Cache-Timing Attack Against HQC

Senyang Huang[1], Rui Qi Sim[2], Chitchanok Chuengsatiansup[3], Qian Guo[1] and Thomas Johansson[1]

[1] Lund University
[2] The University of Adelaide
[3] The University of Melbourne

Abstract. In this paper, we present the first chosen-ciphertext (CC) cache-timing attacks on the reference implementation of HQC. We build a cache-timing based distinguisher for implementing a plaintext-checking (PC) oracle. The PC oracle uses side-channel information to check if a given ciphertext decrypts to a given message. This is done by identifying a vulnerability during the generating process of two vectors in the reference implementation of HQC. We also propose a new method of using PC oracles for chosen-ciphertext side-channel attacks against HQC, which may have independent interest.

We show a general proof-of-concept attack, where we use the Flush+Reload technique and also derive, in more detail, a practical attack on an HQC execution on Intel SGX, where the Prime+Probe technique is used. We show the exact path to do key recovery by explaining the detailed steps, using the PC oracle. In both scenarios, the new attack requires $53,857$ traces on average with much fewer PC oracle calls than the timing attack of Guo et al. CHES 2022 on an HQC implementation.

Keywords: Side-channel attacks · Code-based cryptography · NIST PQC standardization · HQC

1 Introduction

Quantum computing is rising as a new fundamental area in computer science research. Security estimates for most of our deployed cryptographic primitives are seriously affected by this development. Predictions of future developments are contested [Kal20] and it is hard to know when a large-scale quantum computer outperforming classic ones in cryptanalysis will appear. However, it is important and well established that the transition to post-quantum secure cryptographic algorithms must happen well before this time. Encrypted sensitive data might be stored for cryptanalysis at some time in the future.

Newly proposed cryptographic primitives must consider possible attacks from classical adversaries as well as from quantum ones. But protection against theoretical attacks is not sufficient. Cryptographic algorithms need to be implemented when deployed, and deployed implementations need to withstand also other practical implementation attacks, such as side-channel attacks [Koc96] and fault attacks [BDL97].

An important aspect of a cryptographic design is the difficulty and complexity of securing an implementation of the design against relevant implementation attacks. It is often emphasized that a cryptographic algorithm offering simplicity and efficiency in its secure implementation is an important parameter when comparing cryptographic algorithms.

Side-channel attacks [Koc96] assume an additional channel of information for the attacker, often being measurements of power consumption of the device under attack, or measurements of timing variations. Timing attacks was first described by Kocher [Koc96] and they are notably one of the most dangerous implementation attacks. The information channel may come from direct time measurements on the targeted device, but could alternatively involve an attacker only connected to the target device through a communication channel. This may even extend to the possibility of mounting an attack remotely over a network [BB05, BT11, KPVV16, MSEH20, MBA$^+$21]. One of the well known remote exploitations is the Lucky Thirteen attack [AP13] on TLS.

Timing attacks exploit the timing variations which have a dependence with the secret key. Collecting enough timing information might allow the recovery of (a part of) the secret key. These timing variations may be in form of direct variations in the executed code (say from if-then statements), or it might come from other indirect variations on the executing device. A main class of such variations are the cache-timing attacks, where we use the timing difference appearing from the use of cache memory. Caches are typically shared among executing programs, which means that a malicious program can monitor cache activities and extract information on the execution of the other programs using the shared cache. Past research has demonstrated the huge impact of cache attacks on extracting secret keys in cryptographic primitives [CAGTB19, CFSY22, CGYZ22, DDME$^+$18, GPTY18, GPS$^+$20, LYG$^+$15, OST06, PGBY16] or other sensitive information [GRB$^+$17, GSM15, SKH$^+$19, YFT20].

HQC [AAB$^+$20] is a proposed code-based post-quantum secure key encapsulation mechanism (KEM), where the security is based on the hardness of decoding random quasi-cyclic codes in the Hamming metric. It has been selected as a KEM candidate in the fourth round of the NIST PQC standardization project [NIS]. NIST has further claimed to standardize either HQC or BIKE at the end of the fourth round, making HQC one of the most relevant active candidates in this NIST PQC standardization effort.

As such, much attention has been devoted to side-channel cryptanalysis of HQC. So far, the focus has mainly been on power analysis or extracting electromagnetic traces [SRSWZ20, GLG22b, SHR$^+$22, GLG22a]. Timing attacks are in general not directly applicable due to the default option of constant-time implementations. In [GHJ$^+$22], however, a key-recovery timing attack on HQC was described, caused by an implementation weakness in the step that used a procedure called rejection sampling in randomness generation. This is the only known timing attack on HQC and one can protect against it by a more careful implementation of the rejection sampling step. The attack efficiency was further improved in [GNNJ23] with techniques from coding theory.

1.1 Contributions

In this paper, we present the first chosen-ciphertext cache-timing attacks on the reference implementation of HQC. The setting is a chosen-ciphertext side-channel attack, modeled by building a plaintext-checking (PC) oracle. The PC oracle uses side-channel information to check if a given ciphertext decrypts to a given message. With access to the public key, we can encrypt messages. Together with the PC oracle, this is sufficient to perform a key-recovery attack on such post-quantum primitives. We show the exact path to do key-recovery by explaining the detailed steps, spliting into an online phase and an offline phase, using the PC oracle. To the best of our knowledge, this attack is the *first* cache-timing attack among the many chosen-ciphertext side-channel attacks against NIST PQC KEM candidates.

New techniques. We build a cache-timing based distinguisher for implementing the PC oracle. This is done by identifying a vulnerability during the randomness generation process of two vectors in the reference implementation of HQC. In this work, we formally introduce a concept called *cache line indicator* to describe the timing behavior in the implementation of HQC. Deducing the cache line indicator gives the cache-timing based distinguisher, providing us the basis to build the PC oracle for a full key-recovery attack.

Our second contribution is a new key-recovery approach from a PC oracle against HQC. We exploit the sparsity of the HQC key and search for a vector v such that the vectors v and $v - y$ decode to two different messages m_1 and m_2, where one of the two messages are equal to a selected message m. Since one decoding block (called inner code block in the concatenated code construction of HQC) of y is of Hamming weight fewer than 1 with a good probability (of larger than or close to 50%), we could recover these inner blocks of y with further offline processes. We note that similar techniques can be applied to the secret vector x. Thus, one can collect a sufficient number (i.e., close to n) of secret entries and then recover the full key (x, y) with Gaussian Elimination or some light information set decoding procedures.

Implementations. We show a proof-of-concept attack, where we use the Flush+Reload technique [YF14] to realize the cache line indicator. We also derive, in more detail, a practical attack on an HQC execution on Intel SGX, where the Prime+Probe technique [OST06, LYG+15] is used. The corresponding model is discussed and experimental results are presented. In the Flush+Reload setting, we need $53,857$ traces on average (i.e., decryption oracle calls) to fully recover the secret key. We can obtain a similar trace/sample complexity in a practical Prime+Probe attack on an Intel SGX platform using the SGX-Step [VBPS17] framework.

Comparisons with [GHJ+22]. Our attack bears similarities to the one in [GHJ+22] as both employ timing attacks on HQC; nevertheless, the underlying mechanisms of these two attacks significantly differ. Firstly, the work [GHJ+22] observes the number of calls to the sampling function (more calls take longer time), whereas our novel attack focuses on secret-dependent cache-timing access during vector sampling. Second, the attack in [GHJ+22] recovers a portion of the secret y by introducing noise perturbations to repeatedly alter the status of the HQC decryption algorithm (correct or failure) in an online manner; in contrast, we introduce a novel two-stage procedure in which the offline phase identifies a state already at the decoding boundary—where a minor perturbation can affect the decryption result—and the online phase leverages the sparsity of y, using the targeted subvector of y as the perturbation source to detect if the subvector is all zeroes or has a weight of one. In this new method, we also target the secret vector x to recover additional subvectors with a weight limited to one. The new algorithm capitalizes on the extreme sparsity of the secret in HQC, enabling the recovery of secret blocks (including hundreds of secret bit entries) after a reasonable number of repetitions using a different offline-determined error pattern at the decoding boundary each time, in contrast to the bit-wise recovery method in [GHJ+22]. Our newly proposed key recovery method significantly reduces the number of required queries.

1.2 Organisation

The remaining parts of the paper are organized as follows. In Section 2, we present the background information on the HQC scheme, cache-timing attacks, and the Intel SGX platform. Section 3 presents the novel methodology to build a distinguisher to distinguish if the HQC.CPAPKE.Dec function succeeds based on a cache-timing vulnerability in the referenced HQC implementation. This is followed by a full-key recovery attack against HQC from the new cache-timing distinguisher in Section 4. We then present the experimental results in Section 5 and finally conclude the paper in Section 6.

2 Background

In this section, we present necessary background on HQC and cache-timing attacks. We denote $\mathbb{F}_2$ the binary finite field and $\mathcal{R}$ the polynomial ring $\mathbb{F}_2[X]/(X^n - 1)$ for a positive integer n. Given a polynomial $h \in \mathcal{R}$, we denote $w_{\mathrm{H}}(h)$ the Hamming weight of h. The notation $\mathsf{sample}(\mathcal{R})$ means we sample an element from $\mathcal{R}$ uniformly at random and $\mathsf{sample}(\mathcal{R}, w)$ includes the additional constraint that the sampled element needs to have, namely, a Hamming weight of w.

2.1 Hamming Quasi Cyclic – HQC

HQC [AAB+20] is a code-based post-quantum IND-CCA secure KEM, whose security is based on the hardness of decoding random quasi-cyclic codes in the Hamming metric. It has been solicited as a KEM candidate in the fourth round of the NIST PQC standardization [NIS]. NIST claimed to standardize one code-based KEM primitive between HQC and BIKE at the end of the fourth round. Similar to other NIST PQC PKE/KEM candidates, the HQC proposal starts with an IND-CPA version named HQC.CPAPKE and then presents an INC-CCA KEM named HQC.CCAKEM through a CCA transform (in HQC, the Hofheinz-Hövelmanns-Kiltz (HHK) transformation [HHK17] is used).

The PKE version of HQC. The algorithm HQC.PKE (shown in Figure 1) consists of three sub-procedures, HQC.CPAPKE.KeyGen, HQC.CPAPKE.Enc, and HQC.CPAPKE.Dec. In HQC.CPAPKE.KeyGen, the algorithm uniformly samples three polynomial elements h, x, and y, where the Hamming weights of x and y are fixed to be w. Then, the secret key is set to be (x, y) and the public key is $(h, s = x + h \cdot y)$. In HQC.CPAPKE.Enc, the algorithm firstly initializes the pseudo-random number generator (PRNG) by a seed θ. Then, the sampling process becomes deterministic. The algorithm then uniformly samples from $\mathcal{R}$ the polynomials r_1 and r_2 with Hamming weight w_r and e with Hamming weight w_e. The ciphertext is then set to be $c = (u, v)$ with $u = r_1 + h \cdot r_2$ and $v = m\mathbf{G} + s \cdot r_2 + e$. The matrix $\mathbf{G}$ depends on the employed linear code that will be described later. The algorithm HQC.CPAPKE.Dec then applies a decoder with the input $v - u \cdot y$ which is

$$m\mathbf{G} + s \cdot r_2 + e - (r_1 + h \cdot r_2) \cdot y = m\mathbf{G} + x \cdot r_2 - r_1 \cdot y + e,$$

since $s = x + h \cdot y$. If the Hamming weight of the error term

$$e' = x \cdot r_2 - r_1 \cdot y + e$$

is small (i.e., bounded by the employed decoder's capability), then the decoder could correct such an error and the decryption succeeds.

```
Input:
Output: sk, pk
    $h$ = sample($\mathcal{R}$)
    $x$ = sample($\mathcal{R}, \omega$)
    $y$ = sample($\mathcal{R}, \omega$)
    $sk = (x, y)$
    $pk = (h, s = x + h \cdot y)$
```

(a) HQC.CPAPKE.KeyGen

```
Input: $pk$, $m$, $\theta$
Output: $c = (u, v)$
    sampleInit($\theta$)
    $r_1$ = sample($\mathcal{R}, \omega_r$)
    $r_2$ = sample($\mathcal{R}, \omega_r$)
    $e$ = sample($\mathcal{R}, \omega_e$)
    $u = r_1 + h \cdot r_2$
    $v = m\mathbf{G} + s \cdot r_2 + e$
```

(b) HQC.CPAPKE.Enc

```
Input: $sk = (x, y)$, $c = (u, v)$
Output: $m$
    $m$=$\mathcal{C}$.Decode($v - u \cdot y$)
```

(c) HQC.CPAPKE.Dec

Figure 1: HQC.CPAPKE

Code construction. In a recent proposal of HQC from June 2021, the scheme designs a decoding approach with concatenated code with an internal duplicated Reed-Muller code and an outer Reed-Solomon code. The resulting code gives a publicly known generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n_1 n_2}$, where $k = 8k_1$.

We show the concrete parameters of HQC in Table 1. The computations in HQC are done in the ambient space $\mathbb{F}_2^n$ and the remaining $n - n_1 n_2$ useless positions are discarded. The concatenated code C combines an internal duplicated Reed-Muller code with the outer Reed-Solomon code. The inner duplicated Reed-Muller code is defined with parameters $[n_2, 8, n_2/2]$ and the outer Reed-Solomon code is $[n_1, k_1, n_1 - k_1 + 1]$. In the encoding procedure, a message $m \in \mathbb{F}_{2^8}^{k_1}$ is encoded into $m_1 \in \mathbb{F}_{2^8}^{n_1}$ by the outer Reed-Solomon code. Then, the inner duplicated Reed-Muller code encodes each byte $m_{1,i}$ into $\bar{m}_{1,i} \in \mathbb{F}_2^{n_2}$, where $0 \le i < n_1$. Thus, we get $m\mathbf{G} = (\bar{m}_{1,0}, \cdots, \bar{m}_{1,n_1-1})$.

Table 1: The HQC parameter sets [AAB$^+$20]. The base Reed-Muller code is defined by the first-order $[128, 8, 64]$ Reed-Muller code.

	RS-S			Duplicated RM						
Instance	n_1	k_1	d_{RS}	Mult.	n_2	d_{RM}	$n_1 n_2$	n	ω	$\omega_r = \omega_e$
hqc-128	46	16	31	3	384	192	17664	17669	66	75
hqc-192	56	24	33	5	640	320	35840	35851	100	114
hqc-256	90	32	49	5	640	320	57600	57637	131	149

Similarly, to decode $V = v - u \cdot y$, $V \in \mathbb{F}_2^{n_1 n_2}$ is divided into n_1 blocks, i.e., $V = (V_0, \cdots, V_{n_1-1})$, where $V_i \in \mathbb{F}_2^{n_2}$ is called an *inner block* and $0 \le i < n_1$. Each V_i is decoded by the inner duplicated Reed-Muller code into $\bar{V}_i \in \mathbb{F}_2^8$, where $0 \le i < n_1$. Then, $\bar{V}$ is obtained as a string of length $8n_1$ bits, which is denoted as $(\bar{V}_0, \cdots, \bar{V}_{n_1-1})$. For each $i \in [0, n_1)$, we call $\bar{V}_i$ an *internal codeword*. It can be seen that $\bar{V}$ is a noisy codeword of the outer Reed-Solomon code, which can be decoded into k_1 elements over $\mathbb{F}_{256}$ and transformed to k_1 message bytes.

The KEM version of HQC. The KEM construction (shown in Figure 2) is based upon the PKE version of HQC, but additionally calls three independent cryptographic hash functions $\mathcal{G}$, $\mathcal{H}$, and $\mathcal{K}$. Note that the seed θ is generated by m, thus the sampling process only depends on m during the encryption.

```
**Input:** $pk$
**Output:** $K$, $c = (u, v)$, $d$
    $m$=sample($\mathbb{F}_2^k$)
    $\theta = \mathcal{G}(m)$
    $c$ =HQC.CPAPKE.Enc($pk, m, \theta$)
    $K = \mathcal{K}(m, c)$
    $d = \mathcal{H}(m)$
```

(a) HQC.CCAKEM.Encaps

```
**Input:** $sk = (x, y)$, $c = (u, v)$, $d$
**Output:** $K$
    $m'$=HQC.CPAPKE.Dec($sk, c$)
    $\theta' = \mathcal{G}(m')$
    $c'$ =HQC.CPAPKE.Enc($pk, m', \theta'$)
    **if** $c \neq c'$ & $d \neq \mathcal{H}(m')$ **then**
        $K = \perp$
```

(b) HQC.CCAKEM.Decaps

Figure 2: HQC.CCAKEM

2.2 Cache Attacks

Cache. The cache is a small bank of memory that bridges the speed gap between the fast processor and the slow memory by utilizing temporal and spatial locality. That is, the memory is divided into *lines* and recently used lines are stored in the cache. When the processor accesses memory, it first checks whether the requested memory address is in the cache. If it is, indeed, in the cache—or a *cache hit*—the data is retrieved from the cache, resulting in a fast serving time. On the other hand, if the requested memory address is not in the cache—or a *cache miss*—the processor needs to retrieve the data from the main memory and put it into the cache, resulting in a slow serving time.

Set-associative caches. In modern processors, the cache structure is usually *set-associative*. With this structure, the cache is divided into multiple *sets* where each set contains a certain number of *ways*. A memory line can be mapped into a single cache set and can only be stored in the particular set that it is mapped to. The details of the mapping function are considered proprietary knowledge, and vendors do not always publicly disclose such information. Nevertheless, previous research [HWH13, IES15, MLSN+15, YGL+15] has reverse-engineered some of the mapping functions, thus allowing ones to determine the cache set that a given memory line is mapped to.

Cache-based side-channel attacks. Caches are usually shared among multiple programs. This means that a malicious program can monitor cache activities and infer information on the execution of the other programs using the shared cache. Past research has demonstrated the impacts of cache attacks, which range from recovering encryption keys [CAGTB19, CFSY22, CGYZ22, DDME+18, GPTY18, GPS+20, LYG+15, OST06, PGBY16] to other sensitive information [GRB+17, GSM15, SKH+19, YFT20].

Flush+Reload. Flush+Reload [YF14] is cache-based side-channel attack that exploits the timing difference whether the data is in the cache. In the *flush* phase, the attacker performs `clflush` to clear memory lines in the cache. Then, the attacker waits for the victim to execute. Since the memory lines are empty, any victim memory access will put data in the cache. Finally, in the *reload* phase, the attacker re-accesses the memory lines that have been removed in the flush phase. A short access time indicates that the memory line has been brought back into the cache during the victim's execution, while a long access time indicates that that memory resides in the main memory and has not been brought back into the cache. Hence, the attacker can learn which memory line(s) the victim has accessed.

Prime+Probe. Prime+Probe [OST06, LYG+15] is another cache-based side-channel attack. It exploits set-associativity and causes contention. In the *prime* phase, the attacker completely fills the target cache set with its own data. Then, the attacker waits for the victim to execute. Since the target cache set is already full, any memory access caused by the victim must result in attacker's data being removed from the cache. Finally, in the *probe* phrase, the attacker measures the time to access the data used in the prime phase to fill the target cache set. A short access time implies that the data still resides in the cache. On the other hand, a long access time implies that the attacker's data has been evicted by the victim's execution. Therefore, the attacker can infer the cache set that the victim has accessed during the prime and probe phases. Utilizing the mapping between address bit and cache sets, the attacker can get some (partial) information about the address that the victim accessed.

2.3 Intel SGX

Intel Software Guard Extensions (SGX) is a set of extensions of the Intel instruction set that allows running code in trusted execution environments called *enclaves*. SGX aims at protecting the contents inside the enclave from any other code running on the same machine, including protection from the operating system (OS), virtual machine manager, and the high privileged system management mode. To achieve this aim, the processor encrypts the memory space of the enclave and imposes restrictions when switching between enclave and non-enclave.

Note that SGX does not consider the OS as trusted. This means that SGX assumes the OS can be malicious. Since SGX does not provide protection against side-channel attacks [AMG+15, CD16], this has led to a stronger attack model that includes a malicious or compromised OS. In other words, the model assumes that the attacker may control the OS and can use privilege instructions. There exist various attacks that exploit this powerful attacker's ability. Examples include attacks on the page tables [XCP15, BWK+17], branch target buffers [LSG+17], caches [SWG+17, MIE17, BMD+17], and memory false dependency [MES18].

A technique that has been used (e.g., in [LSG+17, MIE17]) to improve the attack is to employ the OS's ability to interrupt the enclave frequently. This allows attackers not only to obtain a high temporal resolution but also to observe memory accesses nearly at every single instruction performed by the victim enclave.

In terms of protections against SGX attacks, several approaches have been proposed. For example, compiler-level page table masking [SCNS16] and modification to page table entries within the SGX hardware [SP17] have been demonstrated to mitigate page table attacks. Other compiler-level protections based on randomization [BCD+17] and binary code retrofitting [WWB+17] have been introduced to prevent cache attacks. Detecting excessive OS interruptions [CZRZ17] has been proposed as a solution to attackers with OS privilege. Note however that the robustness and efficiency of the aforementioned protections are yet to be verified.

3 Plaintext-Checking Oracle from Cache-Timing Leakage

The core problem in a chosen-ciphertext side-channel attack against HQC is to build a plaintext-checking (PC) oracle to check if HQC.CCAKEM.Decaps$(\mathsf{sk}, c) = m$, given c and m. In this section, we first present a vulnerability during the generating process of the two vectors, namely r_1 and e, in the reference implementation of HQC. [1] Then, we introduce an important definition, which is called *cache line indicator*, to describe the cache-timing behavior during the implementation of HQC. Based on the cache-timing leakage from the vulnerability, we develop a PC oracle, which is essential to build our key recovery attack against HQC.

3.1 The Sampling Process of r_1 and e

Now we describe the sampling process of the vectors r_1 and e in the reference implementation of HQC.[2] In the reference implementation, both the vectors r_1 and e are allocated with the data type `uint64_t`. The vectors r_1 and e are sampled by calling `vect_set_random_fixed_weight` in Listing 1. From the code, the function `vect_set_random_fixed_weight_by_coordinates` is called to determine the coordinates of the nonzero components. Then, the corresponding bits are modified in the loop according to the coordinates recorded in the array `tmp`.

It can be observed that in the sampling process, only parts of the vectors are loaded into the cache. When sampling the vectors r_1 and e, only ω_r bits of r_1 and e are modified, which indicates that at most ω_r cache lines are accessed. This is crucial to determine whether HQC.CCAKEM.Decaps$(\mathsf{sk}, c) = m$, given c and m.

[1] The reference version can be found through the link https://pqc-hqc.org/implementation.html.

[2] The generation of the vector r_2 is different from the generation of r_1 and e because r_2 is only used during the multiplication with the vector s and only the nonzero components of r_2 are recorded. The oracle we construct is based on the timing leakage when generating r_1 and e.

```
void vect_set_random_fixed_weight(seedexpander_state *ctx, uint64_t *v, uint16_t
↪  weight) {
    uint32_t tmp[PARAM_OMEGA_R] = {0};
    vect_set_random_fixed_weight_by_coordinates(ctx, tmp, weight);
    for (size_t i = 0; i < weight; ++i) {
        int32_t index = tmp[i] / 64;
        int32_t pos = tmp[i] % 64;
        v[index] |= ((uint64_t) 1) << pos;
    }
}
```

Listing 1: Vulnerable function

3.2 Cache Line Indicator

Now we introduce the definition of the cache line indicator, which describes whether a cache line is accessed during the implementation of the sampling process.

Definition 1. A *cache line indicator* of a vector b is a binary vector $v_b \in \mathbb{F}_2^{\ell_b}$, where ℓ_b is the number of cache lines needed to load the vector b into the cache. In a process where b is involved, if the i-th cache line is accessed, the i-th component of v_b, denoted as v_b^i, is 1; otherwise, v_b^i is 0, where $0 \leq i < \ell_b$.

Given a vector b, the cache line indicator can be deduced directly during the implementation, which is called *offline deduction*. For example, at the beginning of the sampling process, the vector r_1 is initialized as a zero vector. Then, only ω_r bits of r_1 are modified to be one. If the $64i$-th, $\cdots$, $(64i+63)$-th bytes of the vector r_1 are zero, the i-th cache line is not accessed, which means that $v_{r_1}^i = 0$; otherwise, $v_{r_1}^i = 1$, where $0 \leq i < \ell_b$.[3]

As we are targeting at attacking hqc-128, $\ell_{r_1} + \ell_e = 2 \times \lceil 17664/(8 \times 64) \rceil = 70$ cache lines are used to fully load r_1 and e into the cache. In this work, we construct special ciphertexts such that only a part of cache lines are hit during the decapsulation, which means the cache line indicators v_{r_1} and v_e are sparse vectors instead of random vectors. Thus, the cache line indicator can be used to identify the cache behavior during the execution of hqc-128.

The cache line indicator can also be deduced from the cache-timing information leakage applying a side-channel attack, which is called *online query*. In the later section, we show how to apply the Flush+Reload [YF14] and Prime+Probe [LYG+15, OST06] attacks in two different scenarios in more details. Now, we present the general process of deducing the cache line indicators of the vectors r_1 and e with a cache-timing attack.

In an online query, the victim begins to execute the decapsulation function by calling HQC.CCAKEM.Decaps(sk, c), where sk is a secret key and c is a ciphertext. The victim samples r_1. The attacker obtains the timing leakage information through the cache-timing attack, which can help to deduce the cache line indicator of the vectors r_1. The victim samples e. Similarly, the attacker applies the cache-timing attack and obtains the cache line indicator of the vector e.

Now we define a cache-timing oracle $\mathcal{O}(\mathsf{sk}, c)$ to represent the process of an online query, where sk is an unknown secret key and c is a ciphertext. The cache-timing oracle $\mathcal{O}(\mathsf{sk}, c)$ outputs the cache line indicators of r_1 and e, i.e., (v_{r_1}, v_e), during the decapsulation HQC.CCAKEM.Decaps(sk, c).

[3]The cache line size is typically fixed to 64 bytes on x86/x64 CPUs.

3.3 PC Oracle

We denote the PC oracle, which can be used to check if HQC.$\mathsf{CCAKEM.Decaps}(\mathsf{sk}, c) = m$, as $\mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(m, c)$, where c is a ciphertext, m is a message and sk is the unknown secret key. In Algorithm 1, the PC oracle $\mathcal{D}^{\mathcal{O}(\mathsf{sk},c)}(m)$ is constructed with the cache-timing oracle $\mathcal{O}(\mathsf{sk}, c)$ mentioned in Subsection 3.2. As shown in lines 2–3 in Algorithm 1, the attacker computes the initial seed θ from the message m, initializes the sampling process with the seed, and obtains the vectors $\bar{r}_1$ and $\bar{e}$. Then, the attacker deduces the cache line indicators $v_{\bar{r}_1}$ and $v_{\bar{e}}$ from $\bar{r}_1$ and $\bar{e}$ using the offline deduction (See line 4 in Algorithm 1). In line 5 of Algorithm 1, the attacker calls the cache-timing oracle $\mathcal{O}(\mathsf{sk}, c)$, which returns the cache line indicators of r_1 and e, i.e., (v_{r_1}, v_e), during the decapsulation HQC.$\mathsf{CCAKEM.Decaps}(\mathsf{sk}, c)$. From lines 5–9 in Algorithm 1, the attacker compares $(v_{\bar{r}_1}, v_{\bar{e}})$ with (v_{r_1}, v_e). If $v_{r_1} = \bar{v}_{r_1}$ and $v_e = \bar{v}_e$, the PC oracle returns 1 and a zero vector, which indicates that the underlying plaintext during the decapsulation HQC.$\mathsf{CCAKEM.Decaps}(\mathsf{sk}, c)$ is m; otherwise, the PC oracle returns 0 and (v_{r_1}, v_e) to be used in the key-recovery attack to be discussed in Section 4.

Algorithm 1 The PC Oracle $\mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(m, c)$

```
1: procedure $\mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(m, c)$
2:     $\theta = \mathcal{G}(m)$, $\mathsf{sampleInit}(\theta)$, $\bar{r}_1 = \mathsf{sample}(\mathcal{R}, \omega_r)$, $\bar{e} = \mathsf{sample}(\mathcal{R}, \omega_e)$
3:     Deduce the cache line indicators $v_{\bar{r}_1}$ and $v_{\bar{e}}$ from $\bar{r}_1$ and $\bar{e}$.
4:     $(v_{r_1}, v_e) = \mathcal{O}(\mathsf{sk}, c)$.
5:     if $v_{r_1} = v_{\bar{r}_1}$ and $v_e = v_{\bar{e}}$ then
6:         return $(1, \mathbf{0})$
7:     else
8:         return $(0, v_{r_1}, v_e)$
```

Now we analyze the success rate of the PC oracle. We assume that for the ciphertext c, the number of cache lines accessed during the generation of the vectors r_1 and e is no more than T_c, namely $w_{\mathrm{H}}(v_{r_1}) + w_{\mathrm{H}}(v_e) \leq T_c$. The vectors r_1 and e are distributed uniform randomly. Then, the probability that $v_{r_1} = \bar{v}_{r_1}$ and $v_e = \bar{v}_e$ is no more than $P = (\frac{T_c}{\ell_{r_1}+\ell_e})^{w_r+w_e}$. Thus, the accuracy of the PC oracle is at least

$$1 - P = 1 - (\frac{T_c}{\ell_{r_1} + \ell_e})^{w_r+w_e}. \tag{1}$$

Note that when T_c decreases, the success rate of the PC oracle increases.

4 Key-Recovery Attack Against HQC

4.1 Framework

We apply the PC oracle described in Section 3 to build a key-recovery attack against HQC. Our key-recovery attack is inspired by the timing key-recovery attack in the literature [GHJ+22], yet the mechanisms are totally different. That is, instead of observing the number of calls to the sampling function (more calls take longer time) as in [GHJ+22], we observe cache-timing access of sampled vectors. Furthermore, our new key-recovery method is divided into an online and offline phases (whereas [GHJ+22] is all online) and requires significantly fewer queries (compared to [GHJ+22]) thanks to the property of the concatenated code.[4] To be more specific, we construct a chosen ciphertext in the key-recovery attack carefully by utilizing an algebraic structure of the concatenated code

[4]The accuracy of the PC oracle in [GHJ+22] is also nearly 1 according to Table 3 from [GHJ+22].

used in HQC, whereas in the approach of [GHJ+22], the chosen ciphertext are generated randomly, which leads to more queries required. Another advantage of our approach is that the *basis message* and *constant error* constructed in the offline phase can be reused to recover other keys against HQC.

In our technique, the attacker searches for an invalid ciphertext $C = (\mathbf{1}, v)$, called *edge ciphertext*, such that

$$\mathcal{C}.\text{Decode}(v - y) \neq \mathcal{C}.\text{Decode}(v), \tag{2}$$

where $\mathbf{1}$ is the multiplicative identity of $\mathcal{R}$. Note that $\mathcal{C}.\text{Decode}(v - y)$ is the underlying message during the decapsulation of the ciphertext C, which is called an *edge message*. It can be seen that the difference in the decoding results in Inequality 2 is caused by the addition of the secret key y. If the attacker finds an edge ciphertext C such that Inequality 2 holds, it would help the attacker to recover y. The process of recovering x is slightly different from the process of recovering y, which is demonstrated in Appendix A.

To find the edge ciphertext C in Inequality 2, the attacker starts by constructing an invalid ciphertext C_0 with some requirements, which is called a *basis ciphertext*. The underlying message of the basis ciphertext C_0 is called a *basis message*. Then, the attacker injects an error E to the basis ciphertext C_0 such that $C = C_0 \bigoplus E$ is the edge ciphertext.

To efficiently find the inner block of y which makes Inequality 2 hold, we carefully choose the injected error E in our technique. The injected error E is summed up by two vectors, namely, *constant error* and *random error*.

The constant error, denoted as E_c, includes $(\lceil d_{RS}/2 \rceil - 1)$ nonzero inner blocks and $(2n_1 - \lceil d_{RS}/2 \rceil + 1)$ zero inner blocks. For example, as for the variant hqc-128, there are $(\lceil d_{RS}/2 \rceil - 1) = 15$ nonzero inner blocks in the constant error. The purpose of constructing the constant error is to avoid the impact of the corresponding inner blocks of y on the decrypting result of the decapsulation.

The random error, denoted as E_r, contains only one nonzero inner block and $(2n_1 - 1)$ zero inner blocks. The nonzero inner block in the random error is, indeed ,the inner block that causes the difference of the decoding results in Inequality 2.

The key-recovery attack includes two phases, namely, the offline phase and the online phase. As it will be shown in Subsection 4.2, the purpose of the offline phase is to search the basis ciphertext C_0 and construct the constant error E_c. It should be noted that the basis message m along with the constant error E_c can be reused to recover other keys against HQC applying our approach. In the online phase, demonstrated in Subsection 4.3, the attacker builds the random error E_r and recovers the inner blocks of the secret key y with the Hamming weight no larger than 1.

4.2 Offline Phase

Constructing the basis ciphertext C_0. In the offline phase, the method of constructing the basis ciphertext C_0 follows the key-recovery attack in the literature [GHJ+22]. The attacker manually sets r_1 to $\mathbf{1}$ (i.e., the multiplicative identity of $\mathcal{R}$) and both r_2 and e to $\mathbf{0}$ (i.e., the zero vector). In this case, the basis ciphertext is $C_0 = (u, v) = (\mathbf{1}, m\mathbf{G})$. It can be observed from the procedure PKE.Decrypt that $v - uy = m\mathbf{G} - y$. It indicates that the error that the decoder has to correct during the decryption is indeed y, which is the second half of the secret key.

As mentioned in Subsection 3.2, to increase the success rate of the PC oracle deployed in our attack, we require that $w_{\text{H}}(v_{r_1}) + w_{\text{H}}(v_e)$ should be no larger than a threshold T_c, where r_1 and e are the sampled vectors during the decapsulation of C_0. As for our attack against hqc-128, the choice of T_c is to be discussed in Subsection 5.3.

We show the procedure of generating a basis ciphertext C_0 in Algorithm 2. The attacker first randomly generates a message m. As shown in lines 4–5 of Algorithm 2, with a message m the attacker can deduce the cache line indicators of r_1 and e easily applying the offline deduction. The vectors r_1 and e can be directly deduced without knowing y because both $m\mathbf{G} - y$ and $m\mathbf{G}$ are decoded to m. If $w_{\mathrm{H}}(v_{r_1}) + w_{\mathrm{H}}(v_e) \leq T_c$, the algorithm returns the basis ciphertext $C_0 = (\mathbf{1}, m\mathbf{G})$; otherwise, it steps back to line 2 and picks another message m randomly. As can be observed from Algorithm 2, the procedure of generating C_0 is independent of the secret key.

Algorithm 2 Generating a basic ciphertext C_0

```
1: procedure GenText($T_c$)
2:     while 1 do
3:         Randomly pick a message $m$
4:         Compute $\theta = \mathcal{G}(m)$, sampleInit($\theta$), $r_1$ =sample($R, \omega_r$), $e$ =sample($R, \omega_e$)
5:         Deduce $v_{r_1}$ and $v_e$ according to Definition 1.
6:         if $w_{\mathrm{H}}(v_{r_1}) + w_{\mathrm{H}}(v_e) \leq T_c$ then
7:             return $C_0 = (\mathbf{1}, m\mathbf{G})$
8:         else
9:             Continue.
```

Constructing the constant error E_c. The first half of the constant error is zero, and the second half of the constant error contains $(\lceil d_{RS}/2 \rceil - 1)$ nonzero inner blocks. The constant error is constructed block-wise. Next, we show two important definitions, namely, n-bit domain and constant error block.

Definition 2. Given an inner block C_I, the *n-bit domain* of the inner block $C_I \in \mathbb{F}_2^{n_2}$ is a set containing all the binary strings $S \in \mathbb{F}_2^{n_2}$ such that $w_{\mathrm{H}}(C_I - S) \leq n$, which is denoted as $U(C_I, n)$.

Definition 3. Suppose that an inner block of a ciphertext C_0, denoted as C_I, is decoded into an internal codeword V by the Reed-Muller decoder. A *constant error block* e_c is a binary string in $\mathbb{F}_2^{n_2}$ such that each S in $U(C_I + e_c, n)$ is decoded into a different internal codeword $V' \neq V$ using the Reed-Muller decoder. The integer n is called *constant distance.*

Suppose that $C_0^2[i]$ is the i-th inner block of the second half of the ciphertext C_0 and y_i is the i-th inner block of y with $w_{\mathrm{H}}(y_i) < n$ for some $i \in [0, n_1)$. If the attacker obtains a constant error block e_c with a constant distance n, $C_0^2[i] \bigoplus e_c$ and $C_0^2[i] \bigoplus e_c \bigoplus y_i$ are decoded into the same internal codeword during the decapsulation. It indicates that the addition of the i-th block of y will not affect the result of the decapsulation of $C \bigoplus E$, where $E = (\mathbf{0}, E_0)$, the i-th inner block of E_0 is e_i, and the other inner blocks are zero.

In Algorithm 3, we show the procedure of constructing a constant error block. In line 3 of Algorithm 3, the attacker randomly picks a binary string $e_c \in \mathbb{F}_2^{n_2}$ with $w_{\mathrm{H}}(e_c) = L_0$, where L_0 is a preset parameter. Then, the attacker checks whether $C_I + e_c$ is decoded into another internal codeword V_1 different from the original internal codeword V. (See lines 4–5 of Algorithm 3.) If $V_1 \neq V$, the attacker checks whether $U(C_I + e_c, n)$ is a constant domain; otherwise, the attacker goes back to line 3 of Algorithm 3 and generates another binary string (See lines 9–14 of Algorithm 3.). If $U(C_I + e_c, n)$ is a constant domain, Algorithm 3 returns the constant error block e_c; otherwise, the attacker generates a new binary string.

The constant error E_c is constructed from constant error blocks in Algorithm 4. In Algorithm 4, the input I_c is an array of any $(\lceil d_{RS}/2 \rceil - 1)$ integers with a range from 0 to $(n_1 - 1)$, which records the positions of constant error blocks. It can be seen from Algorithm 4 that the attacker calls ConstBlk$(\cdot,\cdot,\cdot)$ to build the corresponding constant error blocks according to the array I_c.

It should be noted that the offline phase can be done in practical time as verified through experiments in Subsection 5.3.

Algorithm 3 Constructing a constant error block

```
1: procedure ConstBlk($C_I$,$n$,$L_0$)
2:     while 1 do
3:         Randomly pick a binary string $e_c \in \mathbb{F}_2^{n_2}$ with $w_{\mathrm{H}}(e_c) = L_0$
4:         Decode $C_I$ using the Reed-Muller decoder and denote the result as $V$.
5:         Decode $C_I + e_c$ using the Reed-Muller decoder and denote the result as $V_1$.
6:         if $V = V_1$ then
7:             continue
8:         else
9:             flag=1
10:            for each binary string $S \in U(C_I + e_c, n)$ do
11:                Decode $S$ using the Reed-Muller decoder and denote the result as $V_2$.
12:                if $V_2 \neq V_1$ then
13:                    flag=0
14:                    break
15:            if flag=1 then
16:                return $e_c$
17:            else
18:                continue
```

Algorithm 4 Constructing the constant error

```
1: procedure ConstError($C_0$,$I_c$,$n$,$L_0$)
2:     $E_c = (\mathbf{0}, \mathbf{0})$.        ▷ $index$ is an array of $(\lceil d_{RS}/2 \rceil - 1)$ integers.
3:     for each integer $i$ in the array $I_c$ do
4:         $E_c^2[i]$ =ConstBlk($C_0[i]$,$n$,$L_0$)  ▷ $E_c^2$ is the second half of $E_c$. $E_c^2[i]$ is the $i$-th
   inner block of $E_c^2$.
5:     return $E_c$
```

4.3 Online Phase

Now we present the process of recovering one inner block of y. Without loss of generality, we show the method of recovering the j-th inner block of y, where $0 \leq j < n_1$ and $j \notin I_c$. The attacker first searches for a random error E_r in a special form and obtains the edge ciphertext $C_1 = C_0 \bigoplus E_c \bigoplus E_r$ to make Inequality 2 hold. Then, the attacker recovers y_j by modifying C_1 and making an offline deduction with the corresponding modified ciphertexts.

To search for an edge ciphertext $C = (\mathbf{1}, v)$ such that Inequality 2 holds, our approach considers the following two cases:

$$\begin{cases} \textbf{Case 1: } \mathcal{C}.\text{Decode}(v) = m, \mathcal{C}.\text{Decode}(v - y) = m'; \\ \textbf{Case 2: } \mathcal{C}.\text{Decode}(v) = m', \mathcal{C}.\text{Decode}(v - y) = m; \end{cases} \tag{3}$$

where m is the basis message and $m \neq m'$.

Given a ciphertext $C = (\mathbf{1}, v)$, the attacker can detect whether it is an edge ciphertext following one of the two cases in Equation 3. As the decoder in HQC is publicly available, the attacker can verify whether $\mathcal{C}.\text{Decode}(v) = m$ by calling the decoder. Verifying whether $\mathcal{C}.\text{Decode}(v-y) = m$ is equivalent to determining whether HQC.$\mathsf{CCAKEM.Decaps}(\mathsf{sk}, C) = m$, which can be detected by applying the PC oracle $\mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(m, C)$.

The method of searching for an edge ciphertext is shown in Algorithm 5. The attacker first initializes the counter variable *cnt* to zero. As shown in lines 4–5 of Algorithm 5, the attacker randomly generates a random error E_r with a single nonzero inner block of Hamming weight L_1, which is a preset parameter. Then, the attacker checks whether the ciphertext $C_1 = C_0 \bigoplus E_c \bigoplus E_r$ follows one of the two cases defined in Equation 3. If C_1 is a Case 1 ciphertext then Algorithm 5 returns 1, C_1 and the cache line indicators v_{r_1}, v_e obtained from the online query. (See lines 9–10 of Algorithm 5.) If C_1 is a Case 2 ciphertext then Algorithm 5 returns $(2, C_1)$. (See lines 11–14 of Algorithm 5.) Otherwise, the attacker increases *cnt* and continues. If the attacker still cannot find an edge ciphertext after randomly generating E_r for T times where T is a preset threshold, Algorithm 5 outputs 0, which means $y_j = 0$. The complexity of Algorithm 5 is at most T times of calling the decoder $\mathcal{C}.\text{Decode}(\cdot)$ and the PC oracle $\mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(\cdot, \cdot)$.

Algorithm 5 Searching for an edge ciphertext C_1

```
1:  procedure BuildError($C_0$, $m$, $E_c$, $j$, $L_1$, $T$)
2:      $cnt = 0$
3:      while $cnt < T$ do
4:          $E_r = \mathbf{0}$
5:          Assign $E_r^2[j]$ as a random binary string such that $w_{\mathrm{H}}\left(E_r^2[j]\right) = L_1$.
6:          $C_1 = C_0 \bigoplus E_c \bigoplus E_r$
7:          $m' = \mathcal{C}.\text{Decode}(C_1^2)$          ▷ $C_1^2$ is the second half of $C_1$.
8:          $(flag, v_{r_1}, v_e) = \mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(m, C_1)$
9:          if $flag = 0$ and $m = m'$ then
10:             return $(1, C_1, v_{r_1}, v_e)$
11:         else if $flag = 1$ and $m \neq m'$ then
12:             return $(2, C_1)$
13:         else
14:             Increase $cnt$ by 1 and continue
15:     return 0
```

The procedure of recovering y_j is shown in Algorithm 6. As presented inside the loop from lines 23–30 in Algorithm 6, the attacker generates L_2 edge ciphertexts to execute the attack and record the possible value of y_j from every iteration, where L_2 is a parameter chosen by the attacker. If all the results from the L_2 iterations suggest that some bits in y_j should be 1, then the attacker concludes that this bit of y_j is 1. (See line 28 and line 30 in Algorithm 6.) In each iteration, the attacker starts by generating an edge ciphertext C_1. If $\textsc{BuildError}(C_0, m, E_c, j, L_1, T)$ returns 0, Algorithm 6 outputs $y_j = 0$. Otherwise, Algorithm 6 continues guessing y_j by doing a modification on the edge ciphertext C_1 with respect to the two different cases.

If $\textsc{BuildError}(C_0, m, E_c, j, L_1, T)$ finds a Case 1 edge ciphertext, the attacker calls $\textsc{GuessCase1}(C_1, v_{r_1}, v_e)$, where v_{r_1} and v_e are the cache line indicators obtained from the online query when the victim calls the decapsulation of C_1. For each $i \in [0, 384)$, the attacker flips the i-th bit of C_1 and executes an offline deduction. (See lines 5–8 of Algorithm 6.) As shown from lines 9–10 in Algorithm 6, if the resulted cache line indicators are exactly v_{r_1} and v_e, the attacker sets the i-th bit of y_j to be 1. Otherwise, the attacker continues to the next guess.

If $\textsc{BuildError}(C_0, m, E_c, j, L_1, T)$ finds a Case 2 edge ciphertext, the attacker calls $\textsc{GuessCase2}(C_1, m)$, where m is the basis message. In this case, for each $i \in [0, 384)$, the attacker flips the i-th bit of C_1 and decodes the corresponding ciphertext C_2. (See lines 15–17 of Algorithm 6.) If the underlying message is indeed m, the i-th bit of y_j is set to be 1. Otherwise, the attacker continues the process.

The procedure of recovering the inner blocks of y with Hamming weight smaller than 1 against hqc-128 is summarized in Algorithm 7. In the offline phase, the attacker starts by constructing a basis ciphertext C_0. The attacker can randomly choose the positions of the nonzero inner blocks of the constant error. Without loss of generality, the attacker can set the indices of the nonzero inner blocks of the constant error to $I_c = \{0, 1, \cdots, 14\}$. As shown in lines 4–5 of Algorithm 7, which is the online phase, for each $i \in [15, 46)$, the attacker assumes that $w_{\mathrm{H}}(y_i) \leq 1$ and tries to recover y_i. Similarly, the attacker switches I_c to be $\{15, 16, \cdots, 29\}$ and builds another constant error to recover the remaining inner blocks of y. (See lines 6–9 of Algorithm 7.)

Note that the complexity of Algorithm 7 is at most $L_2 \cdot (T + 384)$ times of calling the decoder $\mathcal{C}.\text{Decode}(\cdot)$ and $L_2 \cdot T$ times of calling the PC oracle $\mathcal{D}^{\mathcal{O}(\mathsf{sk},\cdot)}(\cdot,\cdot)$. As will be discussed in Subsection 5.3, this procedure can be done in practical time.

With the new technique, the attacker can recover the inner blocks of the secret vector y with Hamming weight no larger than 1. The probability that a randomly sampled inner block in y has a Hamming weight of k is shown in Table 2 of [GLG22c]. We include the table in Appendix B for completeness. According to the table, for the variant hqc-128, about 57.82% on average of the inner blocks of y have Hamming weight no larger than 1, which can be recovered with the new method.

We further observed that instead of fully recovering the secret vector y, one can recover a proportion of both y and x, and obtain the remaining entries in x and y by solving the system of linear equations representing the polynomial equation

$$s = x + h \cdot y \tag{4}$$

in a matrix form, where the public key $\mathsf{pk}= (h, s)$ is publicly available. In general, if we can recover 50% of the entries in y and x in total, the remaining entries can be efficiently recovered via Gaussian Elimination. The recovering procedure also works if we recover a slightly smaller number of entries via a light post-processing Information Set Decoding (ISD) algorithm [EB22].

The idea of using both the x and y secret vectors is a novel approach and it works well for all three HQC parameter sets. For instance, for the variants hqc-128 and hqc-256, the proportion of recovered blocks is 57.82% and 57.20% respectively, both being larger than 50% (meaning only Gaussian Elimination works). For the variant hqc-192, one can only recover 46.5% of the 112 inner blocks according to Table 2 of [GLG22c], i.e., approximately 33331 entries. Thus, about $n_r = n - 33331 = 2520$ additional secret entries

Algorithm 6 Recovering y_j

```
1: procedure GuessCase1($C_1$, $v_{r_1}$, $v_e$)
2:     Initialize $y_j$ as 0.
3:     for each $i \in [0, 384)$ do
4:         $C_2 = C_1$
5:         Filp the $i$-th bit of the $j$-th inner block of $C_2^2$.
6:         $m' = \mathcal{C}$.Decode($C_2^2$).
7:         $\theta = \mathcal{G}(m')$, sampleInit($\theta$), $\hat{r}_1$ =sample($\mathcal{R}, \omega_r$) and $\hat{e}$ =sample($\mathcal{R}, \omega_e$).
8:         Deduce the cache line indicators $v_{\hat{r}_1}$ and $v_{\hat{e}}$ from $\hat{r}_1$ and $\hat{e}$.
9:         if $v_{\hat{r}_1} = v_{r_1}$ and $v_{\hat{e}} = v_e$ then
10:            $y_j[i] = 1$
11:    return $y_j$
12: procedure GuessCase2($C_1$, $m$)
13:    Initialize $y_j$ as 0.
14:    for each $i \in [0, 384)$ do
15:        $C_2 = C_1$
16:        Filp the $i$-th bit of the $j$-th inner block of $C_2^2$.
17:        $m' = \mathcal{C}$.Decode($C_2^2$).
18:        if $m = m'$ then
19:            $y_j[i] = 1$
20:    return $y_j$
21: procedure KeyRecover($C_0$, $E_c$, $m$, $j$, $L_1$, $L_2$, $T$)
22:    Initialize $y_j$ as 0.
23:    for each integer $i \in [0, L_2)$ do
24:        $(flag, C_1, v_{r_1}, v_e)$=BuildError($C_0$, $m$, $E_c$, $j$, $L_1$, $T$)
25:        if $flag = 0$ then
26:            return $y_j = 0$
27:        else if $flag = 1$ then
28:            $y_j = y_j$ & GuessCase1($C_1$, $v_{r_1}$, $v_e$)
29:        else if $flag = 2$ then
30:            $y_j = y_j$ & GuessCase2($C_1$, $m$)
31:    if $y_j = 0$ then
32:        return $w_{\mathrm{H}}(y_j) > 1$
33:    else
34:        return $y_j$
```

Algorithm 7 Recovering inner blocks of y of hamming weight less than 1

```
1: $C_0$ =GenText($T_c$)
2: $I_c = \{0, 1, 2 \cdots, 14\}$
3: $E_c$ =ConstError($C_0$,$I_c$,$n$,$L$)
4: for each integer $i \in [15, 46)$ do
5:     $y_i$ =KeyRecover($C_0$, $E_c$, $m$, $j$, $L_1$, $L_2$, $T$)
6: $I_c = \{15, 16, 17 \cdots, 29\}$
7: $E'_c$ =ConstError($C_0$,$I_c$,$n$,$L$)
8: for each integer $i \in [0, 15)$ do
9:     $y_i$ =KeyRecover($C_0$, $E'_c$, $m$, $i$, $L_1$, $L_2$, $T$)
```

are needed. Because the HQC system is very sparse, one can merely randomly sample n_r secret entries among the unknown ones and hope that they are all zero. The weight of x (or y) is 100, and 30% of the recovered inner blocks are of weight one, so the undecided non-zero positions are 140 in expectation. After substituting the known secret entries, the remaining is thus a decoding problem with code length $n+n_r = 38371$, code dimension 2520, and weight 140 since one needs to solve a sparse equation system with 38371 unknowns, 35851 parity-check equations, and a solution with only 140 non-zero entries. The recent syndrome decoding estimator [EB22] reports that the solving complexity with Stern's algorithm is $2^{48.8}$ bit operations, still practical for a modern computer.

An alternative method. Instead of employing post-processing with ISD, we can search again for an invalid ciphertext (u, v) with the polynomial $u(X) = X^{t_0} \in \mathcal{R}$, where t_0 is an integer. This technique is equivalent to considering a cyclic shift of y since the ambient space is $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$. Note that the multiplication $u \cdot y = u(X) \cdot y$ during the decryption is to cyclically shift y to the left by t_0 bits. This technique could help to determine more zero positions in y, but only requires more online traces. It is interesting to note that the constant errors constructed to recover inner blocks of y can still be reused here. Similar techniques can also be applied to the secret polynomial x. Since this technique can have non-trivial implementation complexity and is unnecessary in our target implementation of hqc-128, we leave the elaborated investigation for future research.

5 Experiments and Results

This section demonstrates our cache attacks based on the cache line indicator. We first describe our attack in a close-to-ideal scenario where we assume the attacker can control the environment so that the obtained cache side-channel information has relatively low noise. This serves as a baseline to evaluate whether we can, indeed, obtain the side-channel information that is essential for our key recovery analysis. We then proceed to a more realistic scenario where we remove such assumption. In both cases, our attack targets the NIST hqc-128 reference implementation.The code for our attack is available at `https://git.sec.cs.adelaide.edu.au/rqsim/cache-timing-attack-against-hqc`.

Besides the cache attacks, we also verify our key-recovery attack with experiments in an ideal scenario. Based on our experimental results, our technique is much more efficient than the query scheme in [GHJ+22]. Specifically, our approach requires only $53,857$ queries, which is much fewer than $866,000$ queries in [GHJ+22].

5.1 Proof-of-Concept Attack

For our proof-of-concept attack, we use the Flush+Reload technique to verify that we can realize the cache line indicator as described in Definition 1. The reason for using Flush+ Reload is that the attack is fairly simple. It achieves the granularity of a cache line as desired (i.e., aligning with the concept of cache line indicator), and it has a very few false positive rate. We would like to emphasize that the purpose of this proof-of-concept attack is to demonstrate that the leakage does happen at the cache.

Threat model. For the proof-of-concept experiment, we adopt a threat model, already explored in the literature, e.g.., [PBY17]. In this model, the attacker uses the Flush+ Reload attack. For that, we assume the attacker can evict memory locations it shares with the victim, e.g., using the `clflush` instruction, and later measure the access time to those locations. Moreover, as in [PBY17], we embed the Flush+Reload in the victim code. We further assume that the attacker can measure every cache access with no errors.

```
1  int monitored_lines = 35;
2
3  for (int i = 0; i < monitored_lines; i++)
4      clflush(&r1[i*8]);
5
6  vect_set_random_fixed_weight(&seedexpander, r1, PARAM_OMEGA_R);
7
8  int fr_r1_accessed_lines[35] = {0};
9  for (int i = 0; i < monitored_lines; i++) {
10     int res = memaccesstime(&r1[i*8]);
11     if (res < LIMIT)
12         fr_r1_accessed_lines[i]++;
13 }
```

Listing 2: Flush+Reload attack code

Experimental setup. We implement the attack on a Dell Inspiron 15-7568 laptop running Ubuntu 18.04.6 LTS. The machine is equipped with an Intel Core i3-6100U CPU having a 3072 KB last-level cache (LLC). We use the Mastik toolkit [Yar16] to perform the Flush+Reload attack. We take multiple measures to reduce measurement noise. That is, we disable automatic power management and Intel Turbo Boost. We also set the Intel scaling driver to performance state, where the CPU is expected to run at the maximum frequency, which we set to 2.20GHz. We further disable the cache prefetcher and separate the core that run the victim and attacker from other programs.

Attack procedure. In our attack, we target cache line accesses when the vectors r_1 and e are sampled, i.e., when the function `vect_set_random_fixed_weight` is executed. Since the attack proceeds similarly for r_1 and e, in the following we only describe the procedure for r_1.

As in a typical cache-based side-channel attack, the attacker first sets the cache into a known state. In our case, we use the `clflush` instruction to evict the vector r_1 from the cache (Listing 2, lines 3–4). After that, the attacker lets the victim perform the sampling of r_1 (Listing 2, line 6). During this sampling, the victim updates some locations in the array, which implies accesses to some cache lines. Finally, the attacker re-accesses the array and measures the access time (Listing 2, lines 8–12). A short access time (i.e., less than `LIMIT`) indicates that the victim has accessed that array slot during the sampling. Note that the time threshold (i.e., `LIMIT`) to determine whether a line had been accessed or not is machine dependent, and the attacker can pre-determine this before the attack.

Results. We run our experiment using 1000 random samples of ciphertext. The experimental results confirm that we can correctly identify the cache line accesses, i.e., if we detect that the cache lines have been accessed, those cache lines were, indeed, accessed. In other words, we have 100% true positive and do not experience any false positive.

5.2 Practical Attack

While the results from the proof-of-concept attack are very accurate, the attack scenario may not be realistic in the sense that we modify the victim program (i.e., to flush cache lines) and assume perfect measurement (i.e., no measurement errors). We now move to a more realistic attack scenario, where we remove the assumption of measuring cache accesses without errors and the requirement to embed the Flush+Reload into the victim code. For our practical attack, we use Prime+Probe to attack a victim code running inside a secure SGX enclave that aims to protect the execution inside it.

Threat model. We assume that the victim implements a virtual Hardware Security Module (HSM) [HSM] using an SGX enclave. That is, to benefit from the security guarantees of SGX, including harware protection against a malicious operating system (OS) and the ability to attest that the code executes within a trusted environment. Thus, we assume that the victim cryptographic code executes within an SGX enclave.

As in prior SGX attacks, we assume a malicious OS, which is under the control of the attacker [MIE17, VBPS17, BMD$^+$17, LSG$^+$17]. Consequently, the attacker can execute priveleged instructions, and, in particular, control the memory layout, including disabling the use of Address Space Layout Randomization (ASLR) in the enclave. Also, the attacker can use SGX-Step [VBPS17] to single-step the enclave. On the other hand, we assume that there are *no* software vulnerabilities in the enclave as well as *no* vulnerabilities in the SGX [BMW$^+$18, vSMK$^+$21, BKS$^+$22]. We further assume that the attacker *cannot* read or write the enclave memory.

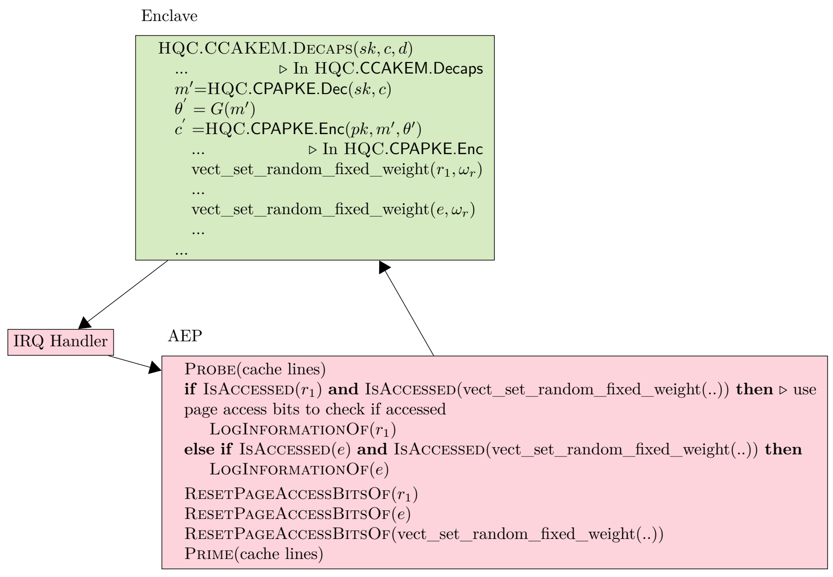
Experimental setup. We implement the attack on a Dell Inspiron 15-7568 laptop running Ubuntu 18.04.6 LTS. The machine is equipped with an Intel Core i3-6100U CPU having a 3072 KB last-level cache (LLC). We verify experimentally that the LLC is divided into four slices, each with 1024 sets, and has an associativity of 12. To reduce noise due to performance tuning, we apply multiple approaches as follows. We disable automatic power management and Intel Turbo Boost. We also set the Intel scaling driver to performance state, where the CPU is expected to run at the maximum frequency, which we set to 2.20GHz. We further disable the cache prefetcher and isolate the core that runs the victim thread.

We use the Mastik toolkit [Yar16] to perform the Prime+Probe attack targeting the last-level cache. We additionally use SGX-Step [VBPS17] to achieve a higher temporal resolution. The SGX-Step also enables the attacker to manipulate the local APIC to interrupt the enclave at timed intervals. At each interrupt, SGX exits the enclave with an Asynchronous Enclave Exit (AEX) procedure that securely saves the execution state. The interrupt handler will return to the Asynchronous Exit Pointer (AEP) before resuming execution in the enclave. Using the SGX-Step framework, we set up the interrupt handler to point to a modified spy function. We further disable Address Space Layout Randomization (ASLR) to obtain the virtual address of the locally defined vectors r_1 and e before running the attack. Note that under the SGX attack, a malicious OS is within the attack model.

Attack preparation. We first identify the cache lines to monitor then use Mastik [Yar16] to create the eviction sets. Before spying on the victim, the attacker needs to identify the cache sets of the vectors r_1 and e in the last-level cache. Note that r_1 and e are local variables, and the attacker can get their virtual address by monitoring its address with Address Space Layout Randomization (ASLR) disabled. After virtual to physical address translation, the attacker determines the cache sets that r_1 and e map to, which are encoded in bits 6–16 of the physical address (assuming a 64-byte cache line size).

The attacker further needs to identify cache slices. However, the cache slices cannot be determined from the address as the hash function that maps the address to the cache slices is undisclosed. There have been previous works, e.g., [SBWE21], that monitored all the possible slices that the target address maps to, and observed accesses only in the lines with the correct mapping. We notice that our machine, however, introduces noisy accesses in cache sets of an incorrect slice mapping, making it difficult to distinguish the correct mapping from the incorrect one. To cope with this issue, we determine the slice mapping of the r_1 and e addresses prior to performing the Prime+Probe attack. The idea is as follows. After initializing the enclave, we call a function that accesses all the elements of r_1 and e in the enclave and perform Prime+Probe on this function using all possible slice mappings. Despite the noise, we are able to distinguish the correct slice mapping since we now know the accesses to r_1 and e, i.e., all the elements. There have been previous works that reverse-engineered the slice hash function [IES15, MSN+15], but we did not have to resort to those approaches.

Attack procedure. The attack procedure is summarized in Figure 3. Similar to previous works [SBWE21, CGYZ22], we apply a controlled-channel attack [XCP15] to stop the enclave at the start of the attack. Then, we use SGX-Step to single-step each instruction. Specifically, we exit the enclave after one instruction when the APIC timer interrupt arrives, after which the interrupt handler redirects to the spy AEP function before returning to the enclave again. In the AEP function, we perform Prime+Probe to observe the cache accesses at the lines that store the vectors. Assuming that vectors r_1, e, and the `vect_set_random_fixed_weight` function are aligned to different pages, we can use the page access bit to identify when to record the Prime+Probe information. For example, if the page access bit for r_1 and `vect_set_random_fixed_weight` are set, we infer that the r_1 vector is being accessed in `vect_set_random_fixed_weight`. We then record the r_1 access information. Before returning to the enclave, we prime the cache sets ready to be probed in the next interrupt. Note that the target function `vect_set_random_fixed_weight` is directly from the HQC reference implementation found in the NIST submission. The only modification we made to the code is removing print statements, which is basically for a debugging purpose.



```
HQC.CCAKEM.DECAPS($sk, c, d$)
    ...                    ▷ In HQC.CCAKEM.Decaps
    $m'$=HQC.CPAPKE.Dec($sk, c$)
    $\theta' = G(m')$
    $c'$ =HQC.CPAPKE.Enc($pk, m', \theta'$)
        ...                ▷ In HQC.CPAPKE.Enc
        vect_set_random_fixed_weight($r_1, \omega_r$)
        ...
        vect_set_random_fixed_weight($e, \omega_r$)
        ...
    ...
```

```
PROBE(cache lines)
if ISACCESSED($r_1$) and ISACCESSED(vect_set_random_fixed_weight(..)) then ▷ use page access bits to check if accessed
    LOGINFORMATIONOF($r_1$)
else if ISACCESSED($e$) and ISACCESSED(vect_set_random_fixed_weight(..)) then
    LOGINFORMATIONOF($e$)
RESETPAGEACCESSBITSOF($r_1$)
RESETPAGEACCESSBITSOF($e$)
RESETPAGEACCESSBITSOF(vect_set_random_fixed_weight(..))
PRIME(cache lines)
```

Figure 3: Attack procedure: performing Prime+Probe in the attacker-modified AEP function. Green box denotes victim activities whereas red boxes denote attacker activities.

Analysis. Thanks to the ability to single-step using the SGX-Step framework, we can observe accesses to cache lines at each access in line 7 of Listing 1. Since our aim is to detect *whether* there is an access rather than *when* the access has occurred, the attacker can, in theory, conclude that there has been an access whenever an access has been detected in a single step. However, in practice, we found that there were noises in the experiments, resulting in a high false positive rate. To be precise, we found that out of 1000 samples of random ciphertext c, approximately 4% of the cache lines across all the samples have been incorrectly identified as access.

To improve the result, we employ a more meticulous analysis technique that could identify as well as filter out the noise. We note that when monitoring accesses at each step in line 7 of Listing 1, we observe the effect of speculative execution. For example, if `vect_set_random_fixed_weight` accesses cache line 0 at iteration step 52, the attacker may also detect accesses at cache line 0 at some prior steps as well. According to this observation, we define the rules to identify accesses in the presence of speculative execution as follows. For each cache line:

1. If we have access in all steps, we are not certain if there had been an access or not. Thus, we mark this cache line as *"not confident"*.
2. Else, if we have n consecutive accesses, we mark it as *"accessed"*.
3. Else, if we have k consecutive accesses at the first k steps, we mark it as *"accessed"*.
4. Else, if we have j consecutive accesses at the first j steps, and the line is not marked as accessed, we mark it as *"not confident"*.
5. Otherwise, we mark it as *"not accessed"*.

In our experiment, we classify accesses using the above rules with $n \geq 4$, $j = 1$ and $1 < k \leq n$.[5] Without taking these measures, the number of false positives would be too high for the attack. Note that we may detect some single accesses at step 0 for a cache line. This could be a genuine access or noise. If this is the only access in the cache line, we need to mark it as not confident

Regarding rule 1, we note that when `vect_set_random_fixed_weight` should, in theory, not access a particular cache line, yet in some cases the attack returns a result where the cache line was detected to have been accessed in all steps. Hence, we classify the access of a cache line with such a pattern as *"not confident"*.

Results. We run the experiment using 1000 random samples of ciphertext. There are 332 samples that we are confident in classifying *all* the cache line accesses, and 442 samples that we are confident in classifying all but one of the 70 cache lines (35 cache lines for each of r_1 and e). The remaining samples are within 8 non-confident cache lines. In this experiment, we correctly identify the cache line access 99.9% on average. To test if our classification rules are overfitting, we perform another batch of 1000 random samples of ciphertext; we also observe a similar result. That is, more than 90% of the samples have the number of not confident lines within 8, and the accuracy of identifying accesses is 99.9%. Figure 4 shows the histogram of the number of non-confident cache lines in the classification. The running time of each sample is roughly 10 seconds, including the use of Mastik and setting up the enclave to be single-stepped.

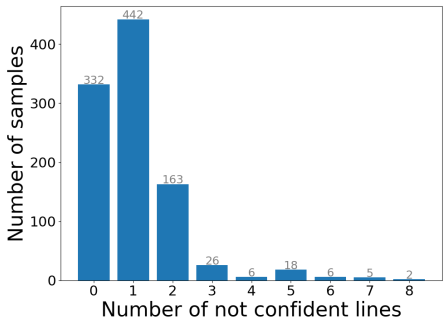


Figure 4: Distribution of the number of not confident cache lines

5.3 Key-Recovery Attack

We verify our key-recovery attack in Algorithm 7 and Algorithm 9 with experiments. In the experiments, we assume an ideal oracle which returns the exact cache indicators

[5]We try different values of n, j, k; the aforementioned combination turns out to provide good accuracy.

after each implementation of hqc-128. The code to verify the attack is available at https://github.com/xiaohuangthu/hqc-cache-timing-attack.

As for the offline phase, we construct 1000 basis ciphertexts applying Algorithm 2. We set T_c to 50, which means at most $T_c = 50$ cache lines out of 70 are hit during the underlying sampling process of a basis ciphertext. According to Equation 1, the accuracy rate of the PC oracle against hqc-128 is $1 - 1.2 \times 10^{-22} \approx 1$. For a random basis ciphertext, the probability that $w_{\mathrm{H}}(v_{r_1}) + w_{\mathrm{H}}(v_e) \le T_c$ is

$$P_G = \frac{\binom{\ell_{r_1}+\ell_e}{T_c} \cdot T_c^{w_r+w_e}}{(\ell_{r_1}+\ell_e)^{w_r+w_e}}. \tag{5}$$

According to Equation 5, the attacker needs to generate $1/P_G \approx 5.15 \times 10^4$ random messages to find such a basis ciphertext. In our experiments, we need 6.23×10^5 random messages, larger than the estimated value but still easily achieved in practice. We show the underlying basis message of one ciphertext in Table 3. We also builds constant errors with $I_c = \{0, 1, 2, \cdots, 14\}$ and $I'_c = \{15, 16, 17, \cdots, 29\}$, respectively for each basis ciphertext in our experiments applying Algorithm 4. We set the parameter L_0 in Algorithm 4 to 200 and the constant distance n to 2. As observed from our experiments, the attacker needs to generate 3.32 random inner blocks to find one constant error block. Therefore, the complexity of the offline phase is mainly determined by the process of generating the basis ciphertext, which costs $6.23 \times 10^5 \approx 2^{19.25}$ calls of the sampling process.

As for the online phase, we verify our result with 1000 different random keys. We recover parts of inner blocks of x and y by implementing Algorithm 9 and Algorithm 7, respectively. In our attacks, we set the Hamming weight of the nonzero inner block in the random error, i.e., L_1, to 150. We also set the number of iterations L_2 to 10 and the threshold T to 600 in Algorithm 9 and Algorithm 7. In our experiments, the attacker can find 53.06 inner blocks of the secret key sk, out of which 49.36 blocks are correct. The attacker makes approximately 53857 online queries in one attack, which sharply decreases 866,000 queries in [GHJ+22]. From our experiments, the attacker obtains more than 46 correct inner blocks in 836 cases out of 1000 different keys. Thus, the success rate of our attack is 83.6%.

Note that if the attacker obtains more than 46 correct inner blocks of sk, the attacker can recover the full secret key by solving the system of linear equations derived from the polynomial equation of Equation 4, with the method discussed in Subsection 4.3. To be more specific, the attacker can randomly select 46 blocks from the 53.06 guessed inner blocks. If all the 46 blocks are correct, the attacker fully recovers the secret key; otherwise, the attacker selects another combination of 46 guessed inner blocks. The probability that the attacker chooses 46 right inner blocks is $\frac{\binom{49}{46}}{\binom{53}{46}} \approx 1.2 \times 10^{-4}$. Thus, to fully recover the secret key, the attacker needs to solve $1/1.2 \times 10^{-4} \approx 8366.43$ systems of linear equations. This procedure can be practically done offline via amortizing computational costs. For instance, the attacker first groups the positions based on whether they are in the 53 guessed inner blocks, and applies Gaussian Elimination to positions outside these blocks to get them into echelon form. Then, the attacker diagonalizes 7 of the 53 blocks, repeating this step about 8366 times, contributing to the primal cost of the post-processing step. The overall cost is roughly bounded by 2^{46} bit operations. We also note here that the choice of parameters may not be the optimal. We leave the problem of finding the optimal parameters as future work.

In the practical scenario, the ideal oracle in our experiments can be replaced with the Prime+Probe cache-timing attack in Subsection 5.2. According to the experimental results, the Prime+Probe cache-timing attack may return “not confident” on the accesses of up to three cache lines. Even with the noise, our attack can still work. Specifically, the attacker can change line 5 in Algorithm 1 with the following two criteria:

1. The number of the undetermined cache lines is no more than 3.
2. $v_{r_1} = v_{\bar{r}_1}$ and $v_e = v_{\bar{e}}$ except for the "not confident" entries.

In this case, we analyse the accuracy of the PC oracle. The instance that the PC oracle cannot distinguish is that the cache line indicators of r_1 and e happen to be the same as the cache line indicator of $\bar{r}_1$ and $\bar{e}$ except for the undetermined entries. We denote the number of undetermined entries as u_d, which is set to 3. Then, the probability of this event is $P' = (\frac{T_c+u_d}{\ell_{r_1}+\ell_e})^{w_r+w_e}$. In this case, the accuracy of the PC oracle is $1 - P'$, which is still close to 1 with the parameters we set. Thus, our key recovery attack will still work in the practical scenario.

6 Conclusion

In this paper, we propose the first chosen-ciphertext cache-timing attacks on the reference implementations of HQC and practically recover the secret key. Our new attack presents a clear attack path. We also verify the attack via a proof-of-concept attack with the Flush+ Reload technique and a more realistic attack using Prime+Probe on an HQC execution on Intel SGX. Our result includes a new efficient method for chosen-ciphertext attacks on HQC with the PC oracles, which can be of independent interest.

Mitigation. Timing attacks can be mitigated by following the constant-time programming paradigm, namely, no secret-dependent control flow, memory access, or variable-time instruction. Accessing all array entries suffices to prevent our attack. The fourth round submission of the HQC reference implementation has integrated this countermeasure by accessing the whole index set instead of solely accessing indices with non-zero entries in a confidential sparse vector, and thus is now secure against the cache-timing attack described in our work. However, the proposed attacks still pose a threat to the PQClean library [6].

The concept of the cache line indicator we introduce may potentially be applicable to other cache-timing attacks. The techniques developed in this work could also be useful in developing new cache-timing attacks against other cryptographic primitives. Our research highlights, once again, the criticality of constant-time implementation practices for cryptographic libraries, particularly in eliminating secret-dependent memory access.

Implementing cryptographic software requires a particular care and an additional consideration since the program performs a computation with sensitive information. There exist tools such as Valgrind [Lana, Lanb] to check for constant-time behavior. There are also programming frameworks, e.g., Jasmin [ABB+17, ABB+20] and FaCT [CSJ+19], that enforce constant-time programming by default. Deploying these tools or frameworks to scrutinize timing vulnerabilities in their code before deployment would help prevent timing side-channel attacks.

Acknowledgement

We thank Yuval Yarom for his thoughtful discussion. We extend our gratitude to all reviewers for their insightful comments and feedback, particularly our shepherd Wenjie Xiong for the guidance and support.

This work was supported by the ELLIIT project; the ARC Discovery Project (project number DP210102670); the Swedish Research Council (grant numbers 2019-04166 and 2021-04602); the Swedish Civil Contingencies Agency (grant number 2020-11632); the

[6] https://github.com/PQClean/PQClean/tree/master/crypto_kem/hqc-rmrs-128/clean (accessed in 2023-04-01).

Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation; and a gift from Google.

References

[AAB+20] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. HQC. Technical report, National Institute of Standards and Technology, 2020. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, 2017.

[ABB+20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *IEEE S&P*, 2020.

[AMG+15] Ittai Anati, Frank McKeen, Shay Gueron, Haitao Huang, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos Rozas, and Hisham Shafi. Intel software guard extensions (Intel SGX). Tutorial slides presented at ISCA, 2015.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BCD+17] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *Eurocrypt*, 1997.

[BKS+22] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. Æpic leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security*, 2022.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Security*, 2017.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, 2011.

[BWK+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.

[CAGTB19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *TCHES*, 2019(4), 2019.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

[CFSY22] Chitchanok Chuengsatiansup, Andrew Feutrill, Rui Qi Sim, and Yuval Yarom. RSA key recovery from digit equivalence information. In *ACNS*, 2022.

[CGYZ22] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the Kalyna key expansion. In *CT-RSA*, 2022.

[CSJ+19] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: a DSL for timing-sensitive computation. In *PLDI*, 2019.

[CZRZ17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *AsiaCCS*, 2017.

[DDME+18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *TCHES*, 2018(2), 2018.

[EB22] Andre Esser and Emanuele Bellini. Syndrome decoding estimator. In *PKC*, 2022.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. *TCHES*, 2022(3), 2022.

[GLG22a] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. Estimating the strength of horizontal correlation attacks in the hamming weight leakage model: A side-channel analysis on hqc kem. *WCC*, 2022.

[GLG22b] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on HQC with chosen ciphertext. In *PQCrypto*, 2022.

[GLG22c] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on HQC with chosen ciphertext. In *PQCrypto*, 2022.

[GNNJ23] Qian Guo, Denis Nabokov, Alexander Nilsson, and Thomas Johansson. Sca-ldpc: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes. Cryptology ePrint Archive, Paper 2023/294, 2023.

[GPS+20] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. Cache vs. key-dependency: Side channeling an implementation of Pilsung. *TCHES*, 2020(1), 2020.

[GPTY18] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, 2018.

[GRB+17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *TCC*, 2017.

[HSM] An Intel SGX based hardware security module backed key management system. https://community.intel.com/t5/Blogs/Tech-Innovation/open-intel/An-Intel-SGX-based-Hardware-Security-Module-backed-Key/post/1360130. Accessed: 2023-04-07.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE S&P*, 2013.

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *DSD*, 2015.

[Kal20] Gil Kalai. The argument against quantum computers, the quantum laws of nature, and Google's supremacy claims. *CoRR*, abs/2008.05188, 2020.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[KPVV16] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *International Conference on Cryptology and Network Security*, 2016.

[Lana] Adam Langley. Checking that functions are constant time with Valgrind. https://github.com/agl/ctgrind. Accessed: 2023-04-07.

[Lanb] Adam Langley. Valgrind with ctgrind. https://github.com/agl/ctgrind. Accessed: 2023-04-07.

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, 2015.

[MBA+21] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In *USENIX Security*, 2021.

[MES18] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, 2018.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *USENIX Security*, 2017.

[MLSN$^+$15] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[MSEH20] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Security*, 2020.

[MSN$^+$15] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[NIS] NIST post-quantum cryptography standardization. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`. Accessed: 2018-09-24.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 2006.

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongswan's implementation of post-quantum signatures. In *CCS*, 2017.

[PGBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In *CCS*, 2016.

[SBWE21] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util: : Lookup: Exploiting key decoding in cryptographic libraries. In *CCS*, 2021.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *AsiaCCS*, 2016.

[SHR$^+$22] Thomas Schamberger, Lukas Holzbaur, Julian Renner, Antonia Wachter-Zeh, and Georg Sigl. A power side-channel attack on the Reed-Muller Reed-Solomon version of the HQC cryptosystem. *Cryptology ePrint Archive*, 2022.

[SKH$^+$19] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, 2019.

[SP17] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks. *CoRR*, abs/1712.08519, 2017.

[SRSWZ20] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the CCA2-secure HQC KEM. In *International Conference on Smart Card Research and Advanced Applications*, 2020.

[SWG$^+$17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[VBPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTex*, 2017.

[vSMK$^+$21] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *IEEE S&P*, 2021.

[WWB$^+$17] Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. Binary code retrofitting and hardening using SGX. In *FEAST@CCS*, 2017.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[Yar16] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. https://cs.adelaide.edu.au/~yval/Mastik, 2016.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[YFT20] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[YGL$^+$15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. ePrint Archive 2015/905, 2015.

A Recovering Inner Blocks of x

Constructing the basis ciphertext C_0. Given the basis message m used in recovering inner blocks of y, the attacker can build another basis ciphertext to recover inner blocks of x. For this time, the attacker manually sets $r_1 = 0$, $r_2 = 0$ and $e = 0$. The basis ciphertext C_0 is in the following form: $C_0^1 = h, C_0^2 = m\mathbf{G} + s$, where h and s are part of the public key pk, i.e., $\mathsf{pk} = (h, s)$. Therefore, the code word to be decoded during the decapsulation of C_0 is $v - u \cdot y = m\mathbf{G} + s - h \cdot y = m\mathbf{G} + x + h \cdot y - h \cdot y = m\mathbf{G} + x$. The error to be corrected by the decoder is exactly x. Similarly as recovering inner blocks of y, the attacker's task is to find the error. Algorithm 8 shows the procedure of constructing C_0.

Algorithm 8 Generating a basic ciphertext C_0

```
1: procedure GenTextX($m$, $\mathsf{pk}$)
2:     return $C_0 = (h, m\mathbf{G} + s)$
```

Recovering inner blocks of x of Hamming weight less than 1. The procedure of recovering an inner block of x is presented in Algorithm 10. It can be seen that recovering x_j is slightly different from recovering y_j in Algorithm 6. As shown in line 6 and line 17, the attacker need to add the second half of the public key s before calling the decoder.

Algorithm 9 Recovering inner blocks of x of hamming weight less than 1

```
1: $C_0$ =GenTextX($m$, $\mathsf{pk}$)
2: $I_c = \{0, 1, 2 \cdots, 14\}$
3: for each integer $i \in [15, 46)$ do
4:     $x_i$ =KeyRecoverX($C_0$, $E_c$, $m$, $j$, $L_1$, $L_2$, $T$)
5: $I_c = \{15, 16, 17 \cdots, 29\}$
6: for each integer $i \in [0, 15)$ do
7:     $x_i$ =KeyRecoverX($C_0$, $E'_c$, $m$, $i$, $L_1$, $L_2$, $T$)
```

The procedure of recovering blocks of x, which is shown in Algorithm 9, is almost the same as the procedure of recovering blocks of y. It should be noted that as the constant errors can be reused in the attacker targeting at x, E_c and E'_c are exactly the same constant errors in Algorithm 7.

Algorithm 10 Recovering x_j

```
 1: procedure GuessCase1($C_1$, $v_{r_1}$, $v_e$,$s$)
 2:     Initialize $x_j$ as 0.
 3:     for each $i \in [0, 384)$ do
 4:         $C_2 = C_1$
 5:         Filp the $i$-th bit of the $j$-th inner block of $C_2^2$.
 6:         $m' = \mathcal{C}$.Decode($C_2^2 + s$).
 7:         $\theta = \mathcal{G}(m')$, sampleInit($\theta$), $\hat{r}_1$ =sample($\mathcal{R}, \omega_r$) and $\hat{e}$ =sample($\mathcal{R}, \omega_e$).
 8:         Deduce the cache line indicators $v_{\hat{r}_1}$ and $v_{\hat{e}}$ from $\hat{r}_1$ and $\hat{e}$.
 9:         if $v_{\hat{r}_1} = v_{r_1}$ and $v_{\hat{e}} = v_e$ then
10:             $x_j[i] = 1$
11:     return $x_j$
12: procedure GuessCase2($C_1$, $m$, $s$)
13:     Initialize $x_j$ as 0.
14:     for each $i \in [0, 384)$ do
15:         $C_2 = C_1$
16:         Filp the $i$-th bit of the $j$-th inner block of $C_2^2$.
17:         $m' = \mathcal{C}$.Decode($C_2^2 + s$).
18:         if $m = m'$ then
19:             $x_j[i] = 1$
20:     return $x_j$
21: procedure KeyRecoverX($C_0$, $E_c$, $m$, $j$, $L_1$, $L_2$, $T$)
22:     Initialize $x_j$ as 0.
23:     for each integer $i \in [0, L_2)$ do
24:         $(flag, C_1, v_{r_1}, v_e)$=BuildError($C_0$, $m$, $E_c$, $j$, $L_1$, $T$)
25:         if $flag = 0$ then
26:             return $x_j = 0$
27:         else if $flag = 1$ then
28:             $x_j = x_j$ & GuessCase1($C_1$, $v_{r_1}$, $v_e$, $s$)
29:         else if $flag = 2$ then
30:             $x_j = x_j$ & GuessCase2($C_1$, $m$, $s$)
31:     if $y_j = 0$ then
32:         return $w_{\mathrm{H}}(x_j) > 1$
33:     else
34:         return $x_j$
```

B Table 2 in [GLG22c]

Table 2: The probability that a randomly sampled inner block in y has a Hamming weight of k

Variants	P_0	P_1	P_2	P_3	P_4	$P_{\leq 5}$
hqc-128	23.44%	34.38%	24.83%	11.77%	4.12%	1.45%
hqc-192	16.50%	30.00%	27.00%	16.04%	7.07%	3.40%
hqc-256	23.14%	34.06%	24.87%	12.02%	4.32%	1.59%

C Basis Message and Constant Errors

Table 3: Basis message

m	b32cf395d8184f48	93401410b1ff6508

Table 4: The nonzero inner blocks of the constant error E_c

$E_c^2[0]$	ac9f0c4b377779e4 85803629d67e337a	d7115a9b4949421e ddce41f18fbb85d4	dfd193ce365da033 53e3738f4db0cd12
$E_c^2[1]$	bf63e4e2b3404a51 775893295fade58c	27b76707c70e1dba b51c557f48b78d2d	125ded31548c7582 5dfff51c51805d55
$E_c^2[2]$	dae47ef360f622f6 13aaf10f9f0cc6f	9788a79f59b08e72 e31a9bb4b4e5d7c6	1d8772aff0807854 7b20aef205eb612b
$E_c^2[3]$	a459924ddc73824a a869016776fd2a09	b42a80cfe299b7be f70dcf7b8e61f313	2f9bc44dec4dd57f 36b6cba105f611aa
$E_c^2[4]$	39e488d02455bc80 eb31bce9360df4c0	e7a43f580c6ac449 3962c17ff6627e98	e7dd4fd16199275d f51feed79cb8b4d9
$E_c^2[5]$	1b3d1bb1fcfbb4c8 102d0011cf5d717e	ce3a1e161b582328 1dafd720f1939977	ce55f2066da6c2eb 55d70bfa7fc73c44
$E_c^2[6]$	5e5b9f8f6345bbfe 82a165d6322c0afa	62b366d2ff717255 e137b05717431e4f	c5243d9531a5ef3d 97988939d0e78422
$E_c^2[7]$	66de937c31661e15 213890aaad48c97a	fac456af77aa9a2f 161af42d22d9cc34	96e136bc5a71e1a5 eabda3bafbfca195
$E_c^2[8]$	3911c21a805de15f 7946a18f13c9a140	95fde9ba619f873b ffba4a4dfbc66de7	9930fb38844acd7f b4cedc841727d0d1
$E_c^2[9]$	aac1ea2b06ac3f5a 75789449bcdffc35	cb64cdd9a6ead100 40eb46c42c665022	962fc46c4efdd4d4 f9dbb7fb9d183b8d
$E_c^2[10]$	ae1b91cd3bd3c6d3 75e26399783f620b	30492abbf9d3b005 fe70b09b8456b5cc	a84925f63df0b71b 4271bdb11abadbe4
$E_c^2[11]$	1a37314e1a092b64 9eb9f357d29ff715	e13d2f079e6c7167 22395a249d3cfe11	1b0bc47c9199c96b b5c9f9d179ca26cd
$E_c^2[12]$	e59ec15c306a24aa b9cff324906ec912	438fa7170930dc19 571785b7d43ebb86	b533bf9d3837751c f899486fbdb572ce
$E_c^2[13]$	5b5b98bbbc8c1b10 526909d5c1e0ff62	eda27a8e3ea6201e 1ef5c1e215ce0b87	d54197c8fe36f206 e9fe7e9f5e063d6c
$E_c^2[14]$	1a9d45047c95663f f56c29f0a73daedc	eda37f438de714a7 acd37519be4da06e	dbcd817484c22d3c c69e91c8d3c74926

Table 5: The nonzero inner blocks of the constant error E'_c

$(E'_c)^2[0]$	4cd38097c327e773 324d97c7f4affef0	97a8e71ea01c4c07 abd7f91eb3477853	98d4541932486793 6ca4f7239293737a
$(E'_c)^2[1]$	bc27c11e92b3fbd4 da8adc4a86fcc4e8	b9e1ff5a4be52f8c d964474217c07dcc	230a82db294d6039 88d7c6fbbc73fa35
$(E'_c)^2[2]$	a3cbee979a5a680b 85f514d661c216a	efa516acd6f8f987 bb65173565522f1d	cc331b8d52341fdc 576d884e8f9cb6a7
$(E'_c)^2[3]$	c1bd761623de5e4a 7f984dab9a4a9605	b77ca8324572e1c0 7d3939a341929549	973e3b7dce783341 5dff9325de4b1f62
$(E'_c)^2[4]$	b69507cb1d5e3994 5df9f266cb3f62fd	67d02b2a86303f0d e671382d10877261	537d7577c5e08f80 b63797c4aa69a4de
$(E'_c)^2[5]$	4aadd79cc0e9be49 64ce246ddc22a9eb	43fe0519f453fbcd ab60a86cea2ffbfb	d4259c0df94932c6 4cbecdd3a88125a8
$(E'_c)^2[6]$	61a8aafc5aca3801 8acae58dcec99da1	4b961e747e054db5 7352f12077364641	539b97a91cf32dbe 51f9f2a975edd7d5
$(E'_c)^2[7]$	4f5fdb11fcc6d305 ee1aa6a08f39f897	e35c1c51e44c0327 8b4c0e3bc6bb2ad6	6a0f6ac6c560ee8f 37897fb4317e50eb
$(E'_c)^2[8]$	5b487516a5eeaf29 2138e5c38e93bcb6	e0fe3e9ee697186c 47d0eea29485537c	c168685d38ce5fbf f461dfba48238f14
$(E'_c)^2[9]$	c202ebe801fae26b 77d3ad8d5f87434a	5bba1474a381c2b7 f94b5ac2dbd7eaf8	c6c2a0d77925d473 778acaaec609989e
$(E'_c)^2[10]$	25a59c639a6d0c7c 5cb1477c5aff04a8	51dc147ff2808dc9 ac4956e3decfafa8	1b2a7ebf833ec7fc a069ae8995a948b9
$(E'_c)^2[11]$	931f3ee0cfc8b39e fa87416079a59115	49712529faaecb9b f1b68f2cd56f94f8	312b5732a23ebd03 1999f54a4abf829e
$(E'_c)^2[12]$	734a7c6bef6d1618 e4187e162eb2f5d8	c4e6f986312cf78a 6bacbb3ebcc3d5e4	954bab9fa85b9127 d168060c0ff57908
$(E'_c)^2[13]$	d492e19b28bff1a7 c1d4dcaf83fd961b	9807cf085a7adaae 7481151ad1da31c9	77ab6eefc8ac0143 dd8e6da33ca15787
$(E'_c)^2[14]$	d8b76989a2bc6e9d 2101b696f473ad36	9fc4a231e46fdedb 1d9438ca06f9b139	9eba83ccc461acea 59db32c58cd8fe87

D Example of Cache Traces and Classification From the Practical Attack

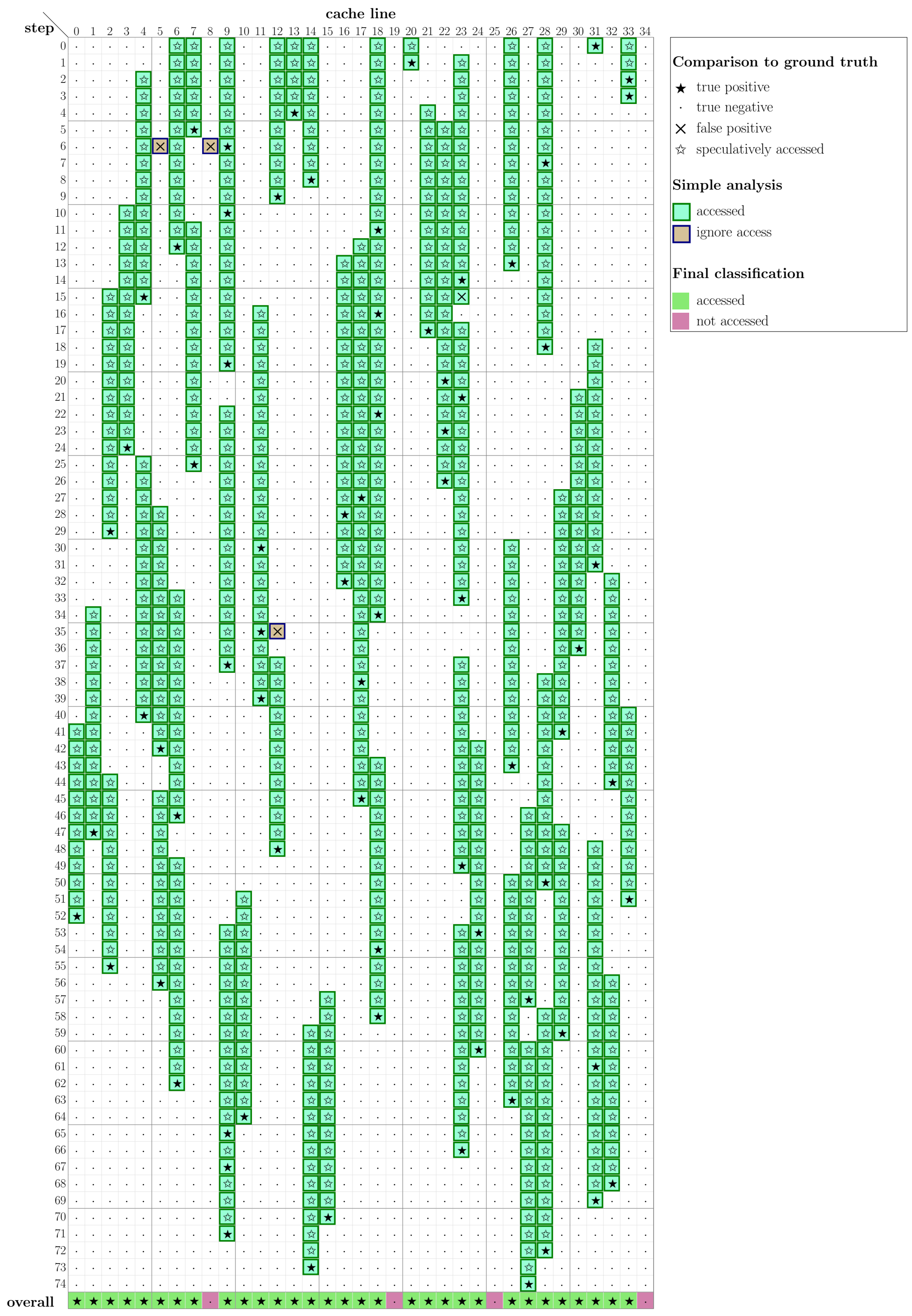


Figure 5: Trace of the cache accesses compared to ground truth at each step using a simple analysis.

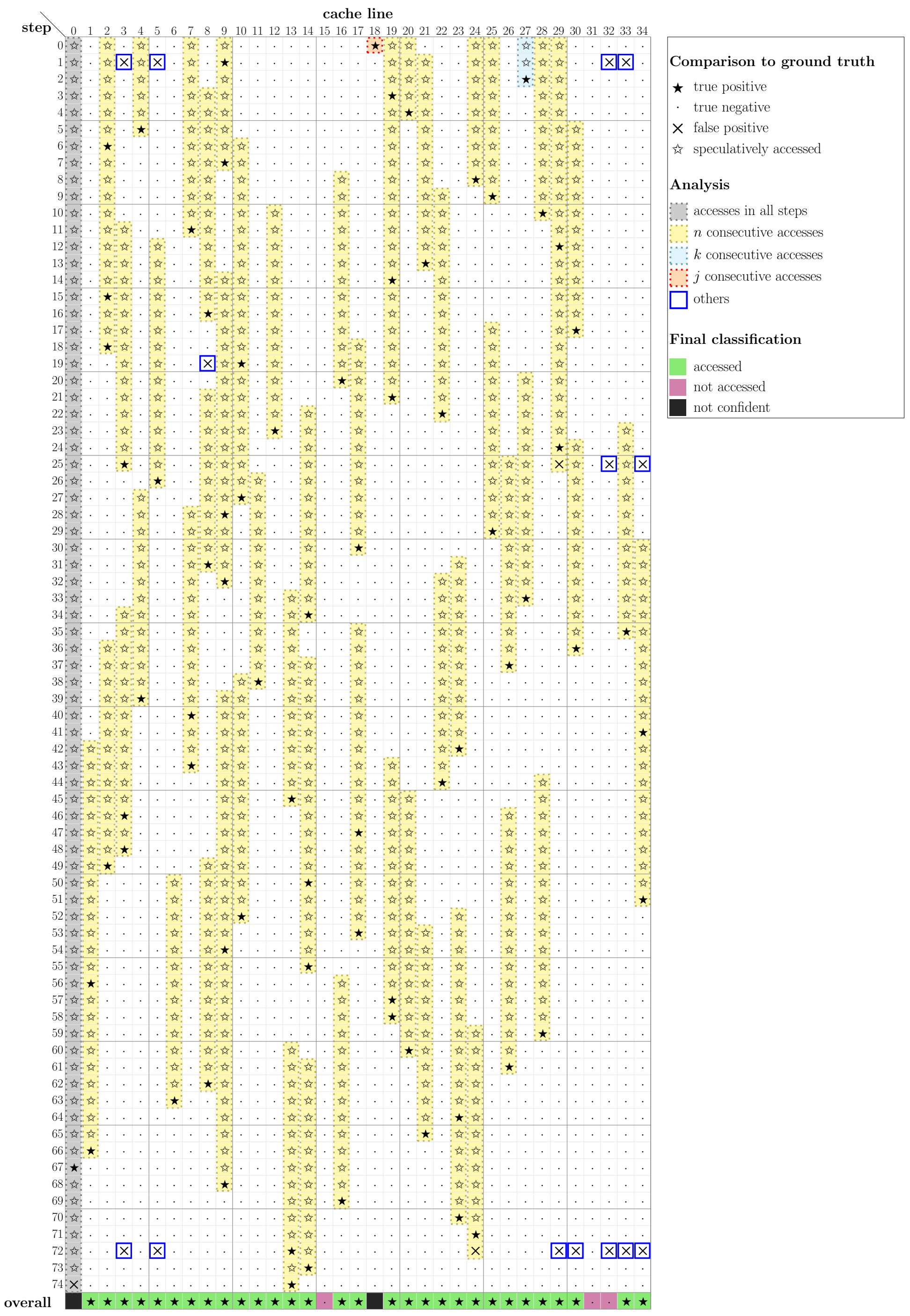


Figure 6: Example of identifying accesses using the rules.

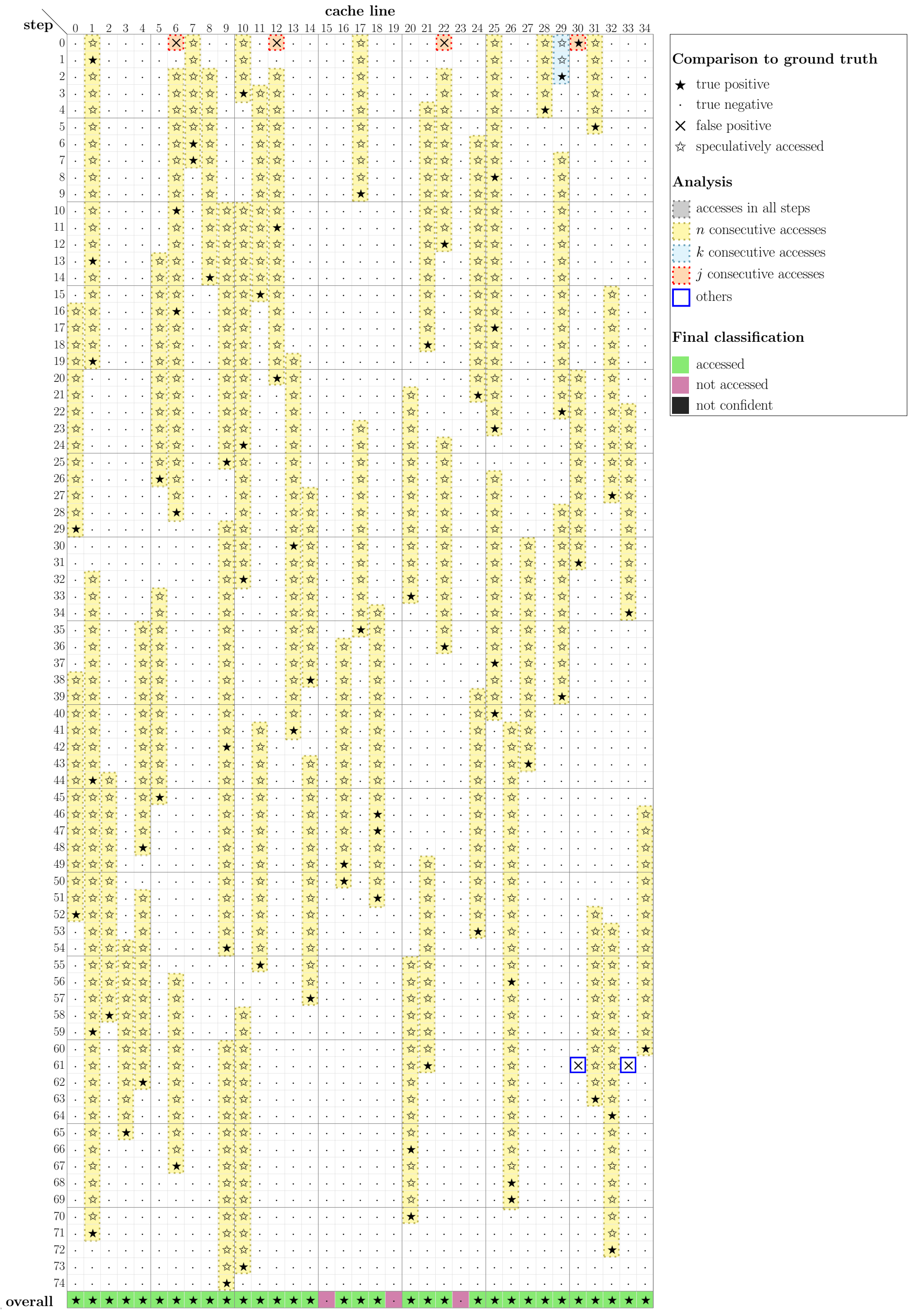


Figure 7: Example of a cache trace with groups of a single access in step 0.