Machine Learning-Based Profiling Attack on the Kyber ML-KEM

Diimppul Bharath
IIT Roorkee
Supervised by: Prof. Rajat Sadukhan

November 24, 2025

Abstract

As Kyber (ML-KEM) becomes standardized by NIST, its implementations' resilience against side-channel attacks (SCA) is a critical concern. This paper presents a machine learning (ML) based profiling attack on the pair-pointwise multiplication (`basemul`) function, using the public dataset from Rezaeezade et al. Unlike classical template attacks that use distance-based scoring, our approach employs a Random Forest classifier to build a 9-class profile based on the Hamming Weight of intermediate values. We replace traditional Sum of Squared Difference (SOSD) for feature selection with an ANOVA-like F-statistic. We demonstrate that scoring key guesses using the model's log-likelihood results in a highly efficient attack, successfully recovering the secret subkeys `a[0]` and `a[1]` with a small number of traces. Our analysis also reveals distinct key equivalence classes, demonstrating the model's ability to learn structural properties of the target's finite field arithmetic.

Keywords: Side-Channel Analysis, Machine Learning, Random Forest, Post-Quantum Cryptography, Kyber, ML-KEM, Profiling Attack.

Contents

1 Introduction

The NIST standardization of post-quantum cryptography (PQC) has led to the selection of CRYSTALS-Kyber (ML-KEM) as a future standard. While mathematically secure, physical implementations of these algorithms are susceptible to side-channel analysis (SCA).

Recent research has provided public datasets for this analysis, notably the one by Rezaeezade et al. [2], which captures power consumption during the `basemul` operation in the Kyber decapsulation process. While classical template attacks have been shown to be effective on this dataset [1], this work explores the application of modern machine learning techniques.

We replace the classical template-building and distance-scoring methodology with a supervised learning approach. We demonstrate that a Random Forest classifier, trained to predict the Hamming Weight of an intermediate value, can be used to perform a highly effective profiling attack. Instead of a distance-based score, we use the total log-likelihood from the classifier's probability outputs to rank key guesses, leading to a rapid key recovery.

2 Background

This section briefly collects the essential concepts the reader should know to understand the methodology that follows: (1) SCA with machine learning, (2) the ANOVA-like between-class sum-of-squares (BSS) used for PoI selection, and (3) the log-likelihood scoring method for classification-based profiling attacks.

2.1 Side-Channel Analysis (SCA) using Machine Learning

Traditional profiling SCA (e.g., template attacks) build a statistical model of leakage for each intermediate value using a profiling device. Modern approaches replace or augment these templates with supervised machine learning models (e.g., Random Forests, SVMs, neural networks). In the profiling phase, labeled traces (where the intermediate variable is known) are used to train a classifier that maps trace features to leakage labels (often Hamming Weight or its bits). In the attack phase, the trained model provides class probabilities for unknown traces; these probabilities are combined across traces and candidate key hypotheses to score and rank guesses. Benefits of ML-based profiling include robustness to high-dimensional traces, automatic feature learning/selection when used with many PoIs or learned features, and flexibility in scoring (probability-based scoring rather than distance metrics).

2.2 ANOVA-like Between-Class Sum of Squares (BSS) for PoI Selection

Feature selection (Points of Interest – PoIs) is critical to reduce dimensionality and focus on time samples that most differentiate leakage classes. An effective metric is an ANOVA-like Between-Class Sum of Squares (BSS) score computed per time sample t:

$$\mathrm{Score}(t) \;=\; \sum_{c=0}^{C-1} n_c\big(\mu_c(t) - \mu_{\mathrm{all}}(t)\big)^2,$$

where C is the number of classes, n_c is the count of traces in class c, $\mu_c(t)$ the mean of class c at sample t, and $\mu_{\text{all}}(t)$ the overall mean at t. This score approximates the numerator of the ANOVA F-statistic (between-class variance). High BSS values indicate time samples where class means diverge most — ideal PoIs for classification. Compared to SOSD or SNR, BSS explicitly accounts for between-class separation and class sizes; it is simple to compute and effective in practice for multiclass HW targets.

2.3 Log-Likelihood Scoring in Classification-Based Attacks

Given a trained classifier that outputs class probabilities $P(c \mid \mathbf{x})$ for each class c and a trace feature vector $\mathbf{x}$, we can score a key hypothesis k_g by summing log-probabilities of the predicted class for that hypothesis across attack traces. If for trace j the hypothetical class under k_g is $c_j(k_g)$, then the total log-likelihood score is:

$$\text{Score}(k_g) = \sum_{j=1}^{N} \log\left(P(c_j(k_g) \mid \mathbf{x}_j)\right).$$

This is equivalent to the log of the product of per-trace probabilities and corresponds to selecting the hypothesis that maximizes the likelihood of the observed traces under the model. Compared with distance-based scoring (e.g., SOSD with Gaussian templates), log-likelihood exploits the full probabilistic output of modern classifiers and often yields better separation between correct and incorrect guesses, especially when class-conditional distributions are non-Gaussian or multimodal.

3 Attack Methodology

3.1 Dataset and Target

We use the public power trace dataset from Rezaeezade et al. [2]. This dataset contains 100,000 power [1] measurements of the Kyber `basemul` function running on an STM32F3 microcontroller. The target operation is the pair-pointwise multiplication, specifically targeting the `fqmul(a[i], b[1])` operations involving the secret key coefficients `a[0]` and `a[1]`.

Preprocessing & Compression Raw traces in the dataset are high-dimensional (each trace originally has 50,000 samples). To reduce noise and speed up downstream processing we optionally compress traces by block-averaging. Following the pipeline used in the technical report, we use a compression factor `cf = 10` (i.e., average every 10 consecutive samples) which reduces each trace length to $\lceil L_{\text{raw}}/\text{cf}\rceil$. No further filtering or alignment was performed in our experiments beyond this step unless explicitly stated.

```python
def compress(raw, cf):
    N, L = raw.shape
    CL = (L + cf - 1) // cf
    out = np.zeros((N, CL), dtype=np.float32)
    for i in range(CL):
        start, end = i*cf, min(L, (i+1)*cf)
        out[:, i] = raw[:, start:end].mean(axis=1)
    return out

# Example usage:
# traces_compressed = compress(traces_raw, cf=10)
```

Listing 1: Block compression

Compression simplifies PoI computation and accelerates classifier training. The compressed traces are standardized later before profiling.

Shuffle and split After optional compression the dataset is randomly shuffled to remove any acquisition-order bias and then split into profiling and attack sets. We use an 80/20 split in most experiments:

```
# assuming traces (N, L) and optional compression already applied
perm = np.random.RandomState(seed=0).permutation(N)
traces_shuffled = traces[perm]
# if additional arrays (nonces, intermediates) exist, shuffle them with same perm

profFrac = 0.8
prof = int(profFrac * N)
att  = N - prof

profiling_traces = traces_shuffled[:prof]
attack_traces    = traces_shuffled[prof:]
```

Listing 2: Shuffling and splitting compressed traces.

This ensures the classifier is trained on profiling traces that are independent of the attack traces.

3.2 Finite-Field Arithmetic

Kyber's arithmetic is performed modulo $q = 3329$ using Montgomery reduction. We implement `montgomery_reduce` and `fqmul` in Python to compute the field products used during the key guess evaluation. This implementation mirrors the logic of the C reference code.

```
KYBER_Q = 3329
QINV = -3327  # modular inverse of q mod 2^16

def int16(x):
    x = x & 0xFFFF
    return x - 0x10000 if x & 0x8000 else x

def uint16(x):
    return x & 0xFFFF

def montgomery_reduce(a):
    t = int16(a)
    t = int16(t * QINV)
    res = (a - (int(t) * KYBER_Q)) >> 16
    return int16(res)

def fqmul(a, b):
    prod = int16(a) * int16(b)
    return uint16(montgomery_reduce(prod))
```

Listing 3: Python implementation of `fqmul` with Montgomery Reduction.

3.3 Leakage Model and Labeling

Our attack profiles the leakage of an intermediate value from the `fqmul` operation. We define our leakage model based on the Hamming Weight (HW) of the Most Significant Byte (MSB) of the intermediate result c (fqmul(a[i],b[1])). Instead of a binary model as used in classical template attacks on this dataset [1], we use a multi-class model with 9 distinct classes, one for each possible HW (0 through 8).

The labels used for training are derived from the dataset's known intermediate values(or we can use fqmul(558,b[1]) and fqmul(17,b[1]) like in [1] Nkotto et al., ePrint 2025/1577). Each label corresponds to the HW of the MSB of the intermediate c:

```
HW_LOOKUP = np.array([bin(i).count("1") for i in range(256)], dtype=np.uint8)
labels = np.zeros(prof, dtype=np.int8)
for i in range(prof):
    val = int(nonces[i, idx])   # intermediate value from dataset
    msb = (val >> 8) & 0xFF
    labels[i] = HW_LOOKUP[msb]
```

Listing 4: Label generation for profiling traces.

Thus, the classifier learns to associate trace samples with the Hamming Weight of the internal operation's MSByte.

3.4 Feature Selection (Points of Interest)

To reduce dimensionality, we select Points of Interest (PoIs) where the leakage between our 9 HW classes is maximal. Instead of the Sum of Squared Difference (SOSD), we compute a "between-class sum of squares" (BSS) score for each time sample t:

$$\text{Score}(t) = \sum_{c=0}^{8} n_c(\mu_c(t) - \mu_{\text{all}}(t))^2$$

where n_c is the number of traces in class c, $\mu_c(t)$ is the mean of traces in class c at time t, and $\mu_{\text{all}}(t)$ is the global mean at time t. High scores identify time samples where class means differ most. The top N indices are chosen as PoIs . In practice, the weighting by n_c has little effect on the ranking of PoIs—removing it (using an unweighted BSS) still yields almost identical peaks and attack performance. The approach remains computationally simple and well suited for multiclass HW targets.

```
def compute_between_class_score(traces, labels, prof):
    mu_all = traces[:prof].mean(axis=0)
    score  = np.zeros(traces.shape[1], dtype=np.float64)
    for c in np.unique(labels[:prof]):
        mask = (labels[:prof] == c)
        mu_c = traces[:prof][mask].mean(axis=0)
        score += np.sum(mask) * (mu_c - mu_all)**2
    return score

score = compute_between_class_score(traces, labels, prof)
pois  = np.argsort(-score)[:NUM_POI]
```

Listing 5: Between-class variance (BSS) based PoI selection.

Figures 1 and 2 illustrate the resulting BSS distributions for the targets `a[0]` and `a[1]`. Their shapes closely resemble the PoI curves reported in [1] Nkotto et al., ePrint 2025/1577, confirming that our preprocessing and PoI selection pipeline reproduces the same leakage regions observed in that study.

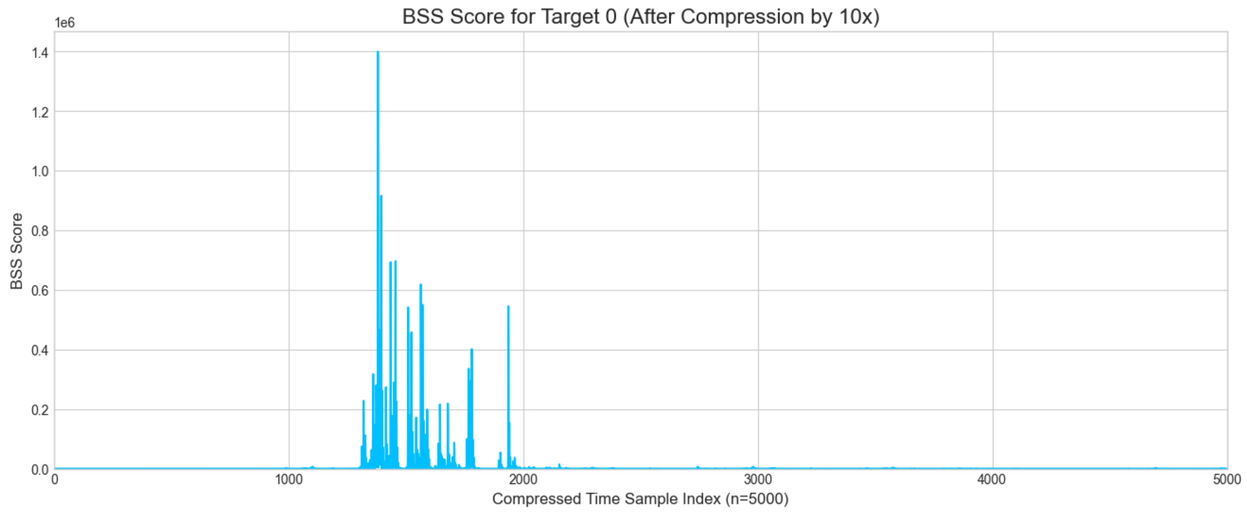


Figure 1: ANOVA-like (BSS) score across all time samples. Peaks indicate optimal PoIs for distinguishing the 9 HW classes for `a[0]`.

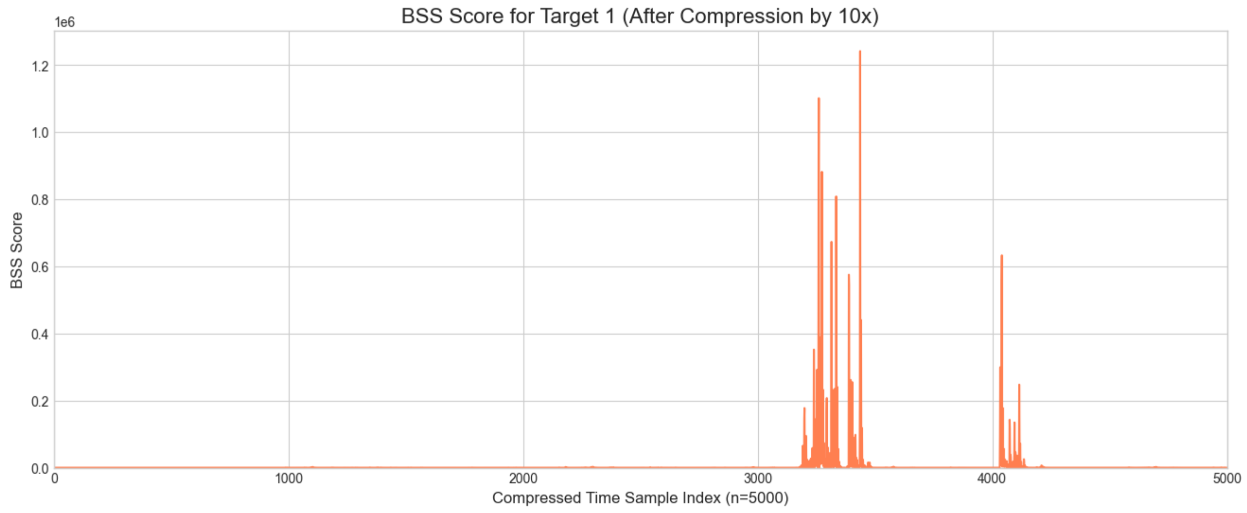


Figure 2: ANOVA-like (BSS) score across all time samples. Peaks indicate optimal PoIs for distinguishing the 9 HW classes for `a[1]`.

A notable finding is the attack's robustness to the number of PoIs selected. We observed that the attack consistently achieves Rank 1 success whether using 500, 200, 100, or as few as 10 PoIs. This indicates that the leakage is highly concentrated in a few key time samples, and our model can effectively capture the pattern with a relatively small feature set.

3.5 Profiling Phase: RandomForest Training

After PoI selection, each trace is reduced to its PoI samples and standardized. The profiling classifier is trained to predict the 9 HW classes using a Random Forest model:

```
X_train = traces[:prof, pois]
X_att   = traces[prof:, pois]

scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_att_s   = scaler.transform(X_att)

clf = RandomForestClassifier(n_estimators=200, n_jobs=-1, random_state=0)
clf.fit(X_train_s, labels)
# clf.predict_proba(X_att_s) gives per-trace probability distributions
```

Listing 6: RandomForest training on profiling traces.

The model outputs per-class probabilities $P_j(\text{HW})$ for each attack trace j. These probabilities are used in log-likelihood computation.

3.6 Attack Phase: Log-Likelihood Scoring

We score all 2^{16} possible key guesses. For a given guess k_g, its score is the total log-likelihood, computed over all N attack traces t_j:

$$\text{Score}(k_g) = \sum_{j=1}^{N} \log \left(P(c(k_g, b_j) \mid t_j) \right)$$

Where:

- b_j is the known ciphertext coefficient for trace t_j.
- $c(k_g, b_j)$ is the hypothetical HW class (0–8) of $\text{fqmul}(k_g, b_j)$.
- $P(c \mid t_j)$ is the classifier's predicted probability.

```
logprobs = np.log(clf.predict_proba(X_att_s))   # (N_attack, 9)
best_guess, best_ll = 0, -np.inf

for a in range(65536):
    products = np.array([fqmul(a, int(b)) for b in nonces[prof:, 1]], dtype=np.
   uint16)
    msb_vals = (products >> 8) & 0xFF
    hw_vals  = HW_LOOKUP[msb_vals]
    ll = np.sum([logprobs[j, hw_vals[j]] for j in range(att)])
    if ll > best_ll:
        best_ll, best_guess = ll, a
```

Listing 7: Log-likelihood scoring and key recovery.

The key guess with the highest (least negative) log-likelihood is identified as the recovered subkey.

3.7 Implementation Notes and Summary

- The dataset must be shuffled before splitting to remove any correlation from trace acquisition order.
- A compression factor of 10 (`cf=10`) offers a good tradeoff between noise reduction and leakage preservation.
- Typical PoI counts between 100 and 500 yield stable performance.
- Precomputing $\log P(c|t_j)$ for all traces accelerates the 2^{16} guess loop.
- The same methodology applies to both target coefficients `a[0]` and `a[1]`.

The complete pipeline is:

1. Load and optionally compress traces.
2. Shuffle and split into profiling/attack sets.
3. Label profiling traces by HW(MSB(c)).
4. Compute between-class (BSS) scores and select PoIs.
5. Standardize features and train RandomForest classifier.
6. Compute log-probabilities for attack traces.
7. Evaluate all 2^{16} key guesses using log-likelihood scoring.
8. Identify the key with maximum likelihood as the recovered secret.

---

4 Results and Analysis

4.1 Key Recovery

The attack was executed using 1000 total traces (800 for profiling, 200 for attack). The ground truth keys (known from the dataset) are `a[0]` $= 558$ and `a[1]` $= 17$.

Our attack successfully recovered both keys at Rank 1. The Top 10 guesses and their log-likelihood (LL) scores are shown in Table 1.

4.2 Analysis of Key Equivalence Classes

A key finding is that the top-ranked incorrect guesses are not random. As seen in Table 1, the top guesses fall into two distinct equivalence classes modulo `KYBER_Q` $= 3329$.

- **For a[0]:** The top guesses are all $\equiv 558 \pmod{3329}$ or $\equiv 2843 \pmod{3329}$.
- **For a[1]:** The top guesses are all $\equiv 17 \pmod{3329}$ or $\equiv 2302 \pmod{3329}$.

Table 1: Top 10 key guesses and their log-likelihood scores (800 profiling traces, 200 attack traces) for poi=200.

Target: a[0] (True = 558)		**Target: a[1] (True = 17)**	
Guess	LogLikelihood	Guess	LogLikelihood
558	**-153.1047**	**17**	**-181.0522**
62765	-221.0199	3346	-216.6137
3887	-221.5682	62224	-248.8193
7216	-287.5244	58895	-316.1385
59436	-322.7492	55566	-384.6001
56107	-390.5059	6675	-386.4654
10545	-423.2735	10004	-423.4791
52778	-458.4242	52237	-452.7894
13874	-490.9208	13333	-491.1058
17203	-524.5801	16662	-524.8121

This reveals a "ghost key" class, which is related to the true key by a constant additive differential:

$$2843 - 558 = \mathbf{2285}$$

$$2302 - 17 = \mathbf{2285}$$

This demonstrates that the leakage of $\texttt{fqmul}(k, b)$ is highly correlated with the leakage of $\texttt{fqmul}(k + 2285, b)$. Our ML model was powerful enough to detect this structural property of the `fqmul` implementation, correctly identifying both the true key class and its "ghost" class.

4.3 Attack Efficiency (Key Rank)

The minimum number of total traces required for a stable Rank-1 recovery is variable. To ensure our profiling and attack sets were independent and not biased by contiguous trace collection, we **randomly shuffled the dataset** before splitting. Using a standard **80/20 split** (80% profiling, 20% attack) on the shuffled data, a stable Rank-1 recovery was achieved with approximately **750 total traces** for a_1(**150 attack traces**). We note that this rank can fluctuate, temporarily increasing before settling at 1.

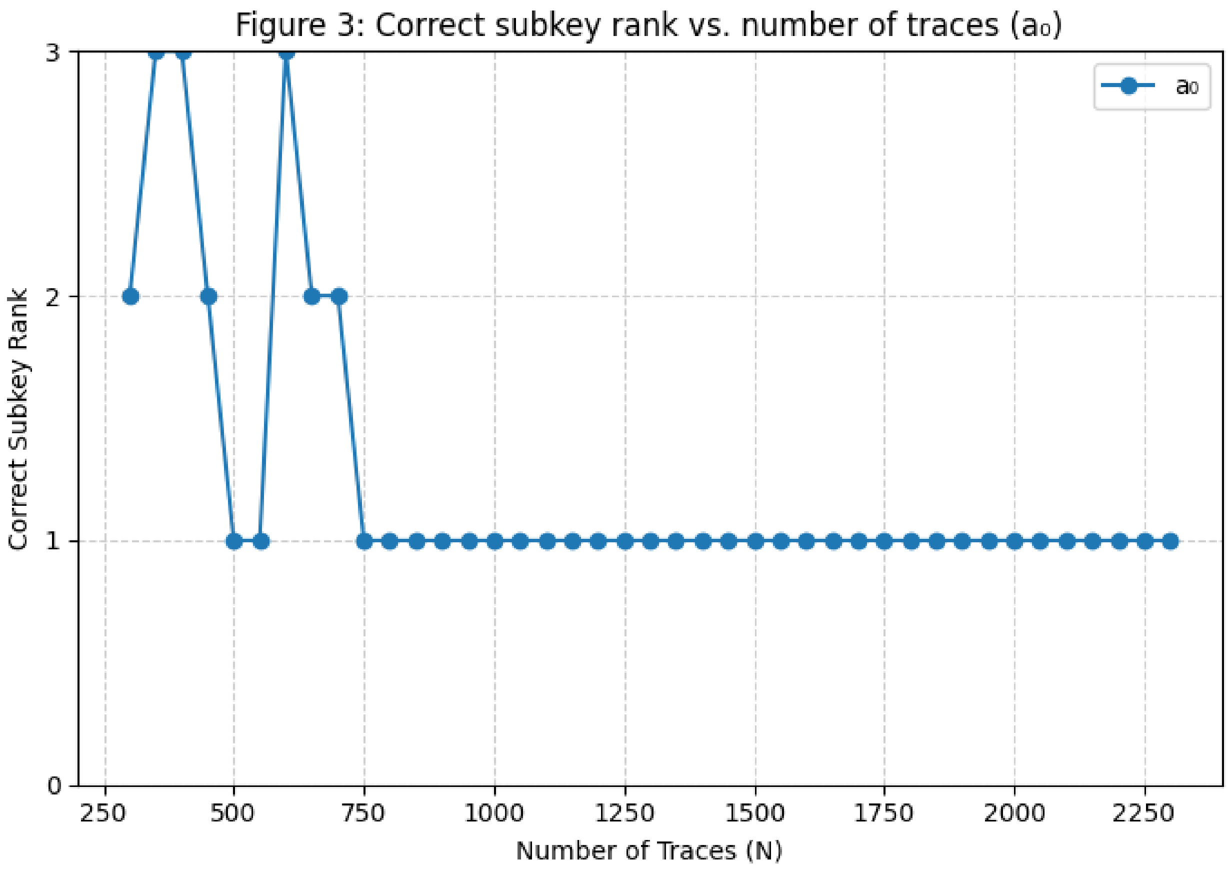
Interestingly, the choice of data split significantly impacts trace efficiency. We found that **different splits lead to different total trace requirements**. For example, when changing the split to **70/30** (70% profiling, 30% attack), the total number of traces required for a successful attack decreased to **less than 750**. This suggests that providing the model with a larger, more diverse set of attack traces (30% vs 20% of the total) can improve scoring accuracy and reduce the overall number of traces needed.

Tie-inclusive ranking: definition and algorithm. We adopt a *tie-inclusive* ranking rule for reporting key ranks. Under this rule the rank of the true key is defined as the number of guesses whose score is $\geq$ the true key's score. Formally, given an array of scores $S[g]$ for all guesses $g \in \{0, \ldots, 65535\}$ and the true guess index $g^\star$, the tie-inclusive rank R is:

$$R = \sum_g \mathbf{1}\{S[g] \geq S[g^\star]\}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. This convention means:

- If the true guess ties with the unique top-scoring guess, $R = 1$.



- If t guesses tie for the top score and the true guess is among them, $R = t$.
- If several guesses have scores strictly greater than the true score, R counts them plus any guesses tied with the true score.

Why this rule? Tie-inclusive ranking communicates how many candidate keys are *as likely or more likely* than the true key according to the model's scoring. For operational decisions (e.g., how many guesses to brute-force after ML scoring), R directly gives the number of candidates that must be considered to guarantee inclusion of the true key.

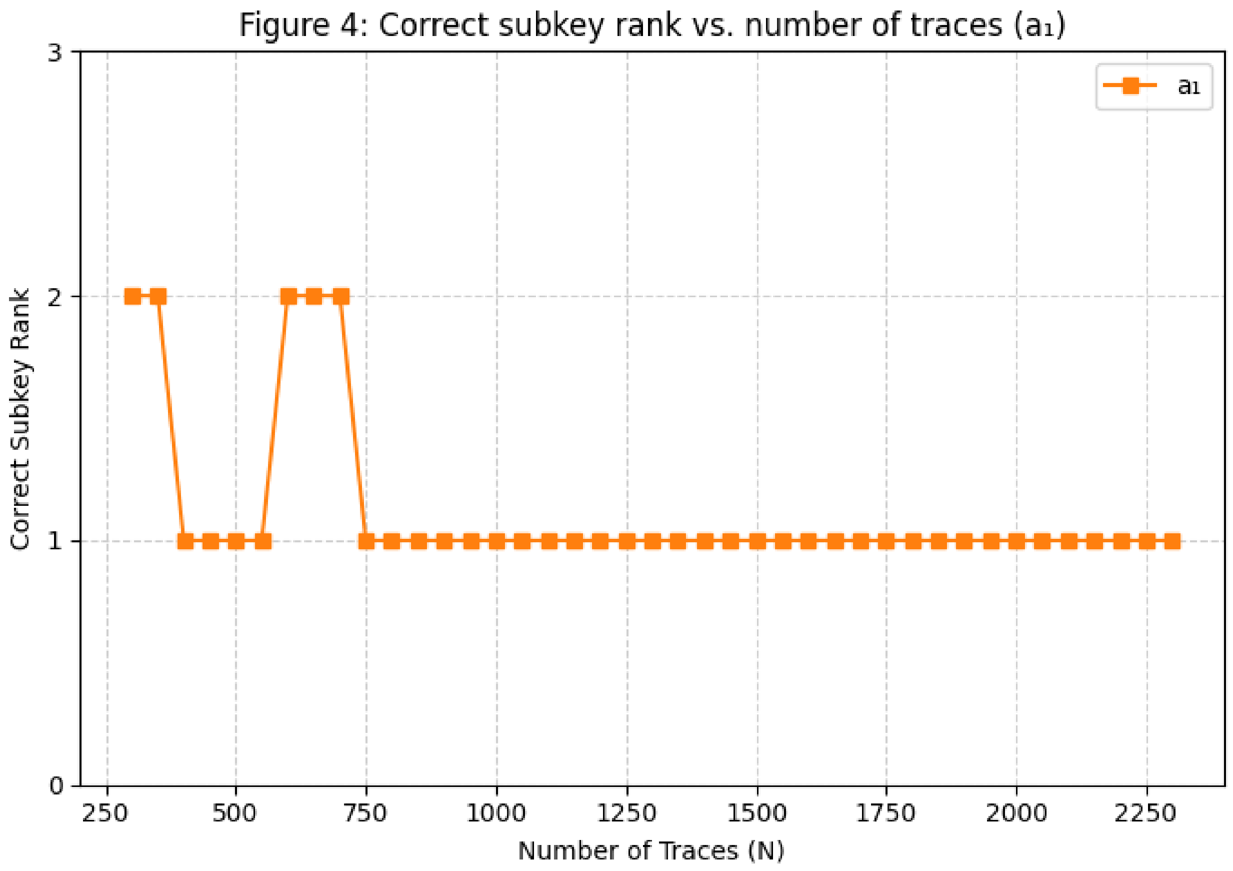
In `numpy` this is a single efficient line:

```
R = int(np.sum(all_scores >= all_scores[true_guess_index]))
```

---

5 Conclusion

We presented a highly effective profiling side-channel attack on the Kyber `basemul` function using a Random Forest classifier. By profiling the 9-class Hamming Weight of the MSByte and using a log-likelihood scoring function, we successfully recovered the secret subkeys `a[0]` and `a[1]` with high confidence.



The complete source code and experimental scripts are available on GitHub [1].

Our ML-based approach proved to be extremely trace-efficient, achieving a Rank-1 recovery with **around 750 total traces** (using an 80/20 split), a number that drops below 750 when adjusting the split to 70/30. Furthermore, our analysis of the ranked key guesses uncovered a constant additive differential (+2285), revealing that the classifier learned deep structural properties of the target's finite field implementation. This work confirms that standard, unprotected implementations of ML-KEM are highly vulnerable to modern machine learning-based attacks.

References

[1] S. Nkotto. "A Template SCA Attack on the Kyber/ML-KEM Pair-Pointwise Multiplication." *Cryptology ePrint Archive, Paper 2025/1577.* 2025.

[2] A. Rezaeezade et al. "Side-Channel Power Trace Dataset for Kyber Pair-Pointwise Multiplication on Cortex-M4." *Cryptology ePrint Archive, Paper 2025/811.* 2025.

[3] L. Weissbart, S. Pk, L. Batina. "One trace is all it takes: Machine Learning-based Side-channel Attack on EdDSA." *Cryptology ePrint Archive, Paper 2019/358.* 2019.

[1]`https://github.com/diimppulg/ML-KEM/tree/main`