Divide and Surrender:
Exploiting Variable Division Instruction Timing in
HQC Key Recovery Attacks

Robin Leander Schröder¹,², Stefan Gast³ and Qian Guo⁴

¹ Fraunhofer SIT, Darmstadt, Germany, leander.schroeder@sit.fraunhofer.de
² Fraunhofer Austria, Vienna, Austria, leander.schroeder@fraunhofer.at
³ Graz University of Technology, Graz, Austria, stefan.gast@iaik.tugraz.at
⁴ Lund University, Lund, Sweden, qian.guo@eit.lth.se

Abstract. We uncover a critical side-channel vulnerability in the Hamming Quasi-Cyclic (HQC) round 4 optimized implementation arising due to the use of the modulo operator. In some cases, compilers optimize uses of the modulo operator with compile-time known divisors into constant-time Barrett reductions. However, this optimization is not guaranteed: for example, when a modulo operation is used in a loop the compiler may emit division (div) instructions which have variable execution time depending on the numerator. When the numerator depends on secret data, this may yield a timing side-channel. We name vulnerabilities of this kind Divide and Surrender (DaS) vulnerabilities.

For processors supporting Simultaneous Multithreading (SMT) we propose a new approach called DIV-SMT which enables precisely measuring small division timing variations using scheduler and/or execution unit contention. We show that using only 100 such side-channel traces we can build a Plaintext-Checking (PC) oracle with above 90% accuracy. Our approach may also prove applicable to other instances of the DaS vulnerability, such as KyberSlash. We stress that exploitation with DIV-SMT requires co-location of the attacker on the same physical core as the victim.

We then apply our methodology to HQC and present a novel way to recover HQC secret keys faster, achieving an 8-fold decrease in the number of idealized oracle queries when compared to previous approaches. Our new PC oracle attack uses our newly developed Zero Tester method to quickly determine whether an entire block of bits contains only zero-bits. The Zero Tester method enables the DIV-SMT powered attack on HQC-128 to complete in under 2 minutes on our targeted AMD Zen2 machine.

Keywords: Post-quantum cryptography · code-based cryptography · HQC · plaintext-checking oracle · timing side-channel attack

1 Introduction

The rise of quantum computing presents a grave threat to current cryptographic infrastructures, primarily based on problems like factoring and discrete logarithms. Shor’s algorithm [Sho94], in particular, significantly compromises these security foundations. In response, the National Institute of Standards and Technology (NIST) initiated a process in 2016 to identify and standardize quantum-resistant public-key cryptographic algorithms, focusing on developing new standards for Key Encapsulation Mechanism (KEM) and digital signatures. This initiative drew a robust response from the cryptographic community and after thorough review and feedback, NIST endorsed CRYSTALS-Kyber [SAB⁺22] as the primary KEM algorithm and CRYSTALS-Dilithium [LDK⁺22], FALCON [PFH⁺22],

Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks

and SPHINCS+ [HBD+22] as key digital signature algorithms. This initiative marks a significant shift towards securing cryptographic practices against emerging quantum computing threats.

One notable aspect of NIST’s selection is its emphasis on lattice-based algorithms like CRYSTALS-Kyber, CRYSTALS-Dilithium, and FALCON, reflecting a preference for lattice-based cryptography. However, to ensure a robust cryptographic future, NIST also prioritizes diversity in cryptographic solutions, hence its ongoing round 4 selection process included non-lattice-based KEM algorithms. This phase spotlighted code-based KEMs such as Classic McEliece [ABC+22], HQC [AAB+22], and BIKE [ABB+22], alongside a now broken isogeny-based KEM, SIKE [JAC+22]. Among these, the HQC proposal [AMBD+18, AGZ20, AAB+22], based on hard decoding problems in coding theory, emerges as a promising candidate [AAC+22].

Given HQC’s potential for NIST standardization, an immediate and critical focus must be directed towards scrutinizing HQC’s security claims. Specifically, secure implementation strategies for HQC on various real-world platforms warrant exhaustive study.

Numerous studies, such as [WTBB+19, PT19, BDH+19, GJ20, GJN20, XIU+21, SHR+22, GLG22, GHJ+22, HSC+23], have extensively explored HQC’s security and implementation aspects, adopting a dynamic attack-and-fix methodology throughout NIST’s initial evaluation rounds. The fourth-round HQC submission includes a single implementation claimed to be constant-time, suggesting resilience against timing attacks [Koc96]—a type of attack that leverages variations in execution time to compromise security. As constant-time implementation is a critical aspect in cryptographic libraries, it is a fascinating research challenge to either validate this security claim or identify a more efficient attack strategy for potent key-recovery attacks.

In our paper, we identify a novel timing side-channel vulnerability in the implementation for HQC, originating from division operations that are not constant-time. This vulnerability enables an efficient key recovery timing attack against the claimed constant-time implementation in the designers’ round 4 submission to NIST. This makes HQC one of the few known cases where division instructions are used in a cryptosystem with secret inputs and cause an exploitable timing difference. We call these vulnerabilities instances of the DaS vulnerability. Other instances are a potential fix for the Lucky13 attack [Lan04] and KyberSlash [Ber23, GTS23]. In KyberSlash similar code structure causes the compiler to emit div instructions where constant-time code could have been generated.

Our approach builds on the established PC oracle-based side-channel attacks, creating a PC oracle that verifies if a particular ciphertext \( c \) decrypts to a given message \( m \). This type of attack can be traced back to the reaction attack, initially introduced by Hall et al. in 1999 [HGS99]. Furthermore, this attack methodology has found broad applicability in the realm of post-quantum cryptography, as evidenced by works such as [GJS16, FHS+17, RRCB20, QCZ+21, NDGJ21, WPH+22], and is applicable to various side-channels such as timing, cache timing, power, and electromagnetic radiation, provided the side-channel leakage is sufficient to construct a PC oracle.

We develop the desired PC oracle using a novel side-channel, called DIV-SMT, which originates from integer modulo operations in some SMT processors such as AMD Zen+/Zen2 processors.

On the x86-64 instruction set architecture, implemented by the AMD Zen+/Zen2 processors, the div instruction is used to calculate both, the quotient and the modulo of the two operands. Hence, divisions and modulo operations in the source code are usually both compiled to div instructions. Compiler optimizations eliminate the div instructions in some cases. Specifically, with a divisor known at compile time, the GCC compiler might apply Barrett Reduction [Bar87] as an optimization. However, as we show in Section 4, this reduction is sometimes not performed, even if it would be possible.

In these widely deployed high-end processors, varying numerator and divisor sizes in
integer divisions can result in single cycle execution time differences. Exploiting such small timing differences is non-trivial and the \texttt{div} instructions only make up a small part of a complete decapsulation in the HQC cryptosystem. Further, advanced CPU features like out-of-order execution can render timing measurement more challenging. Thus, the SMT context becomes crucial here, allowing the attacker and victim threads to run on the same physical core with shared caches, execution units, and schedulers. This setup can create contention on execution units \cite{ABuH18} or schedulers \cite{GJS23}, enabling an adversary to detect when a co-located program executes a specific instruction, such as a multiplication or a division.

In contrast to these prior works, which attacked non-constant-time ECDSA and RSA implementations by detecting the points in time when specific instructions are executed through a timing side channel, our novel approach infers information about the numerators for the same division instruction. Our attack highlights a serious security risk in the HQC implementation submitted to NIST that claims to be constant-time. The claimed constant-time property is crucial for the safe integration of HQC into real-world libraries.

While avoiding non-constant-time instructions is a standard practice in constant-time implementations, division operations in modern high-end CPUs have previously seen little cryptographic research attention due to the lack of documented vulnerabilities and exploits resulting from their timing behavior. To our knowledge, our research presents the first key-recovery attack that clearly demonstrates the potential of exploiting this vulnerability on current, SMT enabled CPUs. Thus, our methodology of distinguishing different numerators in an SMT environment can offer valuable insights for security assessments beyond attacking HQC.

In December 2023, a vulnerability related to the use of modulo operations, potentially compiling into \texttt{div} instructions, was independently identified in the CRYSTALS-Kyber reference implementation by Bernstein \cite{Ber23} and by Tamvada, Kiefer, and Bhargavan \cite{GTS23}. The CRYSTALS-Kyber design team has acknowledged and purportedly rectified this issue. These groups did not present an attack strategy for exploiting this vulnerability in high-end modern CPUs, but their discoveries highlight the broad relevance and impact of our innovative attack methodology in the SMT context, as we deem it likely that our DIV-SMT methodology can also be used to exploit KyberSlash vulnerable implementations of Kyber.

Utilizing the PC oracle derived from the DIV-SMT side-channel, we further advance our research by developing a methodology that significantly reduces the number of PC oracle queries needed to extract HQC’s secret key. This new approach holds significance beyond the newly discovered timing attack exploiting the DaS vulnerability for two main reasons. First, it relies on a PC oracle that can be built from various side-channel leakages. This versatility makes it suitable for a broad spectrum of PC oracle-based side-channel attacks on HQC, thus widening its applicability beyond just timing attacks. Second, since interacting with the oracle is often the primary bottleneck in such attacks, optimizing oracle calls is crucial for efficiency in most of real-world attack scenarios.

We introduce a new technique called Zero Tester for efficiently identifying through PC oracle calls whether a consecutive block, referred to as an ‘inner block’ in HQC’s coding scheme, consists solely of zero-bits. Through the integration of this method with a shifting strategy, we have the capacity to pinpoint numerous zero-bit’s positions inside the secret vector using few queries. This method significantly decreases the speed of full key recovery: Given that the public key already provides \(n\) linear equations for the \(2n\) unknowns in HQC’s secret key, identifying approximately \(n\) out of \(2n\) unknown zero positions is sufficient.

**Summary of Contributions.** Our core contributions are:

- We unveil a novel timing side-channel vulnerability in the supposedly constant-time optimized reference implementation of the HQC round 4 submission to NIST, which
**Divide and Surrender:**

- We propose a new practical exploitation strategy of division timing variance through the use of DIV-SMT side-channels in SMT processors. By leveraging contention between SMT sibling threads, we detect the execution of specific instructions (div) and infer information about the operands processed (specifically, the count of leading zeros in the numerands). This insight lays the groundwork for creating a Plaintext-Checking (PC) oracle, enabling us to execute a key-recovery timing attack.

- We propose a novel key-recovery method for HQC using the PC oracle, significantly reducing the required number of side-channel traces. A detailed comparison with prior studies for HQC-128 key-recovery is presented in Table 1. We conduct extensive simulations to investigate the efficacy of our proposed key-recovery method, employing a simulated PC oracle across different levels of oracle accuracy. Our simulations demonstrate a significant improvement requiring only 11% to 13% of the oracle calls relative to the SCA-LDPC framework presented at ASIACRYPT 2023 [GNNJ23]. Notably, this gain persists even as the oracle’s accuracy decreases to 0.9.

- We demonstrate the practical applicability of our attack through evaluation on an AMD Zen2 CPU. Out of 1000 attacks conducted, three-quarters lead to successful full key recovery, achieving a median attack time of 111 seconds.


**Responsible Disclosure.** We disclosed the issue to the HQC design team in February 2023. After we submitted our paper and published it on IACR eprint [SGG24] the scheme designers released a fixed optimized reference implementation on 2024-02-23 [GH24]. The designers chose to manually implement the Barrett reductions (cf. Section 6.1) [HQC24]. Furthermore, we disclosed the issue to the PQClean team in March 2023 [amb23]. We believe our disclosure allowed implementers to implement a presumably constant-time version of the vector sampling algorithm using explicit Barrett reductions.

**Outline.** The rest of this paper unfolds as follows: Section 2 provides the essential background, covering both the HQC KEM proposal and relevant aspects of CPU architectures. We present our main attack strategies in Section 3. This is followed by the introduction of a novel PC oracle in Section 4, specifically designed to exploit a division timing side-channel to challenge the constant-time claims of an implementation submitted to NIST. Experimental results are reported in Section 5, followed by a comprehensive discussion in Section 6. Section 7 draws conclusions.

## 2 Preliminaries

In this section, we present some preliminary background on the HQC proposal, CPU Pipelines, and SMT.

**Notations.** Consider $\mathbb{F}_2$ as the binary finite field. The Hamming weight of a binary vector is defined as the number of non-zero entries in the vector. In the HQC scheme, we employ a cyclic polynomial ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$, where $n$ is an integer in $\mathbb{Z}$. Elements
2.1 HQC

HQC [AAB+22], a code-based post-quantum IND-CCA secure KEM, leverages the complexity of decoding random quasi-cyclic codes in the Hamming metric for its security. It was proposed as a KEM candidate in the fourth round of the NIST Post-Quantum Cryptography (PQC) standardization [Cen23]. At the conclusion of the fourth round, NIST intended to standardize a single code-based KEM primitive, either HQC or BIKE. Similar to other NIST PQC Public Key Encryption (PKE)/KEM candidates, HQC proposal commences with an IND-CPA version, termed HQC.CPAPKE, and subsequently introduces an INC-CCA KEM, called HQC.CCAKEM, through a CCA transformation (HQC utilizes the Hofheinz-Hövelmanns-Kiltz (HHK) transformation [HHK17]).

2.1.1 The PKE Version of HQC

The HQC.PKE algorithm comprises three sub-procedures: PKE.KeyGen, PKE.Encrypt, and PKE.Decrypt. Within PKE.KeyGen, the algorithm uniformly samples three polynomial elements—\( h, x, \) and \( y \), with \( x \) and \( y \) maintaining fixed Hamming weights of \( w \). The secret key is designated as \( (x, y) \) and the public key as \( (h, s = x + h \cdot y) \). PKE.Encrypt begins by initializing the pseudo-random number generator (PRNG) with a seed \( \theta \) making the sampling process deterministic. Subsequently, the algorithm uniformly samples the polynomials \( r_1 \) and \( r_2 \) from \( \mathcal{R} \), each possessing a Hamming weight of \( w_r \), and \( e \) with Hamming weight \( w_e \). The ciphertext is designated as \( (u, v) \), where \( u = r_1 + h \cdot r_2 \) and \( v = mG + s \cdot r_2 + e \). The matrix \( G \) relies on the linear code being utilized, which we will describe later. PKE.Decrypt employs a decoder on the input \( v - u \cdot y \), resulting in the equation \( mG + s \cdot r_2 + e - (r_1 + h \cdot r_2) \cdot y = mG + x \cdot r_2 - r_1 \cdot y + e \), given that \( s = x + h \cdot y \). Should the Hamming weight of the error term \( e' = x \cdot r_2 - r_1 \cdot y + e \) be small (i.e., within the decoding capability of the employed decoder), the decoder could correct such an error, leading to successful decryption.

Beginning with the October 2020 release, HQC transitioned to a design incorporating a decoding strategy that utilizes a concatenated code combining an internal duplicated Reed-Muller (RM) code and an outer Reed-Solomon (RS) code. The resultant code produces a publicly known generator matrix \( G \in \mathbb{F}_2^{n_1 \times n_2} \), where \( k = 8k_1 \).

The specifics of the HQC parameters are detailed in Table 2. HQC computations occur within the ambient space \( \mathbb{F}_2^{n} \), with any remaining \( n - n_1n_2 \) positions of no value discarded. The concatenated code, \( C \), merges an internal duplicated RM code with an outer RS code. The internal duplicated RM code possesses parameters \( [n_2, 8, n_2/2] \), and the outer RS code is defined as \([n_1, k_1, n_1 - k_1 + 1]\).

During the encoding process, a message \( m \in \mathbb{F}_2^{n_1} \) translates into \( m_1 \in \mathbb{F}_2^{n_2} \) by the outer Reed-Solomon code. The internal duplicated Reed-Muller code then encodes each byte \( m_{1,i} \) into \( \tilde{m}_{1,i} \in \mathbb{F}_2^{n_2} \), where \( 0 \leq i < n_1 \). Consequently, we attain \( mG = (\tilde{m}_{1,0}, \cdots, \tilde{m}_{1, n_1 - 1}) \).

To decode \( V \) we use \( V - u \cdot y \), \( V \in \mathbb{F}_2^{m_1n_2} \) is partitioned into \( n_1 \) blocks, denoted as \( V = (V_0, \cdots, V_{n_1 - 1}) \), with each \( V_i \in \mathbb{F}_2^{n_2} \) defined as an ‘inner block’, where \( 0 \leq i < n_1 \). Each \( V_i \) undergoes decoding by the internal duplicated Reed-Muller code into \( \tilde{V}_i \in \mathbb{F}_2^{n_2} \), where \( 0 \leq i < n_1 \). Thereafter, \( \tilde{V} \) is compiled as a string of \( 8n_1 \) bits, denoted as \( (\tilde{V}_0, \cdots, \tilde{V}_{n_1 - 1}) \). For each \( i \in [0, n_1] \), \( V_i \) is termed an ‘inner codeword’. It is observable that \( \tilde{V} \) is a noisy codeword of the outer Reed-Solomon code, which can be decoded into \( k_1 \) elements over \( \mathbb{F}_{256} \) and transformed into \( k_1 \) message bytes.
### Table 2: The HQC parameter sets [AAB+22]. The base Reed-Muller code is the first-order $[128,8,64]$ Reed-Muller code.

<table>
<thead>
<tr>
<th>Instance</th>
<th>$n_1$</th>
<th>$k_1$</th>
<th>$d_{RS}$</th>
<th>Mult. $n_2$</th>
<th>$d_{RM}$</th>
<th>$n_1n_2$</th>
<th>$n$</th>
<th>$\omega$</th>
<th>$\omega_r = \omega_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HQC-128</td>
<td>46</td>
<td>16</td>
<td>31</td>
<td>3</td>
<td>384</td>
<td>192</td>
<td>17664</td>
<td>17669</td>
<td>66</td>
</tr>
<tr>
<td>HQC-192</td>
<td>56</td>
<td>24</td>
<td>33</td>
<td>5</td>
<td>640</td>
<td>320</td>
<td>35840</td>
<td>35851</td>
<td>100</td>
</tr>
<tr>
<td>HQC-256</td>
<td>90</td>
<td>32</td>
<td>49</td>
<td>5</td>
<td>640</td>
<td>320</td>
<td>57600</td>
<td>57637</td>
<td>131</td>
</tr>
</tbody>
</table>

**Input:** $pk$

**Output:** $K, c = (u,v), d$

1: $m \leftarrow \mathcal{F}_k^2$
2: salt $\leftarrow \mathcal{F}_1^{128}$
3: $\theta \leftarrow \mathcal{G}(m||pk||salt)$
4: $c \leftarrow \text{PKE.Encrypt}(pk,m,\theta)$
5: $K \leftarrow \mathcal{K}(m,c)$

(a) KEM.Encaps

**Input:** $sk = (x,y), c = (u,v), \text{salt}$

**Output:** $K$

1: $m' \leftarrow \text{PKE.Decrypt}(sk,c)$
2: $\theta' \leftarrow \mathcal{G}(m'||pk||salt)$
3: $c' \leftarrow \text{PKE.Encrypt}(pk,m',\theta')$
4: if $m' = \perp \lor c \neq c'$ then
5:  $K \leftarrow \mathcal{K}(\sigma,c)$
6: else
7:  $K \leftarrow \mathcal{K}(m',c)$

(b) KEM.Decaps

**Figure 1:** HQC.CCAKEM

### 2.1.2 The KEM Version of HQC

The HQC KEM, depicted in Figure 1, is constructed from the HQC PKE scheme using a HHK transform. Crucially the encryption is now de-randomized and the decapsulation performs a re-encryption step to verify the validity of the ciphertext. Additionally the decapsulation invokes two distinct cryptographic hash functions: $\mathcal{G}$ and $\mathcal{K}$. The seed $\theta$ is derived from the message $m$, the public key and a random 128-bit salt. The seed $\theta$ and the inputs that are used to derive it are vital to the exploitation of the scheme, as it determines the side-channel behavior of a ciphertext (cf. Section 4).

### 2.2 CPU Pipelines

Modern superscalar CPUs execute multiple instructions in parallel [Fog21]. The CPU frontend decodes instructions into micro-ops (µops) and forwards them to the backend via a dedicated dispatch buffer. The CPU backend has multiple execution units, processing µops in parallel. Simple µops, (e.g., additions of two registers) can typically be executed by multiple execution units, whereas more complex µops (e.g., divisions) require a specialized execution unit [AMD23, Int23]. The execution units are controlled by one [Int23] or multiple [AMD20a, AMD20b, AMD23] schedulers that retrieve µops from the dispatch buffer and determine which µops are ready for execution, based on their operand dependencies, enabling out-of-order execution. The results of the executed µops become architecturally
visible after they retire in the instruction stream order.

2.3 Simultaneous Multithreading (SMT)

CPUs reach their maximum performance if they utilize all execution units simultaneously. Usually, this is not achieved with a single instruction stream. Therefore, many modern CPUs execute multiple instruction streams on the same core, sharing caches, execution units and schedulers. Most Intel and AMD CPUs support two SMT threads on the same core [Int23, AMD23]. Some Power10 CPUs can support up to 8 SMT threads per core [pow]. While this increases performance, it also enables multiple side channels [Sze19, TRVT22]. Particularly, contention on execution units [ABuH18] or schedulers [GJS+23] enables an adversary to observe that a co-located program executes a specific machine instruction. This in turn allows an adversary to recover secret keys when attacking non-constant-time ECDSA and RSA implementations [ABuH18, GJS+23].

3 New Key-Recovery Attack

In this section, we present a novel key-recovery attack on HQC using a PC oracle. First, we present a general description of the threat model. Next, we review several existing attacks in this context. Finally, we detail our new method, which efficiently identifies blocks in the secret vector that have a zero Hamming weight.

3.1 The Generic Threat Model

In our study, we investigate a side-channel-assisted chosen-ciphertext attack model targeting HQC’s decapsulation algorithm. Here, the attacker picks specific ciphertexts and closely observes various side-channel data emanating from the targeted device. This could include timing, cache-timing, or even power and electromagnetic leakages. Our general threat model posits that the attacker can construct a PC oracle $O_{\text{HQC}}$ from side-channel leakage. This oracle confirms or denies whether $\text{PKE}_{}(sk, c) = m$, where $c$ represents the ciphertext and $m$ a message vector. We assume that the adversary can choose both the ciphertext and the message. This oracle has an accuracy level of $\rho$, meaning it returns a correct decision with a probability of $\rho$ and an incorrect one with a probability of $1 - \rho$. Leveraging such an oracle, our focus shifts to methods for full secret key recovery. Importantly, our key recovery technique is not limited to the specific types of side-channel leakages involved.

It is important to clarify that the above framework outlines our generic threat model. For any specific attack scenario, a more detailed threat model must be articulated, as the adversary’s methodology for constructing the PC oracle could vary substantially depending on the concrete attack. For instance, in Section 4.2.1, we elaborate on the threat model pertinent to our newly identified timing attacks originating from division timing side-channels, where both the adversary and the victim need to run in the SMT setting.

3.2 Relevant Attacks from Generic PC Oracle

We briefly review three relevant attacks, each discussed in the following references: [GHJ+22], [HSC+23], and [GNNJ23].

3.2.1 The Key-Recovery Attack from [GHJ+22]

In [GHJ+22], the attacker purposefully sets $r_1$ to 1 (the multiplicative identity of $R$) while ensuring both $r_2$ and $e$ are set to 0 (the zero vector). As a result, the foundational
ciphertext is determined as $C_0 = (u, v) = (1, m\mathbf{G})$. An analysis of the PKE.Decrypt procedure reveals that $v + uy = m\mathbf{G} + y$. This finding suggests that the error the decoder must address during decryption corresponds to $y$, which forms the latter half of the secret key.

Let us clarify that in our descriptions, all attack strategies operate under a common assumption. They all presume that the decryption outcome hinges on a single inner block decoding of the RM code. In other words, the attacks have already introduced enough errors to push the outer RS decoding to its boundary for correct decoding.

The attacker's strategy involves introducing additional errors in a targeted inner RM block that push the RM decoder towards its boundary limits. If even a single bit change alters the decryption result—without loss of generality, from a successful to a failed state—it strongly suggests that the introduced change has increased the weight of the error. That is, a bit in the error vector may have been flipped from 0 to 1. Once the decoding boundary has been reached and a decoding failure occurs, they flip each bit to test if it results in a decoding success again. If the latter happens, the bit was an error-bit with high probability. Since all added vectors are recorded, this enables a highly probable prediction of a specific bit in the secret $y$. The strategy comes at the cost of high online query complexity, as each test of the decryption status requires a separate call to the oracle or the PKE.Decrypt procedure with the secret key.

### 3.2.2 The Two-Phase Attack from [HSC+23]

In [HSC+23], Huang et al. introduced a two-phase key-recovery attack that employs the PC oracle. The attack comprises both online and offline stages. During the online stage, the attacker's objective is to identify a specially crafted, invalid ciphertext $c = (u, v) = (1, v)$ such that adding the secret key's $y$ results in a decoding failure: $C.\text{Decode}(v + y) \neq C.\text{Decode}(v)$. The attacker repeated a certain number of attempts to find such a ciphertext. After multiple attempts to locate such a ciphertext, the attacker, if successful, engages in educated guessing to discern if a weight-1 pattern explains the decoding behavior. To enhance the likelihood of identifying the correct pattern, the attacker can repeat this procedure. If numerous attempts yield no such ciphertext $c$, the conclusion is that the inner block has a weight of 0, signifying an all-zero vector block.

Similarly, the attacker can employ this method to recover the inner blocks of $x$ with weight bounded by 1. Due the sparsity of the key, close to (or more than) 50% of the secret positions are recovered. With the public key relation that $x + h \cdot y = s$, the full secret key can be recovered by Gaussian elimination or slight post-processing such as information set decoding.

The primary advantage of this method over the key recovery method from [GHJ+22] is that when performing educated guess for the value of one inner block of $y$ (or $x$), only the public decoder of the employed RS RM codes is used, which can be done offline without interaction with the oracle. Thus, the online query complexity can be much lower.

### 3.2.3 The SCA-LDPC Framework [GNNJ23]

In [GNNJ23], the authors introduce SCA-LDPC, a generic framework grounded in coding theory for key-recovery chosen-ciphertext side-channel attacks on lattice-based and code-based KEMs using a PC-like oracle. When applied to HQC, this framework significantly reduces query complexity compared to the attacks outlined in [GHJ+22] and [HSC+23].

In contrast to the other attacks, the approach in [GNNJ23] generates a low-Hamming-weight vector $h_l$ and sets $r_1 = h_l$, while keeping $r_2$ and $e$ as zero vectors. Consequently, the decryption function takes $v + uy = m\mathbf{G} + h_l \cdot y$ as its input. Unlike the method in [GHJ+22], which inefficiently recovers only a single $y$-vector entry per PC oracle call, the SCA-LDPC attack yields more valuable information. This is because each entry in $h_l \cdot y$
We next introduce a novel method that tests whether a block of the secret key has weight zero. To implement this idea we find a sequence of errors induced by the secret key. Since the secret key is sparse the weight of the error is unsatisfiable since the additional errors could unflip error bits set by the secret key. This shifting idea was already introduced in [HSC+23]; however, it is made viable through our new zero-testers and we fully develop and implement the idea.

3.3 New Method: Zero Testers with Shifting

We next introduce a novel method that tests whether \( n_2 \) consecutive bits have weight zero with greater query efficiency. While this approach aligns with the attack described in [HSC+23], it diverges in its methods and objectives. Rather than identifying patterns with a bounded weight of 1, our method focuses on efficiently determining blocks with weight zero.

**Definition 1** (Block). A block is a collection of \( n_2 \) consecutive bits. Typically blocks are a slice of an element of \( \mathcal{R} \). The block may wrap around at the edges — e.g. the last bit, and the first \( n_2 - 1 \) bits of \( x \in \mathcal{R} \) may form a block.

**Definition 2** (Zero-Block). A zero-block is a block that has zero Hamming weight, i.e. all bits are zero.

The zero-blocks we identify may be at any offset in parts of the secret key’s \( x \) and \( y \). We are not limited by the RM code’s codeword offsets, as we can shift the secret key’s components using \( u \) and \( r_2 \). Specifically, we can craft ciphertext such that the victim’s decoder will correct an error corresponding to shifted versions of \( x \) or \( y \).

To construct a ciphertext that contains \( y \) shifted by \( k \) positions as its error, we can set \( r_2 = 0 \) and \( u = X^k \) and compute \( v \) honestly as per the scheme by \( v = mG + s \cdot r_2 + e \). During decapsulation, this will result in \( C.\text{Decode}(v - u \cdot y) \), where \( v - u \cdot y = mG + s \cdot r_2 + e - X^k \cdot y = mG + e - X^k \cdot y \). The error \( e \) is then used by the zero-tester method to ascertain information about the first block of this shifted version of \( y \). Analogously, for \( x \) one may set \( r_2 = X^k \) and \( u = X^k \cdot h \), which results in the computation of \( v - u \cdot y = mG + s \cdot X^k + e - X^k \cdot h \cdot y = mG + x \cdot X^k + e \) by \( s = x + h \cdot y \). Both cases result in shifted versions of the secret key to become the error that the victim’s decoder needs to correct. This shifting idea was already introduced in [HSC+23]; however, it is made viable through our new zero-testers and we fully develop and implement the idea.

A zero-tester can be used to identify whether a block has weight zero. Zero-testers are errors we add to the ciphertext, such that a decoding failure indicates whether the tested block in the original error of ciphertext can be a zero-block or not — i.e. it indicates whether a block of the secret key has weight zero. To implement this idea we find a sequence of errors (testers) that can be added to the ciphertext’s \( y \). The errors \( e_i \) are chosen such that adding one or more error bits to them results in a RM decoding failure. This allows efficient testing of whether a block of \( n_2 \)-bits of the secret key is zero. Using these testers we can perform the attack. The requirement for a single error to detect all possible errors is unsatisfiable since the additional errors could unflip error bits set by the error \( e \). A more reasonable requirement is that each tester should detect a large subset of possible errors induced by the secret key. Since the secret key is sparse the weight of the \( n_2 \)-bit blocks is typically \( \leq 5 \). Ideally, the union of the sets of secret key errors that the testers detect makes up the vast majority of possible errors that we are likely to encounter. Furthermore, we would like to find the shortest sequence of testers that allows us to detect the whole set of possible errors.
The number of oracle calls our attack requires is highly sensitive to the number of tests required per block. In contrast to other methods, we do not need to search for set-bits - with our zero-testers we can recover \( n_2 \) bits with very few queries. Each test requires a single oracle call. For zero-blocks all tests (3 for HQC-128) must be performed to confirm that it does indeed contain no set bits. Since the majority of blocks have a weight larger than 0, we can filter these out early. Only approx. 40% of blocks pass the first stage of the zero-tester in HQC-128. This means for the majority of blocks only a single oracle call needs to be performed. The second stage again only lets 31% of the remaining blocks pass. Thus, even though we use 3 testers for HQC-128, only 1 oracle call is performed per shift for the majority of cases.

The full attack then consists of shifting the respective part of the secret key into a position and performing zero-tests on the first block of the ciphertext. The core procedure of our attack is represented as pseudocode in Figure 2.

### 3.3.1 Finding Suitable Zero-Testers

Our method of finding a sequence of testers is akin to a mutation fuzzer. We start with random testers of a weight \( w \in \{\lfloor n_2 \cdot 0.4 \rfloor, \ldots, \lfloor n_2 \cdot 0.5 \rfloor \} \). Then we perform many mutations on these testers and select the mutation that performs the best. Experimentally we found that mutations that flip 1 or 2 bits yield the best results.

Our evaluation criterion is also probabilistic: we sample a given number of low-weight errors according to the probability distribution of their occurrence in the secret key. This probability distribution can be approximated well using the coefficients of the polynomial given by \( f \):

\[
f(x) = ((1 - p) + p \cdot x)^{n_2}
\]

where \( p = \frac{\omega}{n} \). The \( j \)th coefficient of \( f \) yields the probability that \( j \) bits are set in an array of \( n_2 \) bits where each bit is i.i.d. with probability \( p \) of being set.

After applying a mutation we estimate the success probability of an attack as the probability that the sequence of testers will not fail for the number of queries required by the attack (e.g. 1000). If this estimated attack success probability is greater than 99% we terminate.

If we see no improvement in some iterations of the mutation finding process, we restart the process entirely with newly chosen random errors. This prevents remaining stuck in local optima.

### 3.4 Selecting Suitable Shifts

During the attack, we shift the secret key to test the weight of different blocks of the key. This raises the question of which of all possible \( n \) shifts should be tested first to increase the efficiency of the attack.

#### 3.4.1 Static Shift Selection

A simple approach is to use a fixed sequence of shifts every time we perform the attack. For HQC-128 we used the shifts 0, 192, 92, 297, 238, and 115 plus an offset of \( b n_2 \) for the current block \( b \in \{0, \dots, n_1 - 1\} \). These shifts were chosen by greedily picking a shift that performed well in simulation, under the assumption that the previous shifts have already been used to test each block. If these shifts don’t suffice we pick random new shifts that have not been tested before.

#### 3.4.2 Dynamic Shift Selection

When performing the attack we obtain new information about the key with each query. Based on the queries performed and their observed responses, we can attempt to search for
a shift that will maximize the information gained. One such method could be derived from belief propagation, as the problem shares a similar structure. Once we have an algorithm for estimating the posterior conditional probability distribution of a zero-test for a shift $s$ given the query response pairs observed so far, we could combine it with game tree search algorithms, e.g. a variant of expectimax. This may yield further improved shifts. However, this would come at a significant computational cost, as the branching factor is $n \geq 17669$.

### 3.4.3 Recoverable Keys and Success Chance

Our attack depends on that there exist sufficiently many zero-blocks in the two parts of the secret key $x$ and $y$ such that we can recover slightly more than 50% of the secret key. This means there may be some keys that cannot be recovered by our attack, since some keys may not have enough zero-blocks. In the simulation we show that for HQC-128 and HQC-256 this is not an issue, and we can recover 98% and 99% of keys respectively, assuming we try all possible shifts exhaustively. For HQC-192 we can only recover approx. 15% of keys, but a large portion of the key can be recovered and be used to reduce the complexity of other attacks.

To obtain the full key in post-processing one can solve the linear equations given by the public key relation $s = \begin{bmatrix} I_n & \text{rot}(h) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$. In practice, the resulting $n \times n$ matrix over $\mathbb{F}_2$ is often not invertible. Thus, one needs to recover additional zero-bits until there is an invertible submatrix. We recover 5 additional bits, which yields a probability of approx. 97% that the $n \times (n-5)$ matrix contains an invertible $(n-5) \times (n-5)$ submatrix. This probability is computed as $p(n, m) = \prod_{i=1}^{m} (1 - 2^{-i-1})$ [Fer], where $m = n - 5$.

### 4 Division Timing Side-Channel

In [GHJ+22] the authors reveal a timing-side channel stemming from the use of variable-time constant-weight vector sampling. The constant-weight vector sampling is performed as part of the re-encryption step to sample the vectors $r_1$, $r_2$, and $e$. The re-encryption step is part of the decapsulation, as shown in Figure 3. The identified vulnerability was that the sampling process used rejection sampling, which is non-constant-time in the used randomness. The vector sampling process must be constant-time in the used randomness because the randomness depends on the seed $\theta$, which is derived from secret inputs (the message). Leaking any timing information about $\theta$ leads to a distinguisher which can be used as a PC oracle. Such a PC oracle completely breaks the cryptosystem leading to full key-recovery.

To remedy the timing side-channel the authors of the scheme implemented a constant-time constant-weight vector sampling algorithm described in [Sen21]. This algorithm requires modulo reductions of random numbers, which were derived from the seed $\theta$. Unfortunately, the compiler (gcc v13.1.1) uses division instructions to implement these modulo reductions. Division instructions typically take a variable number of cycles on a CPU, depending on the size of the operands. (cf. Section 4.1). The vulnerable code snippet may be found in Figure 4. We identified the vulnerability through manual analysis of the code’s disassembly.

In principle modern compilers have the ability to convert modulo reductions modulo a compile-time known constant into constant-time code using Barrett reductions. This optimization typically occurs when there are single modulo or division operations, that are not within a loop. However, in this case, the compiler does not do so, even when using the -funroll-all-loops flag. This flag causes the loop to be unrolled, but the compiler still generates division instructions instead of Barrett reductions.
Algorithm 1: PC-Oracle Key Recovery Attack

Data: $h$ from the public key, a suitable message $m$ and salt, shifts, zero testers, and scheme parameters $n, n_2, \delta$

Result: bit_status array: zero-bits in the secret key

1. $\text{bit\_status}_{i,j} \leftarrow \text{"Unknown" } \forall i \in \{\text{"X"}, \text{"Y"}\}, j \in \{0, \ldots, n-1\}$

2. for $k \in \text{shifts}$ do
   3. if $\bigwedge_{j \in \{s-n_2+1, \ldots, s\}} \text{bit\_status}_{\text{key\_part}, j} = \text{"KnownZero"}$ then
      4. // we already know each bit in this block to be zero
      5. continue
   6. end
   7. classification $\leftarrow \text{"ZeroBlock"}$
   8. for tester $\in \text{zero\_testers}$ do
      9. set error $e$ with $\delta = \left\lfloor \frac{dn_s}{2} \right\rfloor$ RM-code blocks flipped, such that the corruption of the first block will cause an RS decoder failure.
   10. set the first block of $e$ to the used tester
   11. if key\_part = “X” then
      12. $r_2 \leftarrow X^k$
      13. $u \leftarrow X^k \cdot h$
   14. else if key\_part = “Y” then
      15. $r_2 \leftarrow 0$
      16. $u \leftarrow X^k$
   17. end
   18. construct ciphertext $c$ as $(u, mG + s \cdot r_2 + e, \text{salt})$
   19. if $\neg O_{HQC}(c)$ then
      20. classification $\leftarrow \text{"NonZeroBlock"}$
      21. break
   22. end
   23. end
   24. if classification = “ZeroBlock” then
      25. for $j \in \{s-n_2+1, \ldots, s\}$ do
         26. $\text{bit\_status}_{\text{key\_part}, j} \leftarrow \text{"KnownZero"}$
      27. end
   28. end
   29. if $\left( \sum_{i\in \{X,Y\}} \sum_{j\in\{0,\ldots,n-1\}} \text{bit\_status}_{i,j} = \text{"KnownZero"} \right) \geq n + 5$ then
      30. return bit\_status
   31. end
32. return bit\_status

Figure 2: The core of our attack procedure on HQC represented as pseudocode. For brevity we assume that array accesses into bit\_status wrap around modulo $n$. 
Figure 3: Simplified illustration of HQC decapsulation. Parts with previously identified timing side-channel vulnerabilities are marked in red. Our identified side-channel is in the vector-sampling part of the re-encryption, similar to [GHJ+22]. It was introduced in the patch to [GHJ+22].

[1] [WTBB+19, PT19]; [2] [GJN20]; [3] [GHJ+22].

\[
\text{for (size_t } i = 0; i < \text{weight}; ++i)\
\quad \text{tmp}[i] = i + \text{rand}_u32[i] \mod (\text{PARAM}_N - i);
\]

Figure 4: Vulnerable code snippet generating div instructions.

The timing variation caused by the divisions may be exacerbated if the target system is embedded, and has worse division performance than high-performance x86 microarchitectures. For our analyses and exploitation, we focus on high-performance x86_64 systems, as the vulnerable optimized HQC implementation is written for processors supporting the AVX2 vector instruction-set extension.

4.1 Analyzing Numerator Dependent Division Throughput

To characterize the side-channel leakage that we want to exploit we perform experiments regarding the division throughput on different x86_64 processor microarchitectures.

We measure the throughput of an instruction sequence containing a division using nanoBench[AR20]. The measured instruction sequence is shown in Figure 6.

The instruction sequence was measured on Zen3 (AMD Ryzen 9 5900X), Zen2 (AMD Ryzen 9 3900X), Zen+ (AMD Ryzen 7 3700U) and Intel 8th Gen (Intel i7-8550U). The measured throughput includes not only the division itself but also the preceding instructions. On AMD architectures the \texttt{076.00 LsNotHaltedCyc} performance counter was used. On Intel we used \texttt{3C.00 CORE_CYCLES}. The benchmark results can be reproduced using the code artifact.

The AMD and Intel optimization manuals are relatively sparse on details regarding latency and throughput of divisions. The AMD Optimization manual for the Family 19h line of processors (Zen3, Zen3+, and Zen4) states that: “The hardware integer divider unit has a typical latency of 8 cycles plus 1 cycle for every 9 bits of quotient. The divider allows limited overlap between two consecutive independent divide operations. ‘Typical’ 64-bit divides allow a throughput of one divide per 8 cycles (where the actual throughput is data dependent)”. For the Family 17h (Zen, Zen+, and Zen2) the optimization manual states: “The radix-4 hardware integer divider unit can compute 2 bits of results per cycle”.

To exploit the variable division runtime we target a Zen2 machine. We choose Zen2 due to the variable division timing even for 32-bit numerators.

Using just timing information on the vector sampling part of the HQC decapsulation on an otherwise idle machine, we are able to distinguish a fast vector sampling from a random one. We obtain an accuracy of up to 80% (both classes represented equally). However, when measuring the execution time of the entire decapsulation, the timing signal becomes too weak for practical exploitation. Even 100000 timing measurements do not suffice to gain any significant distinguishing advantage.
Figure 5: Empirical results of numerator dependent throughput of divisions on different microarchitectures for a fixed divisor (17669). Some high-performance x86 microarchitectures can perform 32-bit divisions by the chosen fixed divisor with numerator-independent throughput, while Zen2 and Zen+ have varying division throughput.

```assembly
mov r10, 17669 ; divisor
mov rax, numerator
xor rdx, rdx
idiv r10
```

Figure 6: Measured instruction sequence, where numerator is replaced with a concrete numerator.
4.2 DIV-SMT

We introduce DIV-SMT, which enables us to precisely measure only the contribution of the \texttt{div} instructions to the timing signal, and filter out noise caused by the other parts of the cryptosystem - less than 2\% of the decapsulation runtime are dedicated to the divisions relevant to our attack. Like prior works [ABuH+18, GJS+23], we use contention among SMT sibling threads to detect the execution of a certain instruction (\texttt{div}). In addition and in contrast to these works, we also use contention to infer information about the processed operands (the number of leading zeros in the numerators).

4.2.1 Threat Model

DIV-SMT is relevant to all contexts where the attacker can execute code co-located to the victim. Crucially, the CPU must support SMT and have it enabled. Disabling SMT comes with a typically high but workload-dependent cost [RPNT05, Lar18]. Especially in systems with high SMT thread count like Power10, which sports up to 8 SMT threads per core it is very desirable to leave SMT enabled. Further, the attacker must not be impeded by scheduling countermeasures such as Linux’s Core Scheduling [lin]. Core Scheduling may prevent the attacker from running on the same core as the victim, but also comes with a performance cost. In certain cases, the performance may be worse than with SMT disabled due to the additional scheduling overhead [Fag19].

4.2.2 Side-Channel Measurement

The setup for DIV-SMT is as follows: the attacker runs a thread pinned to the sibling SMT thread of the victim’s thread. For our attack simulation, we also pin the victim’s thread. In more restricted scenarios, the attacker may not have permissions to pin other user’s processes. To counteract this, the attacker may start many DIV-SMT receivers [GJM+24]. If possible, these receivers may be pinned to a different physical core to reduce measurement noise.

The attacker executes division instructions in a loop, measures the cycle counter and stores the counter for measurement in an array. For the cycle counter measurement, we use the \texttt{rdpru} instruction. The \texttt{rdpru} instruction allows us to read the Actual Performance Frequency Clock Counter (APERF) performance-counter from user-space which counts the number of cycles executed by the specific core the executing thread is running on [LGS22]. This is opposed to the \texttt{rdtsc} instruction which reads a global core-frequency-independent timestamp counter. Further, in our experiments, \texttt{rdpru} executes faster than \texttt{rdtsc} which enables higher temporal resolution in our SMT side-channel traces. An excerpt from the attacking loop code may be found in Figure 7.

When the victim is executing divisions, the execution unit for divisions is busy, and the attacker’s divisions are also slowed down. Slowing the attacker’s divisions down causes an increase in the cycle delta between measurements for the attacker. When the victim does not use the execution unit for divisions, the attacker’s divisions complete at a faster rate enabling us to filter out parts of the cryptosystem that do not use divisions and are therefore irrelevant to our attack. We use an empirically determined constant threshold to classify whether the sibling SMT thread is performing divisions. We then sum up these division timings to obtain a total division runtime for a single trace. After collecting 100 traces and computing each trace’s total division runtime, we compute the median total division runtime. This division runtime is then thresholded to form a classifier.

In principle, we use the contention timing side-channel for two distinct purposes. Similar to e.g. PortSmash [ABuH+18] and SQUIP [GJS+23], we use the contention side-channel to detect the execution of division instructions. Additionally, and unlike them, we use the side-channel to infer information about the processed operands, an aspect of SMT contention attacks, which, to the best of our knowledge, has not been exploited before.
fn rdpru32() -> u32 {
    let lo;
    unsafe {
        asm!("rdpru",
            out("eax") lo,
            out("edx") _,
            in("rcx") 1,
        );
    }
    lo
}

let mut i = 0;
while !done_decaps.load(Ordering::SeqCst) && i < len {
    unsafe { *measurements.get_unchecked_mut(i) = rdpru32() }
    let mut _dividend = 1;
    let mut _remainder: u64 = 0;
    let divisor = 1;
    unsafe {
        asm!(
            ".rept 5",
            "xor edx, edx",
            "div rcx",
            ".endr",
            inout("rax") _dividend,
            inout("rdx") _remainder,
            in("rcx") divisor,
        );
    }
    i += 1;
}

Figure 7: Excerpt from the attacking code showing the measurement loop and use of the rdpru instruction. In the loop body the APERF performance-counter is stored into the measurement array and 5 divisions are executed. The loop continues until either the maximum trace length is reached (in which case the trace is discarded), or the decapsulation has finished.

4.2.3 Practical Exploitation in HQC

For our HQC attack, we compute a threshold using ciphertexts where it is known whether a decoding failure occurs. Such ciphertexts are easy to construct by setting the error to a large or small value. The classifier then determines whether the total division runtime is faster than a random message or not. If it is faster, we know that decoding succeeded.

With the SMT based side-channel, we obtain above 90% accuracy with 100 traces. To further increase the accuracy, we measure the true-positive-rate and true-negative-rate of the classifier, and make a decision based on multiple classifications by computing the probability of a positive or negative class, given the observed classifications. A majority vote could also be used and would likely perform similarly.

Finding suitable zero testers is the most costly part of the attack, which is why we would like to only perform it once. By fixing a single message first we can achieve that goal, since the zero testers depend solely on the first RM codeword.

In the round 4 submission of HQC the public key and salt are added as inputs to derive the seed $\theta$, which in turn determines the generated numerators. The addition of the salt allows us to brute-force a salt, that, together with the fixed message and a given public key, generates suitable numerators. This process is also illustrated in Figure 8. In general, we want to find a salt that minimizes the division runtime of the generated numerators as this will increase the distinguishing capability. In our experiments we brute-force salts for
Figure 8: We first choose an arbitrary message for which we intend to find suitable zero testers. Once we’ve found suitable zero testers for this message we can attack any public key by varying the salt. By changing the salt we change which numerators are generated. These are used during the re-encryption step and leak timing information about the decoded message. We need to distinguish our chosen numerators from random different ones. Therefore, we want to find a salt that generates fast divisions with small numerators.

Figure 9: Oracle call requirements for attacking HQC-128 under various oracle accuracies, with whiskers representing the 5th and 95th percentile. We performed 1000 simulated attacks per oracle accuracy.

5 Experimental Results

In this section, we present the experimental results of our research, organized into two distinct subsections. We begin by showing the simulation results, which illustrate the enhanced performance of our method across a range of oracle accuracies. Subsequently, we present real-world attack results on HQC-128 executed on an AMD Zen2 platform, demonstrating its practical effectiveness and confirming that our simulation findings align with real-world applications.

5.1 Attack Simulation

To provide evidence that our new attack strategy works, we perform attacks using a simulated side-channel oracle. The simulated oracle obtains the true information that is to be leaked and adds oracle accuracy dependent noise to it. The true information in our case is whether the decoder outputs the original message or not. We simulate the attack in
Table 3: Comparison of median oracle calls required in our attack vs. SCA-LDPC. For our attack we additionally show the success rate. Attacks are classified as successful if they managed to recover $n + 5$ zero bits from the secret key and no bits were incorrectly identified as a zero bit. These $n + 5$ zero bits are sufficient in approx. 97% of cases as discussed in Section 3.4.3.

<table>
<thead>
<tr>
<th></th>
<th>$O_{HQC}^{\text{perfect}}$</th>
<th>$O_{HQC}^{\text{ideal}}$</th>
<th>$O_{HQC}^{0.995}$</th>
<th>$O_{HQC}^{0.95}$</th>
<th>$O_{HQC}^{0.9}$</th>
<th>$O_{HQC}^{0.515}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCA-LDPC\cite{GNNJ23}</td>
<td>9000</td>
<td>10000</td>
<td>18000</td>
<td>35250</td>
<td>59500</td>
<td>n/a</td>
</tr>
<tr>
<td>Our Work</td>
<td>1094</td>
<td>1142</td>
<td>2246</td>
<td>4922</td>
<td>6951</td>
<td>5,728,728</td>
</tr>
<tr>
<td>Success Rate</td>
<td>78.00%</td>
<td>77.30%</td>
<td>76.90%</td>
<td>77.10%</td>
<td>76.60%</td>
<td>77.13%</td>
</tr>
</tbody>
</table>

6 different scenarios: perfect, ideal, 0.995, 0.95, 0.9 and 0.515, denoted by $O_{HQC}^{\text{perfect}}$, $O_{HQC}^{\text{ideal}}$, $O_{HQC}^{0.995}$, $O_{HQC}^{0.95}$, $O_{HQC}^{0.9}$ and $O_{HQC}^{0.515}$, respectively.

The perfect side-channel scenario is unobtainable in practice: this oracle simply reveals the true information—whether the decoder outputs the original message or not. Nonetheless, this concept plays a crucial role in the misuse or key-mismatch attack scenarios \cite{BDH19}.

The ideal scenario is relevant to the round 3 implementation of HQC—it is similar to the perfect oracle, except that a message-dependent deterministic failure rate of 0.0058 for decoding failures is simulated. Specifically, when the decoder outputs a message $m'$ different from the original message $m$ there’s a chance of 0.0058 that the ideal oracle will output that $m = m'$, even though they are different. This simulates the case where the message $m'$ has the same timing behavior as the message $m$ and is therefore indistinguishable. We use this oracle for the sake of comparison to previous works, as in its current form it no longer applies to the HQC round 4 implementation, since the timing vulnerability discovered in \cite{GHJ22} has been patched.

The other oracles build on the ideal oracle and add independently sampled random noise: for an accuracy level of $\rho = 0.995$ there’s a $1 - \rho = 0.005$ chance that the result will be flipped.

In Figure 9, we show the number of oracle calls the attack requires to complete an attack under different noise conditions. The oracle here is simulated and uses a random number generator to simulate the random failures during a real-world side-channel attack. It can be seen that the distribution has a long tail, which consists of attacks that fail because they cannot find a sufficient number of zero-blocks.

In Table 3, we compare the median number of oracle calls required by our attack versus the SCA-LDPC framework. Our findings indicate a substantial improvement with the new Zero Tester method, which necessitates only 11% to 13% of the oracle calls compared to the SCA-LDPC framework. Notably, this ratio of efficiency gains remains consistent even as the oracle’s accuracy decreases to 0.9.

We simulate the performance of the PC oracle at the accuracy of 0.515. As noted in \cite{UXT22}, this specific accuracy level corresponds to the effectiveness of a Neural Network model in constructing a PC oracle from a masked hardware implementation. For such an oracle $O_{HQC}^{0.515}$, estimating the number of oracle calls required by the SCA-LDPC framework is resource-intensive, a task not undertaken in \cite{GNNJ23}. In contrast, our simulations demonstrate a median requirement of only 5,728,728 oracle calls.

If desired one may reduce the number of oracle calls further by trading it for computation time. Although the trade-off comes with quickly diminishing returns as Information-Set Decoding (ISD) \cite{Pra62, Ste89} comes with high costs. In Table 3, we simulate a success probability ranging between 76% and 78%. Enhancing the success rate is achievable through employing more resource-intensive ISD for post-processing, as opposed to Gaussian Elimination.
5.2 Real-World Attacks in an AMD Zen2 Platform

We performed 1000 attacks using our DIV-SMT side-channel oracle. The targeted machine is a Zen2 machine featuring an AMD Ryzen 7 3700X 8-Core Processor. The attacks ran concurrently on 8 cores, each using 2 SMT threads. 75.3% of the attacks succeeded in recovering $n + 5$ zero-bits from the secret key. This success rate comes close to the 76% to 78% success rate that we observe in simulation (cf. Table 3). A median of 465 409 DIV-SMT traces were used to form responses to a median of 2 577 oracle calls in a median attack runtime of 111 seconds. The number of traces includes a median number of 204 180 calibration traces. Using the calibrated threshold the SMT oracle was tested to have an accuracy of 98.8%.

Table 4: Approximation of the probability that a randomly sampled inner block in the secret vector has a certain Hamming weight [GLG22]. $P_i$ denotes the probability that the block has a Hamming weight of $i$.

<table>
<thead>
<tr>
<th>Variants</th>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_{\geq 5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HQC-128</td>
<td>23.44%</td>
<td>34.38%</td>
<td>24.83%</td>
<td>11.77%</td>
<td>4.12%</td>
<td>1.45%</td>
</tr>
<tr>
<td>HQC-192</td>
<td>16.50%</td>
<td>30.00%</td>
<td>27.00%</td>
<td>16.04%</td>
<td>7.07%</td>
<td>3.40%</td>
</tr>
<tr>
<td>HQC-256</td>
<td>23.14%</td>
<td>34.06%</td>
<td>24.87%</td>
<td>12.02%</td>
<td>4.32%</td>
<td>1.59%</td>
</tr>
</tbody>
</table>

6 Discussions

In this section, we discuss the discovered vulnerability and the attack methodology from various perspectives. We first address countermeasures against the DaS vulnerability in HQC. Subsequently, we compare our new key-recovery method to the state-of-the-art SCA-LDPC framework, elucidating the sources of performance improvement. Lastly, we explore the potential limitations of the new Zero Tester method.

6.1 Countermeasures

In addressing the DaS vulnerability within HQC, two main countermeasures emerge:

1. Manually code the necessary Barrett reductions.

2. Use multiplication and bit-shift as in Bit Flipping Key Encapsulation (BIKE)

The PQClean implementers, after our public disclosure [amb23], chose the first approach – it stays true to the specification of the scheme, but this solution comes with additional implementation effort.

BIKE replaces the modulo reduction with rounding; instead of $i + (s_i \mod n - i)$ they compute $i + \left\lfloor \frac{(n-i)s_i}{2^n} \right\rfloor$. The latter function can be implemented using an integer multiplication and a bit-shift. This function generates the same noticeable bias as the modulo reduction when compared to a uniform distribution. Note, that the bias is not a security issue [Sen21]. We advise to use the method used by BIKE as it is easier to implement (especially cross-platform), less error-prone, and most likely faster on most target architectures.

We advise a change in the specification in any case: implementers of the modulo reduction in the specification are likely to make the same mistakes unless additional guidance is given.
6.2 Comparison to SCA-LDPC

The SCA-LDPC framework is a generic framework that can apply not only to code-based schemes but also to lattice-based schemes with a larger alphabet and a more dense secret distribution. Our new PC key-recovery method is tied to HQC and exploits the extreme sparsity of the secret vector and the concatenated code construction used in HQC. The latter allows us to check whether blocks of consecutive bits in the secret key have weight 0, and due to the low weight of the secret key, this is often the case.

Moreover, in [GNNJ23], an information-theoretical lower bound is discussed through the calculation of Shannon’s entropy of the secret vectors $x$ or $y$. Utilizing LDPC codes, the SCA-LDPC framework achieves near-optimal performance. This is characterized by the number of necessary parity checks being only a small constant times greater than the theoretical lower bound as per Shannon’s theory. However, this information-theoretical bound does not extend to our new attack method. Our method exploits the public-key information $s = x + h \cdot y$. Given these additional equations, the secret key can be uniquely identified even without relying on side-channel leakages, assuming there are no constraints on the computational resources available.

6.3 Limitations

Last, we discuss the limitations of the new Zero Tester method and the potential for improvement. The efficacy of our approach is contingent upon finding long sequences of all-zero entries between two ones in the secret polynomials $x$ and $y$. The probability that an inner RM block within the secret vector is entirely zero is over 23% for HQC-128 and HQC-256, decreasing to 16.5% for HQC-192, as shown in Table 4. This reduction in likelihood explains the limited applicability of the new method to HQC-192, where it is effective for only about 15% of keys, as discussed in Section 3.4.3. For the remaining keys, the positions recovered are not adequate to achieve full-key recovery using Gaussian Elimination. Resorting to the more computationally demanding ISD algorithms for post-processing could enable the recovery of a greater number of keys. We defer a detailed quantitative analysis of this aspect to future research.

7 Conclusion

In this work, we have identified a novel timing side-channel vulnerability, an instance of the DaS vulnerability class, within the optimized, supposedly constant-time, reference implementation of the HQC round 4 submission to NIST. This vulnerability, characterized by its generic nature as demonstrated by similar issues in cryptographic implementations like KyberSlash, poses a significant security risk. Our proposed DIV-SMT methodology effectively targets this vulnerability on SMT-enabled processors by leveraging contention among SMT sibling threads. This process not only identifies specific instructions but also deduces operand information, thereby establishing a PC oracle distinguisher that enables a key-recovery timing attack.

Additionally, we have introduced a new key recovery approach for HQC leveraging the PC oracle, which significantly lowers the requirement for side-channel traces. Through comprehensive simulations using a simulated PC oracle at various levels of accuracy, coupled with practical attacks executed on an AMD Zen2 CPU, our research confirms both the effectiveness and practical viability of our novel attack strategy.
References


Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks


[GH24] Philippe Gaborit and HQC team. Technical update in the proof of hqc, 2024. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/sRZk_CAHUWU/m/oLX_2ns9BAI.


Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks


