IACR
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–22. DOI:XXXXXXXX

Find the Bad Apples: An efficient method for perfect key recovery under imperfect SCA oracles – A case study of Kyber

Muyan Shen[1], Chi Cheng[1,✉], Xiaohan Zhang[1], Qian Guo[2] and Tao Jiang[3]

[1] China University of Geosciences, Wuhan, 430074, China
chengchizz@qq.com
[2] Lund University, Lund, Sweden,
[3] Huazhong University of Science and Technology

Abstract. Side-channel resilience is a crucial feature when assessing whether a post-quantum cryptographic proposal is sufficiently mature to be deployed. In this paper, we propose a generic and efficient adaptive approach to improve the sample complexity (i.e., the required number of traces) of plaintext-checking (PC) oracle-based side-channel attacks (SCAs), a major class of key recovery chosen-ciphertext SCAs on lattice-based key encapsulation mechanisms. This new approach is preferable when the constructed PC oracle is imperfect, which is common in practice, and its basic idea is to design new detection codes that can determine erroneous positions in the initially recovered secret key. These secret entries are further corrected with a small number of additional traces. This work benefits from the generality of PC oracle and thus is applicable to various schemes and implementations. We instantiated the proposed generic attack framework on Kyber512 and fully implemented this attack instance. Through extensive computer simulations and also a real-world experiment with electromagnetic (EM) leakages from an ARM-Cortext-M4 platform, we demonstrated that the newly proposed attack could greatly improve the state-of-the-art in terms of the required number of traces. For instance, the new attack requires only 41% of the EM traces needed in a majority-voting attack in our experiments, where the raw oracle accuracy is fixed.

Keywords: Lattice-based cryptography · Side-channel attacks · Plaintext-checking oracle · NIST Post-Quantum cryptography standardization · Kyber · Key mismatch attacks.

1 Introduction

To continue protecting our data once quantum computers become mature, we need to transit from current factoring or discrete-log-based public key cryptography to post-quantum cryptography (PQC). Just recently, the US National Institute of Standards and Technology (NIST) and the Department of Homeland Security (DHS) have collaborated and released a roadmap to transition to the PQC standard [ND21], which is anticipated to be ready by 2024. Their goal is to complete the transition by 2030.

Starting in 2016, NIST's PQC selection process has attracted attention from all over the world [Moo16]. Currently, there are 4 finalists and 5 alternative candidates for Public Key Encryption (PKE) or Key Encapsulation Mechanism (KEM) on the third-round list [MAA+20]. Lattice-based KEMs occupy the majority, i.e., 3 out of 4 finalists, demonstrating their fundamental role in the PQC standard. One of the most promising candidates among them is Kyber [ABD+19], the KEM part of the Cryptographic Suite for Algebraic Cipher Suite (CRYSTALS). Besides other desirable security properties, NIST

has placed a priority on the resistance of these KEMs to side-channel attacks (SCAs) before deploying these PQC algorithms in the real world, especially in the scenario where an attacker can physically access an embedded device.

SCAs were firstly introduced by Kocher in 1996 [Koc96]. By focusing on side-channel measurements from the timing, power consumption, or electromagnetic (EM) emanation, the implemented cryptographic algorithms leak information about the long-term secret key or message. In the light of this thought, several SCAs against lattice-based KEMs in the NIST PQC standardization process have been proposed (e.g., [DTVV19, GJN20, RRCB20, XPSR+21, NDGJ21, HHP+21, REB+22, UXT+22]). Most of them are chosen-ciphertext attacks (CCAs) since NIST PQC KEMs generally target provable CCA-security, which can be achieved by using the Fujisaki-Okamoto (FO) transformation.

Key recovery SCAs against lattice-based KEMs recovering the long-term secret key is a central research topic in this field since key recovery is a much stronger attack model than only message recovery. We can classify these attacks into two main types. The first type of attack [GJN20, BDH+21] builds an oracle to check whether the decryption is successful, thus being closer to reaction attacks [Ble98, HGS99, MU10], an attack model weaker than the CCA. We call this class of attacks reaction-type SCAs. This type of attack is generic, and there is no need to design message-recovery techniques.

The second type of attack, initially proposed by D'Anvers et al. [DTVV19], connects the entries in the long-term secret key to certain chosen messages and achieves key recovery through a message-recovery approach. We call this class of attacks message-recovery-type SCAs. The initial message-recovery-type attacks [DTVV19, RRCB20], also named plaintext-checking (PC) oracle-based SCA in [RR21], can only gain at most one bit of the secret information from one decryption function call. Later, more advanced attacks [XPSR+21, NDGJ21, REB+22] were discovered, which could recover a large chunk of the message/secret vectors simultaneously. However, it is worth noting that the latter are much more powerful but rely on strong leakages from specific implementations. For example, Xu et al. used only 4 traces to get full-key recovery for reference C version of Kyber512, but the traces rise to hundreds when targeting the assembly-optimized version [XPSR+21]. The compiler-optimization levels also make a huge impact on the number of traces, resulting in 8 traces (-O0) and 960 traces (-O3), respectively. Just recently, Ueno et al. showed that the generality of PC oracle could help launch a generic SCA against most NIST PQC candidate KEMs [UXT+22]. What makes their work more interesting is the realization of a deep-learning-based distinguisher with high accuracy, which helps build the PC oracle to attack the lattice-based KEMs even when there are SCA counter-measurements like masking in the implementation.

Although there has been much work on the PC oracle-based SCAs, the way these attacks deal with oracle inaccuracy needs more attention. The oracle inaccuracy may occur due to the environmental noises, or simply the measurement limitations in implementing the PC oracle, such as the inaccuracy in the deep-learning-based SCA distinguisher in [ISUH21, UXT+22]. Since the practical SCA oracle cannot be perfect, the recovered secret key may have slight or big corruption due to the inaccuracy rate. To successfully recover the full or almost full secret key, previous works aim to improve the oracle accuracy with techniques like majority-voting. Compared with the recovery under a perfect oracle, we need more traces under an imperfect oracle. For example, in [RRCB20], Ravi et al. needs 2560 traces to recover the full key of Kyber512 when the PC oracle is perfect. But to migrate the measurement errors of SCA, they repeatedly query the oracle three times and then use the majority-voting to recover the full key. This causes three times total traces, i.e. $2560 \cdot 3 = 7680$. Therefore, an interesting problem is, can we find a more efficient way to launch a perfect recovery under an imperfect oracle?

In this paper, we investigate the message-recovery-type SCA against Kyber, aiming to improve the sample complexity when an imperfect measuring is assumed. Our focal

point is the PC oracle-based SCAs, as these attacks have more generic applications (e.g., [RDB+21]) compared with its improved version that can in parallel recover a large number of bits of information. Also, PC oracle-based SCAs could exploit a long leakage trace (e.g., corresponding to the whole FO transform) to recover just one bit of the secret information, making such attacks difficult to be thwarted [ABH+22]. This research is of great practical significance since real platforms can be either too noisy, or incur measurement errors, to seriously hurt the accuracy of measurements.

1.1 Contributions

We present a new *checking* approach in the PC oracle-based SCAs to efficiently find the problematic entries in the recovered secret key. These entries are further corrected with a small number of additional traces. Compared with the most used method performing majority-voting with multiple traces to increase the attack success probability, the new adaptive method shows a substantial improvement in sample complexity.

The main contributions of this paper are summarized in the following.

- Firstly, we propose a general SCA framework with improved sample complexity that could be widely applied for attacking NIST lattice-based KEMs. In the high-oracle-accuracy region, we treat the detection of corruptions as a coding problem and propose an efficient method to find the erroneous locations. The novel idea is that if the targeted secret block is erroneously recovered, then the check procedure will return a codeword different from the designed ones. We then extend the attack to the low-oracle-accuracy region and propose a new smart approach called *mixed voting* to improve the decision accuracy using a confidence array.

- Furthermore, we instantiate the described attack framework on Kyber512 and show the details in each step of the new procedure.

- We perform extensive simulations for various oracle-accuracy levels and mount a real-world EM attack against the *pqm4* optimized implementation of Kyber512 running on a STM32 platform with an ARM Cortex-M4 microcontroller. Our experimental results show that the new checking approach can improve the majority-voting method significantly, i.e., the required number of traces is approximately halved for the simulated instances. We make our code and data open-source. They are available at https://github.com/7a17/Find-the-Bad-Apples/.

1.2 Related Works

In [RR21], Ravi and Roy categorized the major SCAs on lattice-based KEMs into three classes, decryption-failure (DF) oracle-based, PC or key-mismatch oracle-based, and full-decryption (FD) oracle-based. DF oracle-based SCA is reaction-type, while PC and FD oracle-based SCAs are message-recovery-types.

DF-oracle-based SCAs were initially proposed in [GJN20], where a generic SCA model focusing on the leakage of the FO transformation is presented. This attack model was instantiated as a timing attack [GJN20] on FrodoKEM and also timing attacks [GHJ+21] on code-based NIST candidates HQC and BIKE with variable execution time of the rejection sampling procedure. Attacks and protections against power/EM adversaries were further investigated in [BDH+21, UXT+22].

A series of similar analyses in the presence of PC oracle has also been presented. For example, D'Anvers et al. [DTVV19] exploited the variable runtime information of its non-constant-time decapsulation implementation on the LAC and successfully recovered its long-term secret key. At CHES 2020, Ravi et al. proposed a generic EM chosen-ciphertext SCA by exploiting the leaked information about Fujisaki-Okamoto (FO) transform or Error

Correcting Codes (ECC) and applied it to six CCA-secure lattice-based KEMs [RRCB20]. Afterwards, Qin et al. [QCZ+21] proposed a systematic approach to evaluate the key reuse resilience of CPA-secure lattice-based KEMs and further mounted it on side-channel attacks against the CCA-secure ones, which greatly reduced the needed side-channel traces/queries compared with Ravi et al. In [ZCD21], Zhang et al. investigated the key reuse resilience of the CPA-secure NTRU-HRSS KEM, whose method can also attack the CCA-secure NTRU-HRSS KEM with the help of side-channel leakages. Recently, Ravi et al. further successfully mounted the previous attacking method on two NTRU-based schemes, which are NTRU and NTRU Prime [REB+22].

The idea of FD oracle-based SCA is first proposed by Xu et al. [XPSR+21] and they demonstrated that an adversary only needs 8 traces in recovering the secret key of Kyber512 KEM for the pqm4 ARM-specific implementation at -O0. Compared with the PC-oracle-based SCA, Xu et al. can gain the complete message information for the chosen ciphertext, which is equivalent to launching multiple chosen-ciphertext attacks simultaneously. Subsequently, Ravi et al. fully exploited the vulnerabilities of the message decoding function on the lattice-based KEMs and launched the corresponding side-channel attacks for different implementations of CCA-secure lattice-based KEMs [RBRC21]. Their attack can be extended to several implementations with side-channel countermeasures such as shuffling and masking but only perform message recovery. More recently, there is a side-channel attack successfully recovering the long-term secret key for the masked implementation of SABER [NDGJ21], which removes Ravi et al.'s restriction.

Organizations. The remaining of the paper is organized as follows. We present the necessary background in Section 2 and the new improved approach in Section 3. This is followed by a concrete instantiation for attacking Kyber512 in Section 4 and the experimental results in Section 5. Finally, we conclude the work in Section 6.

2 Previous PC-based SCA against Kyber

2.1 Kyber and the PC oracle

The security of Kyber is based on the hardness of solving the module learning with errors (M-LWE) problem. From linear algebra, we know that it is easy to retrieve s from linear equations $\mathbf{b} = \mathbf{As}$. Here $\mathbf{A}$ is a matrix and $\mathbf{b}$ and $\mathbf{s}$ are vectors. But if we add noises even with small coefficients, the resulted LWE problem, i.e. recovering $\mathbf{b} = \mathbf{As} + \mathbf{e}$ can be hard. The Ring-LWE problem is to replace matrices and vectors with polynomials, thus significantly reducing the computation and communication costs. The M-LWE problem can be viewed as a combination of the LWE problem and Ring-LWE problem, thus Kyber enjoys the advantages of relatively easy scalability and high efficiency.

To be specific, Kyber is defined over a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. Here $q = 3329$ is a modulo and $n = 256$. For every polynomial $f(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \in \mathbb{R}_q$, each coefficient $a_i \in \mathbb{Z}_q$ $(0 \leq i \leq n-1)$, where $\mathbb{Z}_q$ represents a ring with all elements are integers modulo q. All the polynomial additions and multiplications are operated modulo $x^n + 1$. We interchangeably use $\mathbf{c} \in \mathcal{R}_q$ and its vector form $(\mathbf{c}[0], \cdots, \mathbf{c}[n-1])$ to represent a polynomial. For a matrix $\mathbf{A} \in R_q^{k \times k}$, $\mathbf{s}, \mathbf{e} \in \mathcal{B}_\eta^k$, where $\mathcal{B}_\eta$ means the centered binomial distribution, the M-LWE problem is to distinguish $(\mathbf{A}, \mathbf{B} = \mathbf{As} + \mathbf{e}) \in R_q^{k \times k} \times R_q^k$ from uniformly selected $(\mathbf{A}, \mathbf{B}) \in R_q^{k \times k} \times R_q^k$.

The security of Kyber can be shifted by simply modifying k. More specifically, there are three security levels in Kyber: Kyber-512, Kyber-768, and Kyber-1024, corresponding to $k = 2$, $k = 3$, and $k = 4$, respectively. Generally, a KEM consists of key generation, encapsulation, and decapsulation. But a PC-based SCA is done against the decapsulation part. Thus, in Algorithm 1, we only depict the main parts of encapsulation and decapsulation of Kyber, ignoring details such as the Number Theoretic Transform (NTT).

Algorithm 1 The encapsulation and decapsulation in Kyber

◇ **CCAKEM.Encaps**
Input: Public Key $\mathbf{p}$
Output: Ciphertext $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$
Output: Shared Key K
$\mathbf{m} \xleftarrow{\$} \{0,1\}^{256}$
$(\bar{K}, r) = \mathrm{G}(\mathbf{m}, \mathrm{H}(\mathbf{p}))$
▷ $\mathrm{CPA.Enc}(\mathbf{p}, \mathbf{m}, r)$
  $\mathbf{A} \xleftarrow{\$} R_q^{k \times k}$
  $\mathbf{r}, \mathbf{e}_1 \xleftarrow{\$} R_q^k$, $\mathbf{e}_2 \xleftarrow{\$} R_q$
  $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
  $\mathbf{v} = \mathbf{p}^T\mathbf{r} + \mathbf{e}_2 + \mathbf{Decomp}_q(\mathbf{m}, 1)$
  $\mathbf{c}_1 = \mathbf{Comp}_q(\mathbf{u}, d_\mathbf{u})$
  $\mathbf{c}_2 = \mathbf{Comp}_q(\mathbf{v}, d_v)$
$K = \mathrm{KDF}(\bar{K}, \mathrm{H}(\mathbf{c}_1, \mathbf{c}_2))$

◇ **CCAKEM.Decaps**
Input: Ciphertext $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$
Input: Secret Key $\mathbf{s}$
Output: Shared Key K
▷ $\mathrm{CPA.Dec}(\mathbf{s}, \mathbf{ct})$
  $\mathbf{u} = \mathbf{Decomp}_q(\mathbf{c}_1, d_u)$
  $\mathbf{v} = \mathbf{Decomp}_q(\mathbf{c}_2, d_v)$
  $\mathbf{m}' = \mathbf{Comp}_q(\mathbf{v} - \mathbf{s}^T\mathbf{u}, 1)$
$(\bar{K}', r') = \mathrm{G}(\mathbf{m}', \mathrm{H}(\mathbf{p}))$
$(\mathbf{c}_1', \mathbf{c}_2') = \mathrm{CPA.Enc}(\mathbf{p}, \mathbf{m}', r')$
if $(\mathbf{c}_1, \mathbf{c}_2) = (\mathbf{c}_1', \mathbf{c}_2')$ **then**
  $K = \mathrm{KDF}(\bar{K}', \mathrm{H}(\mathbf{c}_1', \mathbf{c}_2'))$
else
  $K = \mathrm{KDF}(z, \mathrm{H}(\mathbf{c}_1', \mathbf{c}_2'))$
end if

In the following, let $\lceil \cdot \rfloor$ denote the rounding function, we first define two functions, $\mathbf{Comp}_q(x, d)$ and $\mathbf{Decomp}_q(x, d)$.

Definition 1. The Compression function is defined as: $\mathbb{Z}_q \to \mathbb{Z}_{2^d}$

$$\mathbf{Comp}_q(x, d) = \left\lceil \frac{2^d}{q} \cdot x \right\rfloor \pmod{2^d}. \tag{1}$$

Definition 2. The Decompression function is defined as: $\mathbb{Z}_{2^d} \to \mathbb{Z}_q$

$$\mathbf{Decomp}_q(x, d) = \left\lceil \frac{q}{2^d} \cdot x \right\rfloor. \tag{2}$$

Similarly, when the inputs of $\mathbf{Comp}_q(x, d)$ and $\mathbf{Decomp}_q(x, d)$ are polynomials, i.e., $x \in R_q^k$, the above operation is separately done on each coefficient. Let $\xleftarrow{\$}$ represent random selection and T represent the transpose of a matrix. In both the encapsulation and decapsulation, two hash functions G and H, as well as a key derivation function KDF, are used. CCA-secure Kyber is achieved through the well-known Fujisaki-Okamoto (FO) transform based on a CPA-secure PKE. In the encapsulation, a CPA-secure encryption algorithm is used to output $\mathbf{c}_1$ and $\mathbf{c}_2$. Here d_u and d_v used in $\mathbf{Comp}_q(x, d)$ and $\mathbf{Decomp}_q(x, d)$ functions are also determined by different security levels. For example, in Kyber-512, $d_u = 10$, $d_v = 3$. In the decapsulation, a CPA-secure decryption algorithm is firstly used to obtain $(\bar{K}', r')$, which is then re-encrypted to get $(\mathbf{c}_1', \mathbf{c}_2')$.

The CPA-secure KEMs are vulnerable to chosen-ciphertext attacks when the secret key is reused. These attacks are generally operated in a key-mismatch or PC Oracle. Algorithm 2 first depicts the PC oracle $\mathcal{O}$, in which the adversary sends ciphertext $\mathbf{ct}$ and a reference plaintext $\mathbf{m}$ to the oracle. The oracle tells whether $\mathbf{m}$ equals the CPA decryption result $\mathbf{m}'$ or not. On the right part of Algorithm 2, we introduce the process of recovering the secret key by employing the response sequence from PC oracle[1] $\mathcal{O}$. In the recovery process, special ciphertexts are crafted to combine every possible coefficient value (such as $[-3, 3]$ in Kyber512) with certain oracle response sequence. For example, if $\mathcal{O}$ is always accurate, Ravi et al. needed 5 queries to recover one coefficient and $256 \cdot 2 \cdot 5 = 2560$ queries in total for the second round Kyber512. After that, Qin et al. reduced the queries to an average of 1312 for the third round Kyber512 by using the optimal binary recovery tree. We next briefly introduce the main process proposed by Qin et al., an improved method close to Huffman coding to achieve key recovery, and more details can be found in [QCZ+21].

[1] We present here a general description of the PC oracle. In PC oracle-based CCA SCAs, the message $\mathbf{m}$ is limited to be chosen from a set of $\{\mathbf{m}_0, \mathbf{m}_1\}$ and the oracle outputs the chosen message.

Algorithm 2 PC oracle $\mathcal{O}$ and the key recovery process

◇ **PC oracle** $\mathcal{O}$
Input: Ciphertext $\mathbf{ct}$
Input: Message $\mathbf{m}$
Output: 0 or 1
1: $\mathbf{m}' \leftarrow$ CPA.Dec($\mathbf{s}, \mathbf{ct}$)
2: **if** $\mathbf{m}' = \mathbf{m}$ **then**
3: **Return** 1
4: **else**
5: **Return** 0
6: **end if**

◇ **KeyRecovery**
Input: PC oracle $\mathcal{O}$
Output: Secret Key $\mathbf{s}$
1: **for** All coefficients location loc in $\mathbf{s}$ **do**
2: ▷ **CoefficientRecovery**(loc)
3: $seq =$ “”
4: **while** $\mathbf{s}[loc]$ cannot be recovered from seq **do**
5: Generate $(\mathbf{ct}, \mathbf{m})$ from loc and seq
6: $res = \mathcal{O}(\mathbf{ct}, \mathbf{m})$
7: Append res to the response sequence seq
8: **end while**
9: Set $\mathbf{s}[loc]$ the corresponding value of seq
10: **end for**

2.2 The attack on Kyber from [QCZ+21]

We now take Kyber512 as an example to explain the main attack procedure proposed in [QCZ+21], which will be employed as the initial processing steps in our new attack. We start with building a PC oracle $\mathcal{O}$ shown on the left of Algorithm 2 by instantiating the CPA.Dec() function with Kyber.CPA.Dec().

The victim Alice's secret key is $\mathbf{s}$ and $\mathbf{s} = (\mathbf{s_0}, \mathbf{s_1})$ in Kyber512. On the right of Algorithm 2, the **KeyRecovery** is used to recover $\mathbf{s}$. The aim of the **CoefficientRecovery** is to select proper $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$ and $\mathbf{m}$ as inputs to $\mathcal{O}$. Then, the attacker is able to recover $\mathbf{s}$ from the response sequence seq of $\mathcal{O}$. In the following, we show the approach to recover the first coefficient $\mathbf{s_0}[0]$ in $\mathbf{s_0}$, and the remaining coefficients can be recovered similarly.

To launch the attack, the attacker lets $\mathbf{m} = (1, 0, \cdots, 0)$ and $\mathbf{u} = (\mathbf{u_0}, \mathbf{u_1})$, where $\mathbf{u_0} = (\lceil \frac{q}{16} \rfloor, 0, \ldots, 0)$ and $\mathbf{u_1} = \mathbf{0}$. Then, the attacker is able to calculate $\mathbf{c}_1 = \mathbf{Comp}_q(\mathbf{u}, d_\mathbf{u})$, as well as setting $\mathbf{c}_2 = (g, 0, \ldots, 0)$. Here g is later set and used to recover $\mathbf{s_0}[0]$.

Next, $\mathbf{ct} = (\mathbf{c}_1, \mathbf{c}_2)$ is sent as input to the Oracle, which is employed to generate $\mathbf{u} = \mathbf{Decomp}_q(\mathbf{c}_1, d_u)$ and $\mathbf{v} = \mathbf{Decomp}_q(\mathbf{c}_2, d_v)$. Thus, a relationship between $\mathbf{m}'[0]$ and $\mathbf{s_0}[0]$ can be built as follows:

$$\begin{aligned}
\mathbf{m}'[0] =& \mathbf{Comp}_q((\mathbf{v} - \mathbf{s}^T\mathbf{u})[0], 1) && (3)\\
=& \left\lceil \frac{2}{q} \left(\mathbf{v}[0] - (\mathbf{s}^T\mathbf{u})[0]\right) \right\rfloor \bmod 2 && (4)\\
=& \left\lceil \frac{2}{q} (\mathbf{v}[0] - (\mathbf{s_0}[0]\mathbf{u_0}[0] + \mathbf{s_1}[0]\mathbf{u_1}[0])) \right\rfloor \bmod 2 && (5)\\
=& \left\lceil \frac{2}{q} (\mathbf{v}[0] - \mathbf{s_0}[0]\mathbf{u_0}[0]) \right\rfloor \bmod 2 && (6)\\
=& \left\lceil \frac{2}{q} \left(\left\lceil \frac{q}{16} g \right\rfloor - \mathbf{s_0}[0] \left\lceil \frac{q}{16} \right\rfloor \right) \right\rfloor \bmod 2. && (7)
\end{aligned}$$

The last equation holds due to the fact that $\mathbf{v}[0] = \lceil \frac{q}{16} g \rfloor$ and $\mathbf{u_0}[0] = \lceil \frac{q}{16} \rfloor$.

Further, the attacker could adaptively set different g to recover $\mathbf{s_0}[0]$ based on the sequence outputted from the oracle. Take $g = 4$ as an example: recall that each $\mathbf{s_0}[0]$ is selected from $[-3, 3]$. When $\mathbf{s_0}[0] \in [-3, -1]$, $\mathbf{m}' = (1, 0, \ldots, 0)$, which means $\mathbf{m} = \mathbf{m}'$. So, the output of the oracle is 1. Similarly, when $\mathbf{s_0}[0] \in [0, 3]$, $\mathbf{m}' = (0, 0, \ldots, 0)$, which reults the output of the oracle to be 0. From the above analysis, the attacker succeeds in deciding which subinterval, $[-3, -1]$ or $[0, 3]$, $\mathbf{s_0}[0]$ is in using a query. If the attacker chooses proper g, then he could recover $\mathbf{s_0}[0]$ with as few queries as possible. This is achieved by using the the optimal binary recovery tree in [QCZ+21].

In Table 1, the selections of g and the corresponding changes of States are depicted.

For Kyber512, the States in Table 1 are shifted in accordance with the output of the oracle. To be specific, the attacker first lets $g = 4$ and observes the output of the oracle. If the oracle outputs 1, the attacker goes to State 5 and further sets $g = 3$. Then, if the oracle outputs 0, the attacker succeeds in determining $\mathbf{s_0}[0] = -1$. If the oracle outputs 1, the attacker goes to State 6 and sets $g = 2$. Finally, the attacker is able to decide $\mathbf{s_0}[0] = -2$ or $\mathbf{s_0}[0] = -3$ according to the output of the oracle. In fact, in this way, the attacker succeeds in building a relationship between each symbol in $[-3, -1]$ and the response sequence. For example, $\mathbf{s_0}[0] = -1$ corresponds to 10, $\mathbf{s_0}[0] = -2$ corresponds to 110, while 111 is the corresponding response sequence of $\mathbf{s_0}[0] = -3$.

Finally, all the coefficients can be recovered and the total numbers of queries needed for Kyber512 is 1312.

Table 1: Selections of g and the corresponding States

	State 1	State 2	State 3	State 4	State 5	State 6
g	4	5	6	7	3	2
$\mathcal{O} \to 0$	State 2	State 3	State 4	$\mathbf{s_0}[0] = 3$	$\mathbf{s_0}[0] = -1$	$\mathbf{s_0}[0] = -2$
$\mathcal{O} \to 1$	State 5	$\mathbf{s_0}[0] = 0$	$\mathbf{s_0}[0] = 1$	$\mathbf{s_0}[0] = 2$	State 6	$\mathbf{s_0}[0] = -3$

2.3 PC oracle-based SCA attacks

In the CCA-KEMs such as Kyber and Saber, by employing the FO transform, the above attacks do not work. However, with the help of side-channel information, such as analyzing the power consumption or electromagnetism waveforms during decapsulation, the adversary could directly spot the CPA-secure operations inside the CCA-secure function, and launch the same attacks.

At CHES 2020, Ravi et al. introduced a PC-based SCA attack on NIST KEMs by exploiting the information of the re-encryption procedure in the FO transform [RRCB20]. Again we take the attack against Kyber as an example, the adversary simply let $\mathbf{m'}$ be $\mathbf{m_0} = (0, 0, 0, 0, \cdots)$ or $\mathbf{m_1} = (1, 0, 0, 0, \cdots)$. Correspondingly, the waveform of the re-encryption procedure (i.e lines 5-6 on the right of Algorithm 1) is also fixed to two types, allowing us to enact a PC oracle $\mathcal{O}_{SCA}$ with a side-channel waveform distinguisher. In [RRCB20], Ravi et al. designed the SCA distinguisher mainly by calculating the Euclidean distance to the profiled waveform templates. More specifically, they firstly repeatedly collect re-encryption waveforms of $\mathbf{m'} = \mathbf{m_0}$ and $\mathbf{m'} = \mathbf{m_1}$. Then they run a Test Vector Leakage Assessment (TVLA) between the two waveform sets to select the interested points. In the next attacking stage, they achieve binary-classification by calculating the Euclidean distance between the interested points of collected waveform and two waveform templates. We summarize attacking procedure as $\mathcal{O}_{SCA}$ on the left of Algorithm 3. Note that since we restrict $\mathbf{m'}$ to be $\mathbf{m_0}$ or $\mathbf{m_1}$, the problem of distinguishing whether $\mathbf{m'} = \mathbf{m_0}$ can be simplified to deciding which template of $\mathbf{m_0}$ or $\mathbf{m_1}$ has more close Euclidean distance to the collected waveform.

Affected by the noises from the environment or the masking countermeasures, as well as accuracy problems in the side-channel distinguisher itself, $\mathcal{O}_{SCA}$ is imperfect and cannot always tell the truth. We set its accuracy as α_o. Even if α_o reaches 0.990, if we simply instantiate the **KeyRecovery** in Algorithm 2 with $\mathcal{O}_{SCA}$ to recover the secret key, on average we will meet 13.0 error coefficients among all the recovered 512 coefficients (with ciphertext construction method in [QCZ+21]). Since we cannot decide the positions of the errors, the brute force complexity is quite high, which can be estimated as:

$$\binom{512}{13} \cdot 7^{13} \approx 2^{120.7}.$$

Algorithm 3 PC oracle $\mathcal{O}_{SCA}$ and the Practical Key Recovery process under SCA

◇ **PC oracle $\mathcal{O}_{SCA}$ enacted by SCA**
Input: Ciphertext **ct**
Input: Profiled waveforms $\mathbf{W_{m_0}}$, $\mathbf{W_{m_1}}$
Output: 0 or 1
1: Query the device with **ct** and collect waveform $\mathbf{W_{m'}}$ from FO re-encryption
2: **if** $\mathbf{Dist}(\mathbf{W_{m'}}, \mathbf{W_{m_1}}) < \mathbf{Dist}(\mathbf{W_{m'}}, \mathbf{W_{m_0}})$
3: **Return** 1
4: **else**
5: **Return** 0

◇ **RoughKeyRecovery**
Input: An imperfect PC oracle $\mathcal{O}_{SCA}$
Output: Secret Key **s**
1: **Return KeyRecovery**($\mathcal{O}_{SCA}$)

◇ **SCA PC oracle $\mathcal{O}_V$ with majority-voting**
Input: An imperfect PC oracle $\mathcal{O}_{SCA}$
Output: 0 or 1
1: Query the $\mathcal{O}_{SCA}$ **t** times and add the response value to *sum*
2: **if** $sum > \mathbf{t}/2$ **then**
3: **Return** 1
4: **else**
5: **Return** 0
6: **end if**

◇ **PracticalKeyRecovery**
Input: oracle $\mathcal{O}_V$ with majority-voting
Output: Secret Key **s**
1: **Return KeyRecovery**($\mathcal{O}_V$)

Therefore, additional techniques are needed to strengthen the recovery procedure. A commonly used technique, which is also applied in Ravi et al.'s attack, is majority-voting. With a majority-voting of **t** times, we can obtain a more accurate oracle $\mathcal{O}_V$ with accuracy α_{ov}. That is,

$$\alpha_{ov} = 1 - \sum_{s=0}^{\lfloor t/2 \rfloor} \left(\begin{array}{c} t \\ s \end{array} \right) \alpha_o^s (1-\alpha_o)^{t-s}.$$

In the right part of Algorithm 3, we show the oracle $\mathcal{O}_V$ with a majority-voting and the practical key recovery under SCA. With majority-voting of $\mathbf{t} = 3$, the $\mathcal{O}_{SCA}$ with $\alpha_o = 0.990$ mentioned above can be transformed to $\mathcal{O}_V$ with $\alpha_o = 0.9997$. If we instantiate the **KeyRecovery** with $\mathcal{O}_V$ to recover the secret key, we will get less than 1.0 error coefficients among all the 512 coefficients on average. We set it a successful full-key recovery since the remaining complexity now becomes

$$\binom{512}{1} \cdot 7^1 \approx 2^{11.8}.$$

At CHES2022, Ueno et al. mainly use the deep-learning technique to design the side-channel distinguisher and achieve a similar binary-classification (line 2 on the left of Algorithm 3) [UXT+22]. They use Negative Log-Likelihood (NLL) with $\mathbf{t}'$ traces to gain an oracle with higher accuracy and thus perform key recovery under this oracle. Their overall idea is highly similar to the procedures on the right of Algorithm 3.

Both of their methods need **t** or $\mathbf{t}'$ times the total traces required from a perfect SCA oracle. For example, Ravi et al. apply majority-voting with $\mathbf{t} = 3$ for practical key recovery and need $2560 \cdot 3 = 7680$ queries. Ueno et al. apply the NLL with $\mathbf{t}' = 2$ for key recovery for non-protected software and need $1536 \cdot 2 = 3072$ queries.

In the next section, we propose a more efficient full-key recovery strategy compared with the previously mentioned ones.

3 Our adaptive full-key recovery strategy

In this part, we describe the general full-key recovery framework of the new attack. We start with introducing the basic ideas.

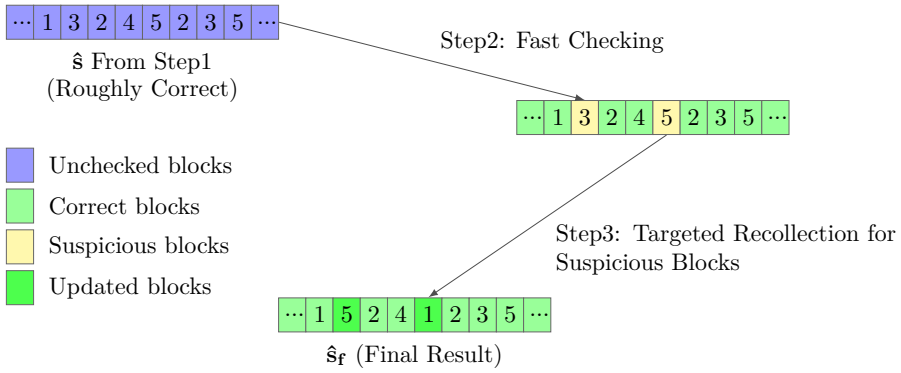

Figure 1: Our full-key recovery strategy

3.1 Our basic idea

In Figure 1, we illustrate the basic idea of our full-key recovery strategy. Instead of strengthening the accuracy of the oracle, the basic idea of our full-key recovery strategy is to detect error positions in the roughly recovered secret key, and then use an improved method to correct the errors. Let the real secret key in the targeted KEM be $\mathbf{s}$, we summarize our main procedure as follows:

Step 1: Perform **RoughKeyRecovery** in Algorithm 3 with $\mathcal{O}_{SCA}$ to get a roughly correct $\hat{\mathbf{s}}$.

Step 2: Based on the knowledge of $\hat{\mathbf{s}}$, we apply a fast checking method to detect the errors.

Step 3: For all suspicious blocks we found in $\hat{\mathbf{s}}$, we perform targeted recollection on them. Finally, we update the coefficient blocks in $\hat{\mathbf{s}}$ and get the final $\hat{\mathbf{s}}_{\mathbf{f}}$.

The major advantage of our full-key recovery strategy is that we make the best use of current information of the roughly correct $\hat{\mathbf{s}}$, and just perform recollection for the suspicious coefficients. During this procedure, we need a *checking* method to construct special ciphertexts to distinguish the locations of error coefficients. A *checking* method should meet the following requirements:

- The *checking* method should be fast and requires as less queries as possible.
- Different from the previous **KeyRecovery** methods, the *checking* method just needs to detect the positions of error coefficients rather than correcting these erroneous coefficients.

In the following, we propose our fast checking method in detail, which plays a central role in our full-key recovery.

3.2 Our fast checking method

The main idea of our fast checking method is to treat the detection of corruptions in $\hat{\mathbf{s}}$ as a coding problem. First, we divide coefficients of $\hat{\mathbf{s}}$ into blocks, while each block may contain one or several coefficients. Then, we treat the possible values of a coefficient block as symbols and the oracle responses (*the 0,1 sequence*) as codewords. To successfully check the targeted block, we generate proper ciphertexts to encode the targeted block symbol to a special codeword $\mathbf{c}_{\mathbf{succ}}$. At the same time encoding the remaining blocks to other different codewords. As shown in Figure 2, the block symbols can be numbers between 1

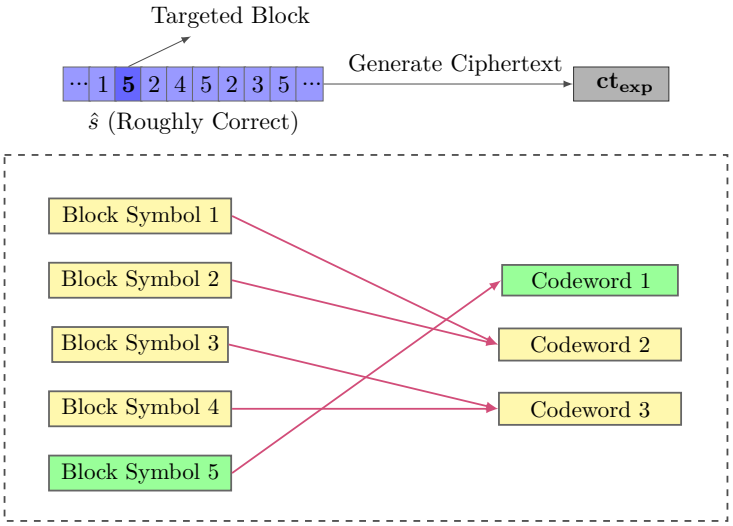

Figure 2: The expected coding for $\mathbf{ct_{exp}}$

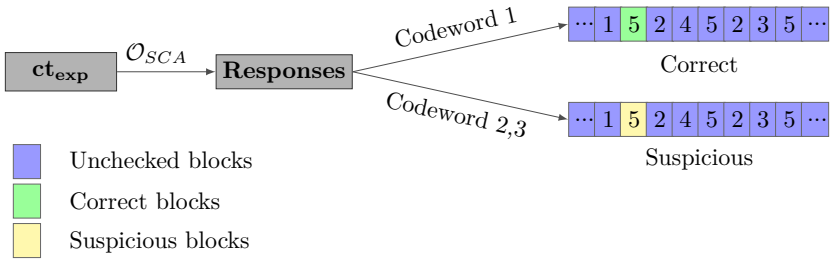

Figure 3: Procedure of checking the targeted block

and 5. To check block symbol 5, we expect a $\mathbf{ct_{exp}}$ to encode it into a special "Codeword 1", while other block symbols are encoded into "Codeword 2" or "Codeword 3".

By setting the oracle responses we collect in checking procedure as $\mathbf{c_{get}}$, we get:

$$\begin{cases} \text{there are } \textbf{no errors} \text{ in the block}, & \mathbf{c_{get}} = \mathbf{c_{succ}}, \\ \text{there are } \textbf{errors} \text{ in the block}, & \mathbf{c_{get}} \neq \mathbf{c_{succ}}. \end{cases}$$

Figure 3 depicts the procedure of checking the targeted block with symbol 5. First, we query the oracle with $\mathbf{ct_{exp}}$ and analyze the response sequence. When we receive "Codeword 1" we set the targeted block as correct, otherwise, we mark the block as suspicious. By continuously querying the oracle, we can check the whole recovered $\hat{\mathbf{s}}$.

3.3 Our adaptive key recovery strategy for different accuracy levels

Algorithm 4 Our Adaptive Full-Key Recovery Strategy for Different Accuracy Level

◇ **AdaptiveFullKeyRec**
Input: An imperfect PC oracle $\mathcal{O}_{SCA}$
Input: Accuracy bound α_b
Output: Recovered secret key $\mathbf{s_f}$
1: $\mathbf{s_f} = \textbf{RoughKeyRecovery}(\mathcal{O}_{SCA})$
2: $\mathbf{E_b} = \textbf{CheckSkByBlock}(\hat{\mathbf{s}}_\mathbf{f})$
3: Estimate α_o of $\mathcal{O}_{SCA}$ from $len(\mathbf{E_b})$
4: **if** $\alpha_o \geq \alpha_b$ **then**
5: **goto FullKeyRecHighAccu**
6: **else**
7: **goto FullKeyRecLowerAccu**
8: **end if**

FullKeyRecHighAccu:
9: **for** All block $B \in \mathbf{E_b}$ **do**
10: **for** All coefficient locations loc in B **do**
11: $\mathbf{s_f}[loc] = \textbf{CoefficientRecovery}(loc)$
12: **end for**
13: **end for**
14: **return** $\mathbf{s_f}$

FullKeyRecLowerAccu:
15: Precompute $\mathbf{cc_1}, \mathbf{cc_2}, \textbf{Threshold}[\,]$ from α_o
16: $round = 0$
17: **while** Not all **Finished**$[loc] = true$ **do**
18: **for** All block $B \notin \mathbf{E_b}$ **do**
19: **for** All coefficient locations loc in B **do**
20: **confidence**$[loc][\hat{\mathbf{s}}_\mathbf{f}[loc]]$ += $\mathbf{cc_1}$
21: **end for**
22: **end for**
23: **for** All block $B \in \mathbf{E_b}$ **do**
24: **for** All coefficient locations loc in B **do**
25: $nv = \textbf{CoefficientRecovery}(loc)$
26: **confidence**$[loc][nv]$ += $\mathbf{cc_2}$
27: **end for**
28: **end for**
29: $cct = \textbf{Threshold}[round]$
30: **for** All coefficients location loc in $\mathbf{s}$ **do**
31: Set vl the most confident coefficient value
32: Update $\hat{\mathbf{s}}_\mathbf{f}[loc]$ with vl
33: **if confidence**$[loc][vl] > cct$ **then**
34: **Finished**$[loc] = true$
35: **end if**
36: **end for**
37: $\mathbf{E_b} = \textbf{CheckSkByBlock}(\hat{\mathbf{s}}_\mathbf{f})$
38: $round$++
39: **end while**
40: **return** $\mathbf{s_f}$

In practice, due to different noise levels or distinguisher accuracy, the resulted oracle accuracy can be different. For example, in [UXT+22], Ueno et al. realized a side-channel distinguisher with a trained Neural-Network (NN) model. The model accuracy (also the oracle accuracy) for non-protected implementation is 0.998, and for masked implementation is 0.960. In the following, we propose our adaptive key recovery strategy, which can deal with different accuracy levels in an adaptive way.

In Algorithm 4, we show the pseudocode of our full-key recovery strategy for high and lower oracle accuracy, respectively. Our strategy is adaptive, in which we could first estimate the oracle accuracy α_o from the erroneous block number (i.e. line 3 in Algorithm 4). Then we compare it with a accuracy bound α_b to determine the accurate level and turn to **FullKeyRecHighAccu** or **FullKeyRecLowerAccu** for high or lower accuracy conditions. When the accuracy of the oracle is high (such as 0.998), our full-key recovery strategy in Subsection 3.1 works well, but it may fail to achieve full-key recovery for lower oracle accuracy (such as 0.960). This is because the checking and the recollection procedure are also done under the SCA oracle, which may also be corrupted. The corruptions on

these two procedures are negligible under a high oracle accuracy but need to be considered when the oracle accuracy is low.

To solve this problem, we introduce a *mixed voting* technique to extend our full-key recovery strategy suitable for lower oracle accuracy. Our main idea is to make full use of the information in the checking and recollection phases. More specifically, we define a **confidence** array to store the votes of every possible value of every coefficient position. For the coefficients in correct blocks, we add $\mathbf{cc_1}$ votes to the values in targeted positions. For the coefficients in erroneous blocks, we run recollection for the targeted positions (i.e. perform **CoefficientRecovery** in Algorithm 2) and cast $\mathbf{cc_2}$ votes for the recovered value from recollection. We repeat this procedure round by round, and when the confidence of a certain candidate value reaches **Threshold**[*round*], we set it as right. Here, $\mathbf{cc_1}, \mathbf{cc_2}$ and **Threshold** can be precomputed from the oracle accuracy α_o. The **RoughKeyRecovery**, **CheckSkByBlock**, **CoefficientRecovery**, α_b and the estimation method of α_o can be instantiated according to different recovery and checking method for the targeted KEM. In the next Section, we will take Kyber512 as an example to give more details.

4 Our adaptive full-key recovery for Kyber

In this chapter, we show how to launch a practical PC oracle-based full-key recovery against Kyber. The remaining challenge is how to select proper parameters.

4.1 Restrictions on ciphertext construction and our solution

For Kyber, the PC oracle requires that the decrypted message $\mathbf{m}'$ (line 4 in Algorithm 1) to be set as $(1, 0, 0, 0, \cdots)$ or $(0, 0, 0, 0, \cdots)$. That is, the first coefficient of $\mathbf{m}'$ can be either 0 or 1, while the other coefficients should be all 0. To achieve this goal, we need to carefully select the ciphertext **ct**, which results in limited choices of **ct**. In some cases, the expected ciphertexts for our designed code may not exist.

In Table 2, we give two examples of the designed codeword for Kyber512, considering $\mathbf{BlockLen} = 1$, here **BlockLen** represents the number of coefficients in a block. Recall that in Kyber the coefficients lie in the interval $[-3, 3]$. From these two examples, we could distinguish the target value **1**, since **1** is uniquely decoded by codeword "0" in **Example 1** and codeword "01" in **Example 2**.

Table 2: Designing codewords to distinguish **1**

	-3	**-2**	**-1**	**0**	**1**	**2**	**3**
Example 1	"1"	"1"	"1"	"1"	"0"	"1"	"1"
Example 2	"11"	"11"	"11"	"11"	"01"	"00"	"00"

Example 1 requires us to find a ciphertext **ct** with which the resulted oracle response is $\{1, 1, 1, 1, 0, 1, 1\}$ for coefficient values in $[-3, 3]$. However, in practice we can never find such **ct**. **Example 2** requires us to find two ciphertexts resulting in $\{1, 1, 1, 1, 0, 0, 0\}$ and $\{1, 1, 1, 1, 1, 0, 0\}$, respectively, and such ciphertexts are available. However, with larger blocks, it is quite hard to find the qualified ciphertexts directly from the initially designed codewords. Hence, we need to find all candidate ciphertexts and try to generate qualified codewords from them. To conclude, we show our main steps as follows:

Step 1: Find as many ciphertexts as possible satisfying the PC oracle restrictions for the concrete **BlockLen**. We store all of them in **ct_list**.

Step 2: For all pairs of ciphertexts in **ct_list**, we calculate the response sequence for all block symbols and store them in **c_list**. Then we try to find the special codeword $\mathbf{c_{succ}}$ from **c_list**.

In the following, we first discuss how to craft all the possible ciphertexts for Step 1. Then we show how to check 4 coefficients by 2 queries, which is the most efficient way we could achieve.

4.2 Crafting ciphertexts for Kyber512

In Kyber512, let the decompressed $\mathbf{ct} = (\mathbf{u}, \mathbf{v})$, $\mathbf{u} = (\mathbf{u_0}, \mathbf{u_1})$ and the reused $\mathbf{s} = (\mathbf{s_0}, \mathbf{s_1})$. Here $\mathbf{u_0}$, $\mathbf{u_1}$, $\mathbf{v}$, $\mathbf{s_0}$, and $\mathbf{s_1}$ are all polynomials with 256 coefficients. From the decryption procedure of Kyber, we know that

$$\mathbf{m}' = \mathbf{Comp}_q(\mathbf{v} - \mathbf{s}^T\mathbf{u}, 1) = \left\lfloor \frac{2}{q}(\mathbf{v} - \mathbf{s}^T\mathbf{u}) \right\rceil \bmod 2 \tag{8}$$

$$= \left\lfloor \frac{2}{q}(\mathbf{v} - (\mathbf{s_0}\mathbf{u_0} + \mathbf{s_1}\mathbf{u_1})) \right\rceil \bmod 2. \tag{9}$$

We take the recovery of coefficients in $\mathbf{s_0}$ as an example. We need some attacking parameters setting as $\mathbf{u_{atk}}$ and $\mathrm{v_{atk}}$, where $\mathbf{u_{atk}}$ is an array with **BlockLen** integers to craft $\mathbf{u}$ in $\mathbf{ct}$, and $\mathrm{v_{atk}}$ is an integer to craft $\mathbf{v}$ in $\mathbf{ct}$.

Assume $\mathbf{BlockLen} = 1$, i.e. only one coefficient in the block. To get a relationship for $\mathbf{s_0}[t],\ t \in [0, 255]$, the attacker firstly set $\mathbf{u_1} = \mathbf{0}$ to ensure $\mathbf{s_1}$ has nothing to do with $\mathbf{m}'$. Moreover, for $i \in [1, 255]$, we set $\mathbf{v}[i] = 0,\ i \in [1, 255]$ and $\mathbf{u_0} = \mathbf{u_{atk}}[0]$ if $t = 0$ or $\mathbf{u_0} = -\mathbf{u_{atk}}[0] \cdot x^{256-t}$ if $t \neq 0$. Take $t = 0$ as an example, now

$$\mathbf{m}'[i] = \left\lfloor -\frac{2}{q}\mathbf{s_0}[i]\mathbf{u_{atk}}[0] \right\rceil \bmod 2,\ i \in [1, 255],\ t = 0.$$

To ensure $\mathbf{m}'[i] = 0$, for $i \in [1, 255]$, since $\mathbf{s_0}[i] \in [-3, 3]$, we only need to let

$$\frac{2}{q}(3\mathbf{u_{atk}}[0]) < \frac{1}{2} \quad \text{i.e. } \mathbf{u_{atk}}[0] < \frac{q}{12}.$$

When $\mathbf{BlockLen} = 2$, to find a relationship for $\mathbf{s_0}[t], \mathbf{s_0}[t+1]$, the attacker similarly set $\mathbf{u_1} = \mathbf{0}$, $\mathbf{v}[i] = 0,\ i \in [1, 255]$, and $\mathbf{u_0} = \mathbf{u_{atk}}[0] - \mathbf{u_{atk}}[1] \cdot x^{255}$ if $t = 0$ or $\mathbf{u_0} = -\mathbf{u_{atk}}[0] \cdot x^{256-t} - \mathbf{u_{atk}}[1] \cdot x^{255-t}$ if $t \neq 0$. Take $t = 0$ as an example, now

$$\mathbf{m}'[i] = \left\lfloor \frac{2}{q}(-\mathbf{s_0}[i]\mathbf{u_{atk}}[0] - \mathbf{s_0}[i+1]\mathbf{u_{atk}}[1]) \right\rceil \bmod 2,\ i \in [1, 255],\ t = 0$$

Since $(\mathbf{s_0}[i], \mathbf{s_0}[i+1]) \in [-3, 3] \times [-3, 3]$, to achieve $\mathbf{m}'[i] = 0$, we also need

$$\frac{2}{q}(3\mathbf{u_{atk}}[0] + 3\mathbf{u_{atk}}[1]) < \frac{1}{2} \quad \text{i.e. } \mathbf{u_{atk}}[0] + \mathbf{u_{atk}}[1] < \frac{q}{12}. \tag{10}$$

When $\mathbf{BlockLen} = 4$, similarly we require

$$\mathbf{u_{atk}}[0] + \mathbf{u_{atk}}[1] + \mathbf{u_{atk}}[2] + \mathbf{u_{atk}}[3] < \frac{q}{12}. \tag{11}$$

Algorithm 5 depicts the ciphertext generation procedure with attacking parameters. We use **sp** and **bp** to locate the targeted block.

4.3 Checking 2 or 4 coefficients by 2 queries

When $\mathbf{BlockLen} = 2$, let **bp** represent the index of the beginning of the block. We take how to check $(\mathbf{s_0}[\mathbf{bp}], \mathbf{s_0}[\mathbf{bp}+1])$ as an example. As shown in Algorithm 5, by setting $\mathbf{ct}$ to the following value:

$$\mathbf{ct} = \mathbf{GeneCiphertext}_{\mathbf{2,0,bp}}(\mathbf{u_{atk}}, \mathrm{v_{atk}}),$$

Algorithm 5 Generating ciphertexts of PC oracle from given parameters

Input: $\mathbf{sp} = 0$ or 1 for recovering $\mathbf{s_0}$ or $\mathbf{s_1}$ in $\mathbf{s}$, $\mathbf{bp}$ = index of the beginning of the block
Input: Attacking parameters $\mathbf{u_{atk}}, \mathrm{v_{atk}}$
Output: Ciphertext $\mathbf{ct}$
◇ $\mathbf{GeneCiphertext_{BlockLen,sp,bp}}(\mathbf{u_{atk}}, \mathrm{v_{atk}})$
$(\mathbf{u_0}, \mathbf{u_1}) = (\mathbf{0}, \mathbf{0})$; $\mathbf{v} = \mathbf{0}$
if $\mathbf{bp} = 0$ **then**
  $\mathbf{u_{sp}} = \mathbf{u_{atk}}[0] - \sum_{n=1}^{\mathbf{BlockLen}-1} \mathbf{u_{atk}}[n] \cdot x^{256-n}$
else
  $\mathbf{u_{sp}} = -\sum_{n=0}^{\mathbf{BlockLen}-1} \mathbf{u_{atk}}[n] \cdot x^{256-\mathbf{bp}-n}$
end if
$\mathbf{v}[0] = \mathrm{v_{atk}}$
return $((\mathbf{u_0}, \mathbf{u_1}), \mathbf{v})$

the attacker knows the following equation:

$$\mathcal{O}(\mathbf{ct}) = \mathbf{m}'[0] = \left\lfloor \frac{2}{q}(\mathrm{v_{atk}} - (\mathbf{s_0}[\mathbf{bp}]\mathbf{u_{atk}}[0] + \mathbf{s_0}[\mathbf{bp}+1]\mathbf{u_{atk}}[1]) \right\rceil \bmod 2.$$

Recall that each $\mathbf{s_0}[\mathbf{bp}]$ lies in $[-3, 3]$. To check all 7×7 value of $(\mathbf{s_0}[\mathbf{bp}], \mathbf{s_0}[\mathbf{bp}+1])$, we also need 7×7 $(\mathbf{ct_1}, \mathbf{ct_2})$. In Algorithm 6 we express our brute force strategy as pseudocode and successfully find all of them.

Algorithm 6 Generating proper ciphertexts to check blocks with $\mathbf{BlockLen} = 2$

Output: Parameters to check all possible block symbols, with $\mathbf{BlockLen} = 2$
for All available $(\mathbf{u_{atk}}[0], \mathbf{u_{atk}}[1])$ satisfying Eq. (10) and all available $\mathrm{v_{atk}}$ **do**
  $\mathbf{ct} = \mathbf{GeneCiphertext}_{2,0,0}((\mathbf{u_{atk}}[0], \mathbf{u_{atk}}[1]), \mathrm{v_{atk}})$
  Append $\mathbf{ct}$ to $\mathbf{ct\_list}$
end for
for All pairs $(\mathbf{ct_1}, \mathbf{ct_2})$ in $\mathbf{ct\_list}$ **do**
  Calculate the response sequence $\mathcal{O}(\mathbf{ct_1})||\mathcal{O}(\mathbf{ct_2})$ for all block symbols and store to $\mathbf{c\_list}$
  if Special codeword $\mathbf{c_{succ}}$ is found from $\mathbf{c\_list}$ **then**
    Append the corresponding block symbols, $(\mathbf{ct_1}, \mathbf{ct_2})$ and $\mathbf{c_{succ}}$ to $\mathbf{result}$
  end if
end for
return result

Suppose we have known the roughly correct $\hat{\mathbf{s}} = (\hat{\mathbf{s}}_\mathbf{0}, \hat{\mathbf{s}}_\mathbf{1})$, and we want to check whether $(\hat{\mathbf{s}}_0[0], \hat{\mathbf{s}}_0[1])$ is equal to $(0, 2)$ or not. We execute the following steps:

1. Query the oracle with $\mathbf{ct_1} = \mathbf{GeneCiphertext}_{\mathbf{2,0,0}}((32, 48), 5)$ and store the response $\mathcal{O}(\mathbf{ct_1})$.

2. Query the oracle with $\mathbf{ct_2} = \mathbf{GeneCiphertext}_{\mathbf{2,0,0}}((22, 43), 5)$ and store the response $\mathcal{O}(\mathbf{ct_2})$. In Table 3, we list all the response sequences $\mathcal{O}(\mathbf{ct_1})||\mathcal{O}(\mathbf{ct_2})$ for all possible block symbols. Here we can see that $(0, 2)$ corresponds to a special codeword "01", while other block symbols are encoded to "11" or "00".

3. Since $\mathbf{c_{succ}}$ = "01", so

$$(\mathbf{s_0}[0], \mathbf{s_0}[1]) = \begin{cases} (0, 2)\ , & \mathbf{c_{get}} = \text{"01"}\ , \\ others, & \mathbf{c_{get}} = others. \end{cases}$$

Table 3: $\mathcal{O}(\mathbf{ct_1})||\mathcal{O}(\mathbf{ct_2})$ for $[-3,3]\times[-3,3]$

$s_0[0]\backslash s_0[1]$	*-3*	*-2*	*-1*	*0*	*1*	*2*	*3*
-3	“11”	“11”	“11”	“11”	“11”	“11”	“11”
-2	“11”	“11”	“11”	“11”	“11”	“11”	“11”
-1	“11”	“11”	“11”	“11”	“11”	“11”	“11”
0	“11”	“11”	“11”	“11”	“11”	“01”	“00”
1	“11”	“11”	“11”	“11”	“00”	“00”	“00”
2	“11”	“11”	“00”	“00”	“00”	“00”	“00”
3	“11”	“00”	“00”	“00”	“00”	“00”	“00”

To accelerate the checking process, we increase the **BlockLen** to be 4. Then, we show how to check 4 coefficients with 2 queries, which can significantly improve the efficiency. Similarly, by setting **ct** as

$$\mathbf{ct} = \mathbf{GeneCiphertext}_{\mathbf{4,0,bp}}(\mathbf{u_{atk}}, \mathrm{v_{atk}}),$$

the attacker knows the following equation:

$$\mathcal{O}(\mathbf{ct}) = \left\lfloor \frac{2}{q}(\mathrm{v_{atk}} - (\sum_{n=0}^{3} \mathbf{s_0}[\mathbf{bp}+n]\mathbf{u_{atk}}[n]) \right\rceil \bmod 2.$$

To check all 7^4 values of $(\mathbf{s_0}[\mathbf{bp}], \mathbf{s_0}[\mathbf{bp}+1], \mathbf{s_0}[\mathbf{bp}+2], \mathbf{s_0}[\mathbf{bp}+3])$, we also need to generate proper 7^4 ciphertext pairs $(\mathbf{ct_1}, \mathbf{ct_2})$. However, the computational complexity can be quite high if we apply a similar brute-force in Algorithm 6. After carefully analyzing the table of oracle responses and the corresponding ciphertext **ct**, we observe that the problem can be solved by using the idea of divide-and-conquer. That is, we first get the needed attacking parameter $\mathbf{u_{atk}}[0], \mathbf{u_{atk}}[1], \mathrm{v_{atk}}$ using Algorithm 6. Then, with the found $\mathbf{u_{atk}}[0], \mathbf{u_{atk}}[1], \mathrm{v_{atk}}$, we try to find proper $\mathbf{u_{atk}}[2], \mathbf{u_{atk}}[3]$ satisfying Eq. (11). Next we select those $(\mathbf{u_{atk}}[0], \mathbf{u_{atk}}[1], \mathbf{u_{atk}}[2], \mathbf{u_{atk}}[3], \mathrm{v_{atk}})$ which could generate the special code we want. Our main process is given in Algorithm 7.

Algorithm 7 Generating proper ciphertexts for checking blocks with **BlockLen** = 4

Output: Parameters to check all possible block symbols, with **BlockLen** = 4
1: Run Algorithm 6 to get **result_bl2**
2: **for** Every tuple $(\mathbf{ct_1}, \mathbf{ct_2})$ in **result_bl2 do**
3: **for** All available extra $(\mathbf{u_{atk}}[2], \mathbf{u_{atk}}[3])$ to $\mathbf{ct_1}$ and $\mathbf{ct_2}$ with satisfying Eq. (11) **do**
4: $\mathbf{ct} = \mathbf{GeneCiphertext}_{4,0,0}((\mathbf{u_{atk}}[0], \mathbf{u_{atk}}[1], \mathbf{u_{atk}}[2], \mathbf{u_{atk}}[3]), \mathrm{v_{atk}})$
5: Append **ct** to **ct_list**
6: **end for**
7: **for** All pairs $(\mathbf{ct_1}, \mathbf{ct_2})$ in **ct_list do**
8: Calculate the response sequence $\mathcal{O}(\mathbf{ct_1})||\mathcal{O}(\mathbf{ct_2})$ for all symbols and store them in **c_list**
9: **if** Special codeword $\mathbf{c_{succ}}$ is found from **c_list then**
10: Append the corresponding block symbols, $(\mathbf{ct_1}, \mathbf{ct_2})$ and $\mathbf{c_{succ}}$ to **result**
11: **end if**
12: **end for**
13: **end for**
14: **return result**

Suppose we want to check if $(\hat{\mathbf{s}}_\mathbf{0}[0], \hat{\mathbf{s}}_\mathbf{0}[1], \hat{\mathbf{s}}_\mathbf{0}[2], \hat{\mathbf{s}}_\mathbf{0}[3])$ really equals to $(-2,-1,0,1)$ for example. We execute the following steps:

1. Query the oracle with $\mathbf{ct_1} = \mathbf{GeneCiphertext_{4,0,0}}((10, 1, 66, 4), 13)$ and store the response $\mathcal{O}(\mathbf{ct_1})$.

2. Query the oracle with $\mathbf{ct_1} = \mathbf{GeneCiphertext_{4,0,0}}((10, 1, 65, 5), 13)$ and store the response $\mathcal{O}(\mathbf{ct_2})$. In Table 4, we list codewords and their corresponding counts for all possible block symbols. We can see that $(-2, -1, 0, 1)$ corresponds to a special codeword "01", while other block symbols are encoded to "11", "00" or "01".

3. Since $\mathbf{c_{succ}}$ = "01", so

$$(\mathbf{s_0}[0], \mathbf{s_0}[1], \mathbf{s_0}[2], \mathbf{s_0}[3]) = \begin{cases} (-2, -1, 0, 1)\ , & \mathbf{c_{get}} = \text{"01"}\ , \\ others, & \mathbf{c_{get}} = others. \end{cases}$$

Table 4: $\mathcal{O}(\mathbf{ct_1})||\mathcal{O}(\mathbf{ct_2})$ for $[-3, 3]^4$

Codewords	"00"	**"01"**	"10"	"11"
Counts	1532	**1**	9	859

Finally, we are able to find 2382 $(\mathbf{ct_1}, \mathbf{ct_2})$ sequences, and thus, we can check 2382 block symbols in all possible 7^4 (= 2401) ones. For the blocks in which symbols cannot respond to available $(\mathbf{ct_1}, \mathbf{ct_2})$, we can simply redivide these blocks. In this way, we are able to check all the coefficients. A natural question here is, can we go on increasing the block length to further improve the efficiency? The increment in the block length also brings much burden in finding proper ciphertexts, and the searching complexity for checking 8 coefficients is already prohibitively high. We need to balance efficiency and searching complexity for ciphertexts.

4.4 Instantiation of full-key recovery against Kyber512

We instantiate the **CheckSkByBlock** in Algorithm 4 with the method of "Checking 4 coefficients by 2 queries". Also, we instantiate **RoughKeyRecovery** and **CoefficientRecovery** with the proposed attack method in [QCZ+21] for Kyber512, which is the most efficient PC oracle-based recovery method to the best of our knowledge (see Section 2.2). We set the bound α_b to 0.990 and we can estimate α_o by $\alpha_o \approx 1 - len(\mathbf{E_b})/Q_{RR}$. Here Q_{RR} represents the number of queries needed in **RoughKeyRecovery**, and for Kyber512 $Q_{RR} = 1312$.

For **FullKeyRecLowerAccu**, the $\mathbf{cc_1}, \mathbf{cc_2}$ and **Threshold**[ ] can be determined in advance through simulations. That is, we randomly generate a set of 20 secret keys and select the best $\mathbf{cc_1}, \mathbf{cc_2}$ and **Threshold**[ ] which result in the lowest total number of traces. Table 5 gives the precomputed $\mathbf{cc_1}, \mathbf{cc_2}$, and **Threshold**[ ] for $\alpha_o = 0.960$, $\alpha_o = 0.950$ and $\alpha_o = 0.900$.

Table 5: The precomputed parameters for $\alpha_o = 0.960$, $\alpha_o = 0.950$ and $\alpha_o = 0.900$

	$\mathbf{cc_1}$	$\mathbf{cc_2}$	**Threshold**[ ]
$\alpha_o = 0.960$	4	3	$\{5, 9, 11, 14, 14, 18, 18, 18, 18, 22, 22, ...\}$
$\alpha_o = 0.950$	4	3	$\{5, 9, 12, 13, 13, 16, 17, 17, 21, 21, 21, ...\}$
$\alpha_o = 0.900$	3	4	$\{5, 9, 13, 16, 20, 20, 20, 22, 24, 25, 26, ...\}$

5 Experiments and Analysis

In this section, we present the experimental results and analysis, including results from software simulations and also real-world experiments on an ARM Cortex-M4 microcontroller. Our aim is to understand the performance of the new algorithm, so we first set several oracle accuracy levels and then run computer simulations to simulate the sample complexity under different accuracy levels. We last launch an EM attack to demonstrate that the simulation results match the real-world scenarios.

5.1 Software Simulations

5.1.1 Simulation settings

We firstly introduce our software simulations to show the efficiency of our full-key recovery method under practical SCA. Recall that $\mathcal{O}$ is the ideal PC oracle, to simulate the practical $\mathcal{O}_{SCA}$, we let the outputs of $\mathcal{O}_{SCA}$ equal the outputs of $\mathcal{O}$ with a probability α_o. Then the two outputs are different with $1-\alpha_o$ probability. That is,

$$\mathcal{O}_{SCA}(\mathbf{ct}) = \begin{cases} \mathcal{O}(\mathbf{ct}) \ , & \text{with } \alpha_{\mathrm{o}} \text{ probability,} \\ !\mathcal{O}(\mathbf{ct}) \ , & \text{with } (1-\alpha_{\mathrm{o}}) \text{ probability.} \end{cases}$$

Now the probability α_o represents the accuracy of the practical $\mathcal{O}_{SCA}$ in each query. In our experiments, we let $\alpha_o = 0.995, 0.950, 0.900$ to simulate different accuracy levels. Our method is adaptive since we first evaluate the accuracy level and compare it with a bound $\alpha_b = 0.990$. In case $\alpha_o = 0.995 > \alpha_b$, we set it as high oracle accuracy, and then use the **FullKeyRecHighAccu** in Algorithm 4 to launch the attack. In other cases, $\alpha_o = 0.950$ and $\alpha_o = 0.900$ simulate the conditions with lower accuracy, and we use the **FullKeyRecLowerAccu** in Algorithm 4 to launch the attack.

5.1.2 Traces evaluation under different accuracy levels

To show the advantage of our proposed full-key recovery strategy against majority-voting, we run tests with 10,000 randomly generated secret keys. We let **#TotalTrace** represent the average number of total traces for the full-key recovery. We also use **#ErrCof** to represent the average number of error coefficients in the finally recovered secret key. Both majority-voting and our proposed method aim to reduce **#ErrCof** to an amount less than 1.0. All the reported numbers are calculated by taking an average. As shown in Table 6, compared to majority-voting, our method achieves similar **#ErrCof** but reducing $58.2\%, 57.8\%, 46.1\%$ total traces when $\alpha_o = 0.995, 0.950, 0.900$, respectively. These results fully show the efficiency of our method in a wide range of oracle accuracy.

5.1.3 Improving Ueno et al.'s work

In [UXT+22], Ueno et al. employed a side-channel distinguisher with trained Neural Network models. Their distinguisher corresponds to our imperfect oracle, and their model accuracy (also the oracle accuracy) for different implementations is given in Table 7. We can see that for non-protected software implementation, their distinguisher lie in the high accuracy region, while for masked implementation, their distinguisher achieves lower accuracy. To achieve full-key recovery, they use negative log likelihood (NLL) to enact a more accurate oracle. More specifically, they use NLL to merge 2 and 5 traces for an oracle response for the non-protected and masked software implementation, and the total number of the required traces are $2 \cdot 1536 = 3072$ and $5 \cdot 1536 = 7680$, respectively.

[2] Ueno et al. did not give the exact number of the **#ErrCof** in their experiments, but their method can achieve nearly 0 errors.

Table 6: Comparison between majority-voting and our proposed method for full-key recovery under $\mathcal{O}_{SCA}$ ($\mathbf{t}$ represents the number of votes cast)

$\alpha_o = 0.995$	**#TotalTrace**	**#ErrCof**
Majority Voting ($\mathbf{t} = 3$)	3936.5 (ref)	0.10/512
Our Method	1645.1 (−58.2%)	0.51/512

$\alpha_o = 0.950$	**#TotalTrace**	**#ErrCof**
Majority Voting ($\mathbf{t} = 7$)	9185.0 (ref)	0.25/512
Our Method	3874.5 (−57.8%)	0.15/512

$\alpha_o = 0.900$	**#TotalTrace**	**#ErrCof**
Majority Voting ($\mathbf{t} = 11$)	14433.3 (ref)	0.39/512
Our Method	7773.9 (−46.1%)	0.25/512

Table 7: Accuracy of the NN side-channel distinguisher in [UXT+22]

	Non-protected software	Masked software
Accuracy	0.998	0.960

Table 8: Comparisons between Ueno et al.'s work and our proposed method for full-key recovery under $\mathcal{O}_{SCA}$

$\alpha_o = 0.998$	**#TotalTrace**	**#ErrCof**
Ueno et al.'s	3072.0 (ref)	0.00/512[2]
Our Method	1663.3 (−45.9%)	0.04/512

$\alpha_o = 0.960$	**#TotalTrace**	**#ErrCof**
Ueno et al.'s	7680.0 (ref)	0.00/512
Our Method	3424.9 (−55.4%)	0.05/512

To show that our method could further optimize Ueno et al.'s work, we use the software simulation mentioned above with $\alpha_o = 0.998$ and 0.960, and run tests with 10,000 randomly generated secret keys. The result is averaged and given in Table 8. Compared to Ueno et al.'s results, our method reduces 45.9%, 55.4% total traces with $\alpha_o = 0.998$ and 0.960, respectively. At the same time, similar to the case in [UXT+22], we could recover nearly all the coefficients, i.e. on average only 0.04 or 0.05 error coefficients occur among the 512 coefficients.

5.2 Real-world experiments

5.2.1 Experiment Setup

For real-world validation, as shown in Figure 4, we conduct our experiments on an STM32F407G board, which is equipped with an ARM Cortex-M4 microcontroller. We compile the ARM-optimized Kyber512 implementations from the *pqm4* library and run it on the microcontroller. The clock frequency of the target board is set to be 24 MHz. A PicoScope 3403D oscilloscope and a CYBERTEK EM5030-3 EM Probe are used to collect the EM traces. We connect the probe and the oscilloscope with a CYBERTEK EM5020A signal amplifer. The traces are sampled at 1 GHz.

To instantiate the PC oracle-based SCA distinguisher, we use the same method as that in [RRCB20], which is introduced in the first part of Subection 2.2. The right part of Figure 4 shows one of the TVLA results between the $\mathbf{m_0}$ and $\mathbf{m_1}$ templates.

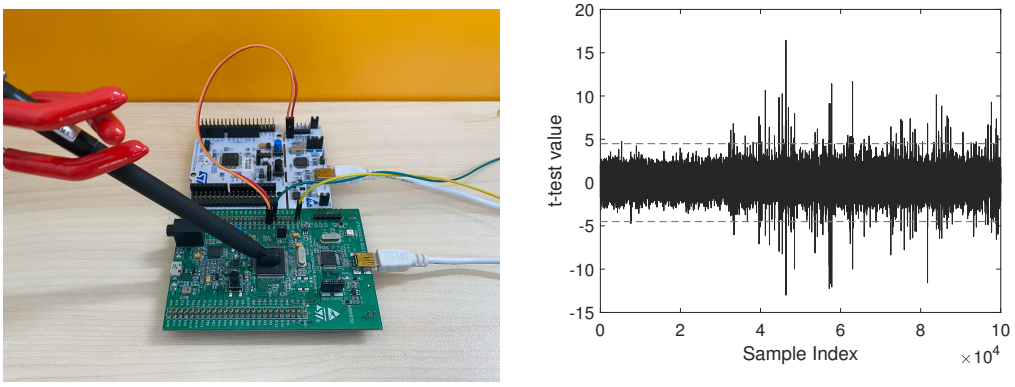

Figure 4: Our SCA equipment (left) and TVLA results between re-encryption of $\mathbf{m_0}$ and $\mathbf{m_1}$ (right)

5.2.2 Experimental Results

We run tests for both the majority-voting approach and our proposed full-key recovery strategy, with 10 randomly generated secret keys. We first implement the **RoughKeyRecovery** and **CheckSkByBlock** (i.e. the result of lines 1-2 in Algorithm 4), then we estimate the oracle accuracy from the erroneous block numbers. The estimated oracle accuracy in our SCA setup is 0.996, and thus we choose to use the **FullKeyRecHighAccu** version of our adaptive full-key recovery strategy.

Table 9: Comparison between majority-voting and our proposed method for full-key recovery in real-world experiments and simulations

	#TotalTrace (Real world)	**#ErrCof** (Real world)	**#TotalTrace** (Simulations)	**#ErrCof** (Simulations)
Majority Voting ($\mathbf{t} = 3$)	3943.5 (ref)	0.20/512	3936.5	0.06/512
Our Method	1618.9 (-58.9%)	0.40/512	1629.9	0.34/512

The experimental results are averaged and shown in Table 9, where both of the two methods lower the average error coefficients to an amount of less than 1.0. Note that under such an experimental setup, with neither the majority-voting nor the new attack strategy applied, we will expect around 4.1 error coefficients. We can see from the table that the real-world experimental results match the computer simulations very well (i.e., with a difference of less than 1%). Also, compared with the majority-voting approach, our new method reduces 58.9% of the required EM traces under our real-world setup.

6 Conclusions

We have presented a novel adaptive approach for improving the sample complexity of the PC oracle-based CCA SCAs on lattice-based KEMs, when the constructed PC oracle is

imperfect. The proposed framework consists of two variants targeting the low/high oracle-accuracy regions, respectively, and is instantiated on Kyber, a candidate in the finalists of the NIST PQC standardization project. Targeting Kyber512, we ran extensive computer simulations and also mounted an EM attack against the optimized implementation in the pqm4 library running on a STM32F407G board with an ARM Cortex-M4 CPU. The experimental results demonstrated that the new approach could provide a substantial gain with respect to the required number of traces.

Our new approach could be applied to other LWE/LWR-based KEMs such as another NIST PQC finalist Saber [DKRV19]; we already described the general idea in Section 3, but the algorithmic details can be involved. Moreover, it is an interesting future direction to extend the new attack framework to NTRU-type KEMs. Last, this work has made the known protections such as masking and shuffling more vulnerable since the adversary needs fewer attack traces. It is always appealing but challenging to design safer and more efficient countermeasures.

References

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

[ABH+22] Melissa Azouaoui, Olivier Bronchain, ClÃĭment Hoffmann, Yulia Kuzovkova, Tobias Schneider, and FranÃğois-Xavier Standaert. Systematic study of decryption and re-encryption leakage: the case of kyber. Cryptology ePrint Archive, Report 2022/036, 2022. https://ia.cr/2022/036.

[BDH+21] Shivam Bhasin, Jan-Pieter DâĂŹAnvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 334–359, 2021.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Mod-lwr based kem algorithm specification and supporting documentation. In *Submission to the NIST post-quantum project (2019)*, 2019. https://www.esat.kuleuven.be/cosic/publications/article-3055.pdf.

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*, pages 2–9, 2019.

[GHJ+21] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander SchrÃűder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike. Cryptology ePrint Archive, Report 2021/1485, 2021. https://ia.cr/2021/1485.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on frodokem. In *Annual International Cryptology Conference*, pages 359–386. Springer, 2020.

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In *ICICS*, 1999.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked cca2 secure kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 88–113, 2021.

[ISUH21] Akira Ito, Kotaro Saito, Rei Ueno, and Naofumi Homma. Imbalanced data problems in deep learning-based side-channel attacks: analysis and solution. *IEEE Transactions on Information Forensics and Security*, 16:3790–3802, 2021.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, et al. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*. US Department of Commerce, National Institute of Standards and Technology, 2020. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf.

[Moo16] Dustin Moody. *Post Quantum Cryptography Standardization: Announcement and outline of NIST's Call for Submissions*. PQCrypto 2016, Fukuoka, Japan, 2016. https://csrc.nist.gov/Presentations/2016/Announcement-and-outline-of-NIST-s-Call-for-Submis.

[MU10] Alfred Menezes and Berkant Ustaoglu. On reusing ephemeral keys in diffie-hellman key agreement protocols. *International Journal of Applied Cryptography*, 2(2):154–158, 2010.

[ND21] NIST and DHS. *Preparing for Post-Quantum Cryptography: Infographic*. 2021. https://www.dhs.gov/sites/default/files/publications/post-quantum_cryptography_infographic_october_2021_508.pdf.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked ind-cca secure saber kem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 4:676âĂŞ707, 2021.

[QCZ+21] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based nist candidate kems. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 92–121. Springer, 2021.

[RBRC21] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) nist pqc candidates for practical message recovery attacks. *IEEE Transactions on Information Forensics and Security*, 2021.

[RDB+21] Prasanna Ravi, Suman Deb, Anubhab Baksi, Anupam Chattopadhyay, Shivam Bhasin, and Avi Mendelson. On threat of hardware trojan to post-quantum lattice-based schemes: A key recovery attack on SABER and beyond. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 11th International Conference, SPACE 2021, Kolkata, India, December 10-13, 2021, Proceedings*, volume 13162 of *Lecture Notes in Computer Science*, pages 81–103. Springer, 2021.

[REB+22] Prasanna Ravi, Martianus Frederic Ezerman, Shivam Bhasin, Anupam Chattopadhyay, and Sujoy Sinha Roy. Will you cross the threshold for me? generic side-channel assisted chosen-ciphertext attacks on ntru-based kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 722–761, 2022.

[RR21] Prasanna Ravi and Sujoy Sinha Roy. Side-channel analysis of lattice-based pqc candidates. NIST Round 3 Seminars - Post-Quantum Cryptography, 2021. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/seminars/mar-2021-ravi-sujoy-presentation.pdf.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/em analysis on post-quantum kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 296–322, 2022.

[XPSR+21] Zhuang Xu, Owen Michael Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Transactions on Computers (Early Access)*, 2021.

[ZCD21] Xiaohan Zhang, Chi Cheng, and Ruoyu Ding. Small leaks sink a great ship: an evaluation of key reuse resilience of pqc third round finalist ntru-hrss. In *International Conference on Information and Communications Security*, pages 283–300. Springer, 2021.