total number of queries will be: $(\lceil \frac{n}{t} \rceil \times \lfloor \log|S_0| \rfloor)+(\lceil \frac{E_1}{t} \rceil)$.

With our method, the number of queries for the average case decreases compared to the state-of-the-art works [56,47].

3.3 Model for Kyber and Saber

Kyber and Saber are based on the module-LWE and module-LWR problems, respectively i.e., here, the modules R_q^l are used for the secret and the ciphertext $\bar{\mathbf{b}}'$ instead of the ring R_q. But if we construct $c=(\bar{\mathbf{b}}', \mathbf{v})$ as follows:

$$\bar{\mathbf{b}}'_i[j]=\begin{cases} k_u, \text{ if } i=0, j=0 \\ 0, \text{ otherwise} \end{cases} \quad \text{and } \mathbf{v}[j]=\begin{cases} k_{v_j}, \text{ if } 0\leq j\leq t-1 \\ v, \text{ otherwise,} \end{cases}$$

where k_u, k_{v_j} are constants. Then the problem reduces to the generic LPR problem, i.e., to the ring problem. Therefore, here the total number of queries will be $l\times$ `the number of queries for LPR`. We use the corresponding d_i from Table 2 and 3 for Kyber768 and the Saber, respectively. We will construct the corresponding binary decision tree from Table 2 and 3 and construct our ciphertext accordingly. For Kyber768, we have seen that for $k_u=38$, $v=14$ and $k_{v_j}=d_i$. From Table 2, we can recover the secret by a similar process mentioned in the previous section.

Number of queries for Kyber768 and Saber: According to Table 2, for Kyber768 S_4 will be the highest distanced node from the root node containing secrets with comparatively low probability and $|S_4|>1$. Also, Table 3 shows that for Saber, S_7 will be that specified node above. For Kyber768 and Saber, $l=3$, we consider our best case and average cases of both the algorithms for $l=3$.

1. Best case: In case of Kyber768, if all the secret values lie in S_0-S_4, then the number of queries will be minimum because, in this case, we need 2 queries to find each block of secrets $\bar{\mathbf{s}}_i[j]$, $\bar{\mathbf{s}}_i[j+1]$, ..., $\bar{\mathbf{s}}_i[j+t-1]$ of blocksize t. The total number of queries will be: $\lceil \frac{n}{t} \rceil \times 3 \times 2$. For Saber this number will be $\lceil \frac{n}{t} \rceil \times 3 \times 3$.
2. Average case: In the case of Kyber768, if E_1 is the expected number of the secret coefficients of each polynomial that lie in the set S_4, the total number of queries will be: $3\times((\lceil \frac{n}{t} \rceil \times 2)+\lceil \frac{E_1}{t})\rceil)$. Similarly, for Saber, if E_1 is the expected number of the secret coefficient of each polynomial that lies in S_7, then the total number of queries will be: $3\times((\lceil \frac{n}{t} \rceil \times 3)+\lceil \frac{E_1}{t} \rceil)$.

3.4 Comparing our attack with the state-of-the-art

In this section, we compare the total number of ciphertexts required to retrieve the whole secret key for the average case in Kyber768 and Saber with our attack and the work by Rajendran et al. [47], which also proposed methods to reduce the number of ciphertexts using parallel plaintext checking oracle model. Even though we need to use the same number of ciphertext as [47] to recover the whole secret key when the parallelization factor $t=1$, our attack model requires less number of ciphertexts than [47] to recover the whole secret key in the average case when the parallelization factor $t>1$. If $t=10$ or 12 or 16, for Kyber768 we use approximately 22% less number of ciphertext than [47]. Also, in Saber, if we take $t=10$, we require $\approx 39\%$ less number of ciphertext than the paper [47] to recover the key. However, we require 57 number of ciphertext to recover the whole secret key of Kyber768 in the average case when the parallelization factor $t=32$. We observe that increasing the parallelization factor t will reduce the number of required ciphertexts. However, in this case, the process of finding the decrypted message from the shared key (offline calculation) will be more costly (takes 2^t comparison). For this reason, we take the value of the parallelization factor t up to 32. But, with a more powerful computer that can do 2^{40} comparison, then we can take the parallelization factor $t=40$. In this case, the number of queries will be 48.

Frequency of fault induction in the attack for Kyber768: We have discussed earlier that to recover the whole secret of the algorithm Kyber768, we require 57 faulted shared keys i.e., 57 many times, we often have to introduce the bit-flip faults at the location of the variable "fail".

4 Realization of the fault model

In this section, we are going to illustrate an end-to-end strategy to demonstrate the fault model in practice.

Table 4: Number of queries required to recover the key for Kyber768 and Saber in total

Scheme		Parallelization factor t					
		1	10	12	16	32	40
Kyber768	This work $3\times((\lceil\frac{256}{t}\rceil\times2)+\lceil\frac{80}{t})\rceil)$	1776	**180**	**153**	**111**	**57**	**48**
	Rajendran et al. [47]	1776	232	197	144	72	63
Saber	This work $3\times((\lceil\frac{256}{t}\rceil\times3)+\lceil\frac{9}{t})\rceil)$	2331	**237**	201	147	75	66
	Rajendran et al. [47]	–	390	–	–	–	–

4.1 Nature of the fault in the attack

In the previous sections, we discuss that our objective is to obtain the output $\mathcal{F}(K', \mathcal{H}(ct))$ by exploiting a fault, where K' is derived from the decrypted message m of the ciphertext ct. This fault uses the plaintext checking oracle $\mathcal{O}_\mu$. To achieve this, it is crucial to neutralize the effectiveness of comparing two ciphertexts, denoted as c and ct, in terms of equality checking. For all security levels of Saber and Kyber, the design employs a `verify` function that takes two ciphertexts, c and ct, along with their lengths and returns 0 if they are equal or 1 otherwise. The result is stored in a variable called "fail". In our attack, we construct ciphertexts in a particular pattern, ensuring that the ciphertexts c and ct are highly likely to be unequal. As a result, the variable "fail" will always be set to 1. This allows us to perform a bit-flip or get stuck at zero at the location of the "fail" variable, thus obtaining our desired output $\mathcal{F}(K', \mathcal{H}(ct))$. If we observe that for our constructed ciphertext ct, the value of the shared key is different from $\mathcal{F}(z, \mathcal{H}(ct))$. At this time, we are ensured that the value of the "fail" variable has changed to 0, and this value is our essential shared key $\mathcal{F}(K', \mathcal{H}(ct))$.

A stuck-at-zero fault is where a signal or a specific bit within a circuit is constantly held at logic zero. This fault can occur due to manufacturing defects, electrical shorts, environmental factors, or other physical issues. In contrast, a bit-flips fault involves the unintentional change of a single bit within a circuit or memory location from its intended value to the opposite value. Both stuck-at-zero and bit-flip faults can have various causes and implications. It is important to note that the specific type and cause of these faults can vary depending on the context, such as the hardware or software implementation, the cryptographic scheme used, and the fault injection techniques employed. Stuck-at-zero and bit-flip faults can lead to unexpected behaviour, data corruption, security vulnerabilities, or system crashes. To ensure system reliability and data integrity, detecting and mitigating these faults often involves employing error detection and correction techniques, such as error-correcting codes, redundant storage methods, or fault-tolerant designs.

In this paper, we choose Dynamic Random Access Memory (DRAM) reliability issue *Rowhammer* to introduce a software-driven hardware fault attack to induce a bit-flip ($1\rightarrow0$) at the address of the "fail" variables. We also present a series of steps that could be followed to incorporate this fault at a precise location in realistic timeframes.

	Model name	RAM size
1.	Intel (R) Core (TM) i7-4770 CPU	4 GB
2.	Intel (R) Core (TM) i7-3770 CPU	8 GB
3.	Intel (R) Core (TM) i5-3330 CPU	4 GB

Table 5: Model details of our target devices

4.2 Our target devices

To demonstrate our attack, we employ a deliberate technique of inducing bit-flips during the decapsulation process of Kyber. In our model, the attacker is assumed to be colocated in the same server as the victim, which performs the decapsulation process of Kyber and Saber. This scenario can also be extended to multiple virtual machines operating on a shared server. In this model, the primary assumption is that the victim and the attacker are co-located on the same physical piece of memory hardware, typically a DRAM and the vulnerable locations are neighbors to each other. This model exists in the current research field of row hammer [16,23] and is also consistent with most microarchitectural attacks [17]. Furthermore, since Kyber and Saber are designed as a CCA-secure scheme, our attack assumes that the attacker can often query the decapsulation process with the constructed ciphertext. We demonstrate our attack against the machines listed in Table 5.

4.3 Probabilities of incorporating precise fault using random Rowhammer

The task of incorporating bit-flips in random locations in memory is common and is very well studied in literature after Rowhammer has been reported in practice, but the hard part is to precisely induce the faults in the location of one's choice. In this paper, considering the target example, if we run the target code of Kyber/Saber multiple times in one process and an unsupervised row hammer code in another process, the address of the variable "fail" coinciding with one of the vulnerable locations, the probability of such event occurrence is considerably low. Suppose there are a total N number of vulnerable locations after hammering randomly among N_1 locations present on a device. Then, the possibility of the variable "fail" being vulnerable $=$ Pr(the location of "fail"=X) $\times$ Pr("fail" coincide in a vulnerable location |the location of "fail"=X) $= \frac{1}{N_1} \times \frac{N}{N_1} = \frac{N}{N_1^2}$, which is very low as $N_1 \gg N$. In our system, we randomly access $N_1 = 2^{30}$ bytes of memory; we discovered $N < 10$ vulnerable locations by accessing the memory randomly. Notably, the number of vulnerable locations (N) is considerably smaller than the total memory access. In order to make this process deterministic, we follow the steps described below.

Using the deterministic process of Rowhammer: We have used the *hammertime* code[4] available at [57] to execute row hammering operations. Through our exploration, we have observed that *hammertime* is a valuable simulator, offering a convenient approach to deterministically evaluate vulnerable locations. This versatile tool is

[4] "https://github.com/vusec/hammertime.git"

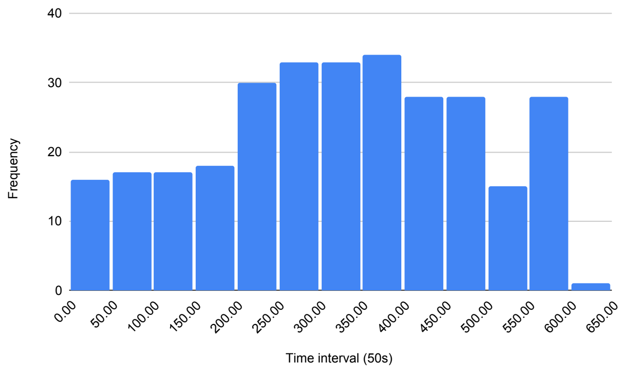

Fig. 5: Frequency of bit-flips in every 50 second

purpose-built for testing, profiling, and simulating the Rowhammer DRAM attack, providing a comprehensive suite of capabilities for assessing the outcomes of exploits.

The provided code presents two types of row hammering techniques: single-sided row hammering and double-sided row hammering. We have employed the single-sided row hammering process outlined in their code for our implementation. In each iteration of this approach, our target is to find the vulnerable rows from an aggressive row's upper or lower rows. Also, in our victim machine, the bit-flip occurs considerably frequently. Figure 5 shows the bit-flip frequency in every 50 second. In the hammertime code, we observe that this code deterministically selects an aggressive row, then fills up the memory with values all 1's (`"0xff"`) in the aggressive row and its neighbouring rows, and repeatedly flushes the corresponding portions of cache memory allocation. Iteratively, it only accesses the addresses with offsets $A = \{a_i\}_i$, where $a_0 = 0$ and $a_{i+1} - a_i = \texttt{0x020}$, for all i to check the bit-flip result. So, we can get the vulnerable address with the offset lie in A by running the hammering code. Figure 6 shows the offsets of the bitflip addresses and their frequency observed in our experiments. We perform a first-level templating of main memory using the *hammertime* code as shown in Figure 6, identifying locations that are vulnerable to Rowhammer. This templating step also aids us in identifying trigger rows so that we can replicate Rowhammer deterministically by re-accessing those aggressor rows again over time. By using the hammering code, we get the vulnerable addresses having different offsets and construct the set A. In this particular attack algorithm, we want the adversary to induce a bit-flip to a known vulnerable location. In order to achieve that, the variable in the decapsulation process (target "fail" variable) must coincide with atleast one offset in the set A of vulnerable addresses in order to precisely induce the fault. In order to increase the reproducibility of the attack over multiple runs, we have assigned the datatype of the variable "fail" in our implementation to "static int" rather than simply using "int". Doing so guarantees that the offset of the "fail" variable remains unchanged throughout the execution. Without loss of generality, if our attack methodology is implemented on any other target secret, then a similar technique could be applied to any global variable or a local

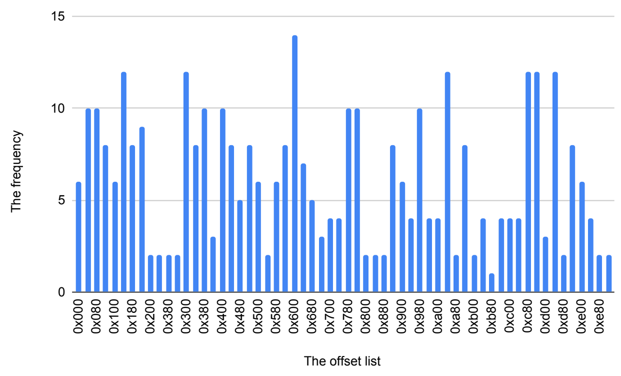

Fig. 6: Frequency of the bit-flips in the corresponding offset.

variable with a static flag for the sake of the reproducibility of our attack. We consider the offset `0x040` of the "fail" variable, which was observed on our executable. This offset can be any value without loss of generality in Kyber/ Saber's implementation, and the appropriate matching offset of the Rowhammer fault can also be selected from the templating phase. In our attack scenario, we select the vulnerable locations offset of `0x040` to show the vulnerability. We construct the following template shown in Figure 7.

The templating method in Rowhammer provides a method that induces a bit-flip from 1 to 0 at the "fail" variable. First, we run the *hammertime* code and observe a bit-flip ($1 \rightarrow 0$) at an address with the offset `0x040`. In this phase, we proceed to unmap the corresponding page of that address and emit a signal, enabling us to execute the victim code in process 2. With a high likelihood, the victim code gets mapped to the unmapped page just being freed by the *hammertime* executable. This will allow the "fail" variable to be sitting in the Rowhammer vulnerable location of the unmapped page. The scenario of page reallocation of the recently unmapped page is commonly encountered using the Page Frame Cache during page allocations involving the buddy allocator [16].

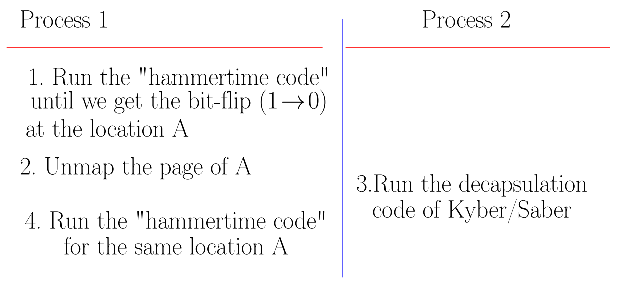
Process 1
Process 2
1. Run the "hammertime code" until we get the bit-flip (1→0) at the location A
2. Unmap the page of A
3.Run the decapsulation code of Kyber/Saber
4. Run the "hammertime code" for the same location A

Fig. 7: Template of generating oracle $\mathcal{O}_\mu$ using Rowhammer

After successfully aligning the "fail" variable with the vulnerable location of Rowhammer, our objective is to actually induce the fault in the target location to change its value to "0". To accomplish this, we need to continue performing row hammering on the same aggressive row that inflicted the Rowhammer in the templating phase. This ensures that the bit-flip occurs at the same vulnerable address, which is now unmapped from the hammering code, but possessed by the target executable of the victim. To achieve this, we made some modifications to the *hammertime* tool, and iterated through the following processes.

An extra *loop* is added inside the *profile_ singlesided* function. Once the target page is unmapped, only then this *loop* will run. The *loop* contains minor modifications to the following functions *fill_ rows* and *c->hamfunc*. This modification involves a checking condition that inside the function *fill_ rows*, we ignore the addresses lying on the unmapped page. This function activates aggressive rows and neighbor rows and as a result, the vulnerable address is affected, leading to a change in its bit from "1" to "0".

After unmapping the page, we run the victim code (decapsulation process with our constructed ciphertext) parallel to the *hammertime code* until we observe the faulty shared key. If we observe a different shared key, then the Rowhammer attempt has been successful and we stop this process. We summarize the whole process as follows:

1. By running the hammering code, vulnerable addresses with offsets from set A are identified, and the "fail" variable is positioned to coincide with one of these vulnerable addresses. A suitable vulnerable location is selected and the corresponding page is unmapped from the code.
2. After unmapping the page, we run the victim code until we do not get the faulty shared key. If we get a different shared key, then we are done.
3. To achieve a bit-flip from "1" to "0" at the "fail" variable, row hammering is continued on the same aggressive row, modifying the fill_row function to fill memory with "0xff" and performing a memory flush on all addresses except the unmapped page corresponding to the vulnerable address.

Figure 8 illustrates the distribution of timings observed for the Rowhammer bit-flip to occur at the vulnerable location through the *hammertime* code after unmapping the vulnerable page. In order to estimate the total time to recover the whole secret key we need 57 independent queries to the oracle. This translates to 57 independent fault occurrences on the "fail" variable in the implementation of the decapsulation algorithm. One such occurrence can be estimated to happen in $<350ms$ with a significantly high probability. So this attack can be realised using an additive progression of timing on respective queries and can be observed in a linear timescale.

5 Discussion and future direction

In this paper, we show an end-to-end software-driven hardware fault on PQ LWE-based KEMs. We choose Saber and Kyber key encapsulation schemes and perform the fault analysis with as much as 39% reduced number of queries for Saber and approximately 23% for Kyber768 on the existing literature. This was achieved by pruning selected leaves of the decisional binary search tree used in the attack. The fault induction using

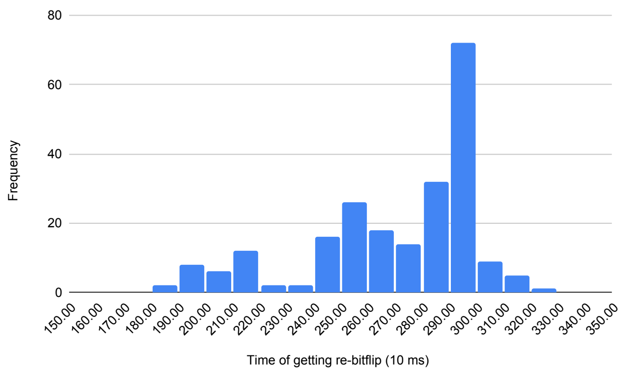

Fig. 8: The value of an interval $[a,b]$ is the number of bit-flips which takes the time $t \in [a,b]$, to make re-bitflip at the same address

the Rowhammer has been known in the literature to appear in random locations of memory due to the reliability issues of commercial DDR RAMs. We follow some precise steps by first templating the memory space, listing out vulnerable addresses of a system, and then precisely locating the target KEM implementation in that vulnerable location. In this context, we use publicly available *Hammertime* code to template the memory space, then make minor modifications to re-induce Rowhammer using the selected aggressor rows on that same location deterministically. This semi-deterministic process is highly useful in conjunction with the paging policies of the Buddy allocator, and then inflicting these bit-flips on the publicly available target implementation.

Though there has been recent work on Frodo KEM [23], where the authors incorporate fault in the key generation phase using Rowhammer. As discussed in Section 3, the key generation of a CCA-secure KEM is a one-time operation and is invoked rarely. Hence, if necessary the key generation can even be done offline in an isolated environment. On the other hand, the decapsulation of a CCA-secure KEM is invoked multiple times to generate the shared secret key from multiple sources. Therefore, in a practical scenario for the sake of performance, the decapsulation cannot run in an isolated environment. Therefore the attack described in [23] is far less realistic than our attack methodology. Further, the authors assume that they can slow down the execution by slowing down components of the target executable. This is already a strong assumption. Additionally, the authors have disabled ASLR (Address Space Layout Randomization) for their experiments which makes the assumptions even stronger and the attack more unrealistic.

5.1 Shuffling and Masking:

Previous attacks [56,47] based on parallel plaintext checking oracle have used side-channel analysis such as EM power analysis. So, these attacks can be prevented using masking countermeasures [48]. Our attack can be conducted on the masked or shuffled implementation of the LWE-based KEMs. Because here, we do not use any side-channel assistance to perform the attack. We induce a bitflip fault to the "fail" variable, which stores the result of the comparison between the public ciphertext and the re-encrypted ciphertext. As a result of this fault, the value of the "fail" variable always remains 0, and that causes decapsulation success. When applying side-channel countermeasures such as masking and shuffling on the decapsulation algorithm of LWE-based KEMs [15,58,35], this fail variable remains unaffected and unmasked, since it is not dependent on the secret. Therefore, the success of our attack does not get affected by generic side-channel countermeasures such as masking or shuffling.

5.2 Extension of our attack on other PQC schemes:

The parallel plaintext checking oracle used in our attack model can be applicable to any LWE-based KEMs. It is not specific to Saber and Kyber. It can be applicable to other LWE-based schemes such as NewHope [3], Lizard citeCheonKLS18, Round5 [9], Frodo [13], Smaug [18](proposed in the ongoing Korean PQC [34] competition), etc. The Rowhammer methodology we propose in this work to introduce fault can also be applicable to other fault attack models where a single or multi-bit fault is required. Popular side-channel countermeasures such as masking and shuffling are ineffective to protect against this attack.

5.3 Combining of lattice reduction techniques with our attack:

There can be some cases when the attacker only has a limited number of accesses to the decapsulation procedure. Then, the attacker can use our attack to recover some of the coefficients of the secret key and then use lattice reduction techniques to recover the rest of the secret key [28]. The LWE-estimator toolbox [2,19] can provide an estimate on the computation effort required to recover the secrets using the lattice reduction techniques. It is up to the attacker to determine the optimum point till when our attack should be stopped and the lattice reduction methods should be used. However, more investigation is needed to combine our attack results with these LWE-estimators to efficiently recover the secret key. We would like to investigate it in the future.

5.4 Possible countermeasures

Although masking or shuffling countermeasures are unable to prevent our attack, there are a few countermeasures that can be useful to thwart our attack. Below, we list these countermeasures in two categories.

- Fault attack countermeasure on the LWE-based schemes: Recently, Berthet et al. [11] propose a countermeasure named quasi-linear masking on Kyber to prevent fault injection attacks together with side-channel attacks. This countermeasure might be used to prevent our attack.
- Rowhammer Countermeasures: There have been various countermeasures of RowHammer attacks proposed in the literature. The authors in the paper [33] proposed Probabilistic Adjacent Row Activation (PARA), where the memory

controller is designed to refresh its adjacent rows with probability p (typically 1/2). The memory controller being probabilistic, the approach does not require a complex data structure for counting the number of row activations. Earlier in [53], it was shown that doubling the refresh rate and removing access to clflush instruction are potential prevention techniques to RowHammer. An interesting countermeasure to rowhammer has been proposed in Anvil [8]. If the cache misses over a time interval is observed to be significantly high, then the software module triggers sampling of the DRAM accesses. ANVIL selectively performs a row refresh if the software module detects repeated accesses to particular rows in the same bank. Another process, Target Row Refresh (TRR), believed to be a definitive solution, can prevent RowHammer bit flips [39] [1]. However, in the paper [26], the authors also find that consumer CPUs rely on in-DRAM TRR and are vulnerable to many-sided RowHammer attacks. They introduce TRRespass, which can autonomously discover intricate hammering patterns to launch real-world attacks on numerous DDR4 DRAM modules available in the market. Till now, there is no concrete solution that can prevent the RowHammer bit flip problem. [1] J.-B. Lee, "Green Memory Solution," in Samsung Electronics, Investor's Forum, 2014.

Acknowledgements

This work was supported in part by Horizon 2020 ERC Advanced Grant (101020005 Belfort), CyberSecurity Research Flanders with reference number VR20192203, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), and Intel Corporation.

Angshuman Karmakar is funded by FWO (Research Foundation – Flanders) as a junior post-doctoral fellow (contract number 203056 / 1241722N LV). Puja Mondal and Angshuman Karmakar are also supported by C3iHub, IIT Kanpur.

References

1. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Online. Accessed 26th June, 2023 (2022), `https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf`
2. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of Learning with Errors. Cryptology ePrint Archive, Report 2015/046 (2015), `https://eprint.iacr.org/2015/046`
3. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum Key Exchange - A New Hope. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 327–343. USENIX Association (2016), `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim`
4. Aranha, D.F., Fouque, P.A., Gérard, B., Kammerer, J.G., Tibouchi, M., Zapalowicz, J.C.: GLV/GLS Decomposition, Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014. pp. 262–281. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

5. Aranha, D.F., Novaes, F.R., Takahashi, A., Tibouchi, M., Yarom, Y.: LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 225–242. CCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3417268, https://doi.org/10.1145/3372297.3417268
6. Aumasson, J.P., Bernstein, D.J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: SPHINCS+: Stateless hash-based signatures, https://sphincs.org/, [Online; accessed 28-June-2023]
7. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Kaliski, B.S., Koç, ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002. pp. 260–275. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
8. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T.: ANVIL: Software-based protection against next-generation rowhammer attacks. ACM SIGPLAN Notices **51**(4), 743–755 (2016)
9. Baan, H., Bhattacharya, S., Fluhrer, S.R., García-Morchón, Ó., Laarhoven, T., Rietman, R., Saarinen, M.O., Tolhuizen, L., Zhang, Z.: Round5: Compact and Fast Post-quantum Public-Key Encryption. In: Ding, J., Steinwandt, R. (eds.) Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers. Lecture Notes in Computer Science, vol. 11505, pp. 83–102. Springer (2019). https://doi.org/10.1007/978-3-030-25510-7_5, https://doi.org/10.1007/978-3-030-25510-7_5
10. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom Functions and Lattices. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7237, pp. 719–737. Springer (2012). https://doi.org/10.1007/978-3-642-29011-4_42, https://doi.org/10.1007/978-3-642-29011-4_42
11. Berthet, P., Tavernier, C., Danger, J., Sauvage, L.: Quasi-linear Masking to Protect Kyber against both SCA and FIA. IACR Cryptol. ePrint Arch. p. 1220 (2023), https://eprint.iacr.org/2023/1220
12. Biehl, I., Meyer, B., Müller, V.: Differential Fault Attacks on Elliptic Curve Cryptosystems. In: Bellare, M. (ed.) Advances in Cryptology — CRYPTO 2000. pp. 131–146. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
13. Bos, J.W., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., Stebila, D.: Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1006–1018. ACM (2016). https://doi.org/10.1145/2976749.2978425, https://doi.org/10.1145/2976749.2978425
14. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Stehlé, D.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM (2017), http://eprint.iacr.org/2017/634
15. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: First- and Higher-Order Implementations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 173–214 (2021). https://doi.org/10.46586/tches.v2021.i4.173-214, https://doi.org/10.46586/tches.v2021.i4.173-214

16. Chakraborty, A., Bhattacharya, S., Saha, S., Mukhopadhyay, D.: ExplFrame: Exploiting Page Frame Cache for Fault Analysis of Block Ciphers. In: 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020. pp. 1303–1306. IEEE (2020). https://doi.org/10.23919/DATE48585.2020.9116219, https://doi.org/10.23919/DATE48585.2020.9116219
17. Chakraborty, A., Bhattacharya, S., Saha, S., Mukhopdhyay, D.: Rowhammer Induced Intermittent Fault Attack on ECC-hardened memory (2020), https://eprint.iacr.org/2020/380
18. Cheon, J.H., Choe, H., Hong, D., Yi, M.: SMAUG: Pushing Lattice-based Key Encapsulation Mechanisms to the Limits. Cryptology ePrint Archive, Paper 2023/739 (2023), https://eprint.iacr.org/2023/739, https://eprint.iacr.org/2023/739
19. Dachman-Soled, D., Ducas, L., Gong, H., Rossi, M.: LWE with Side Information: Attacks and Concrete Security Estimation. Cryptology ePrint Archive, Report 2020/292 (2020), https://eprint.iacr.org/2020/292
20. Daemen, J., Rijmen, V.: Rijndael for AES. In: The Third Advanced Encryption Standard Candidate Conference, April 13-14, 2000, New York, New York, USA. pp. 343–348. National Institute of Standards and Technology, (2000)
21. D'Anvers, J., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM (2018), http://eprint.iacr.org/2018/230
22. Ducas, L., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Dilithium: Digital Signatures from Module Lattices (2017), http://eprint.iacr.org/2017/633
23. Fahr, M., Kippen, H., Kwong, A., Dang, T., Lichtinger, J., Dachman-Soled, D., Genkin, D., Nelson, A., Perlner, R., Yerukhimovich, A., Apon, D.: When frodo flips: End-to-end key recovery on frodokem via rowhammer. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 979–993. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3560673, https://doi.org/10.1145/3548606.3560673
24. Fan, H., Wang, W., Wang, Y.: Cache attack on MISTY1. IACR Cryptol. ePrint Arch. p. 723 (2021), https://eprint.iacr.org/2021/723
25. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU (2018), https://falcon-sign.info/falcon.pdf, [Online; accessed 28-June-2023]
26. Frigo, P., Vannacci, E., Hassan, H., van der Veen, V., Mutlu, O., Giuffrida, C., Bos, H., Razavi, K.: TRRespass: Exploiting the Many Sides of Target Row Refresh. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 747–762. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00090, https://doi.org/10.1109/SP40000.2020.00090
27. Fujisaki, E., Okamoto, T.: Secure Integration of Asymmetric and Symmetric Encryption Schemes. J. Cryptol. **26**(1), 80–101 (2013). https://doi.org/10.1007/s00145-011-9114-1, https://doi.org/10.1007/s00145-011-9114-1
28. Gama, N., Nguyen, P.Q.: Predicting lattice reduction. In: Smart, N.P. (ed.) Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4965, pp. 31–51. Springer (2008). https://doi.org/10.1007/978-3-540-78967-3_3, https://doi.org/10.1007/978-3-540-78967-3_3

29. Guo, Q., Johansson, T., Nilsson, A.: A Key-Recovery Timing Attack on Post-quantum Primitives Using the Fujisaki-Okamoto Transformation and Its Application on FrodoKEM. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12171, pp. 359–386. Springer (2020). https://doi.org/10.1007/978-3-030-56880-1_13
30. Hermelink, J., Pessl, P., Pöppelmann, T.: Fault-Enabled Chosen-Ciphertext Attacks on Kyber. In: Adhikari, A., Küsters, R., Preneel, B. (eds.) Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13143, pp. 311–334. Springer (2021). https://doi.org/10.1007/978-3-030-92518-5_15
31. Islam, S., Mus, K., Singh, R., Schaumont, P., Sunar, B.: Signature Correction Attack on Dilithium Signature Scheme. In: 7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022. pp. 647–663. IEEE (2022). https://doi.org/10.1109/EuroSP53844.2022.00046, https://doi.org/10.1109/EuroSP53844.2022.00046
32. Jiang, H., Zhang, Z., Chen, L., Wang, H., Ma, Z.: Post-quantum IND-CCA-secure KEM without Additional Hash. Cryptology ePrint Archive, Report 2017/1096 (2017), https://eprint.iacr.org/2017/1096
33. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. ACM SIGARCH Computer Architecture News **42**(3), 361–372 (2014)
34. KpqC: Korean post-quantum cryptography competition (2022), https://www.kpqc.or.kr/competition.html, [Online; accessed 28-June-2023]
35. Kundu, S., D'Anvers, J., Beirendonck, M.V., Karmakar, A., Verbauwhede, I.: Higher-Order Masked Saber. In: Galdi, C., Jarecki, S. (eds.) Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13409, pp. 93–116. Springer (2022). https://doi.org/10.1007/978-3-031-14791-3_5, https://doi.org/10.1007/978-3-031-14791-3_5
36. Kwong, A., Genkin, D., Gruss, D., Yarom, Y.: Rambleed: Reading bits in memory without accessing them (05 2020). https://doi.org/10.1109/SP40000.2020.00020
37. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Des. Codes Cryptogr. **75**(3), 565–599 (2015). https://doi.org/10.1007/s10623-014-9938-4, https://doi.org/10.1007/s10623-014-9938-4
38. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Gilbert, H. (ed.) Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6110, pp. 1–23. Springer (2010). https://doi.org/10.1007/978-3-642-13190-5_1, https://doi.org/10.1007/978-3-642-13190-5_1
39. Micron: DDR4 SDRAM Datasheet (2016)
40. Miller, V.S.: Use of Elliptic Curves in Cryptography. In: Williams, H.C. (ed.) Advances in Cryptology — CRYPTO '85 Proceedings. pp. 417–426. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
41. Mujdei, C., Beckers, A., Bermundo, J., Karmakar, A., Wouters, L., Verbauwhede, I.: Side-Channel Analysis of Lattice-Based Post-Quantum Cryptography: Exploiting Polynomial Multiplication. IACR Cryptol. ePrint Arch. p. 474 (2022), https://eprint.iacr.org/2022/474

42. Mus, K., Islam, S., Sunar, B.: QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 1071–1084. CCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3417272, https://doi.org/10.1145/3372297.3417272
43. Mutlu, O., Kim, J.S.: RowHammer: A Retrospective. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(8), 1555–1571 (2020). https://doi.org/10.1109/TCAD.2019.2915318, https://doi.org/10.1109/TCAD.2019.2915318
44. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings. pp. 1–20. Springer (2006)
45. Pessl, P., Prokop, L.: Fault Attacks on CCA-secure Lattice KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2), 37–60 (2021). https://doi.org/10.46586/tches.v2021.i2.37-60, https://doi.org/10.46586/tches.v2021.i2.37-60
46. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. Quantum Inf. Comput. **3**(4), 317–344 (2003). https://doi.org/10.26421/QIC3.4-3, https://doi.org/10.26421/QIC3.4-3
47. Rajendran, G., Ravi, P., D'Anvers, J., Bhasin, S., Chattopadhyay, A.: Pushing the Limits of Generic Side-Channel Attacks on LWE-based KEMs - Parallel PC Oracle Attacks on Kyber KEM and Beyond. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2023**(2), 418–446 (2023). https://doi.org/10.46586/tches.v2023.i2.418-446, https://doi.org/10.46586/tches.v2023.i2.418-446
48. Ravi, P., Chattopadhyay, A., Baksi, A.: Side-channel and Fault-injection attacks over Lattice-based Post-quantum Schemes (Kyber, Dilithium): Survey and New Results. IACR Cryptol. ePrint Arch. p. 737 (2022), https://eprint.iacr.org/2022/737
49. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic Side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3), 307–335 (2020), https://doi.org/10.13154/tches.v2020.i3.307-335
50. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip Feng Shui: Hammering a Needle in the Software Stack. In: Proceedings of the 25th USENIX Conference on Security Symposium. p. 1–18. SEC'16, USENIX Association, USA (2016)
51. Regev, O.: Lecture notes: Lattices in computer science, https://cims.nyu.edu/~regev/teaching/lattices_fall_2009
52. Rivest, R.L., Shamir, A., Adleman, L.M.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Commun. ACM **21**(2), 120–126 (1978). https://doi.org/10.1145/359340.359342, http://doi.acm.org/10.1145/359340.359342
53. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges. Black Hat **15**, 71 (2015)
54. Settana, M., Naila, A., Yaseen, H., Huwaida, T.: Cache-Timing Attack against AES Crypto-Systems Countermeasure Using Weighted Average Masking Time Algorithm. Journal of Information Warfare **15**(1), 104–114 (2016), https://www.jstor.org/stable/26487484
55. Shor, P.W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994. pp. 124–134. IEEE Computer Society (1994). https://doi.org/10.1109/SFCS.1994.365700, https://doi.org/10.1109/SFCS.1994.365700
56. Tanaka, Y., Ueno, R., Xagawa, K., Ito, A., Takahashi, J., Homma, N.: Multiple-Valued Plaintext-Checking Side-Channel Attacks on Post-Quantum KEMs (2022), https://eprint.iacr.org/2022/940

57. Tatar, A., Giuffrida, C., Bos, H., Razavi, K.: Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11050, pp. 47–66. Springer (2018). https://doi.org/10.1007/978-3-030-00470-5_3, https://doi.org/10.1007/978-3-030-00470-5_3
58. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.: Shuffling against side-channel attacks: A comprehensive study with cautionary note. In: Wang, X., Sako, K. (eds.) Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7658, pp. 740–757. Springer (2012). https://doi.org/10.1007/978-3-642-34961-4_44
59. Xiao, Y., Zhang, X., Zhang, Y., Teodorescu, R.: One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 19–35. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao
60. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: Fu, K., Jung, J. (eds.) Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 719–732. USENIX Association (2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

Supplementary material

A Rowhammer

Rowhammer is a phenomenon observed in dynamic random-access memory (DRAM) where repeated access to a specific row can result in bit flips occurring in neighboring rows [43]. This happens because the capacitors in neighboring rows, responsible for storing bit values, discharge slightly due to parasitic currents when the row's word line is activated if this discharge happens frequently enough to lower the voltage below the "charged" threshold before the regular DRAM refresh, which typically occurs every 64ms, the logical value of the bit can flip. There are two types of methods for row hammering: single-sided rowhammering and double-sided rowhammering.

Single-sided Rowhammering: Single-sided rowhammering is a technique where an attacker repeatedly accesses memory rows on only one side of a target row without accessing the rows above or below within the same bank. By rapidly accessing specific rows, the attacker aims to induce electrical interference and disturb neighboring cells, potentially causing bit flips and altering data stored in adjacent rows. This technique relies on the inherent electrical interactions between closely located memory cells, taking advantage of the sensitivity of DRAM cells to repeated accesses.

Double-sided Rowhammering: Double-sided rowhammering is a more aggressive variant of the rowhammering technique. In this approach, an attacker repeatedly accesses memory rows above and below a target row within the same bank. By accessing these rows simultaneously, the attacker intensifies the electrical interactions and disturbances within the DRAM cells. This increases the likelihood of inducing bit flips in the target and adjacent rows. Double-sided row hammer leverages the interplay of electrical charges in closely positioned memory cells to exploit the vulnerability of DRAM and manipulate data stored in memory.

B Fujisaki-Okamoto (FO) transformation

We use this FO transformation proposed by [32] to construct a CCA secure key-encapsulation mechanism (KEM) over the CPA secure `LPR.PKE` scheme. The algorithms of this KEM are shown in Figure 2. The KEM algorithm contains three algorithms: (i) key generation (`KEM.KeyGen`), (ii) encapsulation (`KEM.Encaps`), and (iii) decapsulation (`KEM.Decaps`). The algorithm `KEM.KeyGen` produces a public key pk and a secret key sk' by employing the `LPR.PKE.KeyGen` algorithm. In this context, sk refers to the secret key generated through the `LPR.PKE.KeyGen` algorithm, z is a random bit string of length n, and $\mathcal{H}$ is a hash function. The secret key sk' is computed by concatenating sk, pk, $\mathcal{H}(pk)$, and z. The `KEM.Encaps` algorithm takes the public key pk as input and generates a random message bit string m of length n. Then it uses the hash function $\mathcal{G}$ to compute K' and a random coin string r. After that, it encrypts the message m using the `LPR.PKE.Enc` algorithm with public key pk, message m, and random coin string r to produce the ciphertext ct. Finally, it applies a function $\mathcal{F}$ to K' and the ciphertext ct to produce the shared key K. The `KEM.Decaps` algorithm takes the ciphertext ct and secret key sk' as inputs. It first decrypts the ciphertext using the `LPR.PKE.Dec` algorithm with secret key sk and ciphertext ct to produce the decrypted message m'. It then re-encrypts the message m' using the `LPR.PKE.Enc` algorithm with public key pk and random coin string r to produce the ciphertext c. It then checks whether ct and c are equal. If they are, it applies the function $\mathcal{F}$ to K' and the hash of the ciphertext $\mathcal{H}(ct)$ to produce the shared key K. Otherwise, it applies the function $\mathcal{F}$ to the random bit string z and the hash of the ciphertext $\mathcal{H}(ct)$ to produce an invalid shared key K. Finally, it returns the shared key K.