KyberSlash: Exploiting secret-dependent division timings in Kyber implementations

Daniel J. Bernstein[1,2], Karthikeyan Bhargavan[3,4], Shivam Bhasin[5,7], Anupam Chattopadhyay[6,7], Tee Kiah Chia[7], Matthias J. Kannwischer[8], Franziskus Kiefer[4], Thales B. Paiva[9,10,11], Prasanna Ravi[6,7] and Goutam Tamvada[4]

[1] University of Illinois at Chicago, Chicago, IL 60607-7045, USA
[2] Academia Sinica, Taiwan
[3] Inria, Paris, France
[4] Cryspen, Berlin, Germany
[5] National Integrated Centre for Evaluation, Nanyang Technological University, Singapore
[6] College of Computing and Data Science, Nanyang Technological University, Singapore
[7] Temasek Labs, Nanyang Technological University, Singapore
[8] Quantum Safe Migration Center, Chelpis Quantum Tech, Taipei, Taiwan
[9] University of Sao Paulo, Brazil
[10] Fundep, Brazil
[11] CASNAV, Brazil
authorcontact-kyberslash@box.cr.yp.to

15 January 2025

Abstract. This paper presents KyberSlash1 and KyberSlash2 – two timing vulnerabilities in several implementations (including the official reference code) of the Kyber Post-Quantum Key Encapsulation Mechanism, recently standardized as ML-KEM. We demonstrate the exploitability of both KyberSlash1 and KyberSlash2 on two popular platforms: the Raspberry Pi 2 (Arm Cortex-A7) and the Arm Cortex-M4 microprocessor. Kyber secret keys are reliably recovered within minutes for KyberSlash2 and a few hours for KyberSlash1. We responsibly disclosed these vulnerabilities to maintainers of various libraries and they have swiftly been patched. We present two approaches for detecting and avoiding similar vulnerabilities. First, we patch the dynamic analysis tool Valgrind to allow detection of variable-time instructions operating on secret data, and apply it to more than 1000 implementations of cryptographic primitives in SUPERCOP. We report multiple findings. Second, we propose a more rigid approach to guarantee the absence of variable-time instructions in cryptographic software using formal methods.

Keywords: KyberSlash · PQC · Kyber · ML-KEM · Timing attacks · Division timing

1 Introduction

In 2016, the National Institute of Standards and Technology (NIST) launched a global standardization process for Public Key Encryption (PKE), Key Encapsulation Mechanisms (KEM), and Digital Signatures (DS) that can withstand quantum computer attacks, which is widely recognized under the umbrella term "Post-Quantum Cryptography." After years of evaluation, NIST announced the first set of four algorithms to be standardized in July 2022. Among these, one algorithm is selected for Public Key Encryption/Key Encapsulation Mechanisms (PKE/KEM) and three algorithms were selected for Digital Signatures. Kyber [5], a KEM based on the Module Learning With Errors (MLWE)

problem, is being standardized by NIST as ML-KEM in FIPS203 [33]. We expect to soon witness wide-scale adoption of ML-KEM across a wide-spectrum of computing devices, ranging from the high-end general purpose PCs to mobile phone processors all the way until computationally constrained embedded devices.

Since the announcement of the NIST standardization process, Kyber has garnered significant attention regarding its vulnerability to Side-Channel Attacks (SCA) [1]. This concern was a key focus during the NIST PQC standardization process, where the susceptibility of Kyber to SCA and appropriate protection mechanisms were studied by several reported works in literature [41, 48, 53]. Given its anticipated widespread adoption, the safety of Kyber implementations will be even more important in the future.

In this work, we report the discovery of multiple timing vulnerabilities in the official reference implementation of Kyber,[1] as well as several well-known open-source Kyber implementations. See Appendix A for details on the affected implementations. Notably, all these implementations are carefully designed to be constant-time at the level of source code, by avoiding secret dependent branches and memory accesses. However, we identify that compilers can introduce timing vulnerabilities through utilization of instructions that execute in variable-time. In particular, we discover these vulnerabilities being caused by certain subroutines that involve divisions by the Kyber prime $q = 3329$ (written as `/KYBER_Q` in the code).

1.1 Division variations

It is well known that CPU division instructions are slower for some inputs than for others. It has also been known for a long time that these timing variations might be exploitable; see, e.g., [29, "DIV instruction"] measuring timing variations in the context of fixing Lucky Thirteen.

But this general background does not mean that there is a problem with the divisions in the Kyber reference code. For those divisions, the numerator has a limited range, and the denominator is a compile-time constant (whereas in [29] the denominator was variable). The code is not written to use the CPU's division instruction, but rather to use divisions in the C programming language. One can reasonably guess that any modern compiler will optimize the division by a constant into a multiplication by a suitable constant; multiplication instructions are well known to be faster than division instructions.

A programmer can easily try an experiment to check this: write the division in C; compile it; check the resulting assembly to see that, yes, the compiler is using a multiplication instruction rather than a division instruction. A pleasant consequence of this automatic optimization is that the resulting binary is unaffected by any potential timing variation from division instructions. (There is still a problem on some embedded processors with variable-time multiplication instructions—see, e.g., [14] and [38]—but that problem is outside the scope of this paper.)

Unfortunately, the experiment described in the previous paragraph is not sufficiently systematic, and the guess stated above is an oversimplification. Common changes in compiler options can easily end up producing division instructions instead. For example, asking gcc to optimize for code size (`-Os`) generally disables the conversion of divisions into multiplications.[2] This creates a timing vulnerability.

We observe that exactly this vulnerability is triggered by the Kyber reference code. As the vulnerability is caused by the appearance of divisions in Kyber's C code (`/`), we name the vulnerability KyberSlash. We distinguish two forms of KyberSlash, named KyberSlash1

[1] https://github.com/pq-crystals/kyber

[2] Note that `-Os` is a very common compiler option, especially on embedded systems where code size is of particular concern. See, e.g., [23]. Also, `-Os` is just one example of the issue: for example, on 32-bit MIPS CPUs, gcc 14.1.0 from May 2024 produces division instructions even when it is optimizing for speed.

Table 1: Summary of our practical results exploiting KyberSlash1 and KyberSlash2.

Vulnerability	Processor	Kyber variant	Number of decapsulations	Successful key recoveries over number of tests
KyberSlash1	Cortex-A7	Kyber512	1,835,008	10/10
KyberSlash2	Cortex-M4	Kyber768	24,576	10/10
KyberSlash2	Cortex-A55	Kyber768	36,864,000	1/1

and KyberSlash2; these arise from different aspects of the cryptography inside Kyber, and turn out to open up very different exploitation mechanisms.

1.2 Contributions of this paper

First, this paper describes two variants of the KyberSlash vulnerability present in the November 2023 version of the Kyber reference implementation: KyberSlash1 is present in the decryption of the CPA-secure encryption scheme underlying Kyber and directly leaks information about the secret key. KyberSlash2 is present in the encryption of the CPA-secure encryption scheme and leaks information about the ciphertext. While this is unproblematic inside encapsulation as the ciphertext is public, it can be used to construct a plaintext-checking (PC) oracle in decapsulation allowing key recovery.

Second, this paper presents a practical demo showcasing the exploitability of KyberSlash1 on a Raspberry Pi 2 (Arm Cortex-A7). It crafts special ciphertexts and measures the time for decapsulation running on the same processor in a separate process. It successfully recovers a Kyber512 secret key in 10 out of 10 experiments within 2 to 4 hours.

Third, this paper demonstrates the exploitability of KyberSlash2 in a separate demo targeting the Arm Cortex-M4 microcontroller. We craft ciphertexts on a host and send the ciphertexts using serial communication to the target microcontroller which performs a decapsulation and reports back to the host when decapsulation is completed. When timing is performed on the target itself, the attack succeeds for Kyber768 within 4 minutes in 10 out of 10 experiments. Most of this time is spent on transmitting 6144 ciphertexts to the target device. We also demonstrate that the attack still works if timing is performed on a separate attacker device transmitting the ciphertexts to the target device. Consequently, KyberSlash2 is exploitable remotely[3] in certain cases. The performance of our attacks, together with the target devices and Kyber variants, is summarized in Table 1.

Fourth, this paper patches the dynamic analysis tool Valgrind [35] to allow detection of variable-time instructions operating on secret data extending Langley's ctgrind [28] methodology for detecting timing leaks. With the patched Valgrind, and with modified test programs, we are able to detect the vulnerable division operations in the November 2023 version of the Kyber code. We perform a large scale study with the patched Valgrind and apply it to more than 1000 implementations of various cryptographic primitives within SUPERCOP [9] and identify various potential vulnerabilities due to secret-dependent instruction timings.

Finally, this paper proposes a more rigid approach to guarantee the absence of variable-time instructions in cryptographic software by using formal verification.

Our code for the two demos as well as the Valgrind patches are available at https://kyberslash.cr.yp.to.

1.3 Related work

There is a long literature on side-channel attacks. Attacks often rely on access to physical sensors close to the targeted device; see, e.g., van Eck's 1985 paper [51] on electromagnetic leaks from monitors, or, as one of many recent examples, consider the EM probe in [43,

[3] Here we follow the traditional distinction between "local" attacks (prerequisite: attacker can run code on the victim device) and "remote" attacks (no such prerequisite). See, e.g., [27].

Section 5]. Modeling and protecting against these information leaks is difficult, with protections continually being broken (see, e.g., [47]) and with security seemingly relying on the hope that attackers are too far away to carry out attacks. Sometimes attacks exploit a lack of access control for physical sensors *built into* the targeted device, such as the power monitors exploited in [31].

Many other attacks rely purely on timing information. An early example is a timing attack recovering TENEX passwords; see, e.g., [12]. Within cryptography, broad awareness of the power of timing attacks began with Kocher's 1996 paper [26]. Specific sources of timing variation listed in [26] include "branching and conditional statements" (exploited in that paper), "RAM cache hits" (exploited in [8], [37], and [49]), and "processor instructions (such as multiplication and division) that run in non-fixed time", along with a general warning about "compiler optimizations" as a source of "unexpected timing variations". Many timing attacks exploit the fact that attackers are very often allowed to run computations on the target machine (see, e.g., [55]); there have also been some remote timing attacks relying on timing information naturally percolating through networks (see, e.g., [11]). Another avenue for timing attacks comes from the fact that many CPUs vary clock speeds depending on power consumption by default, creating a channel from power monitors to timing; see, e.g., [54] and [32].

By now there are many examples of side-channel attacks against post-quantum cryptography, including systems submitted to the NIST post-quantum competition. For example, [13] targeted non-constant-time error-correcting codes used in LAC, a lattice-based KEM; [19] targeted a non-constant-time ciphertext-comparison operation within FrodoKEM, another lattice-based KEM; and [36] targeted non-constant-time decoding in HQC, a code-based KEM.

Side-channel attacks against KEMs frequently work as follows. The attacker sends maliciously crafted ciphertexts to the decapsulation procedure, such that the decrypted message and its associated variables are related to a targeted portion of the secret key. The attacker uses side channels to obtain information about whether the message decrypted correctly. This reveals incremental information about the secret key, leading to full key recovery once there are enough ciphertexts. Our attack demos follow this pattern but exploit a different side channel, obtaining the first successful timing attacks against the reference implementation of Kyber.

2 Notation

For any prime q, we denote the field of integers modulo q as $\mathbb{Z}_q$. When n is a fixed positive integer, we let R_q denote the polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$. Then, R_q^k is the module of rank k whose scalars are polynomials in R_q. Polynomials $a \in R_q$ are denoted using lowercase letters. Vectors $\mathbf{a} \in R_q^k$ and matrices $\mathbf{A} \in R_q^{k \times k}$ are denoted in bold using lowercase and uppercase, respectively. When $\mathbf{u}, \mathbf{v} \in R_q^k$, we let $\langle \mathbf{u}, \mathbf{v} \rangle \in R_q$ denote their dot product. The ith entry of vector $\mathbf{a} \in R_q^k$ is denoted as $\mathbf{a}[i]$. Similarly, for a polynomial $a \in R_q$, we use $a[i]$ to denote its coefficient associated with the power x^i.

We denote by $\mathcal{B}_\eta$ the centered binomial distribution (CBD) with range $[-\eta, \eta]$. For a concise notation, we let $\mathbf{a} \leftarrow \mathcal{B}_\eta(R_q^k)$ mean that each coefficient from each polynomial of vector $\mathbf{a} \in R_q^k$ is drawn according to $\mathcal{B}_\eta$. Furthermore, we write $\mathbf{a} \leftarrow \mathcal{B}_\eta^r(R_q^k)$ to denote a derandomized sampling where the randomness comes from a string r. Furthermore, $y \leftarrow \mathrm{Compress}(x, d)$ denotes the lossy compression of x to d bits, where $d < \lceil \log_2 q \rceil$. The compression function is defined as $\mathrm{Compress}(x, d) = \left\lfloor \left(2^d/q\right) x \right\rceil \bmod 2^d$, where $\lfloor \cdot \rceil$ denotes the rounding function that rounds up on ties. The decompression is defined as $x' = \mathrm{Decompress}(y, d) = \left\lfloor \left(q/2^d\right) y \right\rceil$.

Table 2: Kyber parameters for each security level [45].

NIST security	Parameter set	k	η_1	η_2	$d_{\mathbf{u}}$	d_v	Failure probability
Level 1	Kyber512	2	3	2	10	4	2^{-139}
Level 3	Kyber768	3	2	2	10	4	2^{-165}
Level 5	Kyber1024	4	2	2	11	5	2^{-175}

3 Kyber

Kyber is a KEM designed for CCA security based on the Module-Learning With Errors problem (MLWE) [30, 45]. It offers parameter sets designed for NIST security levels 1, 3, and 5. For each security level, it uses fixed parameters $q = 3329$ and $n = 256$ that define the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, over which most of the operations are performed.

Given a desired security level, the setup takes public parameters $k, \eta_1, \eta_2, d_{\mathbf{u}}$, and d_v from Table 2. Parameter k defines the rank of the modules used in the scheme. Parameters η_1 and η_2 define the centered binomial distributions $\mathcal{B}_{\eta_1}$ and $\mathcal{B}_{\eta_2}$ used to generate coefficients with small norm in $\mathbb{Z}_q$. Integers $d_{\mathbf{u}}$ and d_v are the number of bits into which coefficients from the two parts of the ciphertext are compressed.

Kyber uses an encoding procedure that allows it to recover the message even after some noise accumulates during the encryption and decryption. It encodes a 256-bit message $\mathbf{m} \in \mathbb{Z}_2^{256}$ into a polynomial in R_q as $\text{Encode}(\mathbf{m}) = m_0 + m_1 x + \cdots + m_{n-1}x^{n-1} \in R_q$, where $m_i = \mathbf{m}[i]\lceil q/2 \rceil$. In other words, if bit $\mathbf{m}[i] = 0$ then $m_i = 0$, otherwise $m_i = \lceil q/2 \rceil$. A simple decoding procedure can then be applied to a polynomial m'. Namely, the decoding function outputs a 256-bit message $\mathbf{m}' = \text{Decode}(m')$ from a noisy polynomial m' by simply checking if each coefficient of m' is closer to 0 or to $q/2$, modulo q, and decoding it to 0 or 1, correspondingly.

Similar to most proposed post-quantum KEMs, Kyber is built in two layers. The bottom layer consists of a public-key encryption (PKE) scheme that is designed to be secure against passive adversaries, or, more precisely, against chosen-plaintext attacks (CPA). The top layer, which is a key encapsulation mechanism (KEM) designed to be secure against more powerful chosen-ciphertext attacks (CCA), is then constructed by applying a variation of the Fujisaki-Okamoto [18, 21] security conversion over the PKE scheme. Sections 3.1 and 3.2 provide the details of these two layers of algorithms.

3.1 Kyber's auxiliary PKE algorithms designed for CPA security

PKE schemes are defined by three algorithms: key generation, encryption and decryption. These are described by the corresponding procedures listed in Figure 1a. The key generation procedure is essentially a creation of an instance of the MLWE problem that protects the secret key. Similarly, the encryption procedure consists of generating another MLWE instance, which now protects the message $\mathbf{m}$ from being recovered from the ciphertext by someone who does not know the secret key $\mathbf{sk}$.

To see why decryption works, first notice that the ciphertext compression and decompression adds a small noise when going from $(\mathbf{u}, v)$ to $(\mathbf{u}', v')$. Now, by expanding $m' = v - \langle \mathbf{u}', \mathbf{s}' \rangle$, one obtains $m' = \text{Encode}(\mathbf{m}) + \langle \mathbf{e}, \mathbf{r} \rangle - \langle \mathbf{s}, \mathbf{e}_1 + \Delta\mathbf{u} \rangle + e_2 + \Delta v$, where $\Delta\mathbf{u} = \mathbf{u}' - \mathbf{u}$ and $\Delta v = v' - v$. That is, m' is the sum of the encoded message and a polynomial that is constructed from products and sums of elements whose coefficients came from centered binomial distributions, and are, therefore, small. Kyber security parameters are responsible for ensuring that the coefficients in Δm are small enough so that decryption errors occur only with negligible probability, as shown in Table 2.

(a) Kyber's PKE algorithms.

procedure PKE.KeyGen
- $\mathbf{A} \leftarrow$ random element from $R_q^{k \times k}$
- $\mathbf{s} \leftarrow \mathcal{B}_{\eta_1}\left(R_q^k\right)$
- $\mathbf{e} \leftarrow \mathcal{B}_{\eta_1}\left(R_q^k\right)$
- $\mathbf{t} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e}$
- $\mathbf{pk} \leftarrow (\mathbf{A}, \mathbf{t})$
- $\mathbf{sk} \leftarrow \mathbf{s}$
- **return** $(\mathbf{pk}, \mathbf{sk})$

procedure PKE.Encrypt($\mathbf{pk}, \mathbf{m}, r$)
- $\mathbf{A}, \mathbf{t} \leftarrow \mathbf{pk}$
- ▷PRF is used for sampling
- $\mathbf{r} \leftarrow \mathcal{B}_{\eta_1}^{\mathrm{PRF}(r,0)}\left(R_q^k\right)$
- $\mathbf{e}_1 \leftarrow \mathcal{B}_{\eta_2}^{\mathrm{PRF}(r,1)}\left(R_q^k\right)$
- $e_2 \leftarrow \mathcal{B}_{\eta_2}^{\mathrm{PRF}(r,2)}(R_q)$
- $\mathbf{u} \leftarrow \mathbf{A}^{\mathrm{T}}\mathbf{r} + \mathbf{e}_1$
- $v \leftarrow \mathrm{Encode}(\mathbf{m}) + \langle \mathbf{t}, \mathbf{r} \rangle + e_2$
- $\mathbf{c_u} \leftarrow \mathrm{Compress}(\mathbf{u}, d_{\mathbf{u}})$ ▷ KyberSlash2
- $c_v \leftarrow \mathrm{Compress}(v, d_v)$ ▷ KyberSlash2
- **return** $(\mathbf{c_u}, c_v)$

procedure PKE.Decrypt($\mathbf{sk}, (\mathbf{c_u}, c_v)$)
- $\mathbf{u}' \leftarrow \mathrm{Decompress}(\mathbf{c_u}, d_{\mathbf{u}})$
- $v' \leftarrow \mathrm{Decompress}(c_v, d_v)$
- $m' \leftarrow v - \langle \mathbf{u}', \mathbf{s}' \rangle$
- **return** $\mathrm{Decode}(m')$ ▷ KyberSlash1

(b) Kyber's KEM algorithms.

procedure KEM.KeyGen
- $\mathbf{pk}, \mathbf{sk}_{\mathrm{PKE}} \leftarrow$ PKE.KeyGen
- $z \leftarrow$ random 256-bit string
- $\mathbf{sk} \leftarrow (\mathbf{sk}_{\mathrm{PKE}}, \mathbf{pk}, z)$
- **return** $(\mathbf{pk}, \mathbf{sk})$

procedure KEM.Encaps($\mathbf{pk}$)
- $\mathbf{m} \leftarrow$ random 256-bit string
- $\bar{K}, r \leftarrow G(\mathbf{m}, H(\mathbf{pk}))$
- $\mathbf{c} \leftarrow$ PKE.Encrypt($\mathbf{pk}, \mathbf{m}, r$)
- $K \leftarrow \mathrm{KDF}\left(\bar{K}, H(\mathbf{c})\right)$
- **return** $(\mathbf{c}, K)$

procedure KEM.Decaps($\mathbf{sk}, \mathbf{c}$)
- $\mathbf{sk}_{\mathrm{PKE}}, \mathbf{pk}, z \leftarrow \mathbf{sk}$
- $\mathbf{m}' \leftarrow$ PKE.Decrypt($\mathbf{sk}_{\mathrm{PKE}}, \mathbf{c}$)
- $\bar{K}', r' \leftarrow G(\mathbf{m}', H(\mathbf{pk}))$
- $\mathbf{c}' \leftarrow$ PKE.Encrypt($\mathbf{pk}, \mathbf{m}', r'$)
- **if** $\mathbf{c} = \mathbf{c}'$ **then**
  - **return** $K \leftarrow \mathrm{KDF}\left(\bar{K}', H(\mathbf{c})\right)$
- **return** $K \leftarrow \mathrm{KDF}(z, H(\mathbf{c}))$

Figure 1: Kyber's algorithms with notes indicating where KyberSlash1 and KyberSlash2 appear.

3.2 Kyber's KEM algorithms designed for CCA security

The Kyber KEM is defined in Figure 1b. The scheme is constructed using an implicit rejection [21] variant of the Fujisaki-Okamoto [18] transform, which is designed to achieve CCA security by combining hash functions H and G with a PKE designed for CPA security. The KEM key generation is essentially the same as its PKE counterpart, except for the fact that a secret string z and the public key $\mathbf{pk}$ are packed into the secret key $\mathbf{sk}$, to allow for additional verification procedures. The KEM encapsulation takes only $\mathbf{pk}$ and returns a ciphertext $\mathbf{c}$ and a shared key K, that is computed using a key derivation function (KDF). The main objective of the encapsulation is to make the randomness used for encrypting message $\mathbf{m}$ depend on $\mathbf{m}$ itself by defining r based on $G(\mathbf{m}, H(\mathbf{pk}))$, and sampling the disposable values used for encryption using a cryptographic pseudorandom function PRF. This allows for a quick check, during decapsulation, to see whether a ciphertext $\mathbf{c}$ is valid.

Suppose $\mathbf{c}$ is a chosen ciphertext that was manipulated by the attacker. First, we decrypt $\mathbf{c}$ obtaining $\mathbf{m}'$, then we reencrypt $\mathbf{m}'$. Now, even if the ciphertext $\mathbf{c}$ can be successfully decrypted by the PKE algorithm, if we get a different ciphertext after reencrypting $\mathbf{m}'$ using randomness r' defined by $G(\mathbf{m}', H(\mathbf{pk}))$, then we consider $\mathbf{c}$ to an invalid ciphertext. Now, to avoid giving information to an attacker about the validity of the ciphertext they sent, a procedure called implicit rejection is used for deriving the shared key. If the ciphertext was considered valid, then we compute the shared secret based on K' that was derived from $\mathbf{m}'$. Otherwise, we build a fake shared secret applying the KDF to z and $\mathbf{c}$. Since the attacker does not know z, the fake shared secret K is indistinguishable from a valid one, thus not revealing additional information, and, since the output is deterministic, repeating the same challenge will result in exactly the same K.

```
1  void poly_tomsg(uint8_t msg[32], const poly *a)
2  {
3    unsigned int i,j;
4    uint16_t t;
5    for(i=0;i<KYBER_N/8;i++) {
6      msg[i] = 0;
7      for(j=0;j<8;j++) {
8        t  = a→coeffs[8*i+j];
9        t += ((int16_t)t >> 15) & KYBER_Q;
10       /* Division by Kyber Prime */
11       t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
12       msg[i] |= t << j;
13     }
14   }
15 }
```

Figure 2: C code snippet of message decoding operation, containing the vulnerable division operation by the Kyber prime `KYBER_Q`.

4 KyberSlash

We first start by briefly explaining the adversary model for our attack: The attacker attempts to recover the long-term secret key used by the target's decapsulation procedure of Kyber. We assume that the attacker has the ability to communicate with the target decapsulation procedure with chosen ciphertexts. This is a standard adversarial model used in several chosen ciphertext based side-channel attacks [10, 40, 50]. We assume that the attacker is able to observe the execution timing of the decapsulation procedure.

We identified two timing vulnerabilities, which we call KyberSlash1 and KyberSlash2, in implementations of division operations in Kyber. Sections 4.1 and 4.2 explain KyberSlash1 and KyberSlash2 respectively.

4.1 KyberSlash1: Leakage from message decoding

The first timing vulnerability is present within the message decoding operation within the decryption procedure (Line 24 in PKE.Decrypt procedure in Fig. 1a). This operation denoted as Decode (m'), essentially converts every coefficient of $m' \in R_q$ into corresponding bit of the decrypted message $\mathbf{m}' \in \mathbb{Z}_2^{256}$. This decoding operation for each coefficient of m' is computed as follows: $\mathbf{m}'[i] = (((m'[i] \ll 1) + \texttt{KYBER\_Q/2})/\texttt{KYBER\_Q}) \,\&\, 1$.

This operation should be implemented in constant time since the message polynomial $m' \in R_q$ is sensitive, and incremental information about m' for chosen ciphertexts can be used to recover the secret key $\mathbf{sk}$ [43, 44, 50]. Refer to Fig 2 for the C code snippet of the message decoding procedure, taken from the official reference implementation of Kyber. Notice that this operation contains a division by the Kyber prime (i.e. `KYBER_Q`) in Line 11 in Fig. 2. We have added highlighting to our figures to emphasize divisions with secret inputs. We compiled the code using `gcc 14.1` for the x86-64 architecture using the `-Os` compiler optimization flag, instructing `gcc` to optimize for code size. Part of the resulting assembly is shown in Fig. 3 and an `idiv` operation presenting a timing leak is highlighted in red (Line 8). Previous versions of `gcc` result in similar code containing `idiv` instructions. It is important to note that this behavior is not observed for compiler optimization flags `-O0,-O1,-O2,-O3`. We remark that similar behavior was observed for Arm Cortex-A55 (Snapdragon 888) and Arm Cortex-A72 (BCM2835). See Appendix D.

4.2 KyberSlash2: Leakage from ciphertext compression

The second timing vulnerability is present within the ciphertext compression operation within the encryption procedure (Line 17-18 in CPA.Encrypt procedure in Fig. 1a). The compression procedure essentially compresses each coefficient of the input $u \in R_q$ as follows: $\texttt{compress\_q}(u[i], d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d$.

```
1  ...
2  and ax, 3329
3  add eax, edx
4  movzx eax, ax
5  lea eax, [rax+1664+rax]
6  cdq
7  /* Variable-Time Division Instruction */
8  idiv r10d
9  and eax, 1
10 sal eax, cl
11 inc ecx
12 ...
```

Figure 3: Assembly code snippet of the message decoding operation for a single coefficient, when compiled with `gcc 14.1` for the x86-64 architecture using the `-Os` compiler optimization flag.

```
1  void poly_compress(uint8_t r[128], const poly *a)
2  {
3    unsigned int i,j; int16_t u; uint8_t t[8];
4    for(i=0;i<KYBER_N/8;i++) {
5      for(j=0;j<8;j++) {
6        u  = a→coeffs[8*i+j];
7        u += (u >> 15) & KYBER_Q;
8        /* Division by Kyber Prime */
9        t[j] = ((((uint16_t)u << 4) + KYBER_Q/2)/KYBER_Q) & 15;
10     }
11     r[0] = t[0] | (t[1] << 4);
12     r[1] = t[2] | (t[3] << 4);
13     r[2] = t[4] | (t[5] << 4);
14     r[3] = t[6] | (t[7] << 4);
15     r += 4;
16   }
17 }
```

Figure 4: C code snippet of the ciphertext compression operation, containing the vulnerable division operation by the Kyber prime `KYBER_Q`.

The ciphertext compression operation should also be implemented in constant time, as it leaks information about the recomputed ciphertext within the decapsulation procedure (Line 16 of KEM.Decaps procedure in Fig. 1b). The recomputed ciphertext is considered sensitive, and leaks information about the secret key for chosen ciphertexts. Refer to Fig. 4 for the C code snippet for ciphertext compression operation from the official reference implementation of Kyber. This operation contains a division by the Kyber prime `KYBER_Q`, similar to that of the message decoding procedure, that is highlighted in red (Line 9). Refer to Fig. 5 for the assembly code snippet of a single iteration of the message decoding operation when compiled with `gcc` and `-Os` where the `idiv` operation is highlighted in red (Line 7). We observe similar divisions appearing in other platforms when the `-Os` flag is used: on AArch64 using arm64 gcc 14.1.0, and on Arm Cortex-M4 CPU (`-mcpu=cortex-m4`) using arm-none-eabi-gcc 14.1.0. See Appendix E.

5 Exploiting KyberSlash1

This section describes how KyberSlash1 can be exploited. We illustrate the exploitability of the decapsulation-timing variations in the end-of-November-2023 official Kyber512 reference code running under Raspbian (`gcc` 8.3.0) on a Raspberry Pi 2 with a BCM2836 CPU, a quad-core Cortex-A7 running at 900MHz.

5.1 Attack methods

KyberSlash1 leakage is present in the decryption procedure of the PKE underlying Kyber.

```
...
movzx eax, ax
sal eax, 4
add eax, 1664
cdq
/* Variable-Time Division Instruction */
idiv r9d
and eax, 15
mov BYTE PTR [rsp-8+rsi], al
...
```

Figure 5: Assembly code snippet of a single iteration of ciphertext compression operation, when compiled with `gcc 14.1` for the x86-64 architecture using the `-Os` compiler optimization flag.

By carefully crafting ciphertexts, the leakage reveals information about the secret key directly. We first describe the timing behavior of divisions on our target platform and then how exploitable ciphertexts for this target platform can be selected.

5.1.1 Soft divisions

On this platform, `gcc -Os` converts each division into a call to a division subroutine `divsi3`. The CPU includes a hardware division instruction, but by default `gcc` compiles for an ABI that does not guarantee division instructions.

Checking the cost of the `divsi3` subroutine for divisions of n by 3329, for each n in the range of interest, shows that there is a jump by 20 cycles when the numerator n reaches 3329, a further jump by 2 cycles when n reaches 4096, and a further jump by 1 cycle when n reaches 8192.

5.1.2 Ciphertext selection

As noted above, this is a demo exploiting KyberSlash1, specifically the division in the line `t = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1` applied to each coefficient in the noisy message polynomial $m' = v' - \langle \mathbf{s}, \mathbf{u}' \rangle$, where $\mathbf{s}$ is the secret key and $(\mathbf{u}', v')$ is the decompressed ciphertext.

Each input coefficient t in m' turns into a division of $2t+1664$ by $q = 3329$. Consequently, there is a big jump in division cost on this platform when t reaches 833, and there are smaller jumps when t reaches 1216 and 3264. The demo chooses ciphertexts $(\mathbf{u}, v)$ so that these timings reveal the coefficients in $\mathbf{s}$, as explained in the following paragraphs.

A Kyber512 ciphertext $(\mathbf{u}, v)$ consists of three elements $\mathbf{u}[0], \mathbf{u}[1], v$ of $R_q = \mathbb{Z}_q[x]/(x^{256}+1)$, that is, 256-coefficient polynomials, with each coefficient being an integer modulo $q = 3329$. There are some constraints on the coefficients because ciphertext compression enforces rounding. The secret key $\mathbf{s}$ consists of two elements $\mathbf{s}[0], \mathbf{s}[1]$ of R_q, each coefficient being between -3 and 3. The polynomial m' mentioned above is $m' = v' - \mathbf{s}[0]\mathbf{u}'[0] - \mathbf{s}[1]\mathbf{u}'[1]$.

Consider what decryption does when $\mathbf{u}'[0] = 72x^{100}$, $\mathbf{u}'[1] = 0$, and $v = 2081 + 2081x + \cdots + 2081x^{254} + 208x^{255}$. Note that the final coefficient of v is 208, not 2081. All of the coefficients listed here are compatible with the constraints on compressed ciphertexts.

The coefficient of x^{255} in m' has the form $m'[255] = 208 - 72\mathbf{s}[0][155]$. If $\mathbf{s}[0][155]$ happens to be 3, then $m'[255] \bmod 3329$ is 3321, producing a slow division. Otherwise $m'[255]$ is between 64 and 424, producing a fast division. Other coefficients in m' are between 1865 and 2297, producing slow divisions with no obvious dependence upon secrets.

Consider an optimistic model of total decapsulation time as a constant plus the time for this division. Then $\mathbf{s}[0][155] = 3$ produces slow decapsulation, while other possibilities for $\mathbf{s}[0][155]$ produce fast decapsulation, so decapsulation timings immediately distinguish $\mathbf{s}[0][155] = 3$ from the other possibilities. Replacing 72 with -72 similarly distinguishes -3 from $-2, -1, 0, 1, 2, 3$; replacing 72 with 107, which is another allowed coefficient,

distinguishes $2, 3$ from $-3, -2, -1, 0, 1$; etc. Replacing $72x^{100}$ with $72x^{101}$ targets $\mathbf{s}[0][154]$ instead of $\mathbf{s}[0][155]$. Exchanging the roles of $\mathbf{u}'[0]$ and $\mathbf{u}'[1]$ targets $\mathbf{s}[1]$ instead of $\mathbf{s}[0]$.

Converting this into a complete attack demo was a conceptually straightforward matter of filtering out the noise that appears in real timings. The details are in Appendix C. Optimizing this demo turned out to be unimportant, since KyberSlash2 allows a more powerful attack approach, explained in Section 6.

5.2 Experimental results

The demo software consists of two files: a script `demo1-pi2.sh` and the main attack code `demo1-pi2.c`. Executing `sh demo1-pi2.sh; sh demo1-pi2.sh; sh demo1-pi2.sh` runs three attack experiments if both files are in the current directory. The script automatically downloads the target Kyber code, compiles it, and runs the demo. The script assumes that the packages `git`, `time`, and `build-essential` are installed.

A typical experiment takes a few hours. A successful experiment, i.e., an experiment recovering the full Kyber512 secret key, prints `yes, eve succeeded`. The demo was observed succeeding ten times in ten experiments, with run times of 2:13:53, 2:04:45, 3:32:45, 1:59:11, 3:23:30, 2:06:31, 2:34:05, 2:04:46, 3:16:21, 2:23:04.

The demo is not guaranteed to succeed: it gives up if it has not found the key from timings of $7 \cdot 2^{18}$ decapsulations. An earlier version of the demo, differing only in the mechanism used to check the computer's clock, succeeded only twice in three experiments.

6 Exploiting KyberSlash2

In this section, first we discuss how an attacker can recover the key using leakage from KyberSlash2, that occurs during ciphertext compression, then we present our experimental results when attacking real-world implementations.

6.1 Attack methods

Since KyberSlash2 appears during reencryption (as presented in Section 4), we can exploit the timing information to mount a plaintext-checking (PC) oracle attack [10,20,40,42,48,50]. In the following sections, we describe how to instantiate a robust PC-oracle capable of extracting key information under noisy settings.

6.1.1 PC-oracle attack using decapsulation time

Let us begin by defining how to build ciphertexts whose decapsulation timings allow the attacker to learn information on the secret key.

Chosen ciphertexts to extract key information. Each malicious ciphertext is defined by 5 parameters: a 256-bit message $\hat{\mathbf{m}} \in \mathbb{Z}_2^{256}$, followed by four integers $\hat{u}, \hat{v}, \hat{\imath}$, and $\hat{\jmath}$. To build the malicious ciphertext, first we compute $(\mathbf{u}, v)$ as

$$v = \text{Encode}\,(\hat{\mathbf{m}}) + \hat{v}, \text{ and } \mathbf{u}[i] = \begin{cases} -\hat{u}x^{(256-\hat{\jmath})}, & \text{if } i = \hat{\imath}, \\ 0, & \text{otherwise.} \end{cases}$$

Remember that polynomial operations in Kyber are done in $R_q = \mathbb{Z}_q[x]/\left(x^{256}+1\right)$. Then, the malicious ciphertext is the compression of $(\mathbf{u}, v)$, as defined by Kyber.

To avoid having to deal with additional noise from compression and decompression, it makes sense to choose attacking parameters $\hat{u}$ and $\hat{v}$ that are not significantly affected by

Table 3: The effect of $(\hat{u}, \hat{v})$ in the decoded messages for each possible secret coefficient, considering Kyber768 [42].

$\mathbf{s}[\hat{\imath}][\hat{\jmath}]$	Attack parameters $(\hat{u}, \hat{v})$			
	$(207, 937)$	$(2, 729)$	$(106, 521)$	$(106, -728)$
-2	$\hat{\mathbf{m}}_1$	$\hat{\mathbf{m}}_1$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$
-1	$\hat{\mathbf{m}}_1$	$\hat{\mathbf{m}}_1$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$
0	$\hat{\mathbf{m}}_1$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$
1	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$
2	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_0$	$\hat{\mathbf{m}}_1$

the lossy compression. This way, during decapsulation, the malicious ciphertext produces the following noisy message

$$m' = v - \langle \mathbf{s}, \mathbf{u} \rangle = \text{Encode}\,(\hat{\mathbf{m}}) + \hat{v} + \hat{u}x^{(256-\hat{\jmath})}\mathbf{s}[\hat{\imath}].$$

More specifically, each coefficient of the noisy message m' is defined as

$$m'[j] = \begin{cases} \hat{\mathbf{m}}[0]\lceil q/2 \rceil + \hat{v} - \hat{u}\mathbf{s}[\hat{\imath}][\hat{\jmath}], \text{ if } j = 0, \\ \hat{\mathbf{m}}[j]\lceil q/2 \rceil - \hat{u}\mathbf{s}[\hat{\imath}][j], \text{ otherwise.} \end{cases}$$

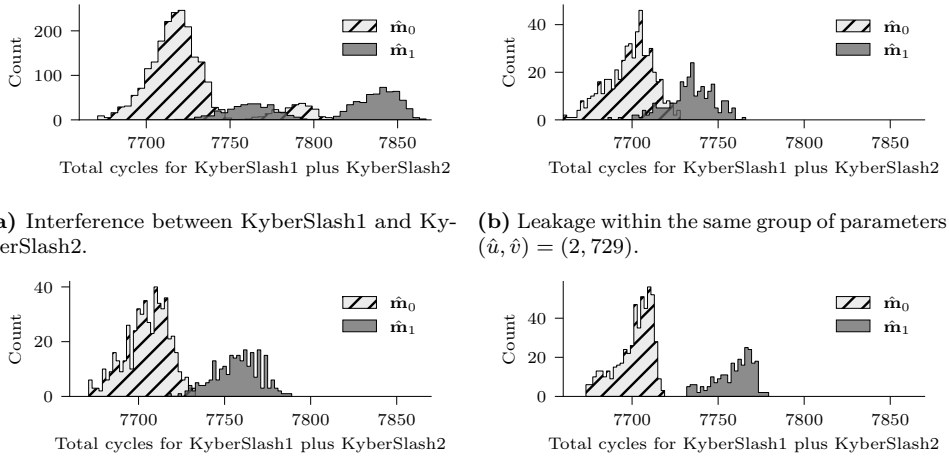
We can now see that, since $\mathbf{s}[\hat{\imath}]$ is a polynomial of small coefficients, if $\hat{u}$ is sufficiently small, then $m'[j]$ will be correctly decoded, for all $1 \leq j < 256$. However, whether $m'[0]$ would be correctly decoded depends on how large the noise defined by $\hat{v}, \hat{u}$ and $\mathbf{s}[\hat{\imath}][\hat{\jmath}]$ is. Therefore, if an attacker is careful in their selection of $\hat{v}, \hat{u}$, they may be able to learn the coefficient $\mathbf{s}[\hat{\imath}][\hat{\jmath}]$ when they have the information on whether m' is correctly decrypted to $\hat{\mathbf{m}}$ or not.

This is the core observation used by what are called plaintext-checking (PC) oracle attacks [10, 20, 40, 42, 48, 50]. PC-oracle attacks are a generic class of attacks that assume access to an oracle that, given a ciphertext $\mathbf{c}$ and a message $\hat{\mathbf{m}}$, returns whether $\mathbf{c}$ was decrypted to $\hat{\mathbf{m}}$ or not. Next we discuss how to build a PC-oracle using the decapsulation time, and how it can be used to recover the secret key.

PC-oracle attack exploiting KyberSlash2. Take a pair of 256-bit messages $\hat{\mathbf{m}}_0$ and $\hat{\mathbf{m}}_1$ that differ only in their first bits, and assume that $\hat{\mathbf{m}}_0[0] = 0$ and $\hat{\mathbf{m}}_1[0] = 1$. Then, it is possible [40, 42] to find a small set of pairs $(\hat{u}, \hat{v})$ such that the knowledge on whether the malicious ciphertext built with parameters $(\hat{\mathbf{m}}_0, \hat{u}, \hat{v}, \hat{\imath}, \hat{\jmath})$ is decrypted to $\hat{\mathbf{m}}_0$ or $\hat{\mathbf{m}}_1$ completely characterizes the secret key coefficient $\mathbf{s}[\hat{\imath}][\hat{\jmath}]$. In our attack, we used the same parameters $(\hat{u}, \hat{v})$ as the ones used by Ravi et al. [42], which are shown in Table 3.

Now, to use these ciphertexts to mount an attack relying on KyberSlash2, we proceed as follows. Generate a pair of messages $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ differing only in their first bits, and let $\mathbf{c}_0$ and $\mathbf{c}_1$ be their corresponding uncompressed encryptions using Kyber's CPA encryption algorithm. Let t_0 and t_1 be the decapsulation times when $\hat{\mathbf{m}}_0$ and $\hat{\mathbf{m}}_1$ are observed after decryption of the malicious ciphertext generated with parameters $(\hat{\mathbf{m}}_0, \hat{u}, \hat{v}, \hat{\imath}, \hat{\jmath})$. Notice that the randomness used when computing ciphertexts $\mathbf{c}_0$ and $\mathbf{c}_1$ come from the hashes of $\hat{\mathbf{m}}_0$ and $\hat{\mathbf{m}}_1$, respectively, thus the encryption of $\mathbf{c}_0$ and $\mathbf{c}_1$ should not share any noticeable similarities.

In an idealized scenario where perfect timing is available and KyberSlash2 is the only leakage from the implementation, then t_0 is expected to be slightly different than t_1. If $t_0 = t_1$, then we can simply restart the process with different pairs $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ until such a difference is observed. This means that, from the decapsulation time, the attacker can infer whether $\hat{\mathbf{m}}_0$ or $\hat{\mathbf{m}}_1$ was observed during decryption. Now, given a pair of indexes $(\hat{\imath}, \hat{\jmath})$, the attacker can verify, for each of the 4 parameters $(\hat{u}, \hat{v})$ from Table 3, which row matches their observations, thus learning secret coefficient $\mathbf{s}[\hat{\imath}][\hat{\jmath}]$. By iterating over the possible kn index parameters $(\hat{\imath}, \hat{\jmath})$, the attacker then learns the full secret key $\mathbf{s}$.



(a) Interference between KyberSlash1 and KyberSlash2.

(b) Leakage within the same group of parameters $(\hat{u}, \hat{v}) = (2, 729)$.

(c) Leakage for $(\hat{u}, \hat{v}) = (2, 729)$ considering the most separated pair $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ of messages among $\pi = 500$ randomly generated pairs.

(d) Average leakage for $(\hat{u}, \hat{v}) = (2, 729)$ using $\mu = 20$ pairs $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$.

Figure 6: The series of observations that allow us to increase the separation of the distributions that we need to distinguish for key recovery. We considered parameters for Kyber768, and the values are simulated using our model for the Cortex-M4 division time.

There are, however, two problems when using this approach in real attacks: in some setups, time measurements come with noise, and the leakages from KyberSlash1 and KyberSlash2 may interfere. In the following, we show how to recover the key in real-world noisy environments.

6.1.2 Key recovery under noisy setups

From the last sections, we know that the problem of key recovery reduces to classifying the observed decapsulation time into $\hat{\mathbf{m}}_0$ or $\hat{\mathbf{m}}_1$. However, because of the interference between KyberSlash1 and KyberSlash2, the distributions that we need to distinguish may not be well separated, as shown in Figure 6a. We now present a series of observations that allow us to simplify this classification by using a careful choice of ciphertexts to be decapsulated.

Separating the analysis for pairs $(\hat{u}, \hat{v})$. The most important observation is that the leakage from KyberSlash1 is very dependent on the value of $\hat{u}$. This happens because $\hat{u}$ scales the secret coefficients, resulting in different baselines for the coefficients upon which the message decoding procedure will act. Therefore, we should analyze the leakage distribution for each of the pairs $(\hat{u}, \hat{v})$ separately. Figure 6b illustrates how, by focusing on only one pair, namely $\hat{u}, \hat{v} = (2, 729)$, we get a simpler pair of distributions to distinguish.

Dealing with random noise. The simplest way to deal with random timing noise is to repeat the measurement a number ρ times and compute some robust measure (e.g. median) over the observed values. However, since the leakage from KyberSlash1 is completely determined by the ciphertext, we cannot get rid of it by repeating the measurement for the same ciphertext. One way of lowering the effect of both measurement and KyberSlash1 noise in the separation is to be more careful in the selection of the pair of messages $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$.

Our idea is to generate a number π of message pairs $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$, and select the pair whose KyberSlash2 leakage is separated by the largest number of cycles. Notice that this can be done offline, using only the target's public key, and a model of the division timing for the target device. For the Arm Cortex-M4, the Technical Reference Manual [4] states that a

Table 4: Clock cycles of `udiv` instruction with numerator n and denominator d on Arm Cortex-M4 (STM32F407VG). For a simpler description, we let $d_{\text{FL}} = 2^{\lfloor \log_2 d \rfloor}$.

Case	Clock cycles	Range of n with $d = 3329$
$d = 0$ or $n = 0$	2	0
$n/d_{\text{FL}} < 1$	3	1 to $(2^{11} - 1)$
$n/d_{\text{FL}} < 2^4$	5	2^{11} to $(2^{15} - 1)$
$n/d_{\text{FL}} < 2^8$	6	2^{15} to $(2^{19} - 1)$
$n/d_{\text{FL}} < 2^{12}$	7	2^{19} to $(2^{23} - 1)$
$n/d_{\text{FL}} < 2^{16}$	8	2^{23} to $(2^{27} - 1)$
$n/d_{\text{FL}} < 2^{20}$	9	2^{27} to $(2^{31} - 1)$
$n/d_{\text{FL}} < 2^{24}$	10	2^{31} to $(2^{32} - 1)$
$n/d_{\text{FL}} < 2^{28}$	11	–
$n/d_{\text{FL}} \geq 2^{28}$	12	–

`udiv` instruction takes 2–12 cycles depending on input data. We have reverse engineered the division timings for the common `STM32F407VG` (present on the `STM32F407-Discovery` board) and show the results on Table 4. We performed similar (but less extensive) experiments on the `STM32L476RG` suggesting that the ultra-low-power L4 series has the same division timings. While we present timings for arbitrary denominators, for KyberSlash only the column with $d = 3329$ is relevant showing cross-over points at $1, 2^{11}, 2^{15}, 2^{19}, 2^{23}, 2^{27}, 2^{31}$. For $d = 3329$, we have confirmed these timings through exhaustive search over the entire numerator space. This shows that for a fixed denominator, the division timing grows monotonically in the value of the numerator allowing to binary search the cross-over points. For variable denominator, we have picked random denominators and specially formed denominators (very small values and powers of two) and searched for the corresponding cross-over points using binary search. We performed a similar reverse engineering for the more complex application-profile processors Arm Cortex-A55 and Arm Cortex-A72. Even though a similar behavior was observed, there were significant differences in the numerator and denominator thresholds for the changes in the number of cycles. We present the results in Appendix B.

We can then use the division time model for the target microarchitecture to select the best pair $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ of messages out of π pairs generated at random. Figure 6c shows how the selection among $\pi = 500$ pairs allows to better distinguish between the two messages.

Reducing the noise from KyberSlash1. We observe that it is possible to actively reduce the noise from KyberSlash1 if, instead of using only one message pair $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ for building the malicious ciphertexts, we use a collection of μ pairs. Because each message pair has its own KyberSlash1 leakage baseline, using more pairs and taking the average leakage allow us to significantly reduce the deterministic noise. The effect of averaging the result for $\mu = 20$ pairs is illustrated in Figure 6d.

Key recovery. Synthesizing our methods for dealing with noise, we have the following parameters:

- ρ is the number of repetitions we use for each ciphertext to lower random measurement noise;
- π is the number of candidate pairs $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ we test offline to select only the one whose KyberSlash2 leakage is the most separated. We have used 100,000 for all experiments as it results in sufficient separation and ciphertext generation still terminates within seconds.
- μ is the number of pairs $(\hat{\mathbf{m}}_0, \hat{\mathbf{m}}_1)$ that we actually use when performing the attack.

Therefore, to attack Kyber768 using this setup, the number of decapsulations is $4\rho\mu kn$, where the factor of 4 comes from the number of pairs $(\hat{u}, \hat{v})$ needed to distinguish the secret coefficients (see Table 3).

Now, given the decapsulation time for each of the $4\rho\mu kn$ ciphertexts, we proceed as follows. First we take the median of the ρ repetitions of each ciphertext and consider this value as the observed time. Then we group each set of ciphertexts with the same parameters $(\hat{u}, \hat{v}, \hat{\imath}, \hat{\jmath})$, and take the average decapsulation time of the μ resulting values. We are now left with a set of $4kn$ values that we need to classify.

For each pair $(\hat{u}, \hat{v})$, we use a Gaussian mixture model (GMM) to give the probability that the decapsulation time is associated with $\hat{\mathbf{m}}_0$ or $\hat{\mathbf{m}}_1$. We remark that the GMM is a rather simple model that does not require any training or complicated parameters. We also tested the performance of k-means unsupervised clustering, but it did not perform as well as the GMM. Furthermore, the probabilities that GMM outputs are very useful when evaluating the likelihood of each row in Table 3 since, not only we are able to find the most suitable value of $\mathbf{s}[\hat{\imath}][\hat{\jmath}]$, we can also rank the different possibilities values by their likelihoods.

6.2 Experimental results

We present software demonstrating our attack on Cortex-M4 implementations from pqm4 [25] described in [22]. We target the Kyber768 parameter set, but the attack script is straightforwardly modified to all parameter sets of Kyber. The vulnerable functions (`poly_tomsg`, `poly_compress`, `polyvec_compress`) are verbatim copies of the Kyber reference implementations. We use the `4956a30` version of pqm4 which is the commit before the KyberSlash fixes have been ported to pqm4. We use the Arm GNU compiler toolchain version `13.2.1.Rel1` from [3]. We target the `STM32F407VG` Cortex-M4 (present on the `STM32F407-Discovery` board).

We implement a simple Python script `m4.py` that can be used to perform the attacks. It takes care of generating the corresponding ciphertexts, assembling the software to be run on the board, flashing the software to the board, executing the experiment, and attempting key recovery. In the end it reports if the secret key was recovered successfully. We present two versions of attack software and describe the experiments and results in more detail in the following.

6.2.1 Local attacker (demo2a)

The first version performs cycle counting directly on the target device returning exact cycle counts of the decapsulation to the attacker (the host). Due to the simplicity of the Cortex-M4 microarchitecture this results in predictable and deterministic cycle counts exactly matching our expectations for timing differences due to KyberSlash1 and KyberSlash2.

As we do not have any noise in this setup, the attack succeeds with a minimum number of measurements: We use the lowest sensible number of messages ($\mu = 2$), and only perform a single measurement ($\rho = 1$) per ciphertext resulting in $n \cdot k \cdot 8 = 6144$ decapsulations. We make use of the 32-bit `DWT_CYCCNT` cycle counter on the target device allowing us to obtain the exact number of clock cycles a decapsulation took. This cycle count is then sent back to the host using USART which attempts the key recovery. We clock the target at the maximum clock frequency of 168 MHz at which one decapsulation requires approximately 900,000 clock cycles.

An end to end attack takes approximately four minutes out of which only 33 seconds are spent on actual decapsulations, while the remaining overhead is dominated by serial communication with the target device. We achieve significantly higher baud rates using a USB-TTL adapter with a FDTI FT232 chipset rather than the cheaper and more common

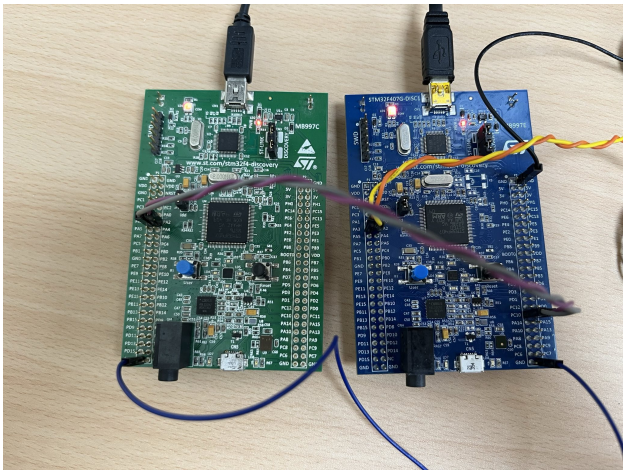


Figure 7: demo2b experiment setup. The target device (left) receives ciphertext from the attacker device (right) through the purple and gray jumper cables (USART3 to USART2) and reports back after decapsulation has completed. The attacker device receives the ciphertexts generated on the host through USART2 (yellow and orange jumper cables), forwards them to the target device through USART3, and reports the timings back to the host.

USB-TTL adapters with a Prolific PL2303 chipset. We determine 806,400 bps as the maximum baudrate for which our setup works reliably.

We perform 10 experiments and successfully recover the secret key in each of them. Executing `./m4.py -i 1 -n 2` runs the attack with the above parameters.

6.2.2 Remote attacker (demo2b)

The second more practical version of the attack software does not make use of the cycle counter on the target device, but instead performs the timing on the device interacting with the target device. The target reports back using USART after completing the decapsulation, but does not report the cycle counts passed. Note that this adds some noise especially due to the USART clock being much slower than the core clock. For simplicity, we make use of a second Cortex-M4 performing the timing and passing it on to a host laptop. The setup is shown in Figure 7.

The experiments proceed similarly as before, except that there is an intermediate Cortex-M4 that acts as a proxy for the ciphertexts and performs the timing. To improve the reliability of the timing, we make use of interrupts for receiving from the target device (USART3). Between the two boards, we use the highest baud rate that reliably works, which is 768,000 bps for our setup. Furthermore, we clock the target device at a lower clock frequency (24 MHz) than the attacker device (168 MHz) improving the accuracy of our timings.

We achieve reliable key recovery when using $\mu = 2$ and $\rho = 4$. A full experiment takes around 20 minutes out of which 17 minutes are spent on decapsulations. Note that, for $\rho > 1$, we only transmit the ciphertext once which may not be possible for a real attack. One could also increase μ, rather than ρ, however, increasing μ in our setup results in a much more significant increase in runtime due to the slow serial communication. We perform 10 experiments and successfully recover the secret key in each of them. Executing `./m4.py -i 4 -n 2 -r` runs the remote attack.

6.2.3 Other platforms

We have performed similar experiments on the Arm Cortex-A55 (which is part of the Qualcomm Snapdragon 888 present in many smartphones) and successfully recovered the

secret key when obtaining cycle counts using Perf. However, the measurements in that case contain significantly more noise than in the Cortex-M4 experiments, therefore we require significantly more decapsulations for recovering the key. Nevertheless, in this setup the transmission of the ciphertext is less time consuming than in the M4 experiments, and it is hence preferable to increase μ as discussed before. We have successfully recovered a secret key for $\mu = 40$ and $\rho = 300$ once, but have not yet performed any larger experiments.

7 Detecting secret-dependent divisions

There are many software-analysis tools aimed at systematically ensuring that secrets are not used as branch conditions or as memory addresses, the two most famous sources of timing attacks. The best usability scores in a recent study [17] come from a dynamic-analysis tool, TIMECOP, which has been applied to many existing cryptographic implementations. On the other hand, dynamic-analysis tools inherently catch only what is visible during specific program runs. Those runs do not always achieve the necessary path coverage; the best guarantees require static analysis.

This section investigates what can be done to systematically ensure that secrets are not used as inputs to division instructions. Section 7.1 covers dynamic analysis of many existing cryptographic implementations, and Section 7.2 covers high assurance as a spinoff of formal verification.

7.1 Dynamic scanning using Valgrind

TIMECOP was introduced in [34] as a patch to the SUPERCOP test framework [9]. An evolution of TIMECOP is now maintained as part of SUPERCOP. We have modified TIMECOP to check not just for secret branch conditions and secret load/store addresses but also for secret divisions.

7.1.1 Patching Valgrind

Internally, like various other constant-timeness tools going back to ctgrind [28], TIMECOP runs cryptographic software under the Valgrind tool – specifically, under Valgrind's Memcheck memory error detector [46]. Our main patch is to Memcheck, and is designed to also be usable via Valgrind-based tools other than TIMECOP, or by developers using Valgrind directly for ad-hoc tests; see Section 7.1.2.

Memcheck applies binary instrumentation to a program to detect, among other things, whether a program uses uninitialized values in non-trivial ways. In particular, it tests for uninitialized branch conditions and uninitialized load/store addresses. Tools such as TIMECOP mark secret data as uninitialized via a Valgrind "client request" [46, §3.1]. Memcheck does not test for uninitialized divisions; this is why a patch is needed.

Our patch, extending a simple 2015 prototype from Dove and Vasiliev [16], issues a warning when a client program uses a division instruction – such as `sdiv` or `udiv` on AArch64, or `idiv` or `div` on x86-64 – to operate on a secret (or uninitialized) data item. The patch also includes preliminary support for catching other variable-latency instructions. We developed the patch for Valgrind 3.22.0 (released October 2023), and it continues to work with Valgrind 3.23.0 (released April 2024).

The patch also modifies Memcheck to print a distinct error message for these operations, to make the operations easier to spot by human readers and scripts. To allow smooth future integration into upstream Valgrind, the patch checks for variable-latency instructions as a run-time option, skipped by default but enabled by a new Valgrind client request `VALGRIND_ENABLE_TIMECOP_MODE`.

```
==7174== Conditional jump or move depends
         on uninitialised value(s)
==7174==    at 0x108BBC: rej_uniform (indcpa.c:140) ...
==7174== Variable-latency instruction operand
         of size 4 is secret/uninitialised
==7174==    at 0x1090CC: pqcrystals_kyber512_ref_
            polyvec_compress (polyvec.c:48) ...
==7174== Variable-latency instruction operand ...
==7174==    at 0x109358: ...poly_compress (poly.c:28) ...
==7174== Variable-latency instruction operand ...
==7174==    at 0x10952C: ...poly_tomsg (poly.c:191) ...
```

Figure 8: Sample of Valgrind log showing detection of variable-latency instructions, in modified `test_kyber.c` with Kyber512, compiled with `gcc` 11.2.1 for AArch64 with `-Os`

7.1.2 Small-scale example: Kyber

We modified the test program `ref/test/test_kyber.c` from the November 2023 Kyber reference code, to invoke the Valgrind client request for the new checking mode, to mark the random number generator's output as "uninitialized" (i.e. potentially secret), and to mark public key data as "initialized".

We cross-compiled, for Linux/AArch64, the patched version of Valgrind, as well as the November 2023 version of the Kyber test programs linked with the Kyber512, Kyber768, and Kyber1024 implementations. The Kyber code was built at optimization levels `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`, and with debugging information enabled (`-g`). The builds were done on an Apple MacBook Pro (2018) with an Intel x86-64 Core i7, running Alpine Linux 3.19, and with a `gcc` 11.2.1 cross-compilation toolchain from [52]. We then ran the Kyber test programs under Valgrind, using the QEMU [15] emulator.

The Valgrind runs produced instrumentation logs, as partially shown in Figure 8. For the `-Os` binaries, the runs flagged lines 28 and 191 of `poly.c` (`poly_compress`, `poly_tomsg`), and line 48 of `polyvec.c`, for Kyber512 and Kyber768, and lines 43 and 191 of `poly.c`, and line 24 of `polyvec.c`, for Kyber1024, as being involved in variable time operations. The flagged instructions correspond to loads of operands for the vulnerable divisions, or operands to be combined with results from the divisions (Figure 9). Moreover, all of the KyberSlash divisions were successfully detected by this method.

We thus show that patching Valgrind can be a practical way to uncover this class of timing vulnerabilities.

7.1.3 Large-scale example: SUPERCOP

Beyond the Valgrind patch, we patched SUPERCOP to provide TIMECOP as part of SUPERCOP's multi-core dependency-tracking `data-do` tool for collecting and updating a large database of test results, whereas previously SUPERCOP provided TIMECOP only as part of a single-core non-dependency-tracking `do-part` tool aimed at developers testing their own code.

We completed the patch for a June 2024 development version of SUPERCOP. That version contains 4433 implementations of 1383 cryptographic primitives, all following SUPERCOP's API, which has also been adopted by various cryptographic competitions and cryptographic libraries. Within these 4433 implementations, 1283 are marked as `goal-constbranch` and `goal-constindex`, meaning that they are designed to avoid secret-dependent branches and array indices. This is also what triggers implementations to be considered by TIMECOP; this does not always mean that they pass TIMECOP.

For example, two of the primitives are Kyber512 and Kyber768. The Kyber768 primitive has three implementations in SUPERCOP, in three subdirectories `ref`, `compact`, and `avx2` of the `crypto_kem/kyber768` directory. All of these are marked as `goal-constbranch` and `goal-constindex`. The `compact` implementation passes TIMECOP but the `ref` and `avx2` implementations do not. All of the implementations have rejection-sampling loops;

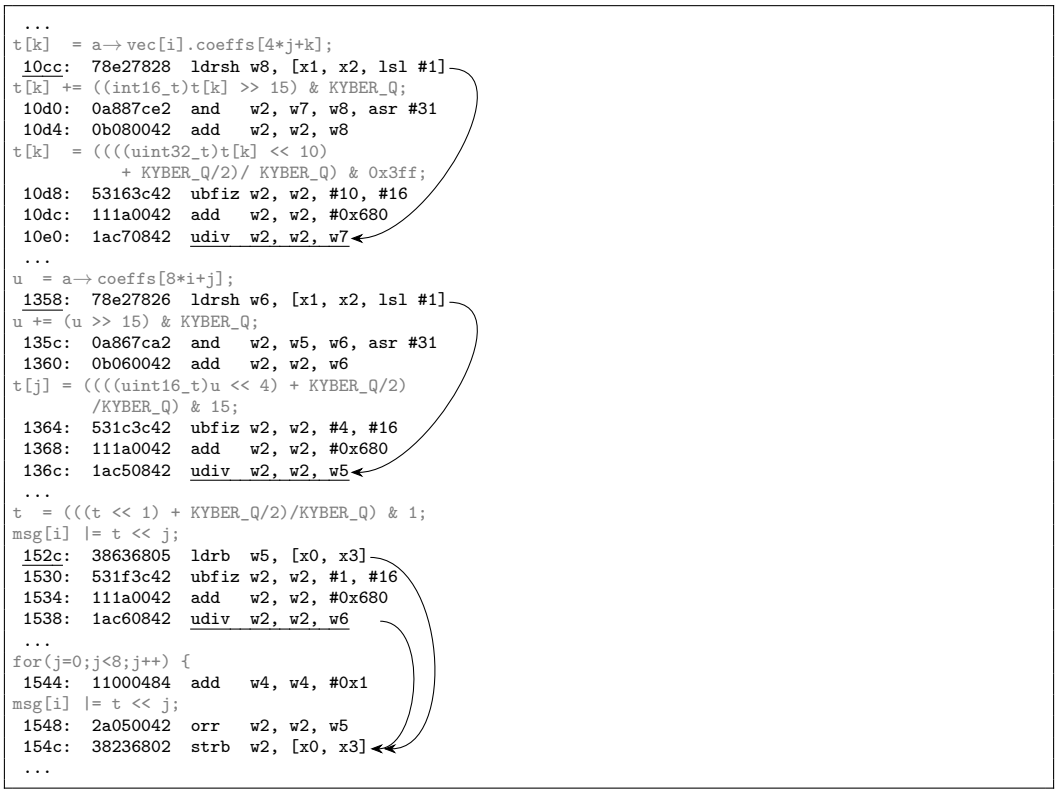


Figure 9: Disassembly showing secret operands flagged by patched Valgrind, and corresponding variable-latency instructions, in modified `test_kyber.c` with Kyber512, compiled with `gcc` 11.2.1 for AArch64 with `-Os`

the reason the `compact` implementation passes TIMECOP is that it has an extra line of code to declassify the rejection condition.

SUPERCOP compiles each implementation with a list of compilers. The default list includes `gcc -O`, `gcc -O2`, `gcc -O3`, `gcc -Os`, in each case with `-march=native` and `-mtune=native` to optimize for the host CPU, `-fwrapv` to avoid a well-known class of vulnerabilities, and `-fPIC -fPIE` for position independence. The default list also includes five `clang` options. It is important to note that analyzing binaries cannot make any guarantees about what will happen when there are changes in compiler options (e.g., a project not using `-fwrapv`), compiler versions, choice of compiler, and CPU; there is simply the hope that trying more combinations will catch more problems.

We restricted SUPERCOP to the 1283 implementations described above—except that we included SUPERCOP's four implementations of New Hope CCA, an ancestor of Kyber, by simply marking them as `goal-constbranch` and `goal-constindex`. As in Section 7.1.2, we added `-g` to the compiler options so that Valgrind output would mention line numbers in source code. We then ran our patched TIMECOP, along with SUPERCOP's usual tests and benchmarks. We used a dual AMD EPYC 7742 (128 cores in total) with 512GB of RAM. We compiled natively to include, e.g., AVX2 implementations; of course, this also meant that the run was excluding, e.g., Arm implementations. The machine owner had disabled overclocking both for security reasons and for hardware-longevity reasons, so the CPUs were limited to 2.245GHz. The machine is running Debian 12, with gcc 12.2.0 and clang 14.0.6. The run completed in 87 minutes of real time, using 5786 minutes of user time and 193 minutes of system time. Spot-checks during the run showed that all cores were in use at the beginning (with variable RAM usage, typically around 20GB in total

for 128 threads), but half the real time was spent waiting for implementations of a few particularly expensive primitives to finish.

This patched TIMECOP run successfully detected various divisions, all of which were specifically the New Hope code with `gcc -Os` (and not `clang -Os`). For example, within `newhope1024cca/avx2`, Valgrind pointed to line 77 of `poly.c`. Manually checking that line finds a division by `NEWHOPE_Q`. Within `newhope1024cca/ref`, Valgrind pointed to lines 16, 41, 82, 83, 84, 85, 115, 116, 354, and 370 of `poly.c`, along with line 215 of `ntt.c`. Manually checking these lines finds that line 16 of `poly.c` is the starting brace of a short function inlined into lines 41, 82, 83, 84, 85, and 115, with a division (actually a mod operator, `%`) on line 19. The other line numbers are directly pointing to divisions in the code.

A separate scan of the New Hope source code finds other division operators, such as an `r / 8` division in `fips202.c`. What distinguishes the TIMECOP results from such a scan is that TIMECOP locates divisions applied to data derived from *secret* inputs.

As a further experiment, we tried adding *all* of the KEMs in SUPERCOP and starting an incremental run. Like the first run, this finished in under 2 hours real time. The output contains 11610 "Variable-latency" lines; the immediately following lines have 2133 different instruction pointers coming from 556 different lines of code in 139 different implementations. A full analysis of those 556 lines of code would be a large project, but here are two illustrative examples. The first report in alphabetical order points to `crypto_kem/hila5/avx2`, specifically a line saying `% (HILA5_Q / 4)` in `kem.c`. The last in alphabetical order points to `crypto_kem/sikep751/ref`, specifically line 263 of `tdiv_qr.c`, which is actually inside the GMP library for big-integer arithmetic. SIKE has been broken in other ways, but this example illustrates the ability of binary analysis to automatically investigate subroutines.

These experiments show that TIMECOP's data-flow analysis, including our patches, can be efficiently applied to large volumes of existing cryptographic software within SUPERCOP's API, in particular producing many examples of data flow from secrets to division instructions. We confirmed that this approach is able to detect the vulnerabilities in the original Kyber code. It is also able to find similar-looking divisions in New Hope, HILA5, and SIKE code. Of course, this does not imply that the examples are exploitable. We also emphasize that the analysis is not a guarantee: it is limited to the binaries created in the TIMECOP run, and to the code paths that are actually taken in the TIMECOP run. Nonetheless, it would have caught the problematic source code in the Kyber implementations and deploying it at a large scale is feasible.

7.2 Using formal methods

Another approach that can help programmers detect and prevent bugs like KyberSlash is the use of formal verification tools. Indeed, KyberSlash1 was first discovered by some of the authors of this paper when they were trying to formally verify a Rust implementation of Kyber.

The security ideal for cryptographic code is *secret independence*, that is, the attacker-observable behavior of a program should not depend on its secret inputs. This means that the program should, of course, not reveal its secrets via its input-output behavior, but also that it does not leak secrets via (say) the program's runtime or memory accesses.

There are several variations and formal definitions of secret independence defined in the literature that cover different subsets of side-channel attacks. The most common definition seeks to prevent branching on secrets, non-constant-time arithmetic operations (such as division) on secrets, and using secrets as array indexes or memory addresses. This discipline is sometimes called cryptographic constant time. [7]

There are a variety of tools that seek to ensure secret independence in cryptographic code. We refer the reader to recent surveys of these tools and evaluations of their usability

for a more complete picture [6, 17, 24]. In the rest of this section, we use the definition of secret independence that is used in the HACL* verified cryptographic library [56], whose formal guarantees are defined and proven for C programs generated from the F* programming language [39].

7.2.1 Secret independence by typechecking

To use any of the formal verification tools on cryptographic code, we must begin by labeling every input and output as either *public* or *secret*. In expressive dependently-typed languages like F*, these secrecy labels can be embedded within the type of each variable, alongside other logical properties needed for correctness (e.g. the range of integers that may be contained in the variable). By default, it is safe to assume that all inputs and outputs within cryptographic code is labeled secret, and the programmer only needs to annotate inputs and outputs that they know to be public.

To verify these labels, we then need to annotate all the primitive operations in the language to reflect our assumptions about whether or not they leak information about their inputs via side-channels. If an operation may leak information about one of its inputs, then that input is labeled as public, preventing cryptographic code from calling it with a secret value. For example, we typically label both inputs to the division operation as public, and on some platforms we may also want to label inputs to certain multiplications as public. The labels given to language primitives and external libraries are *trusted* and hence must be carefully reviewed to ensure that they capture the operational details of the target platforms.

In the F* library used in HACL*, for example, the types `u8` and `i16` are defined to be *secret integers* and arithmetic operations like division and modulus are not defined for them. Furthermore, secret integers cannot be compared or used as array indexes. All these operations are only available for values declared with the public integer types `pub_u8` or `pub_i16`. Public integers can be converted to secret integers, but converting a secret integer to a public integer requires a call to an explicit `declassify` operation, every use of which needs to be carefully audited.

Given such a secrecy labeling for a program and all the libraries it uses, the type checker can statically verify that the program is secret independent, and point out any parts of the code where the discipline is violated. It is worth noting that such a typing discipline does not really need the full expressiveness of F*; it can easily be implemented in any type system that supports abstract types and interfaces, including Rust and Java.

7.2.2 Finding KyberSlash with F*

The first variant of KyberSlash was found during a larger project of formally verifying a Rust implementation of ML-KEM by translating it to F* and then proving its correctness. As a first step towards a correctness proof, we tried to prove that the translated ML-KEM F* code was secret independent. By using the default integer types, all the inputs and outputs in our code were initially labeled as secret. Then, inputs that are public, such as algorithm parameters or public keys are manually labeled as public by changing their types to use public integers. Finally, outputs that need to be revealed to the application, such as ML-KEM ciphertexts, are *declassified* from secret to public.

When we then run the F* tyepchecker on the F* code generated from the Rust implementation, it immediately finds and flags the secret dependent division on line 5:

```
1 let compress_q (coefficient_bits: u8) (fe: u16) =
2   let compressed:u32 = (cast (fe <: u16) <: u32) <<!
3                        (coefficient_bits +! 1uy <: u8) in
4   let compressed:u32 = compressed +! v_FIELD_MODULUS in
5   let compressed:u32 = compressed /!
6                        (v_FIELD_MODULUS <<! 1ul) in
```

```
7 get_n_least_significant_bits coefficient_bits compressed
```

To fix this type error, we could label the input field element as `pub_u16` to indicate that it is public, and then prove that this input is indeed public at all call sites, which would fail since this function is used to compress the IND-CPA message coefficient. And if the function was going to be used with secret inputs, we need to rewrite the Rust code to not use division.

Initially, we did both. We wrote a separate function for compressing message coefficients that treated the input field element as secret, and we kept this function for compressing IND-CPA ciphertext coefficients, since we were (incorrectly) assuming that the ciphertexts were public and declassifying them. Later, when KyberSlash2 was discovered, we fixed our model and moved this declassification from the IND-CPA ciphertexts to the IND-CCA ciphertexts. When we do so, the KyberSlash2 variant also gets flagged by the F* typechecker, and we subsequently reimplemented ciphertext compression as well.

This experience shows both the strengths and the weaknesses of our approach. As long as we correctly annotate and review all public inputs and outputs, the typechecker is able to find secret independence bugs. However, one must be careful when reflecting cryptographic assumptions in the secrecy labeling, or we may miss attacks.

7.2.3 Limitations and Future Directions

When writing code in a compiled language such as Rust or C, the methodology described above ensures that there are no obvious timing leaks. However, even when carefully writing and formally checking the source code, the compiler may produce secret-dependent code.

The formal verification guarantees we obtain from F* above are at the level of source code, and nothing we check here guarantees that the compiler or microarchitecture will not introduce new side-channels that were not visible in the source language semantics.

Modern compilers optimize code aggressively for performance when using high optimization levels. During this process operations such as masking or shifts may be converted into conditional jumps. Techniques as described in 7.1 can be used to analyze the compiled code and detect compiler-introduced secret-dependent operations. To get more formal guarantees, one would need to apply the secret independence checks at the level of machine code, using techniques like those used in the Jasmin assembly implementation of ML-KEM [2].

Acknowledgment

The authors would like to thank Richard Petri and Shih-Ming Yin for their help with setting up the Cortex-M4 demo. Part of this work was funded by FINEP (Financiadora de Estudos e Projetos). Part of this work was funded by the Intel Crypto Frontiers Research Center. Part of this work was funded by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

References

[1] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology, 2022.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma,

Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber episode IV: implementation correctness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):164–193, 2023.

[3] Arm. Arm GNU toolchain downloads. Accessed 2025-01-14. URL: https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads.

[4] Arm Limited. Cortex-M4 Technical Reference Manual. URL: https://developer.arm.com/documentation/ddi0439/latest.

[5] Roberto Avanzi, Joppe W. Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 3.0): Algorithm specifications and supporting documentation (October 1, 2020). *Submission to the NIST post-quantum project*, 2020.

[6] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021.

[7] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, and David Pichardie. System-level non-interference of constant-time cryptography. Part II: verified static analysis and stealth memory. *J. Autom. Reason.*, 64(8):1685–1729, 2020.

[8] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL: https://cr.yp.to/papers.html#cachetiming.

[9] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems, 2024. URL: https://bench.cr.yp.to.

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

[11] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003. URL: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[12] Larry Campbell. Tenex hackery, 1991. URL: https://groups.google.com/g/alt.folklore.computers/c/v9KnB8BIXGY/m/aZ-qDLtD0gAJ.

[13] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, pages 2–9. ACM, 2019. doi:10.1145/3338467.3358948.

[14] Wouter de Groot. A performance study of X25519 on Cortex-M3 and M4, 2015. URL: https://research.tue.nl/en/studentTheses/a-performance-study-of-x25519-on-cortex-m3-and-m4.

[15] QEMU Project Developers. QEMU: A generic and open source machine emulator and virtualizer. Accessed 2025-01-14. URL: https://www.qemu.org.

[16] Justin Dove and Victor Vasiliev. Automated testing against timing attacks. Term project, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2015. https://courses.csail.mit.edu/6.857/2015/projects.

[17] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "These results must be false": A usability evaluation of constant-time analysis tools, 2024. USENIX Security Symposium 2024, to appear. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/fourne.

[18] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[19] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 359–386. Springer, 2020. doi:10.1007/978-3-030-56880-1_13.

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

[21] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 341–371, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70500-2_12.

[22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard arithmetic for lattice-based cryptography. 2022:614–636, Aug. 2022. URL: https://tches.iacr.org/index.php/TCHES/article/view/9833.

[23] Intel Corporation. Enable compiler optimizations. Accessed 2025-01-07. URL: https://www.intel.com/content/www/us/en/docs/programmable/683282/current/enable-compiler-optimizations.html.

[24] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *43rd IEEE Symposium on Security and Privacy*, San Francisco, 2022. IEEE.

[25] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[26] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. doi:10.1007/3-540-68697-5_9.

[27] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 20–38. IEEE, 2020. doi:10.1109/SP40000.2020.00082.

[28] Adam Langley. Checking that functions are constant time with valgrind, 2010. https://www.imperialviolet.org/2010/04/01/ctgrind.html.

[29] Adam Langley. Lucky Thirteen attack on TLS CBC, 2013. URL: https://www.imperialviolet.org/2013/02/04/luckythirteen.html.

[30] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[31] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: software-based power side-channel attacks on x86. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 355–371. IEEE, 2021. doi:10.1109/SP40001.2021.00063.

[32] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1977–1991. ACM, 2022. doi:10.1145/3548606.3560682.

[33] National Institute of Standards and Technology. FIPS203: Module-lattice-based key-encapsulation mechanism standard (initial public draft). Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, 2023-08-24 2023. https://doi.org/10.6028/NIST.FIPS.203.ipd.

[34] Moritz Neikes. TIMECOP: automated dynamic analysis for timing side-channels, 2019. URL: https://www.post-apocalyptic-crypto.org/timecop/.

[35] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, June 2007. https://dl.acm.org/doi/10.1145/1250734.1250746; https://valgrind.org.

[36] Thales Bandiera Paiva and Routo Terada. A timing attack on the HQC encryption scheme. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 551–573. Springer, 2019. doi:10.1007/978-3-030-38471-5_22.

[37] Colin Percival. Cache missing for fun and profit, 2005. URL: https://www.daemonology.net/papers/htt.pdf.

[38] Thomas Pornin. The problem, 2018. URL: https://www.bearssl.org/ctmul.html.

[39] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.*, 1(ICFP):17:1–17:29, 2017. doi:10.1145/3110261.

[40] Gokulnath Rajendran, Prasanna Ravi, Jan-Pieter D'anvers, Shivam Bhasin, and Anupam Chattopadhyay. Pushing the limits of generic side-channel attacks on LWE-based KEMs-parallel PC oracle attacks on KyberKEM and beyond. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[41] Prasanna Ravi, Anupam Chattopadhyay, Jan-Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. *ACM Trans. Embed. Comput. Syst.*, 23(2):35:1–35:54, 2024. doi:10.1145/3603170.

[42] Prasanna Ravi, Suman Deb, Anubhab Baksi, Anupam Chattopadhyay, Shivam Bhasin, and Avi Mendelson. On threat of hardware trojan to post-quantum lattice-based schemes: a key recovery attack on Saber and beyond. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 81–103. Springer, 2021.

[43] Prasanna Ravi, Thales Paiva, Dirmanto Jap, Jan-Pieter D'Anvers, and Shivam Bhasin. Defeating low-cost countermeasures against side-channel attacks in lattice-based encryption: A case study on Crystals-Kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):795–818, 2024. URL: https://doi.org/10.46586/tches.v2024.i2.795-818, doi:10.46586/TCHES.V2024.I2.795-818.

[44] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020. URL: https://doi.org/10.13154/tches.v2020.i3.307-335, doi:10.13154/TCHES.V2020.I3.307-335.

[45] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehle, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2019. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[46] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX'05 Annual Technical Conference*, pages 17–30, April 2005. https://www.usenix.org/legacy/events/usenix05/tech/general/seward.html.

[47] Marvin Staib and Amir Moradi. Deep learning side-channel collision attack. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):422–444, 2023. URL: https://doi.org/10.46586/tches.v2023.i3.422-444, doi:10.46586/TCHES.V2023.I3.422-444.

[48] Yutaro Tanaka, Rei Ueno, Keita Xagawa, Akira Ito, Junko Takahashi, and Naofumi Homma. Multiple-valued plaintext-checking side-channel attacks on post-quantum KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):473–503, 2023.

[49] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(1):37–71, 2010. URL: https://doi.org/10.1007/s00145-009-9049-y, doi:10.1007/S00145-009-9049-Y.

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

[51] Wim van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Comput. Secur.*, 4(4):269–286, 1985. doi:10.1016/0167-4048(85)90046-X.

[52] Zach van Rijn. Your source for static cross- and native- musl-based toolchains. Accessed 2025-01-14. URL: https://musl.cc.

[53] Ruize Wang, Martin Brisfors, and Elena Dubrova. A side-channel attack on a higher-order masked CRYSTALS-Kyber implementation. In Christina Pöpper and Lejla Batina, editors, *Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part III*, volume 14585 of *Lecture Notes in Computer Science*, pages 301–324. Springer, 2024. doi:10.1007/978-3-031-54776-8_12.

[54] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 679–697. USENIX Association, 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen.

[55] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Everywhere all at once: Co-location attacks on public cloud FaaS. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 133–149. ACM, 2024. doi:10.1145/3617232.3624867.

[56] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. pages 1789–1806. ACM, 2017.

A Disclosure

We have reported KyberSlash to the Kyber team privately shortly after discovery. After discussion with the Kyber team it has been agreed to publicize the vulnerabilities immediately. The main consideration for this was that due to the upcoming standardization later in 2024, the current deployment of the affected code is small, but the potential impact of delaying the announcement would be much more devastating. The divisions have been replaced by explicit multiplications in the official reference implementation prior to the announcement.[4]

Starting in December 2023, further patches addressing KyberSlash have been applied to at least the following cryptographic libraries: zig[5], kyber-k2so[6], CIRCL[7], AWS-LC[8], Botan[9], liboqs[10], crystals-go[11], PQClean[12], pqcrypto-kyber[13], pypqc[14], pqm4[15], and

[4] KyberSlash1 vulnerability was patched in the commit dda29cc dated December 1, 2023 and KyberSlash2 vulnerability was patched in the commit 272125f dated December 30, 2023.

[5] https://github.com/ziglang/zig

[6] https://github.com/symbolicsoft/kyber-k2so

[7] https://github.com/cloudflare/circl

[8] https://github.com/aws/aws-lc

[9] https://github.com/randombit/botan

[10] https://github.com/open-quantum-safe/liboqs

[11] https://github.com/kudelskisecurity/crystals-go

[12] https://github.com/PQClean/PQClean

[13] https://github.com/rustpq/pqcrypto

[14] https://github.com/JamesTheAwesomeDude/pypqc

[15] https://github.com/mupq/pqm4

Table 5: Clock cycles of `udiv` instruction with numerator n and denominator d on Arm Cortex-A55 (Snapdragon 888). For a simpler description, we let $d_{\text{FL}} = 2^{\lfloor \log_2 d \rfloor}$.

Case	Clock cycles	Range of n with $d = 3329$
$d = 0$ or $n = 0$	3	0
$n/d_{\text{FL}} < 2^2$	3	1 to $(2^{13} - 1)$
$n/d_{\text{FL}} < 2^6$	4	2^{13} to $(2^{17} - 1)$
$n/d_{\text{FL}} < 2^{10}$	5	2^{17} to $(2^{21} - 1)$
$n/d_{\text{FL}} < 2^{14}$	6	2^{21} to $(2^{25} - 1)$
$n/d_{\text{FL}} < 2^{18}$	7	2^{25} to $(2^{29} - 1)$
$n/d_{\text{FL}} < 2^{22}$	8	2^{29} to $(2^{32} - 1)$
$n/d_{\text{FL}} < 2^{26}$	9	–
$n/d_{\text{FL}} < 2^{30}$	10	–
$n/d_{\text{FL}} \geq 2^{30}$	11	–

Table 6: Clock cycles of `udiv` instruction with numerator n and denominator d on Arm Cortex-A72 (BCM2835). For a simpler description, we let $d_{\text{FL}} = 2^{\lfloor \log_2 d \rfloor}$.

Case	Clock cycles	Range of n with $d = 3329$
$d = 0$ or $n = 0$	3	0
power-of-two d	3	–
$n/d < 1$	3	1 to 3328
$n/d_{\text{FL}} < 2^8$	5	3329 to $(2^{19} - 1)$
$n/d_{\text{FL}} < 2^{12}$	6	2^{19} to $(2^{23} - 1)$
$n/d_{\text{FL}} < 2^{16}$	7	2^{23} to $(2^{27} - 1)$
$n/d_{\text{FL}} < 2^{20}$	8	2^{27} to $(2^{31} - 1)$
$n/d_{\text{FL}} < 2^{24}$	9	2^{31} to $(2^{32} - 1)$
$n/d_{\text{FL}} < 2^{28}$	10	–
$n/d_{\text{FL}} \geq 2^{28}$	11	–

kyberlib[16]. We have not investigated whether these libraries were exploitable before the patches. There are some other Kyber implementations, such as [2], that never included the problematic divisions.

We have communicated with numerous maintainers of potentially affected libraries and feedback has been uniformly positive.

B Division leakage for other devices

Table 5 and Table 6 show the reverse engineered division timings for the Arm Cortex-A55 and the Arm Cortex-A72. The timings look similar to the Arm Cortex-M4 timings, but have different cross-over points. The Arm Cortex-A72 has additional shortcuts for power-of-two denominators.

C Signal processing for the KyberSlash1 demo

The KyberSlash1 demo takes the following steps to filter out noise in timings, although sufficient noise—or noise dependent on the ciphertexts—can make this fail.

The demo collects batches of measurements. After each batch, it uses all measurements so far to formulate a guess for the Kyber secret key. It recomputes the public key from this guess, checks for a match, and stops in case of success. It gives up if it has not succeeded after 512 batches.

[16] https://github.com/sebastienrousseau/kyberlib

Each batch includes 7 choices of ciphertexts $(\mathbf{u}, v)$ targeting each of the 512 positions in the secret. These $7 \cdot 512$ ciphertexts are handled in a random order to limit any impacts from hysteresis. Each ciphertext is tried 16 times in succession, with the timings sorted and only one intermediate timing recorded to remove outliers.

After the batch, the demo computes an interquartile mean of the recorded timings (across all batches) for each of the 7 choices of $(\mathbf{u}, v)$ at each position, obtaining $7 \cdot 512$ interquartile means. At each position, the demo then obtains a guess for this position of the secret key by comparing

- 7 interquartile means $t_0, \ldots, t_6$ from the *observed* timings and
- for each possibility for this position of the secret key: a *model* of the division timings $d_0, \ldots, d_6$ for the same 7 choices of $(\mathbf{u}, v)$.

Specifically, the demo takes the possibility that minimizes the variance of the 7 numbers $t_0 - d_0, \ldots, t_6 - d_6$. The point here is that, in the optimistic model from Section 5.1.2, a correct guess would have these 7 numbers being a constant (and thus variance 0), namely the time for the rest of the decapsulation process. (This is not exactly the same as asking which guess has a model best correlated with the observed timings; it is asking specifically for a *diagonal* correlation. Correlation allows a scaling factor, whereas the model predicts that the scaling factor is 1.)

One expects random noise to settle down at *most* positions before it settles down at *all* positions; a "coupon collector" spends considerable time waiting for the last few coupons. To address this, the demo actually tries multiple guesses after each batch: specifically, it picks 10 positions with the smallest ratios between the second-smallest variance and the smallest variance, and then tries all 2^{10} combinations of first or second guesses at each position.

C.1 Optimizations

No claims of optimality are made for the number of decapsulations used in this demo. Only one timing is kept from each series of 16 timings; presumably the other timings could be used productively, and there is no reason to think that 16 is optimal. The 2^{10} guesses after each batch could be replaced by more guesses and more advanced lattice attacks; it would be interesting to explore how to optimize "soft-decision decoding" in the lattice context, accounting for varying confidence levels at each position. A ciphertext can target many positions at once, say a random selection of about a third of the 256 positions, and attribute the resulting timing to each of those positions; each position receives some noise from the other positions, but a simple model suggests that this will be outweighed by the speedup.

D Assembly Code Snippets for Message Decoding Operation

Fig.10 and Fig. 11 show divisions when compiling Fig 2 for 64-bit and 32-bit Arm processors.

E Assembly Code Snippets for Ciphertext Compression Operation

Fig.12 and Fig. 13 show divisions when compiling Fig 4 for 64-bit and 32-bit Arm processors.

```
...
ldrsh w6, [x1, x2, lsl 1]
ldrh w2, [x1, x2, lsl 1]
and w6, w7, w6, asr 31
add w2, w2, w6
ubfiz w2, w2, 1, 16
add w2, w2, 1664
/* Variable-Time Division Operation */
udiv w2, w2, w7
and w2, w2, 1
lsl w2, w2, w4
...
```

Figure 10: Assembly code snippet of the message decoding operation for a single coefficient, when compiled with `arm64 gcc 14.1.0` for the AArch64 architecture using the `-Os` compiler optimization flag.

```
...
uxtah r3, lr, r3
uxth r3, r3
lsls r3, r3, #1
add r3, r3, #1664
/* Variable-Time Division Instruction */
udiv r3, r3, r5
and r3, r3, #1
lsls r3, r3, r4
orr r3, ip, r3
...
```

Figure 11: Assembly code snippet of the message decoding operation for a single coefficient, when compiled with `arm-none-eabi-gcc 14.1` for Arm Cortex-M4 CPU (`-mcpu=cortex-m4`) using the `-Os` compiler optimization flag.

```
...
ldrsh w6, [x1, x2, lsl 1]
and w2, w5, w6, asr 31
add w2, w2, w6
ubfiz w2, w2, 4, 16
add w2, w2, 1664
/* Variable-Time Division Operation */
udiv w2, w2, w5
and w2, w2, 15
strb w2, [x3, x7]
add x3, x3, 1
cmp x3, 8
...
```

Figure 12: Assembly code snippet of a single iteration of ciphertext compression operation, when compiled with `arm64 gcc 14.1.0` for the AArch64 architecture using the `-Os` compiler optimization flag.

```
...
uxth r3, r3
lsls r3, r3, #4
add r3, r3, #1664
cmp r5, r1
/* Variable-Time Division Instruction */
udiv r3, r3, r4
and r3, r3, #15
strb r3, [r6], #1
bne .L3
..
```

Figure 13: Assembly code snippet of a single iteration of ciphertext compression operation, when compiled with `arm-none-eabi-gcc 14.1` for Arm Cortex-M4 CPU (`-mcpu=cortex-m4`) using the `-Os` compiler optimization flag.