Key recovery on static Kyber based on transient execution attacks

Luccas Ruben J. Constantin-Sukul, Rasmus Ø. Gammelgaard, Alexander N. Henriksen, and Diego F. Aranha[0000−0002−2457−0783]

Aarhus University, Aarhus, Denmark
{luccas,rasmusoeg,ahenriksen}@post.au.dk, dfaranha@cs.au.dk

Abstract Transient execution attacks on modern processors continue to threaten security by stealing sensitive data from other processes running on the same CPU. A recent example is Downfall, which demonstrated how microarchitecture leakage could reveal short AES keys. We explore the possibility of leaking much longer keys for post-quantum cryptography by combining Gather Data Sampling from Downfall with Flush+Reload to mount a key recovery attack against static Kyber. We reassemble private keys from fragments scattered within random noise by exploiting patterns observed across multiple consecutive loads. The whole attack runs in under 40 minutes with success rate between 60% and 70%, no matter the Kyber security level used by the victim. This underscores the implicit reliance of cryptographic algorithms on the underlying microarchitecture for security.

Keywords: Micro-architecture security · Post-quantum cryptography · Key recovery · Kyber.

1 Introduction

Modern processor manufacturers have found numerous techniques to optimize their designs to enhance speed and efficiency by exploiting caching mechanisms, speculative and out-of-order execution. However, these optimizations have not come without adversity. One notable vulnerability in modern CPUs involves *side channels*, which enable attackers to exploit unintended ways to access data beyond security boundaries. Examples of side channels include variations in computation time, fluctuations in power consumption, and changes to the CPU cache.

One important attack vector to mount side-channel attacks is exploiting security issues in the underlying microarchitecture. Attacks such as Meltdown [4], Spectre [3] and GoFetch [2] have abused aggressive processor optimizations to steal sensitive data. Mitigating these attacks is hard, and may force users to update their hardware. Alternatively, they can rely on software or firmware workarounds which often disable optimizations, resulting in considerable performance degradation. Downfall [5] is a recent entry in this class of attacks, now targeting leakage through internal buffers.

This paper explores the use of *Gather Data Sampling (GDS)* from Downfall to steal private keys from implementations of post-quantum cryptography. Our contributions are:

- We propose chaining methods to leak large memory buffers containing secret data through multiple leakage samples collected through GDS. The secret data spans a few hundred bytes, significantly exceeding typical symmetric keys targeted in previous work.
- We weaponize GDS against a high-speed implementation of Kyber [1] in the static setting, realizing in practice an impact hinted in the original Downfall paper [5, §7.2].

In our threat model, the victim and attacker must run on the same physical CPU core as sibling threads, or be context-switching on the same CPU thread. This captures situations where multiple users can access the same machine and thus might share a CPU core, for instance in multi-tenant clouds.

The remaining sections of the paper are organized as follows. Section 2 covers background material in microarchitecture security and MDS attacks. Section 3 describes the Downfall attack, which we reproduce to some extent and for our purposes in Section 4. Section 5 gives an overview of Kyber and explains in detail how GDS from Downfall can be used to recover static Kyber keys from microarchitecture leakage. We discuss experimental results in Section 6 and a comparison with previous work in Section 7, concluding the paper in Section 8.

2 Preliminaries

Memory subsystem. CPUs often support multiple levels of cache between the individual cores and the RAM, which serve to speed up memory access by storing frequently or soon-to-be-used data. Individual cores typically have their own caches (levels L1/L2), and the entire CPU might also have a cache level shared among all cores (L3). When dealing with memory and data sizes on Intel processors, important types are the 2-byte *word*, 4-byte *double-word* (DWORD), and 8-byte *quad-word* (QWORD). Memory is often divided into blocks of size 64 bytes (*cache lines*). If a byte belonging to a block is accessed, the cache will store the entire block. Since caches are small compared to memory, they can only store a few blocks. This means that memory blocks compete for cache lines, and structure is needed in the cache to optimize lookup time and hit rate.

Vector registers. Intel CPUs have a set of registers primarily used for Single-Instruction Multiple-Data (SIMD) instructions, allowing to simultaneously compute a given operation on multiple pieces of data, which is particularly beneficial in optimizing arithmetic-intense software. Vector instructions come in several different sizes, but we will focus on the 256-bit `ymm` vector registers, which most currently running Intel processors support as part of the AVX2/512 extensions. These registers can be interpreted in different ways, such as holding 4 QWORDs or 8 DWORDs, depending on the data size encoded in the instruction.

Multithreading. Multiple threads may be executed on the same physical core simultaneously. They share the same hardware, but architecturally they are isolated from one another, and may only access their own addresses and registers. Many modern processors support two virtual threads running on each physical core, which appear as two separate cores for running software. This feature is called Simultaneous Multi-Threading (SMT) and it improves the overall efficiency by increasing the number of independent instructions in the pipeline.

Speculative Execution. CPU cores might choose to execute otherwise sequential instructions in parallel to gain a significant speedup. This can happen during branching, where both branches might get executed before the condition is resolved, or by predicting data from an otherwise slow load operation, and forwarding it to dependent instructions. The actual predictions might be wrong, which can result in flushing incorrectly executed instructions, or the need to re-execute some instructions with updated data. To maintain isolation, it is important that incorrectly executed instructions do not become architecturally visible to the running software. However, as noted in Downfall and other microarchitecture vulnerabilities, its side effects may still be observed.

Temporal Buffers. Individual CPU cores might use internal buffers to speed up operations. When performing a read, if requested data is not present in the cache, it might instead forward data from a buffer to dependent instructions whilst waiting for the rest of the cache line. When performing a write, it might store data in an internal buffer before committing it to the cache.

Transient execution environments. Instructions which have been speculatively executed are called *transient instructions*. These instructions are accessible in the execution environment whilst the CPU has yet to resolve their validity and possibly flush them. The duration in which the transient data might be accessible is called the *transient window*, which can be increased by provoking cache misses or other micro-architectural effects.

Exploiting cache timings. Caches might enable covert- or side-channel attacks, since the time difference between a cache-hit and a cache-miss is often measurable. Using this together with the ability to flush cache lines often provides an easy mechanism for leaking transient data by encoding it to one of many cache lines. An example of this is seen in Flush+Reload [9], where an attacking thread monitors memory accesses of a victim thread through shared memory pages (e.g. when using standard libraries).

MDS attacks. Microarchitecture Data Sampling (MDS) attacks collect and leak confidential in-flight data from internal CPU buffers across security boundaries. This comes in contrast to most microarchitecture attacks, which exploit speculative execution and/or target data from the CPU caches.

Two recent examples of MDS attacks are RIDL [7] and Zombieload [8]. Rogue In-Flight Data Load (RIDL) is a data speculation attack that carefully crafts a speculative load to an address that the CPU is not prepared to handle (as in a page fault), ultimately triggering the CPU into pulling data from internal buffers, which can then be detected using Flush+Reload. In the cross-process setting, RIDL requires some bytes to be known in advance, such that a *mask-sub-rotate* technique can be used to leak only values consistent with previous knowledge or observations. The attack reported in the original paper is quite inefficient, recovering a total of 26 characters from the sensitive `/etc/shadow` file in 24 hours. ZombieLoad exploits faulting load instructions to transiently compute on values belonging to previous memory operations in the current or sibling thread, allowing them to be recovered. For short cryptographic keys that are entirely available in the transient window, additional redundant data (*domino bytes* composed of nibbles of different bytes) can be transmitted to recover a 16-byte AES key across processes in just 10 seconds.

Even if those attacks are considered to be quite poweftul, being applicable to recover secrets in a number of scenarios involving SGX and MDS-resistant hardware, it is not clear how they can be used to efficiencly leak much longer keys that do not fit a single transient domain.

3 Downfall

Downfall [5] is a series of attacks which exploit the use of temporal buffers in Intel processors. It is observed that these buffers can be shared by several processes running on the same physical core, where some specific buffers reside. For example, vector instructions frequently use these buffers to optimize load times and data processing. When a process uses vector operations, the data in the vector registers potentially goes through a temporal buffer to optimize performance.

Gather Data Sampling (GDS). The main attack presented in Downfall achieves the goal of acquiring data inside a transient window, which should not usually be accessible, by using the x86 assembly instruction *gather*. Afterwards, it seeks to transmit the data using a *covert channel*, hoping it becomes visible after the transient window has closed. The *gather* instruction accesses scattered data (DWORDs or QWORDs) in memory efficiently and loads it into a vector register. This is done using a base pointer, a vector register with offsets from that base pointer, and a mask to potentially ignore some of the loads.

For this instruction to be faster than just multiple loads in a row, some optimizations have been made. One such optimizations is using a temporal buffer to store the bits needed for *gather* so for example if the process gets interrupted, the already loaded bits can be preserved and easily retrieved. Downfall shows that this buffer might leak data between threads when they run on the same core. This leaked data may come from a victim thread that populates a shared temporal buffer, the *Vector Physical Register File*. An attacking thread can then perform a faulting *gather* on, e.g., uncacheable memory, which might cause the

CPU to transiently forward data from the temporal buffer to dependent instructions. GDS is exemplified in Listing 1.1, where a transient window is created, followed by executing a faulting *gather*, which might end up acquiring data from the victim thread. This version is heavily based on Downfall [5, Listing 2], and a modified version of the proof-of-concept code [1].

Listing 1.1. Gather Data Sampling.

```
    # Wipe out noisy values
    .rept 32 # Repeat 32 times
    inc %rax
    vmovups (%rdi), %ymm3
    .endr
    vpxor %ymm1, %ymm1, %ymm1       # Zero index vector
    vpcmpeqb %ymm2, %ymm2, %ymm2    # Ones mask vector
    vmovups (%rdi), %ymm3           # Load word permutation
    vpxor %ymm4, %ymm4, %ymm4       # Zero output vector

    # Step 1: Increase transient window
    # Done using a cache miss and a page fault
    lea known_address, %rdi
    clflush (%rdi)
    mov (%rdi), %rax
    xchg %rax, 0(%rdi)
    mov $0, %rdi
    mov (%rdi), %rax

    # Step 2: Gather with a fault
    mov $0, %r13
    vpgatherqq %ymm2, 0(%r13, %ymm1, 1), %ymm4

    # Step 3: Permute and encode to cache
    vpermq %ymm4, %ymm3, %ymm4      # Permute ymm4
    encode_ymm4                     # Macro for encoding ymm4
```

When the CPU attempts to flush the faulting gather instruction, it realises it forwarded the wrong data, and now has to erase all traces of the transient instructions accordingly. This means the transient data cannot be stored in memory, instead it must be transmitted using a covert channel. For this purpose, a simple Flush+Reload is used.

Cache as a covert channel. To leak 1 byte of transient data, the attacking thread can allocate an array of size 256 x 4096 bytes, or one memory page for each possible value. During a run of GDS where the program might access a transient byte, it makes a memory lookup into exactly one page based on the transient value, which is now the only page in cache. After the transient window closes, the program can then Flush+Reload all pages and infer the byte based on which page is cached. This approach can be scaled according to how many transient bytes are accessible, by simply making another 256 pages for each byte, and leaking them in order. An example for 1 byte can be seen in Listing 1.2.

[1] https://github.com/flowyroll/downfall/blob/main/POC/gds_aes_ni/asm.S

Listing 1.2. Cache as a covert channel.

```
int CACHE_MISS = 230;     // Setup threshold for cache misses
char oracle[256 * 4096];  // Initialise oracle of 256 pages

// A transient window is made and gather is called
...
// Load one page based on transient byte from gather
oracle[transientByte * 4096];
// Flush+Reload checking the first byte in each page for a hit
for (int i = 0; i < 256; i++)
    int reloadTime = flush_reload_t(oracle[i * 4096]);
    if (reloadTime < CACHE_MISS)
        printf("Successfully got byte %i", i);
```

The array is indexed in strides of pages to avoid the *prefetcher*, which works on data within the same page [4]. If performed within a single page, cache hits on consecutive bytes would be observed, and then provoke false positive hits.

4 Reproducing Downfall

Our first task was finding a vulnerable CPU and attempting to minimally reproduce the results in the Downfall paper, as to achieve sufficient conditions to proceed with the attack. We initially performed some experiments using known data to see which settings we should apply to GDS to leak efficiently in our setup. This was done using modified versions of the code provided in the repository[2]. We were mainly interested in figuring out for our specific machine:

- Which victim-side vulnerable instructions leak consistently?
- Which attacker-side *gather* leaks consistently?
- Which victim-attacker setup leaks the most?

A simplified version of the attacker's code using various *gather* instructions can be found in our repository. For the victim, we simply made it repeatedly load known data using different `vmov` or `vpgather` instructions.

4.1 Initial results

We choose to run attack and victim code in two different processes, where the victim runs as the main process on a core, and the attacker as a sibling thread on the same core. This proved to leak more consistently than running on the same logical core and relying on context-switches. Table 1 presents the results. For what instructions proved most to leak the most consistent, we found that `vpgatherqq` was the instruction leaking the most for the attacker, while `vmov` instructions leaked the most for the victim among the evaluated instructions. Furthermore, we found that we were able to consistently leak up to 12 consecutive bytes, and at a rate of about 100 entries/second. For unknown reasons, we were not able to leak up to 22 bytes, as originally claimed in the Downfall paper.

[2] https://github.com/flowyroll/downfall/tree/main/POC/gds_test

Table 1. Combinations of victim/attacker instructions and their hits in Downfall leakage, measured in correct QWORDs leaked per second.

Victim	Attacker	Result
`vpgatherdd`	`vpgatherqq`	30
`vpgatherqq`	`vpgatherqq`	15
`vmovdqa`	`vpgatherqq`	105
`vmovdqu`	`vpgatherqq`	120
`vmovdqu`	`vpgatherdd`	90
`vmovups`	`vpgatherqq`	100

An example of leaked data in 8-byte segments can be seen in Listing 1.3. We choose to split the leaks based on their QWORD index into the loaded `ymm` register, In the example, the victim code repeatedly loads one `ymm` register of known data, represented in hexidecimal as `30313233...4c4d4e4f`. The attacker executes GDS and repeatedly rotates the transient data using `VPERMQ` to access all QWORDs, which are then sorted into Q1-Q4 by index. We call each line in the leak a *segment*, and the number after the leaked data is the number of hits that particular segment received. Leakage of 12-byte segments looked similarly, with the extra 4-bytes from one index (e.g. Q1) almost always being found in the subsequent one (Q2).

Listing 1.3. Example leak of 8-byte segments in hexadecimal representation.

```
Q1                                  Q3
00 3031323334353637 475             00 4041424344454647 681
01 aaaaaaaaaaaaaaaa 38              01 415bc77b1c8dfeff 54
02 e1ffffffffffffff 49              02 ffffffffffffffff 45

Q2                                  Q4
00 0b93ac4623f8ffff 46              00 48494a4b4c4d4e4f 666
01 38393a3b3c3d3e3f 492             01 ffffffffffffffff 58
02 70f8ffffffffffff 42
03 aaaaaaaaaaaaaaaa 25
```

In general, the leaked data is easily distinguished from most noise, and reassembling an entire `ymm` register by brute force is feasible. This is analogous to the Downfall attack against AES, where the keys are short enough to fit in one `ymm` register. However, if the leaked data cannot fit in a single `ymm` register, this simple attack does not provide a way to reassemble the data correctly. For example, if the data is contained in four `ymm` registers, there would be four different 8-12 byte segments in each of Q1-Q4 with many hits, and one would have to try every single combination of recovering four `ymm` registers based on them. Even for relatively few `ymm` registers, this would quickly prove to be infeasible, even though it can be performed offline during post-processing after initially leaking data. This lead to further experiments with the victim running several consecutive loads.

4.2 Finding natural patterns in the leakage

The next step was observing the patterns in leaked data across consecutive loads. When observing 12-byte segments, a lot of hits where consistently in the first 8 bytes of a register (e.g. Q1 in `ymm1`), and then the last 4 bytes were from a different register (e.g. Q2 in `ymm2`). We refer to this data pattern as a *link* in a *chain*. A simplified example with two consecutive loads can be found in Listing 1.4, illustrating the splitting pattern of 8 and 4-byte segments.

Listing 1.4. Example of leakage with chains.

```
# ymm1 = AAAABBBBCCCC...  ymm2 = 111122223333...
# An example of how 2 consecutive loads could chain together
# Simplified for the sake of clarity. Written as bytes, not hexadecimal
Q1                                  Q3
AAAABBBB3333                        EEEEFFFF7777
11112222CCCC                        55556666GGGG
Q2                                  Q4
CCCCDDDD5555                        GGGGHHHH1111
33334444EEEE                        77778888AAAA
```

When running just the simple victim and attacker, these patterns proved to consistently appear when the victim performed several loads in a row. Furthermore, if the victim executed the same amount of loads across different runs, the patterns would always be the same. The patterns also did not seem to appear in any of the noise, meaning the only 32-byte long chains were indeed from the data intended to leak. As it can be seen from the example, the two chains do not exactly correspond to the two `ymm` registers which were loaded, but in this simple case the original registers can easily be extracted once the pattern formed by the chains is known.

Reassembling multiple `ymm` registers worth of data into their original order becomes feasible by first collecting the leakage from consecutive loads, observing the chains, and then fitting the data to the specific observed patterns. This was observed to be consistent across different amounts of loads, and did not require any modification or prior knowledge about the data itself. However, the *order* in which the `ymm` registers were loaded is not preserved and needs to be recovered by other means.

5 Attacking Kyber with MDS

In this section, we use the Downfall exploit to steal private keys from a high-speed implementation of secure post-quantum cryptography. We have chosen Kyber as a target since it was recently standardized by NIST under the name ML-KEM [6]. The attack assumes that a static version of Kyber is used, such that one key pair will be generated and used to encrypt and decrypt multiple times. This is a realistic use in cryptographic protocols where both parties cannot be present simultaneously to execute key exchange (i.e. e-mail encryption) and in setups where you do not want to generate new keys each time.

The attack has an online and an offline phases. The online phase corresponds to leaking temporal buffers with GDS, and the offline phase corresponds to generating and testing candidate private keys until a proper match is found.

5.1 An overview of Kyber

Kyber is a family of post-quantum key-encapsulation mechanisms (KEM) based on hardness assumptions over lattices. It allows two parties to establish a session key to encrypt follow-up communication, coming with standard algorithms such as `KeyGen`, `Encrypt`, `Decrypt`, `Encapsulate`, and `Decapsulate`. The latter two algorithms provide chosen-ciphertext security (CCA), and rely on inner chosen-plaintext-secure (CPA) algorithms through the Fujisaki-Okamoto transform.

Our attack targets the inner decryption procedure, which takes as input the private key *sk*. Private keys take 768, 1152 and 1536 bytes, depending on the standardized security level (respectively *Light*, *Standard*, and *Paranoid*). We stress that these key lengths are much longer than symmetric keys leaked in previous work, thus using the simple GDS attack would make reassembling the key significantly time-consuming. Since Kyber performs dense arithmetic over polynomials, optimized implementations may also employ vectorized instructions. The reference Kyber code contains such an implementation, accelerated with AVX2 instructions, which we target in the following sections.

5.2 Vulnerable code

Vulnerable instructions that handle part of the private key can be found in the `Decrypt` operation from the inner CPA portion, implemented in the function `indcpa_dec`. The vulnerable part was the sequence of instructions unpacking and loading the private key *sk* into vector registers. A simplified version of the call stack can be seen in Listings 1.5 and 1.6 below.

Listing 1.5. Simplified call stack for decryption.

```
void indcpa_dec(uint8_t *m, const uint8_t *c, const uint8_t *sk) {
    polyvec b, skpv;
    poly v, mp;
    unpack_ciphertext(&b, &v, c);
    unpack_sk(&skpv, sk);           // Unpacking the sk
    ...
}
static void unpack_sk(polyvec *sk, const uint8_t *packedsk) {
    polyvec_frombytes(sk, packedsk);
}
void polyvec_frombytes(polyvec *r, const uint8_t *a) {
    for(unsigned int i = 0; i < KYBER_K; i++)
        poly_frombytes(&r->vec[i], a+i*KYBER_POLYBYTES);
}
void poly_frombytes(poly *r, const uint8_t *a) {
    nttfrombytes_avx(r->vec, a, qdata.vec);
}
```

From Listing 1.6, the entire private key is loaded using 6 `ymm` registers at a time, which will be referred as *blocks*. For the *Light*, *Standard* or *Paranoid* security levels the private key will be split into 4, 6, or 8 blocks respectively.

Listing 1.6. Simplified excerpt from `shuffle.S` to unpack/load a private key.

```
nttfrombytes_avx:
    vmovdqa   112*2(%rdx),%ymm0
    call      nttfrombytes128_avx
    add       $256,%rdi
    add       $192,%rsi
    call      nttfrombytes128_avx
    ret

nttfrombytes128_avx:
    #load     %rsi  index into sk
    vmovdqu   (%rsi),%ymm1
    vmovdqu   32(%rsi),%ymm2
    vmovdqu   64(%rsi),%ymm3
    vmovdqu   96(%rsi),%ymm4
    vmovdqu   128(%rsi),%ymm5
    vmovdqu   160(%rsi),%ymm6
    ...
```

In our setting, the victim process generates a static private key at some security level and then repeatedly decrypts random messages. Initial experiments showed that one could *not* clearly distinguish parts of the key from noise by just counting the number of hits, which was the case in the Downfall experiments explained in Section 4. We hypothesize that this was caused by the fact that the victim now runs many other vector instructions for arithmetic, some of which are also vulnerable to GDS, and thus may populate the temporal buffer with additional noise during the attack.

Moreover, without making use of the patterns found from consecutive loads, in the best case with *Light* security we would expect 24 leaked segments in each of the Q1-Q4 groups. Having to assemble by brute force all possible combinations of leaked data would lead to $(24!)^4 \approx 2^{316}$ combinations corresponding to all permutations, which would not be feasible for the offline phase of the attack. Hence, being able to exploit leakage patterns corresponding to *blocks* is critical for reducing the number of permutations and thus the feasibility of the offline phase.

5.3 Using leakage patterns to recover the key

When leaking 12-byte segments from Kyber, we again observed patterns similar to our pure Downfall experiments. It also became clear that just permuting over QWORDs when running the attack was not sufficient with respect to reassembling an entire private key. As discussed in Section 4.2, it was possible to recover the `ymm` registers in the simple case, but not their order. This means that reassembling an entire private key from blocks would require to check all combinations of permutations of registers and blocks, which could quickly make the offline phase infeasible. Another limitation was that chains consisting of more than two consecutive loads often skip entire QWORDs, preventing them from being part of the chain. They would still appear as 8-byte segments, just not as 12-byte links. The specific leakage pattern observed for our target Kyber implementation can be seen in Figure 1.

The obstacle motivated another attack to be executed at the same time, which consists in permuting instead based on DWORD indexes using `vpermd`

and gathered using `vpgatherdd`. This once again produced patterns containing chains, but instead of 8-4 byte *links* between QWORD of the key (Q-chain), we saw 4-4-4 byte *links* between DWORDS of the key (D-chain). Listing 1.7 exemplifies this, where `mapQ` is leakage on QWORDs, and `mapD` on DWORDs. Note both attackers need to run in the same core as sibling threads, introducing a risk of increasing noise, but which did not affect our results.

Listing 1.7. Sample chains in `mapQ` compared to `mapD`.

```
# ymm1 = AAAABBBBCCCC...  ymm2 = 111122223333...
# An example of how consecutive 2 loads could chain together
# Simplified for the sake of clarity, written in byte representation (not hex)
# mapQ                                  # mapD
Q1                                      D1
AAAABBBB3333                            AAAA2222CCCC
11112222CCCC                            1111BBBB3333
Q2                                      D2
CCCCDDDD5555                            BBBB3333DDDD
33334444EEEE                            2222CCCC4444
Q3                                      D3
EEEEFFFF7777                            CCCC4444EEEE
55556666GGGG                            3333DDDD5555
Q4                                      ...
GGGGHHHH1111                            D8
77778888AAAA                            HHHH1111BBBB
                                        8888AAAA2222
```

Combining both sources of leakage fixed both of the limitations. It immediately fixed the problem of some QWORDs not appearing as a link in a chain as seen in Figure 1, since half of the missing QWORD would now appear as part of a link in a D-chain. Whilst this immediately only provides half of the missing data, fortunately the entire missing QWORD appears as a 8-byte segment in `mapQ`. This means the missing half of the QWORD can simply be looked up.

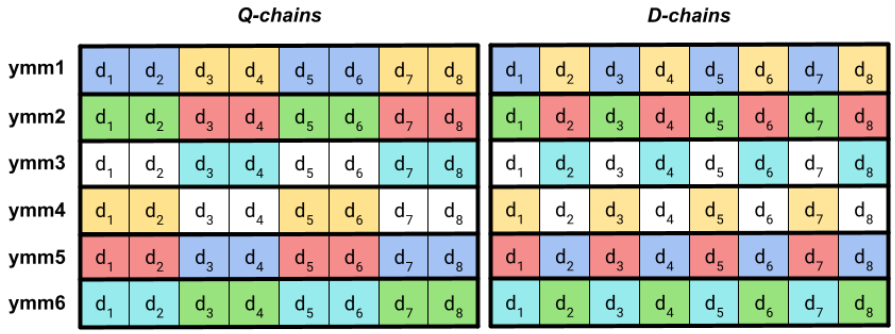

Figure 1. Block of 6 `ymm` registers with DWORDs coloured to show both *Q-chains* and *D-chains* observed in our leaks. On the left, the five *Q-chains* are coloured in, and on the right the five *D-chains*. In white are the QWORDs & DWORDs which do not appear in a chain, but whose patterns ends up looking like a chain.

The problem of ordering the newly constructed `ymm` registers within a block also gets adressed. We see that it is always the 1st and 3rd QWORD of `ymm3`, and 2nd and 4th QWORD of `ymm4` which are not part of a chain. After reconstructing `ymm4`, one can simply traverse the *Q-chain* starting from `ymm4`, and the order of the other `ymm` registers becomes clear.

In detail, the reassembly procedure starts by identifying `ymm4`. This can be done by finding a *Q-chain* involving $q_1 = d_1|d_2$, where d_1 is in some *D-chain*, but d_2 is not. By following the yellow *D-chain* involving d_1 from Figure 2, values d_1, d_3, d_5, and d_7 of `ymm4` can be obtained. The remaining values d_2, d_4, d_6, and d_8 of `ymm4` can be learned by performing lookups in *mapQ* for 8/12-byte segments that start with d_1, d_3, d_5 and d_7, recovering the full `ymm4` register.

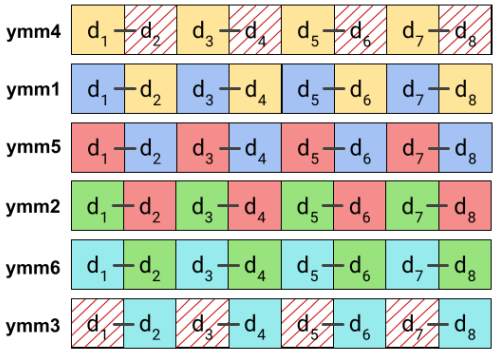

Figure 2. The order we assemble the `ymm` registers into blocks. *D-chains* are coloured. Lines link DWORDs that appear together as QWORDs in `mapQ`.

Since `ymm4` shares the yellow *D-chain* with `ymm1`, half the bytes of `ymm1` can be obtained, namely d_2, d_4, d_6, and d_8. The procedure continues by looking for a *Q-chain* where $q_1 = X|d_2$ and $q_3 = Y|d_6$ to learn $d_1 = X$ and $d_5 = Y$. From the blue *D-chain* that d_1 and d_5 appear in, values d_3 and d_7 can be obtained, now recovering the entire `ymm1` register.

We repeat the process for `ymm5`, `ymm2`, and `ymm6` to obtain the red, green, and turquoise *D-chains*. For the remaining d_1, d_3, d_5 and d_7 in `ymm3`, a lookup is performed in `mapQ` to find 8/12-byte segments that end with d_2, d_4, d_6, and d_8 from the turquoise *D-chain*. A final validation check is performed for `ymm3`, such that a *D-chain* involving its d_1, d_3, d_5 or d_7 does not exist, to ensure that this is a valid candidate for `ymm3`.

5.4 Implementation and optimization

We implemented the reassembly of candidate private keys in Python, using the ideas from the previous subsection. Moreover, we also optimize the number of blocks and reduce the number of permutations of blocks that could be valid private keys.

Filtering. Leakage data contains easily identifiable noise that can be filtered out, such as repetitions of the same bytes (as in `FFFFA1FF...`). Since the private key is randomly sampled, the segments with many repeated bytes can be filtered out to reduce the number of potential chains. Other examples of recurrent noise were identified by running the attack multiple times using different pairs of keys and messages and looking for common segments. The reasoning is that these segments would not be dependent on the key but instead generated every time, e.g. by vulnerable instructions using hardcoded constants.

Gaps in chains. Some parts of the private key might appear only in 8-byte segments if the attack has not run for long enough. Several 12-byte segments are needed to construct the chains, but it might still be contructed in the presence of missing links. For each *Q-chain*, exactly one missing 12-byte link can be tolerated, since we do not need the chain to loop around itself. We still need the missing link to appear as an 8-byte segment, but do not require the trailing DWORD that leads back to the start of the chain.

Even more missing links can be tolerated for each *D-chain*. We observe that each DWORD can appear three times in `mapD`, potentially as the first, middle, or last DWORD of some 12-byte segment. This means we can tolerate that either several gaps of one missing link or one gap of two missing links in a row for each *D-chain*. There is enough overlap to connect the links on either side when one link is missing. If we observe $d_1|d_2|d_3$ and $d_3|d_4|d_5$, but $d_2|d_3|d_4$ is missing, then we can jump the gap and still find the entire chain. We cannot extend the chain further for a gap of two missing links but can tolerate it at the end of the chain, analogous to the problem seen with *Q-chains*.

When assembling both *Q-* and *D-chains*, we attempt to construct chains starting in every map and rotate the found chains to always begin with elements from the first map. This ensures that even if a gap exists in the middle of a chain, the rotation will be attempted where the gap is at the end. This resulted in reduced running time since some missing 12-byte segments can be tolerated.

Verifying blocks. When reconstructing the blocks, we used every correct *D-chain*, but only some of the correct *Q-chains*. We assume all the correct *Q-chains* are present in the leak and use them for verification. Since we know the entire structure of a block, we can infer what *Q-chains* that block should have produced in the leak. We assume the block is invalid if any of these chains are missing in our found *Q-chains*. This is a trade-off that makes our runtime slightly longer but filters out several incorrect blocks.

Usually, some of the blocks we now produce will be similar, deviating only by some flipped bits [3]. We observed that for the valid blocks that will be part of the private key, the flipped bits will be in `ymm3` and/or `ymm4`. This is because some of these bits do not appear in the *Q-chains*, which we use to validate. Because of this, we decided to group blocks based on their first QWORD. Then we only

[3] We could not determine the reason for this effect.

need to attempt the permutations where we take at most one block from each group, under the assumption no two almost identical blocks can appear in the same private key.

Complexity. Since we now (theoretically) know the entire blocks in full, all we have to do is brute force the order of the blocks. If we assume all blocks are correct, the complexity of this process for *Paranoid* security becomes $8! \approx 2^{16}$. In practice, when running the online phase long enough and the offline phase with the aforementioned optimizations, we never saw more than twice the expected amount of blocks. This means, even for *Paranoid*, we only expect to check $\frac{16!}{8!} \approx 2^{29}$ different permutations in the worst case.

Checking the candidate private keys. To finalise the attack, we also had to verify that we could find and reassemble the correct private key among the candidates within a feasible time span. This is performed by generating messages, encrypting them using the public key, and attempting to decrypt the ciphertext to check if the original message is returned.

6 Experimental results

The complete attack code can be found in our repository [4]. The initial Downfall experiments can be found inside the `AllTests` folder. The data corresponding to GDS attacks against Kyber can be found in `KyberAttack` folder. The experiments were executed on a laptop with an i5-11300H CPU, microcode version 0x86. This CPU model was released before Intel released their patches for GDS. We employed Ubuntu 22.04.3 LTS running kernel version 5.15.146.1-2, within WSL version 2.1.5.0. The virtual machine had access to 2 logical cores, 0 and 1, with both located on the same physical core.

We tested the attack for each of the security levels *Light*, *Standard*, and *Paranoid*, since they all require private keys of different lengths, which incur increasing difficulties for the attacker. For each of the security levels, a total of 10 key pairs were tested. All results were run with the attackers compiled using `-O3` optimization, seeing as this gave better throughput. Our methodology consisted of starting the victim process and increasing the time allowed to the attacker by intervals of 5 minutes until the private key could be fully recovered, with a cut-off point at the success rate of 60%. Table 2 contains the experimental results, consisting of the time needed to obtain at least 60% success rate in 10 attempts to recover a full Kyber private key at a certain security level. Furthermore, we note that there was partial success in the failed attempts, with only one or two blocks missing. The offline phase to check candidates by testing permutations ran almost instantaneously in all cases, given the optimizations described previously.

[4] https://gitlab.au.dk/au685153/downfall-bachelor-project

Table 2. Overview of required running time for the different security levels. The success rate is the fraction of 10 Kyber keys fully recovered in a given time limit.

Security Level	Time	Success Rate
Light	30 min	70%
Standard	35 min	70%
Paranoid	40 min	60%

7 Discussion

We compare our attack strategy with two other MDS attacks from the literature.

The RIDL cross-process attack assumes the attacker already knows some part of the data it attempts to leak, such as the string `root:` in the `/etc/shadow` file. This is not possible in our case, since no information about the private key is known *a priori*. Reassembling the key using the *mask-subtract-rotate* technique is thus not feasible, motivating our custom procedure based on cross-referencing *Q-chains* and *D-chains*.

ZombieLoad relies on the fact that the entire key may be present in the transient domain, which means the first 16 bytes leaked are from the key or unrelated noise. Since only one byte is leaked at a time at any index, the attacker can choose between leaking an entire byte of the key or the domino between two bytes of the key for checking correctness.

In our case, it is not clear how to create domino bytes efficiently, because we only obtain a low volume of correct 8- to 12-byte leaks serving as links. Perhaps it is possible using the DWORD setup with `vpgatherdd` and `vpermd`, with the domino bytes contained within 8-byte leaks. While it might end up helping optimizing the attack, it would still not help with reassembling the `ymm` registers within a block in the right order, since the missing QWORDS are used for this. A working combination would thus require 8-byte DWORD links cross-reference with QWORDs from `vpgatherqq`, so both attacks would still be required, but not having to handle 12-byte leaks would make the online phase simpler.

Another important difference in the comparison with ZombieLoad is that our attack is less active, not having to craft and transmit domino bytes, but just monitoring GDS leakage until enough blocks of the private key can be found. Our attack also does not require the whole key to be present at the same time, being capable of chaining different segments found across many iterations of the attack. At a higher level, both attacks are essentially exploiting redundancy at different granularities, overlapping nibbles in the case of ZombieLoad; and chaining patterns in our cross-referencing approach.

In terms of performance, our attack takes a long time to complete, even considering the much longer keys it targets. The main reason behind this is the amount of noise from vector arithmetic instructions in Kyber, and polluting the internal buffers. However, we do not consider that a strong limitation in the static setting. It is unclear if other MDS attacks would perform better under the constraints considered in our paper.

8 Conclusion

We presented a key recovery attack against Kyber in the static setting. By weaponizing GDS from Downfall, we were able to fully recover Kyber keys from microarchitecture leakage with success rate between 60% and 70% in under 40 minutes for all security levels. The main idea consists in running two GDS attackers at different granularities, and cross-referencing fragments of the key until good candidates and a proper match can be found. In comparison to previous work, the proposed attack is able to leak much longer keys in an entirely passive manner by just observing GDS leakage across a long time interval. This naturally comes at the cost of performance.

Future work in this direction could involve attempting different MDS attacks against Kyber or optimized versions of the attack that are more aggressive, such as adapting the idea of transmitting domino bytes to our setting. We expect that our attack can also be effective against other quantum-safe cryptographic algorithms with longer private keys.

Acknowledgments. We thank Daniel Moghimi for clarifying questions about Downfall, and the anonymous reviewers for the constructive feedback. The authors have no competing interests to declare that are relevant to the content of this paper.

References

1. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In: EuroS&P. pp. 353–367. IEEE (2018)
2. Chen, B., Wang, Y., Shome, P., Fletcher, C.W., Kohlbrenner, D., Paccagnella, R., Genkin, D.: Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In: USENIX Security. USENIX Assoc. (2024)
3. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: exploiting speculative execution. Commun. ACM **63**(7), 93–101 (2020)
4. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., Strackx, R.: Meltdown: reading kernel memory from user space. Commun. ACM **63**(6), 46–56 (2020)
5. Moghimi, D.: Downfall: Exploiting speculative data gathering. In: USENIX Security Symposium. pp. 7179–7193. USENIX Association (2023)
6. National Institute of Standards and Technology: FIPS 203 – Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf (2024)
7. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-Flight Data Load. In: IEEE Symposium on Security and Privacy. pp. 88–105. IEEE (2019)
8. Schwarz, M., Lipp, M., Moghimi, D., Bulck, J.V., Stecklina, J., Prescher, T., Gruss, D.: Zombieload: Cross-privilege-boundary data sampling. In: CCS. pp. 753–768. ACM (2019)
9. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security. pp. 719–732. USENIX Assoc. (2014)