Observations on the performance of PQ KEMs

Nir Drucker
University of Haifa, Israel,
and
Amazon, Seattle, USA
drucker.nir@gmail.com

Shay Gueron
University of Haifa, Israel,
and
Amazon, Seattle, USA
shay@math.haifa.ac.il

***Abstract*—This note discusses two aspects of the performance of Round-2 KEM candidates: a) the impact of Simultaneous MultiThreading (SMT); b) the balance between encapsulation and decapsulation.**

- Software performance can sometimes be improved by parallelization of tasks. In some cases this can be achieved by simultaneous execution on logical CPUs (also known as SMT). Since such a technology opens the door to possible security vulnerabilities, its overall benefit needs careful evaluation. We evaluate the hyper-threaded performance of some of the Round-2 KEM candidates proposed to the NIST Post Quantum Cryptography project.

- The common assumption is: that slow decapsulation is performed on a (strong) server side and the weaker client platforms execute the (faster) encapsulation. We argue that this is not necessarily the case in TLS 1.3, which is now suggested as the next generation of secure communication protocols and discuss the implications.

I. Performance measurements on platforms with SMT

SMT is a technology that allows two software threads to run on two logical processors. These processors are associated with a single physical CPU and share its hardware resources. This technology is designed to improve tasks parallelization and therefore the overall throughput. In particular, it targets high-end servers. Intel's processors support this technology [1] (called Hyper-Threading (HT)) and other processors provide their own SMT solutions (e. g., AMD [2] and ARM [3]). In general SMT solutions introduce some security challenges. For example, enabling HT on Debian's systems [4] caused unpredictable behavior. This bug is captured in Intel's erratum [5], [6]. Other examples are security attacks that leverage the HT feature such as the TLBLeed attack [7], the L1 Terminal Fault (L1TF) attack [8], [9] and the PortSmash attack [10].

As a result it is important to carefully evaluate the potential performance gains of SMT-enabled platforms that execute cryptographic code. This information is typically not provided by current benchmark environment that measure primitives run on a single core. For example, SUPERCOP recommends to "Turn off hyperthreading" in order to reduce the randomness of the results [11].

A. Evaluation experiments and results

a) Platform characteristics.: Our experiments run on a platform equipped with the 7^{th} Intel®Core^{TM} Generation (Micro-architecture Codename "Kaby Lake" [KBL]) 3.60 GHz Core^{TM} i$7-7700$. This platform has 16 GB RAM, 32K L1d and L1i cache, 256K L2 cache, and $8,192$K L3 cache, where the Enhanced Intel Speedstep® Technology was disabled and the Intel® Turbo Boost Technology was turned off. Of course, the HT technology was enabled. This platform has 4 physical cores and 8 logical cores.

b) Compilation and OS.: We carried out the experiments on Linux platform (Ubuntu 16.04 LTS) and the code was compiled with GCC in 64-bit mode, using the "-O3" optimization level. In addition, we used the following compilation flags: `-ggdb -maes -mavx2`.

c) The benchmarked schemes.: Our experiments cover the following KEMs (giving in alphabetic order): BIKE [1], FrodoKEM [2], NewHope

[1]Code from additional implementation of BIKE1 [12] with AVX2 and OpenSSL.

[2]Code from [11] dir: `supercop-20190110/crypto_kem/frodokem640/x64/`.

[3], SIKE [4], and Three Bears [5]. We chose these schemes for our experiments because they represent a wide variety of KEM approaches and they vary in their performance. They are also easy to port to our benchmarking environment. We point out that some schemes are not written to be thread-safe (e. g., SABER and LAC, because they use global variables) therefore we could not benchmark them. For the benchmarked schemes we took their additional (AVX2) implementation if such exists, and evaluated the variant with the lowest set of security parameters (the fastest option).

d) Methodology.: We setup our experiments to run the same function on a multiple number of threads $(1/2/4/8/16)$. Recall that the number of physical cores is 4 and the number of logical cores is 8. The processor capabilities are fully utilized when the number of software threads is 8 and above.

Algorithm 1 describes the basic test (called generically F). We used Algorithm 2 to evaluate the number of cycles consumed by a function F. Every function runs 25 times (warm-up), followed by 100 iterations that are clocked (using the $RDTSC$ instruction) and averaged. To minimize the effect of background tasks running on the system, each such experiment is repeated 10 times, and the minimum result is recorded. Note that the warm-up phase has less impact when hyper threading is involved because context switches are possible. However, we still use it in order to have the same methodology for single thread (where warm-up is important) and multi-thread experiments.

Algorithm 1 Single thread function F

```
1: procedure F
2:     (sk, pk) = keygen()
3:     (ss_e, ct) = encaps(pk)
4:     (ss_d) = decaps(ct, sk)
5:     if ss_d ≠ ss_e then abort
```

For multi-threaded environments, two logical processors (associated with a single physical processor) can execute the same code and thus reduce the number of cache misses or execute different code, which can increase cache misses. Therefore, in our experiments we distinguish between two cases: a) performing key generation (keygen) and encapsulation (encaps) once and then run the MEASURE function on the decapsulation (decaps) functionality (Algorithm 3); b) run MEASURE for each function keygen/encaps/decaps separately (Algorithm 4). We use the same methodology as in (a) to measure only the keygen function. Note that servers execute the keygen and decaps parts of the KEM for every connection (we assume here that forward-secrecy is enforced and therefore the keys are ephemeral). The results are given in Table II (for option (a)), and Table I (for option (b)).

In our evaluation process we observed a variability in the measured results. This is due to the time it takes to create and destroy the threads which may be significant relative to the run-time of F. Therefore,

Algorithm 2 Measuring function

```
1: procedure MEASURE(F)
2:     min = ∞
3:     for i = 1 to 25 do
4:         eval(F)                          ▷ Warm-up the CPU caches
5:     for i = 1 to 10 do
6:         total = 0
7:         for i = 1 to 100 do
8:             begin = rdtsc
9:             eval(F)
10:            end = rdtsc
11:            total += end - begin
12:        avg = total/100
13:        if min < avg then
14:            min = avg
15:    return min
```

Algorithm 3 Only decaps is measured in F (called Fdec)

```
1: procedure FDEC
2:     (sk, pk) = keygen()
3:     (ss_e, ct) = encaps(pk)
4:     MEASURE((ss_d) = decaps(ct, sk))
5:     if ss_d ≠ ss_e then abort
```

Algorithm 4 keygen/encaps/decaps are measured in F (called Fall)

```
1: procedure FALL
2:     MEASURE((sk, pk) = keygen())
3:     MEASURE((ss_e, ct) = encaps(pk))
4:     MEASURE((ss_d) = decaps(ct, sk))
5:     if ss_d ≠ ss_e then abort
```

[3]Code from the additional implementations of NewHope [12] dir: `avx2/crypto_kem/newhope512cpa/`.

[4]Code from the additional implementations of SIKE [12] dir: `x64/SIKEp434/`, compiled with `_MULX_` and `_ADOX_` set to TRUE.

[5]Code from the additional implementations of Three Bears [12], dir: `/With_Asm/crypto_kem/BabyBear/`

TABLE I

MULTI-THREADED RUNS OF ALGORITHM 4 (FALL). THE ROWS ARE SORTED BY THE NUMBER OF THREADS AND THEN BY THE NUMBER OF CYCLES OF THE BASELINE FALL ON A SINGLE THREAD.

#Threads	KEM	Func	Cycles of Fall (baseline)	AVG SlowDown per thread	Max SlowDown per thread
8	NewHope	keygen	78,983	1.5	1.8
8	Three Bears	keygen	80,113	1.68	1.92
8	BIKE	keygen	439,776	1.66	1.79
8	FrodoKEM	keygen	1,350,579	1.54	1.57
8	SIKE	keygen	6,767,775	1.93	1.93
8	NewHope	decaps	19,042	1.37	1.82
8	Three Bears	decaps	174,166	1.14	1.91
8	FrodoKEM	decaps	1,927,762	1.3	1.59
8	BIKE	decaps	3,399,593	1.6	1.68
8	SIKE	decaps	11,818,253	1.92	1.93
16	NewHope	keygen	78,983	1.75	1.81
16	Three Bears	keygen	80,113	1.91	1.92
16	BIKE	keygen	439,776	2.62	3.76
16	FrodoKEM	keygen	1,350,579	2.59	3.24
16	SIKE	keygen	6,767,775	3.41	3.84
16	NewHope	decaps	19,042	1.56	1.82
16	Three Bears	decaps	174,166	1.68	1.92
16	FrodoKEM	decaps	1,927,762	1.9	3.22
16	BIKE	decaps	3,399,593	1.91	3.02
16	SIKE	decaps	11,818,253	2.58	3.47

TABLE II

MULTI-THREADED RUNS OF ALGORITHM 3 (FDEC). THE ROWS ARE SORTED BY THE NUMBER OF THREADS AND THEN BY THE NUMBER OF CYCLES OF THE BASELINE FDEC ON A SINGLE THREAD.

#Threads	KEM	Func	Cycles of Fdec (baseline)	AVG SlowDown per thread	Max SlowDown per thread
8	NewHope	keygen	19,042	1.57	1.81
8	Three Bears	keygen	174,166	1.57	1.92
8	FrodoKEM	keygen	1,927,762	1.47	1.56
8	BIKE	keygen	3,399,593	1.36	1.75
8	SIKE	keygen	11,818,253	1.88	1.93
8	NewHope	decaps	19,042	1.38	1.67
8	Three Bears	decaps	174,166	1.29	1.92
8	FrodoKEM	decaps	1,927,762	1.41	1.59
8	BIKE	decaps	3,399,593	1.59	1.67
8	SIKE	decaps	11,818,253	1.83	1.92
16	NewHope	keygen	19,042	1.63	1.84
16	Three Bears	keygen	174,166	1.67	1.92
16	FrodoKEM	keygen	1,927,762	1.66	2.52
16	BIKE	keygen	3,399,593	1.7	2.86
16	SIKE	keygen	11,818,253	2.61	3.72
16	NewHope	decaps	19,042	1.46	1.69
16	Three Bears	decaps	174,166	1.57	1.92
16	FrodoKEM	decaps	1,927,762	1.92	2.83
16	BIKE	decaps	3,399,593	1.92	2.77
16	SIKE	decaps	11,818,253	2.8	3.66

we replaced F with Fall (Algorithm 4). The final results are given in Table III.

Let $t \in \{1, 2, 4, 8, 16\}$ be the number of threads. Denote by τ_t the overall number of measured cycles divided by $(t/4)$ and let $\bar{\tau}_t = \frac{\tau_t}{\tau_1}$. This value is used in Table III. We point out that on a CPU with 4 physical cores, $\tau_1 = \tau_2 = \tau_4$ and thus Table III considers only the ratio between τ_8, τ_{16} and τ_4 (by dividing with $(t/4)$).

TABLE III
THE LATENCY OF ALGORITHM 4 RUNNING ON $t = 4/8/16$ THREADS DIVIDED BY t AND BY THE SINGLE THREAD LATENCY ($\bar{\tau}_t$).

KEM	#Threads (t)	$\bar{\tau}_t$
BIKE	4	1.00
BIKE	8	0.87
BIKE	16	0.87
FrodoKEM	4	1.00
FrodoKEM	8	0.86
FrodoKEM	16	0.86
NewHope	4	1.00
NewHope	8	1.17
NewHope	16	0.97
SIKE	4	1.00
SIKE	8	0.97
SIKE	16	0.97
Three Bears	4	1.00
Three Bears	8	1.12
Three Bears	16	1.01

II. TO ENCAPSULATE OR NOT TO DECAPSULATE - THIS IS THE QUESTION

Evaluating the performance of a KEM is not straightforward because one needs to consider, for example, the platforms (e. g., servers, client-desktops, FPGAs, or IoT devices), the compilers (e. g., GCC, Clang, ICC) and the number of threads (e. g., single (isolated) thread or runs in a multi-thread environment). Table IV summarizes the speedups of several Round-2 candidates when running on two platforms:

- P1: with an Intel Core i7-7800X processor (amd64; SL+512x2 (50654); 2017 Intel Core i7-7800X; 6 x 3500MHz; oki, supercop-20181123).
- P2: with a Cortex A57 processor (aarch64; Cortex-A57 (418fd071); 2015 NVIDIA Tegra X1; 4 x 1734MHz; jetsontx1, supercop-20180818).

The raw measurements values are taken from SUPERCOP [11] (median values). The table shows the results for the Round-2 candidates that are reported by SUPERCOP on P1, P2 (and are not marked there in "red"). The table accounts only for the fastest variant of every candidate. The table shows speedups of 2-3× or more when running on P1.

*a) Which KEM primitive (*encaps *or* decaps*) matters more?:* KEMs are designed for integration in security protocols such as TLS and IKEv2. Today, there are several suggestions that describe a hybrid model that combines a quantum-secure KEM with a classical key exchange scheme (e. g., ECDSA). For example, a hybrid key exchange for TLS 1.2 is described in [13], for TLS 1.3 in [14], [15], and for IKEv2 in [16]. These reports assign different roles to the client and the server sides. In [13] the server generates the keys and decapsulates the ciphertext. The client only perform the encapsulation. By contrast, [15][Section 3.3] proposes that the client would generate the keys and perform the decapsulation. This situation is important in TLS 1.3, where the client initiate the session and expects to be able to send/receive data after only 1-RTT.

Table V presents the ratio between cycles count of the keygen+decaps (or only decaps) and the cycles count of encaps, measured on platforms P1 and P2. The top part of the table refers to the 8 KEMs that are mentioned in Table IV. The bottom part introduces 6 more KEMs for which SUPERCOP reports measurements only on P1. We see diverse results for the ratio decaps/encaps between 0.3 and 17.24. Similarly, the results of (keygen+decaps)/encaps are diverse, although over a wider range. We point out that in all schems (except for babybearephem) the decapsulation is slower than the encapsulation.

Our assumption is that TLS 1.3 is the long term solution that will replace TLS 1.2 and therefore consider scenarios where the server performing the encapsulation and the client performs the key generation and decapsulation. To support forward secrecy, which is a standard required property of key exchange already today, the client will have to

TABLE IV
THE RATIO BETWEEN THE PERFORMANCE ON PLATFORMS P1 AND P2 (SEE EXPLANATION IN THE TEXT) [11]. VALUES GREATER THAN 1 INDICATE BETTER PERFORMANCE ON THE FORMER PLATFORM.

KEM	keygen	encaps	decaps
babybearephem	3.5	3.1	2.2
frodokem640	2.9	3.2	3.2
kyber512	3.6	3.7	4.5
lightsaber	1.4	1.8	2.3
newhope512cca	2.3	2.2	2.4
ntruhrss701	129.4	59.7	95.0
ntrulpr4591761	247.3	266.0	280.2
sntrup4591761	7.0	236.9	331.6

use ephemeral keys i. e., execute keygen in every session. For the server this implies that a typical workload will include multiple encapsulations. This raises two questions: a) whether a future such server should enable HT? and what is the expected performance gain; b) should the designs of new KEMs be focused on optimizing the encapsulation or the key generation + decapsulation. This may affect the way by which NIST evaluate the various KEMs. The McEliece scheme is a clear example, where keygen is extremely slow but encaps and decaps are very fast.

TABLE V
THE PERFORMANCE RATIO BETWEEN keygen+decaps (KG+DEC) OR decaps (DEC) AND encaps (ENC) RUNNING ON PLATFORMS P1, P2 [11]. THE SECOND PART OF THE TABLE INCLUDES ROUND-2 CANDIDATES THAT ONLY RUN ON P1.

	Cortex A57		Intel Core i7-7800X	
system	$\frac{kg+dec}{enc}$	$\frac{dec}{enc}$	$\frac{kg+dec}{enc}$	$\frac{dec}{enc}$
babybearephem	1.1	0.3	1.1	0.5
frodokem640	1.7	1.0	1.7	1.0
kyber512	1.9	1.2	1.7	1.0
lightsaber	1.9	1.4	1.8	1.1
newhope512cca	1.8	1.1	1.7	1.1
ntruhrss701	20.1	3.0	9.8	1.9
ntrulpr4591761	2.0	1.5	2.0	1.4
sntrup4591761	7.0	3.0	138.6	2.1
bike3l1			17.94	17.24
lac128			1.76	1.31
lake1			9.76	4.34
ledakem12			49.13	14.48
locker1			9.68	4.62
sikep503			1.68	1.07

III. CONCLUSION

We provided an evaluation of some post quantum Round-2 KEM candidates in multi-threaded environment. We see that the latency is not affected when the number of threads is smaller or equal to the number of physical cores. However, when the number of software threads increases, the overall latency increases as well, but the throughput is changed only slightly ($0.86\times - 1.17\times$). On the other hand, the latency of running a function (F) on every thread increases by a factor of up to $2\times$ ($3\times$), when the number of threads is $t = 8$ ($t = 16$), respectively.

The task of measuring multi-threaded applications is more subtle than measuring an application that runs on a single thread. The reason is that for measuring such applications, one should take into consideration a larger set of parameters. For example, measuring two applications that use the same code may have less cache misses than applications that do not share their code. On the other hand, in case the same code is used and this code has excessive use of a single execution port in the CPU, the two applications may compete on this resource. Therefore, the way to have a consistent measurement is to synchronize the threads so that the same code would be repeatedly measured on every thread in parallel to the other threads. However, this seems to be impractical.

Our reported measurements describe a scenario where a server only performs a small number of cryptographic functions with a small code foot print. However, in other usages the code of the threaded application is relatively large. In such cases the shared CPU resources can be better utilized through SMT and performance can be expected to be improved. Accurate assessments can be obtained only according to a specific use-case.

Our measurements indicate that if a server is running only cryptographic primitives, enabling the HT technology is questionable. It provides almost the same throughput but with higher latency per thread. This does not seem to justify the risk involved with opening the door to security vulnerabilities that emerged from sharing CPU resources. Apparently, dedicated optimization of cryptographic code for hyper-threaded servers is an interesting direction. This is another parameter by which NIST can assess the efficiency of a KEM.

Table I shows that HT has higher efficiency for primitives with shorter latencies. In TLS 1.3 the server is supposed to execute the encapsulation therefore optimizing the encapsulation would have an impact on a server that uses HT.

Note that this design implies that the low resource client is expected to perform the heavier key generation and decapsulation routines. Forward secrecy forces keys to be ephemeral and therefore the key generation is repeated. This considerations need to be included in the set of criteria for the final selection of the standardized KEMs.

ACKNOWLEDGMENTS

This research was supported by BSF Grant 2018640 and by the Center for Cyber Law & Policy

at the University of Haifa, in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

REFERENCES

[1] Intel, "Intel® Hyper-Threading Technology," https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, 2002, last accessed 22 May 2019.

[2] AMD, "The "Zen" Core Architecture," https://www.amd.com/en/technologies/zen-core, 2018, last accessed 22 May 2019.

[3] ARM, "CPU CORTEX-A65AE," https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a65ae, 2018, last accessed 22 May 2019.

[4] H. de Moraes Holschuh, "[WARNING] Intel Skylake/Kaby Lake processors: broken hyper-threading," https://lists.debian.org/debian-devel/2017/06/msg00308.html, June 2017, last accessed 22 May 2019.

[5] Intel, "Intel® Core™ X-Series Processor Family, Section SKZ6," https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/6th-gen-x-series-spec-update.pdf, February 2019, last accessed 22 May 2019.

[6] ——, "6th Generation Intel® Processor Family, Section SKL150," https://www.intel.com/content/www/us/en/products/docs/processors/core/desktop-6th-gen-core-family-spec-update.html, November 2018, last accessed 22 May 2019.

[7] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 955–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[8] Intel, "L1 Terminal Fault / CVE-2018-3615 , CVE-2018-3620,CVE-2018-3646 / INTEL-SA-00161," https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault, August 2018, last accessed 22 May 2019.

[9] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[10] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. Garca, and N. Tuveri, "Port contention for fun and profit," Cryptology ePrint Archive, Report 2018/1060, 2018. [Online]. Available: https://eprint.iacr.org/2018/1060

[11] D. J. Bernstein and T. L. (editors), "eBACS: ECRYPT Benchmarking of Cryptographic Systems, SUPERCOP," https://bench.cr.yp.to/supercop.html, 2018, last accessed 22 May 2019.

[12] NIST, "Post-Quantum Cryptography - Round 2 Submissions," https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions, 2019, last accessed 22 May 2019.

[13] M. Campagna and E. Crockett, "Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)," Internet Engineering Task Force, Internet-Draft draft-campagna-tls-bike-sike-hybrid-01, May 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid-01

[14] W. Whyte, Z. Zhang, S. Fluhrer, and O. Garcia-Morchon, "Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3," Internet Engineering Task Force, Internet-Draft draft-whyte-qsh-tls13-06, Oct. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls13-06

[15] D. Steblia and S. Gueron, "Design issues for hybrid key exchange in TLS 1.3," Internet Engineering Task Force, Internet-Draft draft-stebila-tls-hybrid-design-00, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-stebila-tls-hybrid-design-00

[16] C. Tjhai, M. Tomlinson, grbartle@cisco.com, S. Fluhrer, D. V. Geest, O. Garcia-Morchon, and V. Smyslov, "Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2)," Internet Engineering Task Force, Internet-Draft draft-tjhai-ipsecme-hybrid-qske-ikev2-03, Jan. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-tjhai-ipsecme-hybrid-qske-ikev2-03