KpqClean Ver2: Comprehensive Benchmarking and Analysis of KpqC Algorithm Round 2 Submissions

Minjoo Sim[1][0000−0001−5242−214X],
Siwoo Eum[1][0000−0002−9583−5427],
Gyeongju Song[1][0000−0002−4337−1843],
Minwoo Lee[2][0000−0002−2356−3055],
Sangwon Kim[2][0009−0000−6313−2236],
Minho Song[2][0009−0007−6277−0069], and
Hwajeong Seo[2][0000−0003−0069−9061]

[1]Department of Information Computer Engineering,
Hansung University, Seoul (02876), South Korea,
[2]Department of Convergence Security,
Hansung University, Seoul (02876), South Korea,
{minjoos9797, shuraatum, thdrudwn98, minunejip, kim3875, smino0906, hwajeong84}@gmail.com

Abstract. From 2022, Korean Post-Quantum Cryptography (KpqC) Competition has been held. Among the Round 1 algorithms of KpqC, eight algorithms were selected in December 2023. To evaluate the algorithms, the performance is critical factor. However, the performance of the algorithms submitted to KpqC was evaluated in different development environments. Consequently, it is difficult to compare the performance of each algorithm fairly, because the measurements were not conducted in the identical development environments. In this paper, we introduce KpqClean ver2, the successor to the KpqClean project. KpqClean ver2 provides comprehensive benchmark analysis results for all KpqC Round 2 algorithms across various environments (Ryzen, Intel, and aarch64). This framework includes both a "clean" implementation and an "avx2" implementation of the KpqC Round 2 candidate algorithms. To benchmark the algorithms, we not only removed external library dependencies from each algorithm but also integrated the same source code for common algorithms (such as AES, SHA2, SHAKE, and etc.) to enable more accurate performance comparisons. The framework automatically recognizes the user's environment, providing easy benchmarking for all users without the need for separate settings. This study also includes memory usage analysis using Valgrind for each algorithm and function usage proportion analysis during the execution of each cryptographic algorithm using Xcode's profiling tool. Finally we show that the practical strength of KpqC algorithms in terms of execution timing and memory usages. This result can be utilized for the understanding of KpqC finalist in terms of performance.

Keywords: Post-quantum Cryptography · Benchmark · KpqC Competition · Cryptography Implementation · Profile Analysis

1 Introduction

With the development of quantum computers, currently used encryption algorithms are facing significant threats. To overcome this challenge, a competition is being held to select a secure encryption algorithm that can be used as a standard in the quantum computer era.

The U.S. National Institute of Standards and Technology (NIST) held a competition in 2016 to select a quantum-resistant encryption standard in response to threats posed by quantum computers to existing encryption algorithms[1]. The first requirement is that the minimum security level should match the security of AES-128, AES-192, and AES-256 from the perspective of an exhaustive key search. Similarly, the algorithm must meet the security level of SHA-256 or SHA-384 from the perspective of collision resistance. The second requirement is to provide sufficiently strong security while ensuring practical cost-effectiveness and performance guarantees. The final requirement is to consider scalability in the algorithm's design.

In June 2022, NIST released four algorithms selected as quantum-resistant cryptography standards. CRYSTALS-KYBER [2] was selected in the public key category, and CRYSTALS-DILITHIUM [3], FALCON [4], and SPHINCS+ [5] were selected in the electronic signature category. However, the 4th round is still in progress to improve the public key algorithm, which lacks a mathematical foundation. The candidate algorithms for the 4th round are SIKE [6], BIKE [7], Classic McEliece [8], and HQC [9], with SIKE having been eliminated early due to security issues [6].

In 2022, the Korean Post-Quantum Cryptography (KpqC) Competition was held in Korea to select an independent quantum-resistant algorithm standard [10]. This competition aims to foster the development of post-quantum secure cryptographic algorithms within the Korean cryptographic community, addressing the challenges posed by advancements in quantum computing. The KpqC competition presented four main evaluation criteria. The first criterion is the security of the cryptography. All KpqC candidate algorithms must satisfy security requirements, and each algorithm's whitepaper must provide proof regarding the defined security properties. The second criterion is efficiency. Since security is a fundamental requirement for KpqC algorithms, once security is assured, efficiency becomes a key factor for competition. The third criterion is usability. Algorithms should not be limited to specific platforms or particular systems, as this would diminish their usability and result in inconvenience. Therefore, it is essential to support as many platforms and systems as possible to enhance the versatility and practicality of the algorithm. The final criterion is originality. The selected algorithm will become a Korean standard cryptography. In Round 1, 7 PKE/KEM algorithms and 9 digital signature algorithms were introduced, for a total of 16 candidate algorithms. The Round 1 selection was announced

in December 2023, and Round 2 is underway for the selected algorithms. A total of 8 algorithms were selected for Round 2, including 4 PKE/KEM algorithms and 4 digital signature algorithms. The selected algorithms are as follows: PALOMA [11], REDOG [12], NTRU+ [13], and SMAUG-T [14] (SMAUG [15] + TiGER [16] merged) were selected for PKE/KEM and HAETAE [17], NCC-Sign [18], MQ-Sign [19], AIMer [20] were selected for digital signature algorithms. The final algorithm will be selected as Korean standard cryptography in 2024. In this paper, we introduce KpqClean ver2, the successor to the KPQClean [21] project. The new project shows the fair performance comparisons on newly updated KpqC algorithms on various platforms.

The structure of the rest of the paper is as follows: Section 2 provides specific details of the NIST PQC Competition and the KpqC Competition. Section 3 details the work done to benchmark the KpqC algorithm. Section 4 presents the benchmark results. Finally, Section 5 concludes the paper by summarizing the main findings and discussing potential directions for future research.

1.1 Contribution

- Benchmark results of "clean" implementation for various environments We not only removed external library dependencies but also integrated them into the source code provided by PQClean for common algorithms (e.g., AES, SHA2, SHAKE, and etc.) to enable more accurate performance comparisons. During this integration process, direct code modifications were made to all KPQC Round 2 algorithms.

In particular, it has been implemented so that users can easily use it without any modifications, not only on Ryzen and Intel processors but also in the aarch64 environment. KpqC ver2 automatically recognizes the user's environment, providing easy benchmarking for all users without the need for separate settings.

- Benchmark results of "avx2" implementation For the KPQC Round 2 algorithm "avx2" implementation, dependencies were removed in the same way as for the "clean" implementation. The avx2 implementations (e.g., KeccakP-1600-time4-SIMD256, etc.) provided by PQClean were also integrated. Additionally, even if the code was not from PQClean, algorithms used by multiple encryption algorithms were integrated into a single implementation. The process of replacing with common code was the same as for the "clean" implementation, with all code modifications performed directly.

- Comprehensive Analysis and Utilization of KpqC We provide comprehensive analysis results to support the future utilization and development of the KpqC algorithm. These include measurements of memory usage during the execution of each algorithm using Valgrind, as well as the proportion of function usage during algorithm implementation, measured using Xcode's Profile tool. These findings can contribute to a better understanding and further development of KpqC algorithm implementations.

2 KpqClean Ver2

We are a follow-up to the KPQClean project for the KpqC competition, which was inspired by PQClean project [22]. KpqClean ver2 follows the pqlean project structure. As with KPQClean project [21] and KPQCLib [23], external library dependencies for each algorithm have been removed. For more accurate performance comparisons, we integrated the same code for common algorithms (e.g., AES, SHA2, SHAKE, etc.) provided by PQClean. Unlike previous projects that only provided a "clean" implementation, we also offer an "avx2" implementation. REDOG is not included because it is still in progress.

2.1 Clean Implementation

Preliminary work was conducted using PQClean's code to measure the performance of KpqC Round 2 algorithms in a consistent environment. Almost all KpqC Round 2 algorithms use Randombytes code (i.e., number generators) that rely on OpenSSL [24] to generate KAT files. To remove these dependencies, we replaced the Randombytes code with the version provided by PQClean, which has no external dependencies.

All OpenSSL dependencies were removed, and the SHAKE code was replaced with PQClean's implementation of fips202. Additionally, the AES, SHA2, and Randombytes code was replaced with common code from the pqclean project.

The specific replacements for each algorithm are as follows: For the PKE/KEM algorithms, NTRU+ and SMAUG-T had their Randombytes, AES, and SHA2 code replaced, while PALOMA did not require replacement. For the digital signature algorithms, AIMer, HAETAE, and NCC-Sign had their Randombytes and fips202 code replaced, and MQ-Sign had its fips202 and AES code replaced.

2.2 AVX2 Implementation

The following outlines the details of the algorithm modifications to align them with the clean version. For the PKE/KEM algorithms, NTRU+ and SMAUG-T (kem 90s) replaced Randombytes, AES, and SHA2, while SMAUG-T (kem) only replaced Randombytes. For the digital signature algorithms, HAETAE and NCC-Sign replaced Randombytes, and MQ-Sign replaced fips202.

The AVX2 replacements for each algorithm are as follows. In the PKE/KEM category, NTRU+ and SMAUG-T (kem 90s) replaced AES-CTR, and SMAUG-T (kem) replaced KeccakP-1600-time4-SIMD256 from PQClean. For the digital signature algorithms, HAETAE replaced AES-CTR and PQClean's KeccakP-1600-time4-SIMD256.

To ensure consistent performance measurement across the same environment, we integrated similar operations within encryption algorithms that are not part of PQClean, so that they all operate under a unified codebase.

The integrated code is as follows: SMAUG-T (kem) and HAETAE were unified under the same fips202x4. However, since HAETAE uses f1600x4.S, this

part was not integrated. Additionally, KeccakP-1600-AVX2 and KeccakP-1600-SnP were also integrated using the same code. AIMer, HAETAE, and NCC-Sign were integrated using fips202 implemented in the same AVX2. Note that this fips202 implementation differs from the clean version. For NCC-Sign, Keccak-1600 was originally implemented with AVX512. However, since AVX512 was not supported in the test environment, it was consolidated into an AVX256 implementation, Keccak-1600-AVX2. For AIMer, KeccakP-1600-times4-SIMD256 and KeccakP-1600-SnP were not integrated, as they were used with some modifications to the original PQClean versions. PALOMA does not have an AVX2 implementation. Finally, MQ-Sign does not integrate with AES, as it uses its own AVX2 implementation within Randombytes.

3 Benchmark Result

We benchmarked the algorithms we worked on in Intel, Ryzen, and aarch64 environments, respectively. The benchmarking environment resembled the specifications outlined in Table 1.

Table 1: Specification of target environments

	Testing Environment1	Testing Environment2	Testing Environment3	Testing Environment4
OS	Ubuntu 22.04	Ubuntu 23.10.1	MacOS Sonoma 14.4.1	Ubuntu 23.10.1
CPU	Ryzen 7 4800H(2.90GHz)	Intel i5-8259U(2.30GHz)	Apple M2(3.23GHz)	Ryzen 7 4800H(2.90GHz)
RAM	16GB	16GB	8GB	16GB
Compiler	gcc 11.4.0	gcc 13.2.0	Apple clang 15.0.0	gcc 13.2.0
Optimization Level	-O3	-O3	-O3	-O3

To obtain the measurements, each algorithm went through 10,000 iterations, and the average number of clock cycles required for each round of operation was calculated. The **-O3** optimization level (fastest) was applied.

The source code for performance measurements distinguishes between x86 and aarch64 (Apple Silicon) architectures and works correctly. We used the **RDTSC** instruction to calculate time consumption. The RDTSC (Read Time-Stamp Counter) instruction is a crucial tool for performance analysis on x86 architectures. It reads the CPU's time-stamp counter, a 64-bit register that counts clock cycles since the last reset, allowing precise measurement of CPU cycles for specific code segments. In the aarch64 environment, to obtain cycle counts, we made use of **m1cycles.c**[1] from [25].

[1] https://github.com/GMUCERG/PQC_NEON/blob/main/neon/kyber/m1cycles.c

Table 2: Testing Environment1(Unit : clock cycle).

PKE/KEM							
Scheme	Impl.	Keygen		Encapsulation		Decapsulation	
NTRU+KEM576	clean	253,031	+1,674,686 -84,309	85,541	+51,107 -3,297	110,140	+140,014 -4,783
NTRU+KEM768	clean	354,704	+662,355 -97,445	106,690	+58,668 -1,014	141,220	+88,721 -3,934
NTRU+KEM864	clean	354,592	+674,995 -99,218	128,823	+89,257 -3,775	170,729	+77,540 -6,328
NTRU+KEM1152	clean	659,905	+3,152,986 -324,346	164,274	+149,390 -4,542	219,767	+80,644 -2,615
NTRU+PKE576	clean	329,712	+2,161,939 -161,860	83,245	+59,319 -1,494	107,302	+89,057 -1,829
NTRU+PKE768	clean	310,102	+552,416 -52,669	107,632	+55,899 -2,014	140,803	+76,320 -4,242
NTRU+PKE864	clean	348,486	+638,703 -89,487	128,465	+83,177 -1,532	169,804	+86,643 -1,807
NTRU+PKE1152	clean	637,652	+2,479,645 -300,933	164,153	+98,964 -3,841	230,312	+180,328 -7,128
PALOMA128	clean	131,189,151	+337,324,670 -48,388,844	133,345	+73,976 -11,400	8,424,379	+3,300,524 -264,417
PALOMA192	clean	650,967,499	+2,007,087,009 -202,508,541	176,834	+32,343 -3,820	41,793,889	+1,177,644 -239,267
PALOMA256	clean	769,657,392	+1,349,832,403 -228,310,941	215,487	+99,424 -10,573	43,735,841	+22,761,855 -510,993
SMAUG-T1	clean	143,625	+305,324 -84,117	66,403	+56,876 -9,157	104474	+104,848 -13,733
SMAUG-T3	clean	193,935	+203,539 -92,957	243,970	+261,094 -97,607	421956	+157,232 -106,755
SMAUG-T5	clean	1,566,958	+702,292 -50,577	1,663,364	+946,346 -116,910	1877628	+1,173,955 -75,887
SMAUG-TiMER	clean	69,234	+342,914 -21,993	76,165	+155,081 -13,351	112282	+169,192 -12,957

Digital Signature							
Scheme	Impl.	Keygen		Sign		Verify	
HAETAE2	clean	1,160,435	+10,259,794 -973,472	5,032,357	+2,519,620 -65,788	158,529	+73,732 -7,178
HAETAE3	clean	2,103,397	+20,388,104 -1,771,318	2,739,863	+729,030 -77,779	275,419	+73,277 -3,254
HAETAE5	clean	1,998,427	+13,745,470 -1,579,232	3,369,567	+541,953 -115,071	342,870	+131,280 -15,489
AIMer128f	clean	159,134	+191,911 -99,278	3,742,915	+1,367,987 -807,883	3,545,576	+319,515 -133,755
AIMer128s	clean	108,704	+368,810 -50,095	26,541,270	+16,357,937 -359,635	27,013,406	+3,403,360 -628,829
AIMer192f	clean	201,497	+487,253 -59,948	8,745,805	+5,257,135 -480,631	7,800,523	+1,378,615 -238,193
AIMer192s	clean	187,591	+383,071 -49,754	65,094,493	+14,206,703 -1,773,834	62,821,167	+13,650,383 -985,018
AIMer256f	clean	413,606	+259,604 -70,304	16,706,302	+3,299,116 -553,621	15,776,925	+10,620,789 -410,898
AIMer256s	clean	454,476	+1,137,595 -111,870	119,340,970	+20,268,336 -107,711,367	119,925,648	+33,733,868 -108,221,678
MQ-Sign-MQLR-256-72-46	clean	74,764,874	+36,781,799 -581,337	463,964	+48,292 -3,647	733,587	+171,880 -5,658
MQ-Sign-MQLR-256-112-72	clean	286,180,511	+97,280,167 -783,333	1,289,973	+91,964 -8,434	2,016,272	+162,672 -14,634
MQ-Sign-MQLR-256-148-96	clean	750,300,850	+79,183,656 -1,667,619	2,762,744	+95,554 -17,807	4,307,611	+209,806 -21,585
MQ-Sign-MQRR-256-72-46	clean	100,975,612	+89,136,585 -533,300	809,523	+55,141 -6,455	706,565	+77,276 -2,851
MQ-Sign-MQRR-256-112-72	clean	387,732,599	+44,852,716 -714,839	2,057,398	+1,481,762 -27,427	1,986,691	+247,411 -13,125
MQ-Sign-MQRR-256-148-96	clean	997,169,889	+150,048,622 -2,407,046	4,298,168	+390,204 -25,453	4,272,864	+280,803 -22,595
NCC-Sign1	clean	300,401	+646,449 -107,145	479,198	+325,929 -10,123	285,502	+125,834 -5,971
NCC-Sign3	clean	352,126	+620,273 -101,305	600,143	+248,629 -7,934	356,543	+106,500 -5,092
NCC-Sign5	clean	455,065	+1,455,368 -65,624	1,585,378	+701,098 -26,541	591,947	+99,558 -11,889

Table 3: Testing Environment2(Unit : clock cycle).

PKE/KEM							
Scheme	Impl.	Keygen		Encapsulation		Decapsulation	
NTRU+KEM576	clean	209,949	+703,153 -58,735	81,678	+144,344 -6,588	108,342	+242,202 -14,578
NTRU+KEM768	clean	232,394	+262,510 -5,060	101,297	+187,479 -3,435	125,636	+191,836 -3,046
NTRU+KEM864	clean	228,042	+343,026 -6,330	109,680	+147,016 -3,320	145,575	+195,881 -4,007
NTRU+KEM1152	clean	558,996	+2,733,988 -255,826	152,931	+162,781 -4,815	212,747	+271,627 -6,285
NTRU+PKE576	clean	207,953	+736,779 -57,425	78,594	+125,638 -2,932	98,820	+99,366 -2,758
NTRU+PKE768	clean	234,498	+384,692 -4,278	102,024	+172,934 -2,000	129,611	+171,113 -1,585
NTRU+PKE864	clean	223,966	+243,950 -5,872	110,043	+180,451 -5,633	142,710	+196,082 -2,650
NTRU+PKE1152	clean	558,631	+2,226,137 -254,665	151,692	+167,950 -3,474	212,804	+213,218 -5,604
PALOMA128	clean	147,357,235	+389,328,913 -58,449,361	87,441	+230,293 -8,077	8,033,779	+3,699,969 -152,779
PALOMA192	clean	700,965,299	+2,476,698,857 -659,158,113	136,963	+120,833 -11,391	41,802,948	+6,515,492 -128,528
PALOMA256	clean	819,112,319	+1,820,540,793 -282,053,355	200,987	+257,089 -54,317	45,735,541	+25,319,729 -2,327,449
SMAUG-T1	clean	128,685	+639,815 -78,981	58,942	+113,882 -9,866	74,463	+146,055 -11,087
SMAUG-T3	clean	176,020	+1,181,616 -86,666	98,497	+180,147 -15,075	128,148	+185,434 -22,976
SMAUG-T5	clean	233,829	+2,058,947 -90,455	165,370	+287,522 -23,316	184,943	+267,087 -21,147
SMAUG-TiMER	clean	61,064	+80,540 -14,158	58,115	+114,315 -9,279	73,249	+159,739 -12,141

Digital Signature							
Scheme	Impl.	Keygen		Sign		Verify	
HAETAE2	clean	1,067,698	+9,540,834 -871,740	1,866,070	+1,657,046 -157,894	170,103	+277,809 -20,503
HAETAE3	clean	2,155,146	+18,433,878 -1,796,444	7,057,356	+2,779,776 -280,660	285,845	+269,579 -7,909
HAETAE5	clean	2,225,344	+11,563,516 -1,779,086	3,389,114	+2,921,706 -196,536	355,867	+427,329 -24,849
AIMer128f	clean	90,968	+ 126,428 - 26,982	3,288,188	+ 3,470,032 - 438,256	2,961,062	+3,117,156 -293,944
AIMer128s	clean	146,339	+ 640,573 - 79,889	23,324,902	+ 20,995,074 - 676,710	22,820,605	+8,788,577 -178,851
AIMer192f	clean	175,871	+ 275,195 - 24,493	7,292,073	+ 3,175,801 - 91,083	6,832,062	+6,176,570 -80,346
AIMer192s	clean	223,472	+ 2,430,944 - 71,518	57,739,339	+35,410,681 -1,241,911	56,700,184	+21,375,740 -406,560
AIMer256f	clean	448,151	+ 3,784,489 - 79,889	15,639,023	+13,579,117 -689,741	14,820,473	+8,329,473 -783,335
AIMer256s	clean	488,242	+ 545,422 - 88,608	116,361,684	+30,120,644 -104,888,491	114,818,217	+19,463,959 -103,397,158
MQ-Sign_MQLR_256_72_46	clean	70,289,113	+11,686,941 -1,296,511	423,941	+341,237 -9,107	656,693	+326,409 -7,989
MQ-Sign_MQLR_256_112_72	clean	270,156,039	+166,579,211 -10,777,551	1,138,677	+683,619 -15,499	1,709,939	+477,371 -7,893
MQ-Sign_MQLR_256_148_96	clean	682,362,334	+208,552,816 -13,388,824	2,370,417	+793,709 -21,759	3,454,074	+1,304,184 -152,396
MQ-Sign_MQRR_256_72_46	clean	101,307,620	+51,567,794 -6,870,156	753,662	+451,436 -19,796	643,609	+447,893 -7,755
MQ-Sign_MQRR_256_112_72	clean	372,556,356	+120,261,636 -13,556,862	1,812,867	+845,917 -17,393	1,747,581	+898,999 -61,451
MQ-Sign_MQRR_256_148_96	clean	913,690,602	+257,111,756 -11,967,402	3,660,222	+1,155,052 -32,918	3,359,634	+926,552 -47,380
NCC-Sign1	clean	205,450	+216,504 -9,328	1,161,522	+681,130 -12,298	288,198	+211,742 -3,862
NCC-Sign3	clean	280,512	+751,724 -13,686	884,954	+770,420 -16,302	391,048	+428,520 -8,464
NCC-Sign5	clean	416,020	+321,480 -19,212	1,049,172	+1,536,890 -104,722	630,294	+400,706 -25,886

Table 4: Testing Environment3(Unit : clock cycle).

PKE/KEM							
Scheme	Impl.	Keygen		Encapsulation		Decapsulation	
NTRU+KEM576	clean	152,653	+370,504 -37,853	57,392	+62,146 -1,157	64,897	+34,112 -736
NTRU+KEM768	clean	170,987	+141,653 -8,859	74,605	+54,620 -856	85,814	+23,360 -988
NTRU+KEM864	clean	175,592	+146,002 -4,080	83,862	+33,761 -1,153	100,583	+11,018 -733
NTRU+KEM1152	clean	425,625	+1,770,371 -197,047	108,935	+14,504 -832	137,606	+15,519 -1,010
NTRU+PKE576	clean	162,109	+571,361 -47,318	56,945	+13,285 -390	65,513	+22,680 -1,047
NTRU+PKE768	clean	172,444	+143,358 -10,304	73,866	+53,067 -729	85,777	+25,486 -802
NTRU+PKE864	clean	180,081	+133,932 -8,616	83,385	+19,992 -604	100,620	+15,877 -518
NTRU+PKE1152	clean	420,607	+1,647,182 -192,038	108,907	+11,733 -536	137,555	+19,417 -514
PALOMA128	clean	119,572,480	+327,879,744 -38,892,410	58,133	+23,738 -1,164	8,091,350	+1,002,047 -52,993
PALOMA192	clean	561,481,159	+1,137,578,065 -178,665,050	83,030	+71,604 -2,536	39,835,958	+8,317,315 -149,888
PALOMA256	clean	662,511,849	+1,552,324,299 -209,806,307	97,140	+21,949 -2,166	41,789,539	+8,327,562 -139,063
SMAUG-T1	clean	39,278	+17,947 -684	39,889	+20,699 -606	49,270	+28,460 -554
SMAUG-T3	clean	66,770	+23,199 -911	64,752	+47,962 -890	78,442	+18,288 -621
SMAUG-T5	clean	119,675	+92,197 -5,370	114,363	+14,157 -1,318	130,111	+43,348 -736
SMAUG-TiMER	clean	40,742	+35,256 -4,301	39,914	+31,513 -689	48,089	+25,117 -646

Digital Signature							
Scheme	Impl.	Keygen		Sign		Verify	
HAETAE2	clean	951,707	+ 4,304,776 - 786,252	1,388,930	+ 93,477 - 9,149	135,534	+ 26,563 - 1,777
HAETAE3	clean	1,686,812	+ 11,883,232 - 1,395,079	3,429,034	+ 155,754 - 14,846	238,747	+ 49,736 - 2,304
HAETAE5	clean	1,993,473	+ 7,778,604 - 1,621,255	4,227,778	+ 174,834 - 19,735	300,372	+ 95,616 - 3,349
AIMer128f	clean	62,227	+40,235 -4,000	2,316,451	+208,839 -10,695	2,153,628	+43,867 -6,903
AIMer128s	clean	61,326	+36,004 -3,175	18,327,463	+271,003 -36,069	18,357,046	+259,231 -38,087
AIMer192f	clean	132,084	+75,762 -6,206	5,920,018	+1,802,046 -21,542	5,505,462	+147,506 -11,000
AIMer192s	clean	131,826	+64,183 -5,693	46,264,704	+7,347,145 -160,832	45,799,558	+9,473,531 -63,067
AIMer256f	clean	301,561	+156,232 -6,853	11,632,128	+173,515 -29,069	10,853,032	+177,752 -20,509
AIMer256s	clean	300,197	+131,455 -5,691	119,041,270	+29,507,476 -346,796	88,530,661	+14,031,610 -159,888
MQ-Sign_MQLR_256_72_46	clean	108,328,299	+ 9,438,458 - 374,162	911,214	+ 301,685 - 3,856	1,383,970	+ 169,608 - 3,873
MQ-Sign_MQLR_256_112_72	clean	548,057,963	+ 101,662,406 - 493,984,383	3,301,232	+ 2,177,370 - 168,977	5,294,775	+ 3,394,860 - 148,453
MQ-Sign_MQLR_256_148_96	clean	1,575,443,966	+ 91,175,611 - 6,529,056	7,186,177	+ 89,944 - 24,204	11,775,183	+ 713,513 - 23,856
MQ-Sign_MQRR_256_72_46	clean	137,903,106	+ 32,745,015 - 3,558,022	1,447,542	+ 951,781 - 64,326	1,449,188	+ 933,045 - 91,464
MQ-Sign_MQRR_256_112_72	clean	650,807,091	+ 77,310,053 - 10,391,443	4,843,332	+ 1,871,382 - 26,251	5,069,042	+ 184,745 - 25,198
MQ-Sign_MQRR_256_148_96	clean	1,822,096,906	+ 139,852,643 - 15,531,536	11,118,530	+ 894,537 - 30,514	11,929,921	+ 366,263 - 164,625
NCC-Sign1	clean	169,791	+ 90,560 - 9,557	776,134	+ 605,924 - 38,488	220,085	+ 186,268 - 4,850
NCC-Sign3	clean	218,513	+ 133,066 - 6,990	718,883	+ 571,758 - 10,965	279,032	+ 191,800 - 1,737
NCC-Sign5	clean	330,636	+ 18,775 - 6,476	746,686	+ 567,287 - 24,467	447,622	+ 367,489 - 9,469

Table 5: Testing Environment2(Unit : clock cycle).

PKE/KEM							
Scheme	Impl.	Keygen		Encapsulation		Decapsulation	
NTRU+KEM576	avx2	74,524	+181,432 -39,838	43,325	+116,319 -20,071	12,933	+39,959 -1,233
NTRU+KEM768	avx2	34,551	+68,949 -13,817	29,121	+57,847 -3,323	16,449	+63,935 -1,627
NTRU+KEM864	avx2	26,132	+62,458 -3,398	32,559	+75,457 -5,829	17,272	+52,338 -1,026
NTRU+KEM1152	avx2	60,927	+422,209 -32,945	40,212	+143,884 -6,074	26,813	+99,627 -5,711
NTRU+PKE576	avx2	28,451	+185,063 -13,931	21,394	+75,004 -2,346	11,979	+19,247 -141
NTRU+PKE768	avx2	29,094	+66,792 -3,492	29,486	+48,630 -4,552	18,613	+98,341 -3,441
NTRU+PKE864	avx2	22,996	+82,442 -3,230	30,464	+150,116 -3,352	19,384	+90,366 -2,718
NTRU+PKE1152	avx2	97,123	+1,208,669 -66,817	40,233	+114,021 -6,451	24,370	+122,616 -2,682
SMAUG-T1	avx2	41,139	+79,395 -2,601	33,704	+91,644 -6,328	644,01	+77,957 -31,009
SMAUG-T3	avx2	64,350	+104,162 -16,956	55,982	+126,894 -10,004	63,344	+171,742 -9,052
SMAUG-T5	avx2	146,222	+322,130 -63,006	107,760	+233,606 -16,474	109,709	+160,651 -15,309
SMAUG-T1 90	avx2	70,489	+223,761 -39,965	35,396	+77,718 -9,278	39,227	+101,975 -8,919
SMAUG-T3 90	avx2	91,033	+286,151 -48,769	45,414	+112,330 -11,298	57,594	+114,514 -12,556
SMAUG-T5 90	avx2	108,447	+428,481 -62,413	62,174	+103,076 -8,870	85,226	+155,500 -16,206

Digital Signature							
Scheme	Impl.	Keygen		Sign		Verify	
HAETAE2	avx2	834,651	+14,765,785 -704,885	4,946,575	+2,634,687 -385,571	67,068	+164,044 -8,002
HAETAE3	avx2	1,423,068	+7,444,214 -1,217,732	374,489	+729,547 -62,213	128,981	+201,363 -36,449
HAETAE5	avx2	1,924,879	+10,273,613 -1,668,673	1,077,729	+1,352,227 -154,947	134,969	+200,435 -24,139
AIMer128f	avx2	40,172	+65,462 -4,580	811,275	+776,393 -51,881	783,010	+821,260 -46,757
AIMer128s	avx2	93,037	+361,621 -45,977	5,889,742	+4,730,638 -96,018	6,494,209	+3,670,069 -755,539
AIMer192f	avx2	99,173	+102,063 -29,347	2,210,305	+1,851,543 -221,417	2,131,677	+1,564,933 -174,807
AIMer192s	avx2	97,972	+259,516 -28,322	15,833,475	+7,631,911 -888,749	15,289,548	+5,757,682 -342,116
AIMer256f	avx2	236,956	+1,771,332 -68,018	4,071,768	+2,804,818 -85,904	3,980,184	+1,617,296 -54,768
AIMer256s	avx2	242,895	+1,707,521 -74,005	29,154,407	+11,831,895 -249,801	28,753,363	+8,120,859 -189,125
MQ-Sign_MQLR_256_72_46	avx2	4,947,896	+3,414,784 -563,616	59,773	+159,945 -11,739	45,235	+127,409 -9,479
MQ-Sign_MQLR_256_112_72	avx2	23,971,764	+74,850,252 -3,429,550	164,022	+297,784 -30,662	162,990	+307,284 -33,124
MQ-Sign_MQLR_256_148_96	avx2	57,884,657	+69,657,957 -5,028,725	260,591	+826,927 -35,381	279,745	+978,227 -52,185
MQ-Sign_MQRR_256_72_46	avx2	7,538,634	+44,762,198 -442,702	76,323	+162,959 -11,189	44,568	+109,752 -9,488
MQ-Sign_MQRR_256_112_72	avx2	32,740,982	+25,672,572 -1,606,868	206,977	+334,305 -15,719	141,516	+277,264 -13,374
MQ-Sign_MQRR_256_148_96	avx2	81,811,186	+56,095,180 -4,311,388	409,799	+554,377 -59,595	290,724	+568,102 -55,886
NCC-Sign1	avx2	137,199	+837,133 -23,725	207,323	+375,175 -24,189	131,963	+260,327 -15,565
NCC-Sign3	avx2	187,859	+1,332,789 -36,395	374,557	+572,997 -22,527	188,051	+296,407 -17,369
NCC-Sign5	avx2	265,387	+1,534,171 -36,517	422,955	+465,541 -27,789	299,315	+860,981 -22,305

Table 6: Testing Environment4 (Unit : clock cycle).

PKE/KEM							
Scheme	Impl.	Keygen		Encapsulation		Decapsulation	
NTRU+KEM576	avx2	73,798	+259,383 -26,731	55,179	+128,449 -9,330	19,814	+60,922 -7,228
NTRU+KEM768	avx2	140,673	+129,781 -33,518	37,193	+80,431 -10,223	17,330	+21,124 -249
NTRU+KEM864	avx2	94,406	+100,677 -24,893	51,431	+94,062 -21,764	19,302	+60,992 -191
NTRU+KEM1152	avx2	158,476	+483,033 -117,992	38,796	+40,374 -371	24,911	+21,124 -377
NTRU+PKE576	avx2	79,520	+227,909 -15,256	61,034	+88,084 -9,704	14,639	+47,282 -1,473
NTRU+PKE768	avx2	141,808	+142,102 -94,045	26,999	+52,664 -493	17,558	+33,611 -187
NTRU+PKE864	avx2	80,727	+160,756 -11,156	64,812	+139,348 -35,261	21,408	+48,291 -1,688
NTRU+PKE1152	avx2	158,938	+554,201 -116,279	44,797	+114,935 -7,039	26,357	+26,754 -1,011
SMAUG-T1	avx2	131,751	+129,800 -25,292	48,594	+110,094 -4,137	55682	+25,924 -524
SMAUG-T3	avx2	162901	+300,867 -91,735	74,948	+81,739 -1,665	91347	+61,657 -1,302
SMAUG-T5	avx2	224491	+312,995 -96,775	135,314	+39,353 -2,088	156083	+32,736 -1,194
SMAUG-T1 90	avx2	72566	+76,639 -2,937	74,438	+195,581 -43,872	42832	+39,383 -811
SMAUG-T3 90	avx2	132416	+91,261 -13,561	69,773	+198,622 -18,211	67223	+57,100 -2,756
SMAUG-T5 90	avx2	153995	+132,032 -95,763	76,148	+37,909 -3,851	102099	+75,903 -3,528

Digital Signature							
Scheme	Impl.	Keygen		Sign		Verify	
HAETAE2	avx2	829,349	+10,808,467 - 696,819	699,815	+216,469 -5,091	71,242	+92,666 -1,381
HAETAE3	avx2	1,463,470	+6,482,066 -1,253,307	391,344	+722,575 -6,688	113,329	+47,679 -1,998
HAETAE5	avx2	1,902,225	+13,926,729 -1,641,428	439,008	+320,604 -5,110	134,080	+71,356 -1,724
AIMer128f	avx2	109,474	+220,894 -19,429	988,392	+2,678,803 -21,793	990,973	+1,791,113 -36,148
AIMer128s	avx2	126,120	+77,054 -14,963	7,384,692	+9,217,199 -52,071	7334755	+529,34,585 -34,585
AIMer192f	avx2	199,741	+211,827 -91,861	2,693,700	+585,823 -17,232	2674129	+244,402 -13,466
AIMer192s	avx2	211,196	+246,917 -103,084	20,005,582	+4,831,700 -79,537	19914200	+748,532 -50,940
AIMer256f	avx2	359,442	+794,758 -88,640	5,216,095	+1,065,740 -24,950	5184187	+1,878,096 -26,392
AIMer256s	avx2	294,201	+1,387,219 -22,935	42,617,762	+8,401,648 -227,331	42199394	+522,971 -70,311
MQ-Sign_MQLR_256_72_46	avx2	3,737,246	+10,631,703 -158,356	46,147	+153,083 -762	35,451	+19,773 -738
MQ-Sign_MQLR_256_112_72	avx2	16,810,793	+45,102,351 -577,347	126,525	+115,538 -7,277	117,008	+139,874 -7,475
MQ-Sign_MQLR_256_148_96	avx2	44,255,025	+92,906,130 -1,345,668	211,273	+703,648 -7,432	219,523	+349,167 -24,469
MQ-Sign_MQRR_256_72_46	avx2	5,690,604	+3,830,502 -121,241	63,070	+65,719 -1,242	35,065	+50,340 -787
MQ-Sign_MQRR_256_112_72	avx2	24,379,382	+8,860,592 -386,754	179,025	+696,775 -12,652	120,588	+758,779 -10,881
MQ-Sign_MQRR_256_148_96	avx2	61,059,896	+101,890,843 -335,404	300,749	+335,714 -14,055	237,074	+421,088 -22,097
NCC-Sign1	avx2	226,966	+463,437 -78,312	248,743	+202,816 -2,156	163,753	+54,240 -1,382
NCC-Sign3	avx2	233,022	+687,931 -36,344	327,080	+345,299 -4,600	213,619	+81,050 -3,398
NCC-Sign5	avx2	398,589	+788,294 -100,730	509,618	+333,934 -4,583	341,277	+48,802 -3,137

3.1 Benchmark Result of Clean

Table 2 presents benchmark results for the Public Key Encryption/Key Encapsulation Mechanism (PKE/KEM) algorithms and Digital Signature algorithms on Ryzen processors. Table 3 presents benchmark results for the PKE/KEM algorithms and Digital Signature algorithms on Intel processors. Table 4 presents benchmark results for the PKE/KEM algorithms and Digital Signature algorithms on M2 chip.

Benchmark results for all environments showed that among the PKE/KEM candidates, SMAUG-T performed best in keygen, encapsulation, and decapsulation. On Ryzen processors, among the DSA candidates, AIMer performed best in keygen, NCC-Sign performed best in sign, and HAETAE performed best in verify. On Intel and M2 processors, among the DSA candidates, NCC-Sign performed best in keygen and sign, while HAETAE performed best in verify. The detailed performance comparison of each algorithm can be found in Table 33 through Table 38 in the Appenix.

3.2 Benchmark Result of AVX2

Table 5 presents benchmark results for the PKE/KEM and Digital Signature algorithms on Intel processors, while Table 6 shows the benchmark results for the same algorithms on Ryzen processors

Benchmark results for all environments showed that among the PKE/KEM candidates, SMAUG-T performed best in keygen, while NTRU+ performed best in both encapsulation and decapsulation. Similarly, among the DSA candidates, NCC-Sign performed best in keygen, encapsulation, and decapsulation. The detailed performance comparison for each algorithm is provided in Table 39 through Table 44 in the Appendix.

4 KpqC 2 Round Algorithm Computation Time Profiling Analysis

In this section, we analyze the usage proportion of functions employed by each algorithm implemented with KPQClean. For this analysis, we utilized the Time Profile tool provided by Xcode. The Time Profile tool measures the time and proportion of each function called during the execution of the project where the algorithm is implemented. Using this tool, we analyzed the execution of each algorithm. The results of this analysis help identify the operations that cause significant overhead in each algorithm, contributing to a better understanding of the algorithms and setting target operations for optimization efforts.

Table 7 shows the proportion of the key generation, encryption, and decryption process as a single process(100%). For NTRU+ in the KEM/PKE algorithm, key generation is the largest part of the total process. SMAUG-T is dominated by encryption and decryption, but we can see that key generation, encryption, and decryption are all performed in similar proportions. PALOMA has the largest proportion of the total process in key generation.

In Tabel 8, DSA algorithms, we can see that the signing process is the largest part of the overall algorithm. However, in the case of AIMer, the key generation process is very small part, and the signing and verification processes are similar.

Table 7: The computational proportion of Key Generation, Encapsulation, and Decapsulation of each KpqC KEM/PKE algorithm.

Scheme	KeyGen.	Encap.	Decap.
NTRU+KEM576	**58.43%**	16.85%	24.72%
NTRU+KEM768	**58.26%**	18.26%	23.48%
NTRU+KEM864	**52.17%**	19.57%	28.26%
NTRU+KEM1152	**63.90%**	16.60%	19.50%
NTRU+PKE576	**59.46%**	24.32%	16.22%
NTRU+PKE768	**50.00%**	23.48%	26.52%
NTRU+PKE864	**40.14%**	23.24%	36.62%
NTRU+PKE1152	**67.10%**	12.99%	19.91%
SMAUG-T1	23.19%	**40.58%**	36.23%
SMAUG-T3	34.45%	17.65%	**47.90%**
SMAUG-T5	29.22%	**39.61%**	31.17%
SMAUG-TiMER	31.75%	31.75%	**36.51%**
PALOMA-128	**95.51%**	0.04%	4.44%
PALOMA-192	**94.74%**	0.01%	5.25%
PALOMA-256	**95.71%**	0.01%	4.28%

Table 8: The computational proportion of Key Generation, Sign, and Verify of each KpqC DSA algorithm.

Scheme	KeyGen.	Sign	Verify
AIMer128f	0.60%	**51.80%**	47.40%
AIMer128s	0.10%	**50.20%**	49.70%
AIMer192f	0.60%	**51.70%**	47.50%
AIMer192s	0.10%	46.90%	**53.00%**
AIMer256f	0.60%	**51.60%**	47.70%
AIMer256s	0.20%	**50.20%**	49.60%
HAETAE2	22.60%	**73.20%**	4.10%
HAETAE3	29.70%	**65.30%**	4.80%
HAETAE5	24.60%	**70.50%**	4.80%
NCC-Sign1	19.40%	**56.50%**	24.10%
NCC-Sign3	17.40%	**63.60%**	18.90%
NCC-Sign5	15.00%	**63.10%**	21.80%

4.1 Computation Proportion of NTRU+ Algorithm

This is an analysis of the overhead of each operation in NTRU+ based on NTRU+KEM576. The results of the profiling analysis are shown in Table 9.

The most significant operation in NTRU+'s key generation process is the Poly_baseinv operation. This function is performed 58% of the time in the key generation process. It computes the inverse of a polynomial over the NTT domain. During the NTRU+ key generation process, the f and g polynomials must be able to have inverses, and this operation is used to verify this. Other operations such as polynomial multiplication, subtraction, and NTT operations are performed the rest of the time.

The largest part of the encryption process in NTRU+ is the AES algorithm operation. AES algorithm operations are performed about 40% of the encryption process. The Hash_h_kem(), Hash_g(), and Hash_f() functions, which use AES, are performed in the encryption process at 27%, 27%, and 13%, respectively. Inside each of these functions, the AES algorithm is used, and in the Hash_f() function, a Shake-related function is used. As a result, AES is used 40% of the time and Shake 26% of the time. Other than AES and Shake, there are NTT operations and multiplication operations in the NTT domain, which are performed at 13% and 13% respectively.

The NTT operation is the most important operation in the decryption process of NTRU+. The NTT operation is performed 32% of the time in the decryption process. Other operations are Hash_g() 18%, invNTT() 18%, Poly_basemul() 18%, and Hash_h_kem() 14%. As a result, when looking at proportion of NTT and invNTT, we can see that NTT operations are the most important operations in the decryption process.

NTRU+PKE shows similar results to NTUR+KEM. As a result, NTT-related operations can be seen as the most overhead operations in NTRU+, with the largest share of operations being NTT-related(NTT conversions, INTT conversions, multiplication over NTT domains, inverse calculations, etc.).

Table 9: Results of NTRU+KEM computational proportion measurement

Key Generation		Encapsulation		Decapsulation	
Function	proportion	Function	proportion	Function	proportion
Poly_baseinv	58%	Hash_h_kem	27%	NTT	32%
Poly_basemul	12%	Hash_g	27%	Hash_g	18%
Poly_reduce	8%	Poly_basemul	20%	invNTT	18%
NTT	6%	Hash_f	13%	Poly_basemul	18%
AES_ctr	6%	NTT	13%	hash_h_kem	14%
etc	4%	-	-	-	-

4.2 Computation Proportion of SMAUG-T Algorithm

Based on SMAUG-T1, the overhead of the computations performed in each step of SMAUG-T is analysed. The results of the profiling analysis are shown in Table 10.

The largest computation in the key generation process of SMAUG-T is the genPubkey() function. The genPubkey() function is performed at 55% of the key generation process. Inside the genPubkey() function, the genAx() and genBx() functions are performed. In addition to the genPubkey() function, the genSx_Vec() function is also performed at 27% of the time, which also uses shake-related functions. As a result, the largest proportion of the key generation process is Shake-related operations.

The largest operation in SMAUG-T encryption process is the Load_from_string_pk() function. Load_from_string_pk() is performed with a proportion of 50 per cent in the encryption process. Internally, it is implemented as a GenAx() function, which performs Shake-related operations. In addition, GenRx_vec() and Sha3_256 functions also perform shake-related operations, and shake-related operations are performed 76% of the time throughout the encryption process. As a result, shake-related operations are the largest part of the encryption process.

The operation that has the largest proportion in the decryption process of SMAUG-T is the Indcpa_enc() function. Indcpa_enc() is performed 81% of the time in the decryption process. Indcpa_enc90 and Indcpa_dec() are the encryption and decryption operations for the CPA-safe Lizard public key scheme. Inside the two functions, many post-multiplication addition operations such as vec_mult_add(), poly_mult_add(), and Shake-related operations are performed. Shake-related operations are performed 35% of the time throughout the decryption process, while Vec_mult_add() is performed 29% of the time. As a result, Shake-related operations are the largest part of the decryption process.

As a result, the largest share of computation in SMAUG-T is dominated by Shake-related functions, regardless of the key generation, encryption, and decryption processes.

Table 10: Results of SMAUGN-T computational proportion measurement

Key Generation		Encapsulation		Decapsulation	
Function	proportion	Function	proportion	Function	proportion
genPubkey(GenBx)	38%	GenRx_vec	46%	GenAx	24%
genSx_vec	31%	load_from_string_pk	29%	genRx_vec	20%
genPubkey(GenAx)	19%	computeC2	11%	computeC1	16%
etc	12%	sha3_256	7%	computeC2	16%
-	-	computeC1	4%	sha3_256	12%
-	-	etc	3%	etc	12%

4.3 Computation Proportion of PALOMA Algorithm

This is the result of analysing the overhead of the operations performed in each process of PALOMA based on PALOMA-128. The results of the profiling analysis are shown in Table 11.

The most significant operation in the key generation process of PALOMA is the Gen_scrambled_code() function operation. Gen_scrambled_code() is performed 70% of the time in the key generation process. Gen_scrambled_code() performs the process of scrambling the parity check matrix, and the Gaussain_row() function has the largest overhead. The reason for the large overhead is that the Gen_scrambled_code() function is implemented as a simple logical operation, but the large overhead is caused by the large number of iterations. In the implementation code, the iteration statement with the largest number of iterations is implemented with 9,846,431 iteration operations. As a result, logical and arithmetic operations are the most dominant operations in the key generation process due to the large number of iterations.

The largest part of PALOMA's encryption process is the Permutation operation. The Permutation operation is performed with a proportion of 44% in the encryption process. The permutation operation first generates errors through LSH, and then performs a permutation operation through the random oracle permutation matrix generated using the generated error vector. In addition, the process used to generate the error backer and random oracle G is performed with a proportion of 22%, and the LSH hash function is used internally. As a result, the permutation operation is the largest part of the encryption process.

The most significant operation in the decryption process of PALOMA is the reconstruction of the error vector e. The process of reconstructing the error vector e is performed 99% of the time in the decryption process. This process consists of Construct_key_eqn(), find_err_vec(), and solve_key_eqn(). The Construct_key_eqn() step is performed with 47% proportion and is responsible for generating the key equation. The find_err_vec() function is 44% of the time and is responsible for finding the polynomial that connects the generated key equation with the error locations. This is how the error vector e is reconstructed. As a result, the largest part of the decryption process is the reconstruction of the error vector e, which is heavily polynomial arithmetic.

As a result, the key generation process in PALOMA has a large overhead, and even though simple logical operations are used, the large number of iterations results in a large overhead.

Table 11: Results of PALOMA computational proportion measurement

Key Generation		Encapsulation		Decapsulation	
Function	proportion	Function	proportion	Function	proportion
gen_Scrambled_code	70%	Perm	45%	Construct_key_eqn	47%
gen_rand_goppa_code	30%	rand_oracle_g	22%	find_err_vec	44%
-	-	gen_err_vec	22%	solve_key_eqn	7%
-	-	encrypt_temp	11%	etc	2%

4.4 Computation Proportion of AIMer Algorithm

Based on AIMer-128s, the overhead of the computations performed by each process in AIMer is analysed. The results of the profiling analysis are shown in Table 12.

The largest proportion of computations in AIMer's key generation process is the AIM2 process. AIM2 is performed 83% of the time in the key generation process. Inside the AIM2 process, the Generate_matrices_L_and_U() function is used 53% of the time, and multiplication operations on GF are used 37% of the time. Inside the Generate_matrices_L_and_U() function, which is used heavily, Shake-related operations are used 90% of the time. As a result, Shake-related operations are the most heavily proportioned part of the key generation process.

The most significant operation in AIMer's signing process is the run_phase_1() function operation. Inside the run_phase_1() function, Expand-related functions (Expand_tree, Expand_tape) account for the majority (53%). Shake-related operations are performed in Expand-related functions, and the share of shake-related operations in all signature processes other than expand-related functions is 63%. Other operations include matrix multiplication, addition, and power on GF. As a result, shake-related operations account for the largest share of the signing process.

The largest part of AIMer's verification process is Expand related operations. Expand operations are performed 53% of the time in the verification process. As mentioned in the verification process, Expand-related operations are performed as Shake-related operations. Therefore, the share of shake-related operations in the verification process is 65%. Similarly, in Aim2_mpc, matrix multiplication, addition, and power operations on GF are performed. As a result, the largest share of the verification process is shake-related operations.

As a result, the largest proportion of computation in AIMer is dominated by Shake-related functions, regardless of the key generation, encryption, and decryption processes.

Table 12: Results of AIMer computational proportion measurement

Key Generation		Sign		Verify	
Function	proportion	Function	proportion	Function	proportion
GF_exp	41%	Aim2_mpc	76%	Aim2_mpc	70%
Generate_matrices_L_and_U	28%	Commit_seed_and_expand_tape	11%	Commit_seed_and_expand_tape	11%
GF_transposed_matmul	17%	Expand_seed	6%	Reconstruct_seed_tree	9%
generate	14%	expand_tree	4%	Gf_mul_add	4%
-	-	etc	3%	etc	6%

4.5 Computation Proportion of HAETAE Algorithm

This is the result of analysing the overhead of the operations performed in each process of HAETAE based on HAETAE-120. The results of the profiling analysis are shown in Table 13.

The largest operation in the key generation process of HAETAE is the Polyvecmk_sqsing_value() function. The Polyvecmk_sqsing_value() function checks whether the generated private key meets certain conditions and is performed 66% of the time in the key generation process. The FFT operation is 87.7% of the time in this process. Other operations are Polyvecmk_uniform_eta(), polyvecmk_uniform_eta(), and Polymatkm_expand(), and Shkae-related operations are mainly used inside the two functions. As a result, the FFT operation is the largest part of the key generation process.

The most important operation in HAETAE's signature process is the hyperball sampling process. The hyperball sampling process is performed 82% of the time in the signature process, and the Sample_gauss_N function is implemented internally. Sample_gauss_N performs operations to sample random numbers from a Gaussian distribution. Under the hood, Shake-related operations are heavily used. As a result, the most heavily proportioned operations in the signing process are shake-related operations.

Shake-related operations are the most important part of HAETAE's verification process. Unpack_sig(), Polymatkm_expand(), Poly_unifrom(), and Poly_unifrom() functions are performed in similar proportions in the verification process. Within each of these functions, shake-related operations are used, and when viewed as a whole, shake-related operations are performed 40% of the time. As a result, shake-related operations are the largest part of the verification process.

As a result, HAETAE is a signing process with a large overhead, and the operations with the largest overhead in the signing process are Shake-related operations.

Table 13: Results of HAETAE computational proportion measurement

Key Generation		Sign		Verify	
Function	proportion	Function	proportion	Function	proportion
Polyvecmk_sqsing_value	66%	Polyfixveclk_smaple_hyperball	82%	unpack_sig	24%
Polyvecmk_uniform_eta	8%	invntt_tomont	4%	Polymatkm_expand	18%
Polymatkm_expand	6%	ntt	2%	Poly_uniform	13%
Polyvecm_ntt	5%	etc	12%	Polyveck_ntt	10%
etc	12%	-	-	etc	35%

4.6 Computation Proportion of MQ-Sign Algorithm

This is an analysis of the computational overhead of each process of MQ-Sign based on MQLR-72-46. The results of the profiling analysis are shown in Table 14.

The largest computation in the key generation process of MQ-Sign is the process of generating the public key. In the process of generating the public key, matrix multiplication and addition operations on GF are performed, and this process is performed with 67% proportion. In addition, the Generate_F function

is performed at 30%, and the AES algorithm operation is performed inside. As a result, the matrix operations on GF are the most important operations in the key generation process.

The largest operation in the signing process of MQ-Sign is the GF256mat_prod_ref() function. The GF256mat_prod_ref() function is performed 73% of the time during the signing process. Inside the function, the GF256v_madd_u32() function is responsible for most of the operations. The GF256v_madd_u32() function, which performs a vector multiplication followed by an addition operation, is also used heavily in other functions used in the signature process, and is performed 85% of the time in the signature process as a whole. As a result, the arithmetic operations between vectors are the most heavily used operations in the signing process.

The largest computation in the verification process of MQ-Sign is the mapping of the public key to the z contained in the signature value. This process is performed 78% of the time in the verification process. This is the process of calculating h using the public key and the z contained in the signature value. The Gf256v_madd() function is heavily used inside this process. As a result, the most heavily proportioned operations in the verification process are arithmetic operations between vectors.

As a result, the operations with the largest overhead in MQ-Sign are the arithmetic operations on GF.

Table 14: Results of MQ-sign computational proportion measurement

Key Generation		Sign		Verify	
Function	proportion	Function	proportion	Function	proportion
Generate_keypair_mqlr	69%	GF256mat_prod_ref	73%	mpkc_pub_map_gf256	76%
generate_F	30%	solve_linear_eq_ref_modify	13%	hash	11%
gf256v_madd_u32	1%	gf256mat_mul_ref	10%	mpkc_pub_map_gf256	11%
-	-	gf256mat_gaussian_elim_ref	4%	-	-

4.7 Computation Proportion of NCC-Sign Algorithm

This is the result of analysing the overhead of operations performed in each process of NCC-Sign based on NCC-Sign-1. The results of the profiling analysis are shown in Table 15.

The operation that has the largest share in the key generation process of NCC-Sign is the Poly_uniform() operation. Poly_uniform() is performed 34% of the time in the key generation process. This is the process of sampling polynomial coefficients, and Shake-related operations are performed internally 60% of the time. In addition, NTT and InvNTT operations are also performed at 30%. As a result, NTT and Shake operations have the largest share of the key generation process.

The NTT operations are the largest part of the signing process in NCC-Sign. NTT operations are performed 36% of the time during the signing process, and

59% of the time when InvNTT operations are combined. In addition, several polynomial operations are performed with a proportion of about 13%. As a result, NTT operations are the largest part of the signature process.

The InvNTT operation has the largest proportion in the verification process of NCC-Sign. The InvNTT operation is performed 31% of the time in the verification process, and 59% of the time when the NTT operations are combined. In addition, the Poly_uniform() function is performed 22% of the time, and Shake-related operations inside this function are performed 80% of the time. As a result, InvNTT operations are the most heavily proportioned operations in the verification process.

As a result, the operations that weigh the most in NCC-Sign are the NTT and InvNTT operations, which all have a large overhead.

Table 15: Results of NCC-Sign computational proportion measurement

Key Generation		Sign		Verify	
Function	proportion	Function	proportion	Function	proportion
Poly_uniform	34%	NTT	36%	invNTT_tomont	31%
invntt_tomnot	17%	invNTT_tomont	23%	NTT	28%
Poly_uniform_eta	15%	Poly_uniform	13%	Poly_uniform	22%
NTT	13%	Poly_base_mul	5%	Poly_use_hint	9%
shake256	7%	Poly_uniform_gamma_1	4%	shake256	3%
etc	14%	etc	19%	etc	7%

5 RAM/ROM Memory Usage Analysis

When running a computer program, memory is consumed, and this is typically closely related to variable usage. Since the amount of memory available on a computer is limited, it is generally advisable to minimize memory usage in programs. However, post-quantum cryptography (PQC) tends to be somewhat inefficient in terms of memory usage due to its overall parameter sizes being larger compared to modern cryptography algorithms.

This section evaluates the memory usage of KpqC candidate algorithms. The tool used to assess RAM usage was Valgrind, a type of Dynamic Binary Instrumentation (DBI) program [26]. Valgrind measures memory consumption during program execution and is employed while running the compiled binary file. Valgrind categorizes the consumed memory into stack (static) and heap (dynamic), with the heap category including any additional heap usage (extra_heap).

ROM usage was measured using the 'size' command. The 'size' command provides the code size, data size, and BSS size of the compiled binary file, making it useful for analyzing the program's ROM usage. The sum of the 'text' and 'data' values in the output of the 'size' command represents the ROM usage, which includes all necessary code and initialized data required during the program's execution.

The environment for RAM/ROM measurement was as follows: OS: Ubuntu 22.04, CPU: Intel i5-8259U (3.80 GHz), RAM: 16GB, Compiler: gcc 11.3.0.

Table 16: Measured RAM Usage of KEM Algorithms(unit:bytes, w: with openssl, wo: without openssl)

Algorithm	**stack+heap+extra_heap**	**stack**	**heap**	**extra_heap**
NTRU+KEM-576	19,632	18,600	1,024	8
NTRU+KEM-768	25,024	23,992	1,024	8
NTRU+KEM-864	27,728	26,696	1,024	8
NTRU+KEM-1152	35,824	34,792	1,024	8
NTRU+PKE-576	19,696	18,664	1,024	8
NTRU+PKE-768	25,120	24,088	1,024	8
NTRU+PKE-864	27,824	26,792	1,024	8
NTRU+PKE-1152	35,904	34,872	1,024	8
PALOMA-w-128	2,201,944	1,943,432	197,377	61,135
PALOMA-w-192	7,900,112	7,641,600	197,377	61,135
PALOMA-w-256	9,282,512	9,024,000	197,377	61,135
PALOMA-wo-128	1,944,464	1,943,432	1,024	8
PALOMA-wo-192	5,309,888	5,308,856	1,024	8
PALOMA-wo-256	9,025,032	9,024,000	1,024	8
SMAUG-T1	12,912	10,824	2,472	72
SMAUG-T3	37,888	35,800	2,472	72
SMAUG-T5	37,888	35,800	2,472	72

The results of RAM usage measurements for KEM algorithms are shown in Table 16. When ranked by memory usage in ascending order, the order is SMAUG-T, NTRU+, and PALOMA. Most public-key encryption algorithms showed relatively low heap usage, indicating minimal dynamic memory allocation. However, the stack usage for some algorithms reached several tens of kilobytes, which could make them unsuitable for operation in embedded processor environments. Therefore, optimizing memory usage may be more important than optimizing computational efficiency in these cases.

The results of RAM usage measurements for DSA algorithms are shown in Table 17. When ranked by memory usage in ascending order, the order is NCCSign, HAETAE, AIMer, and MQSign. Similar to public-key encryption algorithms, most digital signature algorithms exhibited relatively low heap usage, indicating minimal dynamic memory allocation. However, some digital signature algorithms have memory usage reaching the MB range, indicating a need for optimization.

The results of ROM usage measurements for KEM algorithms are shown in Table 18. When ranked by memory usage in ascending order, the order is SMAUG-T, NTRU+, and PALOMA. Embedded systems like the Cortex-M4 typically have flash memory (ROM) ranging from 256KB to 1MB. Since program code and initialized data must operate within this limited memory, large

Table 17: Measured RAM Usage of DSA Algorithms(unit:bytes)

Algorithm	**stack+heap+extra_heap**	**stack**	**heap**	**extra_heap**
AIMer-128f	142,256	141,008	1,232	16
AIMer-128s	923,936	922,904	1,024	8
AIMer-192f	313,600	312,352	1,232	16
AIMer-192s	2,035,488	2,034,456	1,024	8
AIMer-256f	649,872	648,624	1,232	16
AIMer-256s	4,182,504	4,181,264	1,232	8
HAETAE-2	369,736	111,224	197,377	61,135
HAETAE-3	432,600	174,088	197,377	61,135
HAETAE-5	481,896	223,384	197,377	61,135
MQSign-MQLR-256-72-46	1,318,960	590,048	728,858	54
MQSign-MQLR-256-112-72	4,963,632	2,219,328	2,744,250	54
MQSign-MQLR-256-148-96	11,584,144	5,187,168	6,396,930	46
MQSign-MQRR-256-72-46	1,185,680	579,400	606,226	54
MQSign-MQRR-256-112-72	4,478,288	2,193,960	2,284,282	46
MQSign-MQRR-256-148-96	10,473,984	5,143,080	5,330,866	38
NCCSign-1	60,712	60,712	0	0
NCCSign-3	80,248	80,248	0	0
NCCSign-5	120,440	120,440	0	0

Table 18: Measured ROM Usage of KEM Algorithms(unit:bytes, w: with openssl, wo: without openssl)

Algorithm	**ROM**	**text**	**data**
NTRU+KEM576	52,906	52,258	648
NTRU+KEM768	51,234	50,586	648
NTRU+KEM864	57,770	57,122	648
NTRU+KEM1152	53,562	52,914	648
NTRU+PKE576	54,763	54,107	656
NTRU+PKE768	54,923	54,267	656
NTRU+PKE864	61,731	61,075	656
NTRU+PKE1152	59,155	58,499	656
PALOMA-w-128	62,030	61,270	760
PALOMA-w-192	61,782	61,022	760
PALOMA-w-256	61,798	61,038	760
PALOMA-wo-128	58,016	57,328	688
PALOMA-wo-192	57,760	57,072	688
PALOMA-wo-256	57,784	57,096	688
SMAUG-T1	47,611	46,867	744
SMAUG-T3	47,395	46,651	744
SMAUG-T5	46,483	45,739	744

Table 19: Measured ROM Usage of DSA Algorithms(unit:bytes)

Algorithm	ROM	text	data
HAETAE2	7,937	7,048	889
HAETAE3	7,976	7,087	889
HAETAE5	7,970	7,081	889
AIMer128f	79,483	78,811	672
AIMer128s	79,579	78,907	672
AIMer192f	87,827	87,155	672
AIMer192s	87,723	89,381	672
AIMer256f	90,059	89,387	672
AIMer256s	90,067	89,395	672
MQ-Sign_MQLR_256_72_46	176,061	175,325	736
MQ-Sign_MQLR_256_112_72	187,373	186,637	736
MQ-Sign_MQLR_256_148_96	187,693	186,957	736
MQ-Sign_MQRR_256_72_46	173,976	173,232	744
MQ-Sign_MQRR_256_112_72	185,928	185,184	744
MQ-Sign_MQRR_256_148_96	186,392	185,648	744
NCCSign-1	153,209	153,209	9,932
NCCSign-3	144,005	139,193	4,812
NCCSign-5	132,805	113,657	19,148

ROM usage can make program execution challenging. While public-key algorithms generally have ROM sizes in the tens of kilobytes, making them seemingly suitable for embedded systems, it is important to note that larger ROM usage can lead to longer boot times and increased power consumption, requiring careful consideration.

The results of ROM usage measurements for DSA algorithms are shown in Table 19. When ranked by memory usage in ascending order, the order is HAETAE, AIMer, NCCSign, and MQSign. Digital signature algorithms tend to use slightly more ROM compared to public-key encryption algorithms, with some algorithms measuring over 100KB of ROM usage. This could pose constraints when implementing these algorithms in certain embedded systems.

6 Key Size and Performance Analysis: NIST PQC Algorithms vs. KpqC Algorithms

Evaluating the efficiency of cryptography schemes involves various perspectives, and algorithm benchmarking is one of the key approaches. For instance, the size of the key, ciphertext, and signature are critical factors in assessing the practicality of a cryptography scheme.

In this section, we conduct a comparative analysis of four NIST PQC standardized algorithms and three NIST PQC Round 4 candidate algorithms (excluding SIKE) against the KpqC Round 2 candidate algorithms.

Table 20: Key Size Comparison: CRYSTALS-Kyber vs. SMAUG-T & NTRU+(unit:bytes)

Rank	Scheme	Public key	Rank	Scheme	Secret key	Rank	Scheme	Ciphertext
1	**SMAUG-T1**	**672**	1	**SMAUG-TiMER**	**136**	1	**SMAUG-TiMER**	**608**
	SMAUG-TiMER	**672**	2	**SMAUG-T1**	**176**	2	**SMAUG-T1**	**672**
3	KYBER-512	800	3	**SMAUG-T5**	**218**	3	KYBER-512	768
4	**NTRU+KEM576**	**864**	4	**SMAUG-T3**	**236**	4	**NTRU+KEM576**	**864**
	NTRU+PKE576	**864**	5	KYBER-512	1,632		**NTRU+PKE576**	**864**
6	**SMAUG-T3**	**1088**	6	**NTRU+KEM576**	**1,760**	6	**SMAUG-T3**	**1,024**
7	**NTRU+KEM768**	**1152**		**NTRU+PKE576**	**1,760**	7	KYBER-768	1,088
	NTRU+PKE768	**1152**	8	**NTRU+KEM768**	**2,336**	8	**NTRU+KEM768**	**1,152**
9	KYBER-768	1,184		**NTRU+PKE768**	**2,336**		**NTRU+PKE768**	**1,152**
10	**NTRU+KEM864**	**1296**	10	KYBER-768	2,400	10	**NTRU+KEM864**	**1,296**
	NTRU+PKE864	**1296**	11	KYBER-1024	2,592		**NTRU+PKE864**	**1,296**
12	KYBER-1024	1,568	12	**NTRU+KEM864**	**2,624**	12	**SMAUG-T5**	**1,472**
13	**NTRU+KEM1152**	**1728**		**NTRU+PKE864**	**2,624**	13	KYBER-1024	1,568
	NTRU+PKE1152	**1728**	14	**NTRU+KEM1152**	**3,488**	14	**NTRU+KEM1152**	**1,728**
15	**SMAUG-T5**	**1792**		**NTRU+PKE1152**	**3,488**		**NTRU+PKE1152**	**1,728**

6.1 Key Size Comparison Between NIST PQC Algorithms and KpqC Algorithms

In the following sections, we conduct a comparative analysis of selected PQC algorithms. To provide a structured and focused comparison, the algorithms have been grouped into four categories based on their underlying cryptography principles and intended use cases: CRYSTALS-Kyber vs. SMAUG-T & NTRU+, Classic McEliece & HQC & BIKE vs. PALOMA & REDOG, CRYSTALS-Dilithium &, FALCON vs. HAETAE & NCC-Sign, and SPHINCS+ vs. AIMer.

- CRYSTALS-Kyber vs. SMAUG-T & NTRU+ This comparison focuses on lattice-based Key Encapsulation Mechanisms (KEMs), where CRYSTALS-Kyber is contrasted with SMAUG-T and NTRU+. The commonality in their lattice-based foundations allows for a nuanced analysis of their key size. The comparison results for Key Size Comparison: CRYSTALS-Kyber vs. SMAUG-T & NTRU+ are presented in Table 20.

The results of the comparison of algorithms with 128-bit security strength are as follows: SMAUG-T1's public key is shorter than that of Kyber-512, a NIST PQC algorithm, while NTRU+576's public key size is nearly identical to Kyber-512. In summary, SMAUG-T1 has the shortest public key among the compared algorithms.

The results of the comparison of secret key sizes for algorithms with NIST security level 1 (128-bit security strength) are as follows: SMAUG-TiMER has

the shortest secret key, followed by SMAUG-T1, which is shorter than that of Kyber-512. In summary, SMAUG-T1 has a shorter secret key than Kyber-512, with SMAUG-TiMER being the shortest overall.

The results of the comparison of ciphertext sizes for algorithms with NIST security level 1 (128-bit security strength) are as follows: SMAUG-TiMER has the shortest ciphertext, followed by SMAUG-T1, which is shorter than Kyber-512, with NTRU+576 having the longest. In summary, SMAUG-T1's ciphertext is shorter than Kyber-512's, with SMAUG-TiMER having the shortest overall.

Table 21: Key Size Comparison: Classic McEliece & HQC & BIKE vs. PALOMA & REDOG(unit:bytes)

Rank	Scheme	Public key	Rank	Scheme	Secret key	Rank	Scheme	Ciphertext
1	HQC-128	2,249	1	**REDOG-1**	**666**	1	mceliece348864	96
2	**REDOG-1**	**4,270**	2	**REDOG-2**	**1,464**	2	**PALOMA128**	**136**
3	HQC-192	4,522	3	BIKE-1	2,244	3	mceliece460896	156
4	HQC-256	7,245	4	HQC-128	2,289	4	mceliece6688128	194
5	BIKE-1	12,323	5	**REDOG-3**	**2,560**	5	mceliece6960119	208
6	**REDOG-3**	**13,987**	6	BIKE-3	3,346		mceliece8192128	208
7	BIKE-3	24,659	7	HQC-192	4,562	7	**PALOMA192**	**240**
8	**REDOG-5**	**32,634**	8	BIKE-5	4,640		**PALOMA256**	**240**
9	BIKE-5	40,973	9	mceliece348864	6,492	9	**REDOG-1**	**389**
10	mceliece348864	261,120	10	HQC-256	7,285	10	**REDOG-2**	**840**
11	**PALOMA128**	**319,488**	11	mceliece460896	13,608	11	**REDOG-3**	**1,475**
12	mceliece460896	524,160	12	mceliece6688128	13,932	12	HQC-128	4,497
13	**PALOMA192**	**812,032**	13	mceliece6960119	13,948	13	HQC-192	9,042
14	**PALOMA256**	**1,025,024**	14	mceliece8192128	14,120	14	BIKE-1	12,579
15	mceliece6688128	1,044,992	15	**PALOMA128**	**94,528**	15	HQC-256	14,485
16	mceliece6960119	1,047,319	16	**PALOMA192**	**357,568**	16	BIKE-3	24,915
17	mceliece8192128	1,357,824	17	**PALOMA256**	**359,616**	17	BIKE-5	41,229

- Classic McEliece & HQC & BIKE vs. PALOMA & REDOG This comparison focuses on code-based cryptography schemes. Classic McEliece, HQC, and BIKE, which are code-based algorithms, are analyzed against PALOMA and REDOG.

The comparison results for Key Size Comparison: Classic McEliece & HQC & BIKE vs. PALOMA & REDOG are presented in Table 21

The results of the comparison of public key sizes are as follows: HQC-128 has the smallest public key at 2,249 bytes, followed by REDOG-1 at 4,270 bytes. Among all algorithms, HQC-128 offers the smallest public key size.

The results of the comparison of secret key sizes are as follows: REDOG-1 has the shortest secret key at 666 bytes, followed by REDOG-2 at 1,464 bytes. Although BIKE-1 at 2,244 bytes and HQC-128 at 2,249 bytes have larger secret keys than REDOG-2, they are still relatively compact.

The results of the comparison of ciphertext sizes are as follows: mceliece348864 has the smallest ciphertext at 96 bytes, followed by PALOMA128 at 136 bytes. In summary, mceliece348864 has the shortest ciphertext overall, with PALOMA128 also being among the more compact options.

- CRYSTALS-Dilithium &. FALCON vs. HAETAE & NCC-Sign This comparison focuses on lattice-based digital signature schemes. CRYSTALS-Dilithium and FALCON, two prominent lattice-based signature algorithms, are compared with HAETAE and NCC-Sign. The comparison sheds light on their signature sizes. The comparison results for Key Size Comparison: CRYSTALS-Dilithium &. FALCON vs. HAETAE & NCC-Sign are presented in Table 22

The results of the comparison of public key sizes are as follows: FALCON-512 has the smallest public key at 897 bytes, followed by HAETAE2 at 992 bytes. In summary, FALCON-512 has the smallest public key overall, with HAETAE2 also being among the more compact options.

The results of the comparison of secret key sizes are as follows: FALCON-512 has the smallest secret key at 1,281 bytes, followed by HAETAE2 at 1,408 bytes. Although HAETAE3 at 2,112 bytes and FALCON-1024 at 3,305 bytes have larger secret keys than FALCON-512 and HAETAE2, they remain relatively compact compared to other algorithms. In summary, FALCON-512 has the smallest secret key overall, with HAETAE2 also being relatively compact.

The results of the comparison of signature sizes are as follows: FALCON-512 has the smallest signature size at 666 bytes, followed by FALCON-1024 at 1,280 bytes. On the other hand, HAETAE2 and HAETAE3 have slightly larger signature sizes at 1,474 bytes and 2,349 bytes, respectively, while NCC-Sign1 has a significantly larger signature size of 2,912 bytes. In summary, FALCON-512 has the smallest signature size overall, with FALCON-1024 and HAETAE2 being among the more compact options.

- SPHINCS+ vs. AIMer Finally, we compare SPHINCS+, a stateless hash-based signature scheme, with AIMer. The comparison results for key size comparison: SPHINCS+ vs. AIMer are presented in Table 23

The results of the comparison of public key sizes are as follows: AIMer128 and SPHINCS+128 all have the smallest public key sizes at 32 bytes. Following these, AIMer192 and SPHINCS+192 has a slightly larger public key at 48 bytes.

The results of the comparison of secret key sizes are as follows: AIMer128 has the smallest secret keys at 48 bytes, followed by SPHINCS+128 at 64 bytes.

Table 22: Key Size Comparison: CRYSTALS-Dilithium & FALCON vs. HAETAE & NCC-Sign(unit:bytes)

Rank	Scheme	Public key	Rank	Scheme	Secret key	Rank	Scheme	Signature
1	FALCON-512	897	1	FALCON-512	1,281	1	FALCON-512	666
2	**HAETAE2**	**992**	2	**HAETAE2**	**1,408**	2	FALCON-1024	1,280
3	Dilithium-2	1,312	3	**HAETAE3**	**2,112**	3	**HAETAE2**	**1,474**
4	**HAETAE3**	**1,472**	**4**	FALCON-1024	2,305	4	**HAETAE3**	**2,349**
5	**NCC-Sign1**	**1,760**	**5**	Dilithium-2	2,528	5	Dilithium-2	2,420
6	FALCON-1024	1,793	6	**NCC-Sign1**	**2,688**	6	**NCC-Sign1**	**2,912**
7	Dilithium-3	1,952	7	**HAETAE5**	**2,752**	7	**HAETAE5**	**2,948**
8	**HAETAE5**	**2,080**	8	**NCC-Sign3**	**3,552**	8	Dilithium-3	3,293
9	**NCC-Sign3**	**2,336**	9	Dilithium-3	4,000	9	**NCC-Sign3**	**3,872**
10	Dilithium-5	2,592	10	Dilithium-5	4,864	10	Dilithium-5	4,595
11	**NCC-Sign5**	**3,200**	11	**NCC-Sign5**	**5,568**	11	**NCC-Sign5**	**6,080**

The results of the comparison of signature sizes are as follows: AIMer128s has the smallest signature size at 4,160 bytes, followed by AIMer128f at 5,888 bytes. SPHINCS+128s has a signature size of 7,856 bytes. In summary, AIMer128s has the smallest signature size overall.

6.2 Performance Comparison Between NIST PQC Algorithms and KpqC Algorithms

We benchmarked the NIST PQC and KpqC algorithms on both Intel and ARM processors to compare their performance. The benchmarking environment adhered to the specifications described in Table 24.

To obtain the measurements, each algorithm went through 10,000 iterations, and the average number of clock cycles required for each round of operation was calculated. The -O3 optimization level (fastest) was applied.

- CRYSTALS-Kyber vs. SMAUG-T & NTRU+ The comparison results for performance comparison: CRYSTALS-Kyber vs. SMAUG-T & NTRU+ are presented in Table 25 and Table 26. These tables represent the performance measurements conducted on Intel and ARM processors, respectively.

In the key generation, encapsulation, and decapsulation processes, SMAUG-TiMER and SMAUG-T1 consistently demonstrated the best performance on both Intel and ARM processors. SMAUG-TiMER was the most efficient across all operations, with SMAUG-T1 closely following.

Table 23: Key Size Comparison: SPHINCS+ vs. AIMer(unit:bytes)

Rank	Scheme	Public key	Rank	Scheme	Secret key	Rank	Scheme	Signature
1	**AIMer128f**	**32**	1	**AIMer128f**	**48**	1	**AIMer128s**	**4,160**
	AIMer128s	**32**		**AIMer128s**	**48**	2	**AIMer128f**	**5,888**
	SPHINCS+128s	32	3	SPHINCS+128s	64	3	SPHINCS+128s	7,856
	SPHINCS+128f	32		SPHINCS+128f	64	4	**AIMer192s**	**9,120**
5	**AIMer192f**	**48**	5	**AIMer192f**	**72**	5	**AIMer192f**	**13,056**
	AIMer192s	**48**		**AIMer192s**	**72**	6	SPHINCS+192s	16,224
	SPHINCS+192f	48	7	**AIMer256f**	**96**	7	**AIMer256s**	**17,056**
	SPHINCS+192s	48		**AIMer256s**	**96**	8	SPHINCS+128f	17,088
9	SPHINCS+256f	49		SPHINCS+192f	96	9	**AIMer256f**	**25,120**
	SPHINCS+256s	49		SPHINCS+192s	96	10	SPHINCS+256s	29,792
11	**AIMer256f**	**64**	11	SPHINCS+256f	128	11	SPHINCS+192f	35,664
	AIMer256s	**64**		SPHINCS+256s	128	12	SPHINCS+256f	49,856

Table 24: Specifications of Target Environments for Performance Comparison

	Intel processors	ARM processors
OS	Ubuntu 23.10.1	MacOS Sonoma 14.5
CPU	Intel i5-8259U(2.30GHz)	Apple M1(3.2GHz)
RAM	16GB	8GB
Compiler	gcc 13.2.0	Apple clang 15.0.0
Optimization Level	-O3	-O3

In key generation, SMAUG-TiMER required 40,727 cc on Intel and 55,086 cc on ARM, while SMAUG-T1 needed 42,812 cc on Intel and 57,506 cc on ARM. Kyber512, though faster than NTRU+576 in key generation, was less efficient in encapsulation and decapsulation.

For encapsulation, SMAUG-TiMER led with 39,849 cycles on Intel and 39,490 on ARM. SMAUG-T1 showed similar results, followed by NTRU+576, which outperformed Kyber512.

In decapsulation, SMAUG-TiMER again had the lowest cycle count, with SMAUG-T1 close behind. NTRU+576 surpassed Kyber512 in both encapsulation and decapsulation efficiency.

Overall, SMAUG-TiMER and SMAUG-T1 were the top performers across all operations, while Kyber512 excelled only in key generation, with NTRU+576 proving more effective in the remaining processes.

- Classic McEliece & HQC & BIKE vs. PALOMA The comparison results for performance comparison: Classic McEliece & HQC & BIKE vs. PALOMA

Table 25: Performance Comparison: CRYSTALS-Kyber vs. SMAUG-T & NTRU+(Intel)(unit:clock cycles)

Rank	Scheme	Keygen	Rank	Scheme	Encap	Rank	Scheme	Decap
1	**SMAUG-TiMER**	**40,727**	1	**SMAUG-TiMER**	**39,849**	1	**SMAUG-TiMER**	**48,191**
2	**SMAUG-T1**	**42,812**	2	**SMAUG-T1**	**39,907**	2	**SMAUG-T1**	**48,874**
3	**SMAUG-T3**	**69,125**	3	**NTRU+PKE576**	**56,945**	3	**NTRU+KEM576**	**64,897**
4	kyber512	110,653	4	**NTRU+KEM576**	**57,392**	4	**NTRU+PKE576**	**65,513**
5	**SMAUG-T5**	**119,648**	5	**SMAUG-T3**	**64,581**	5	**SMAUG-T3**	**78,558**
6	**NTRU+KEM576**	**152,653**	6	**NTRU+PKE768**	**73,866**	6	**NTRU+PKE768**	**85,777**
7	**NTRU+PKE576**	**162,109**	7	**NTRU+KEM768**	**74,605**	7	**NTRU+KEM768**	**85,814**
8	**NTRU+KEM768**	**170987**	8	**NTRU+PKE864**	**83,385**	8	**NTRU+KEM864**	**100,583**
9	**NTRU+PKE768**	**172444**	9	**NTRU+KEM864**	**83,862**	9	**NTRU+PKE864**	**100,620**
10	**NTRU+KEM864**	**175,592**	10	**NTRU+PKE1152**	**108,907**	10	**SMAUG-T5**	**130,231**
11	**NTRU+PKE864**	**180,081**	11	**NTRU+KEM1152**	**108,935**	11	**NTRU+PKE1152**	**137,555**
12	kyber768	198651	12	**SMAUG-T5**	**114,105**	12	**NTRU+KEM1152**	**137,606**
13	kyber1024	266290	13	kyber512	125,375	13	kyber512	143,529
14	**NTRU+PKE1152**	**420607**	14	kyber768	174,804	14	kyber768	210,180
15	**NTRU+KEM1152**	**425625**	15	kyber1024	244,419	15	kyber1024	290,399

Table 26: Performance Comparison: CRYSTALS-Kyber vs. SMAUG-T & NTRU+(ARM)(unit:clock cycles)

Rank	Scheme	Keygen	Rank	Scheme	Encap	Rank	Scheme	Decap
1	**SMAUG-TiMER**	**55,086**	1	**SMAUG-TiMER**	**39,490**	1	**SMAUG-TiMER**	**49,092**
2	**SMAUG-T1**	**57,506**	2	**SMAUG-T1**	**39,552**	2	**SMAUG-T1**	**50,360**
3	kyber512	70,685	3	**NTRU+KEM576**	**56,914**	3	**NTRU+KEM576**	**64,959**
4	**SMAUG-T3**	**93,432**	4	**NTRU+PKE576**	**56,949**	4	**NTRU+PKE576**	**65,127**
5	kyber768	109,502	5	**SMAUG-T3**	**64,883**	5	**SMAUG-T3**	**79,752**
6	**SMAUG-T5**	**143,646**	6	kyber512	71,735	6	**NTRU+KEM768**	**85,811**
7	**NTRU+PKE576**	**153,560**	7	**NTRU+PKE768**	**73,727**	7	**NTRU+PKE768**	**86,246**
8	**NTRU+KEM576**	**154,387**	8	**NTRU+KEM768**	**74,319**	8	kyber512	87,617
9	**NTRU+PKE768**	**158,368**	9	**NTRU+PKE864**	**83,296**	9	**NTRU+KEM864**	**100,823**
10	**NTRU+KEM768**	**158,560**	10	**NTRU+KEM864**	**83,401**	10	**NTRU+PKE864**	**101,003**
11	**NTRU+KEM864**	**164,433**	11	**NTRU+PKE1152**	**108,924**	11	**SMAUG-T5**	**134,162**
12	kyber1024	165,118	12	**NTRU+KEM1152**	**108,967**	12	kyber768	134,352
13	**NTRU+PKE864**	**201,859**	13	kyber768	111,051	13	**NTRU+KEM1152**	**137,549**
14	**NTRU+KEM1152**	**389,639**	14	**SMAUG-T5**	**116,730**	14	**NTRU+PKE1152**	**138,082**
15	**NTRU+PKE1152**	**474,510**	15	kyber1024	163,171	15	kyber1024	195,524

Table 27: Performance Comparison: Classic McEliece & HQC & BIKE vs. PALOMA(Intel)(unit:clock cycles)

Rank	Scheme	Keygen	Rank	Scheme	Encap	Rank	Scheme	Decap
1	hqc-128	4,308,666	1	**PALOMA128**	**87,441**	1	**PALOMA128**	**8,033,779**
2	hqc-192	12,098,383	2	**PALOMA192**	**136,963**	2	hqc-128	12,077,461
3	hqc-256	22,078,541	3	mceliece348864f	171,826	3	hqc-192	36,473,745
4	mceliece348864f	115,143,829	4	mceliece348864	178,119	4	mceliece348864f	38,731,109
5	**PALOMA128**	**147,357,235**	5	**PALOMA256**	**200,987**	5	mceliece348864	38,777,040
6	mceliece348864	257,322,142	6	mceliece460896f	355,743	6	**PALOMA192**	**41,802,948**
7	mceliece460896f	353,874,031	7	mceliece460896	357,513	7	**PALOMA256**	**45,735,541**
8	mceliece6960119f	568,091,516	8	mceliece8192128f	465,714	8	hqc-256	66,741,738
9	mceliece8192128f	617,997,240	9	mceliece8192128	503,102	9	mceliece460896	93,483,536
10	mceliece6688128f	653,450,392	10	mceliece6688128	563,525	10	mceliece460896f	93,624,278
11	**PALOMA192**	**700,965,299**	11	mceliece6688128f	587,370	11	mceliece6960119f	174,013,196
12	mceliece460896	804,708,960	12	mceliece6960119	1,828,378	12	mceliece6960119	174,166,372
13	**PALOMA256**	**819,112,319**	13	mceliece6960119f	1,847,889	13	mceliece6688128f	179,210,745
14	mceliece6960119	1,640,855,766	14	hqc-128	8,551,947	14	mceliece6688128f	180,083,946
15	mceliece6688128	1,886,622,666	15	hqc-192	24,193,559	15	mceliece8192128f	219,454,147
16	mceliece8192128	1,951,587,331	16	hqc-256	44,083,667	16	mceliece8192128	219,602,742

are presented in Table 27 and Table 28. These tables represent the performance measurements conducted on Intel and ARM processors, respectively.

In key generation, HQC128 performs best on both Intel and ARM processors, requiring 4,308,666 cc on Intel and 1,504,516 cc on ARM, making it the most efficient algorithm among the compared algorithms.

For encapsulation, PALOMA128 outperformed all others with 87,441 cc on Intel and 57,476 cc on ARM. PALOMA192 and PALOMA256 also showed competitive results.

In decapsulation, PALOMA128 led on Intel with 8,033,779 cc, while HQC128 was the top performer on ARM with 4,624,790 cc. This indicates that PALOMA128 is well-suited for Intel, while HQC128 excels on ARM.

PALOMA algorithms consistently outperformed Classic McEliece in both encapsulation and decapsulation across all platforms. For instance, on Intel, PALOMA128 significantly outperformed the highest-ranking Classic McEliece (mceliece348864f) in both operations.

- CRYSTALS-Dilithium &. FALCON vs. HAETAE & NCC-Sign The comparison results for performance comparison: CRYSTALS-Dilithium &. FALCON vs. HAETAE & NCC-Sign are presented in Table 29 and Table 30. These

Table 28: Performance Comparison: Classic McEliece & HQC & BIKE vs. PALOMA(ARM)(unit:clock cycles)

Rank	Scheme	Keyget	Rank	Scheme	Encap	Rank	Scheme	Decap
1	hqc-128	1,504,516	1	**PALOMA128**	**57,476**	1	hqc-128	4,624,790
2	hqc-192	4,499,872	2	**PALOMA192**	**80,913**	2	**PALOMA128**	**8,177,108**
3	hqc-256	8,181,740	3	**PALOMA256**	**95,934**	3	hqc-192	13,685,601
4	mceliece348864f	104,286,620	4	mceliece348864f	169,711	4	hqc-256	15,196,978
5	**PALOMA128**	**121,551,322**	5	mceliece348864	170,002	5	**PALOMA192**	**41,189,957**
6	mceliece348864	220,493,568	6	mceliece8192128f	390,583	6	**PALOMA256**	**43,095,506**
7	mceliece460896f	294,991,120	7	mceliece8192128	390,685	7	mceliece348864f	61,817,900
8	mceliece6960119f	519,756,682	8	mceliece460896f	431,503	8	mceliece348864	61,821,538
9	mceliece6688128f	547,192,847	9	mceliece460896	432,698	9	mceliece460896	129,451,256
10	**PALOMA192**	**579,462,770**	10	mceliece6688128f	661,603	10	mceliece460896f	129,624,278
11	mceliece8192128f	597,865,145	11	mceliece6688128	663,925	11	mceliece6960119	240,555,949
12	mceliece460896	634,445,332	12	mceliece6960119f	2,339,770	12	mceliece6960119f	240,459,344
13	**PALOMA256**	**691,203,675**	13	mceliece6960119	2,358,294	13	mceliece6688128f	248,341,895
14	mceliece6960119	1,337,245,665	14	hqc-128	3,031,270	14	mceliece6688128f	248,477,346
15	mceliece6688128	1,486,936,619	15	hqc-192	9,079,169	15	mceliece8192128f	303,876,177
16	mceliece8192128	1,638,084,312	16	hqc-256	16,738,076	16	mceliece8192128	303,877,879

tables represent the performance measurements conducted on Intel and ARM processors, respectively.

In the key generation process, NCC-Sign1 consistently demonstrated the best performance, requiring only 205,450 clock cycles on Intel and 165,845 clock cycles on ARM, making it the most efficient algorithm in this category. Dilithium2 followed closely, with slightly higher clock cycles on both platforms.

For the signing operation, Dilithium2 emerged as the top performer on both processors, completing the task in 450,483 clock cycles on Intel and 306,143 clock cycles on ARM. NCC-Sign1 also showed strong performance, particularly on the ARM processor, where it ranked second with 358,150 clock cycles.

In the verification process, Falcon512 outperformed other algorithms, demonstrating the highest efficiency with 132,067 clock cycles on Intel and 91,243 clock cycles on ARM. This highlights Falcon512's suitability for applications where fast verification is essential. HAETAE2 and NCC-Sign1 also performed well in verification, ranking second and third on ARM and Intel, respectively.

Overall, the results indicate that NCC-Sign1 excels in key generation, Dilithium2 is most efficient in signing, and Falcon512 leads in verification across both Intel and ARM platforms.

Table 29: Performance Comparison: CRYSTALS-Dilithium &. FALCON vs. HAETAE & NCC-Sign(Intel)(unit:clock cycles)

Rank	Scheme	keygen	Rank	Scheme	Sign	Rank	Scheme	Verify
1	**NCC-Sign1**	**205,450**	1	dilithium2	450,483	1	falcon-512	132,067
2	dilithium2	278,954	2	**NCC-Sign3**	**884,954**	2	falcon-padded-512	133,669
3	**NCC-Sign3**	**280,512**	3	**NCC-Sign5**	**1,049,172**	3	**HAETAE2**	**170,103**
4	**NCC-Sign5**	**416,020**	4	**NCC-Sign1**	**1,161,522**	4	falcon-padded-1024	281,205
5	dilithium3	482,104	5	dilithium5	1,463,427	5	falcon-1024	281,777
6	dilithium5	734,609	6	**HAETAE2**	**1,866,070**	6	dilithium2	281,907
7	**HAETAE2**	**1,067,696**	7	dilithium3	2,465,069	7	**HAETAE3**	**285,845**
8	**HAETAE3**	**2,155,146**	8	**HAETAE5**	**3,389,114**	8	**NCC-Sign1**	**288,198**
9	**HAETAE5**	**2,225,344**	9	**HAETAE3**	**7,057,356**	9	**HAETAE5**	**355,867**
10	falcon-padded-512	37,897,367	10	falcon-512	10,617,704	10	**NCC-Sign3**	**391,048**
11	falcon-512	37,959,474	11	falcon-padded-512	10,959,368	11	dilithium3	440,805
12	falcon-1024	108,091,196	12	falcon-1024	23,159,283	12	**NCC-Sign5**	**630,294**
12	falcon-padded-1024	108,252,565	13	falcon-padded-1024	23,186,413	13	dilithium5	737,849

Table 30: Performance Comparison: CRYSTALS-Dilithium &. FALCON vs. HAETAE & NCC-Sign(ARM)(unit:clock cycles)

Rank	Scheme	keygen	Rank	Scheme	Sign	Rank	Scheme	Verify
1	**NCC-Sign1**	**165,845**	1	Dilithium2	306,143	1	Falcon-512	91,243
2	Dilithium2	190,398	2	**NCC-Sign1**	**358,150**	2	**HAETAE2**	**137,345**
3	**NCC-Sign3**	**218,097**	3	**NCC-Sign3**	**460,644**	3	Falcon-1024	182,233
4	**NCC-Sign5**	**333,251**	4	**NCC-Sign5**	**745,970**	4	Dilithium2	195,294
5	Dilithium3	362,409	5	**HAETAE2**	**968,079**	5	**NCC-Sign1**	**219,515**
6	Dilithium5	540,914	6	Dilithium5	1,023,099	6	**HAETAE3**	**240,094**
7	**HAETAE2**	**975,752**	7	Dilithium3	1,741,397	7	**NCC-Sign3**	**281,971**
8	**HAETAE3**	**1,936,358**	8	**HAETAE3**	**2,147,559**	8	**HAETAE5**	**301,920**
9	**HAETAE5**	**2,115,228**	9	**HAETAE5**	**3,457,214**	9	Dilithium3	321,151
10	Falcon-512	35,583,909	10	Falcon-512	10,044,417	10	**NCC-Sign5**	**448,855**
11	Falcon-1024	106,966,738	11	Falcon-1024	21,844,507	11	Dilithium5	542,833

- SPHINCS+ vs. AIMer The comparison results for performance comparison: SPHINCS+ vs. AIMer are presented in Table 31 and Table 32. These tables

represent the performance measurements conducted on Intel and ARM processors, respectively.

Table 31: Performance Comparison: SPHINCS+ vs. AIMer(Intel)(unit:clock cycles)

Rank	Scheme	keygen	Rank	Scheme	Sign	Rank	Scheme	Verify
1	**AIMer128s**	**85,760**	1	**AIMer128f**	**7,020,534**	1	sphincs-sha2-128s-simple	1,769,297
2	**AIMer128f**	**86,858**	2	**AIMer192f**	**13,314,965**	2	sphincs-sha2-192s-simple	2,394,749
3	**AIMer192s**	**221,051**	3	**AIMer256f**	**34,079,571**	3	sphincs-sha2-128s-simple	2,763,529
4	**AIMer192f**	**221,939**	4	**AIMer128s**	**54,057,979**	4	sphincs-sha2-256s-simple	3,554,691
5	**AIMer256f**	**532,954**	5	sphincs-sha2-128f-simple	81,191,592	5	sphincs-shake-192s-simple	3,872,933
6	**AIMer256s**	**536,662**	6	**AIMer192s**	**103,955,683**	6	sphincs-sha2-128f-simple	4,989,027
7	sphincs-sha2-128f-simple	3,512,168	7	sphincs-shake-128f-simple	131,239,147	7	sphincs-shake-256s-simple	5,538,012
8	sphincs-sha2-192f-simple	5,106,785	8	sphincs-sha2-192f-simple	133,555,092	8	**AIMer128f**	**6,429,942**
9	sphincs-shake-128f-simple	5,676,001	9	sphincs-shake-192f-simple	212,061,614	9	sphincs-sha2-256f-simple	7,081,467
10	sphincs-shake-192f-simple	8,233,255	10	**AIMer256s**	**263,703,008**	10	sphincs-sha2-192f-simple	7,123,456
11	sphincs-sha2-256f-simple	13,540,513	11	sphincs-sha2-256f-simple	272,214,600	11	sphincs-sha2-128f-simple	7,812,153
12	sphincs-shake-256f-simple	21,778,482	12	sphincs-shake-256f-simple	440,803,080	12	sphincs-shake-192f-simple	11,374,784
13	sphincs-sha2-256s-simple	213,627,804	13	sphincs-sha2-128s-simple	1,690,989,832	13	sphincs-shake-256f-simple	11,778,032
14	sphincs-sha2-128s-simple	222,113,099	14	sphincs-sha2-256s-simple	2,630,577,802	14	**AIMer192f**	**13,025,861**
15	sphincs-sha2-192s-simple	323,514,039	15	sphincs-sha2-128s-simple	2,725,259,275	15	**AIMer256f**	**31,939,622**
16	sphincs-shake-256s-simple	346,536,078	16	sphincs-sha2-192s-simple	2,989,370,882	16	**AIMer128s**	**53,876,227**
17	sphincs-shake-128s-simple	359,032,336	17	sphincs-shake-256s-simple	4,124,025,081	17	**AIMer192s**	**102,567,956**
18	sphincs-sha2-192s-simple	524,022,262	18	sphincs-shake-192s-simple	4,710,044,154	18	**AIMer256s**	**257,224,646**

In both Intel and ARM processors, the AIMer algorithms consistently demonstrated superior performance. For key generation, AIMer128 proved to be the most efficient, with all AIMer parameters significantly outperforming those of SPHINCS+.

In signing operations, AIMer128f achieved the best results, and overall, the AIMer algorithms outperformed SPHINCS+ across the board.

However, in the verification process, SPHINCS+ generally exhibited better performance than the AIMer algorithms.

In summary, AIMer algorithms were the most effective in key generation and signing, while SPHINCS+ was more efficient in verification on both Intel and ARM processors.

7 Conclusion

This paper presents the benchmarking efforts performed on the candidate algorithms of KpqC Round 2. KpqClean Ver2, the successor to the KpqClean library, is now available on GitHub[2]. The goal of KpqClean Ver2, as with pre-

[2] https://github.com/kpqc-cryptocraft/KpqClean_ver2

Table 32: Performance Comparison: SPHINCS+ vs. AIMer(ARM)(unit:clock cycles)

Rank	Scheme	keygen(avg)	Rank	Scheme	Sign(avg)	Rank	Scheme	Verify(avg)
1	**AIMer128f**	**56,314**	1	**AIMer128f**	**2,324,875**	1	sphincs-sha2-128s-simple	1,691,069
2	**AIMer128s**	**56,395**	2	**AIMer192f**	**5,887,515**	2	sphincs-shake-128s-simple	1,826,203
3	**AIMer192f**	**123,386**	3	**AIMer256f**	**11,554,222**	3	**AIMer128f**	**2,158,816**
4	**AIMer192s**	**169,004**	4	**AIMer128s**	**18,404,651**	4	sphincs-sha2-192s-simple	2,695,583
5	**AIMer256s**	**291,148**	5	**AIMer192s**	**45,924,420**	5	sphincs-shake-192s-simple	2,711,920
6	**AIMer256f**	**292,269**	6	sphincs-sha2-128f-simple	84,542,863	6	sphincs-sha2-256s-simple	3,758,715
7	sphincs-sha2-128f-simple	3,649,196	7	**AIMer256s**	**88,425,057**	7	sphincs-shake-256s-simple	3,998,496
8	sphincs-shake-128f-simple	3,952,483	8	sphincs-shake-128f-simple	92,399,930	8	sphincs-sha2-128f-simple	5,133,120
9	sphincs-sha2-192f-simple	5,312,027	9	sphincs-sha2-192f-simple	138,373,380	9	**AIMer192f**	**5,482,033**
10	sphincs-shake-192f-simple	5,800,574	10	sphincs-shake-192f-simple	148,824,161	10	sphincs-shake-128f-simple	5,565,225
11	sphincs-sha2-256f-simple	13,962,889	11	sphincs-sha2-256f-simple	284,393,734	11	sphincs-sha2-192f-simple	7,429,704
12	sphincs-shake-256f-simple	15,202,018	12	sphincs-shake-256f-simple	305,656,621	12	sphincs-sha2-256f-simple	7,765,007
13	sphincs-sha2-256s-simple	222,570,090	13	sphincs-sha2-128s-simple	1,750,772,084	13	sphincs-shake-192f-simple	8,013,609
14	sphincs-sha2-128s-simple	230,498,741	14	sphincs-sha2-128s-simple	1,915,535,299	14	sphincs-shake-256f-simple	8,148,539
15	sphincs-shake-256s-simple	242,995,939	15	sphincs-sha2-256s-simple	2,746,781,740	15	**AIMer256f**	**10,770,937**
16	sphincs-sha2-128s-simple	252,420,789	16	sphincs-sha2-128s-simple	2,894,620,586	16	**AIMer128s**	**18,288,170**
17	sphincs-sha2-192s-simple	337,468,644	17	sphincs-sha2-192s-simple	3,103,763,239	17	**AIMer192s**	**45,489,467**
18	sphincs-sha2-192s-simple	367,897,731	18	sphincs-sha2-192s-simple	3,310,939,009	18	**AIMer256s**	**87,568,082**

vious versions, is to provide benchmark results for KpqC candidate algorithms in a standardized environment. This project aims to present benchmark results in an integrated environment while offering a more convenient KpqC library. Currently, it includes "clean" and "avx2" implementations, with plans to add an "m4" implementation in the future. The comprehensive benchmark analysis results can serve as valuable insights for understanding and researching the KpqC cryptographic algorithm in the future. Our efforts are aimed at increasing interest among researchers in the field of KpqC and providing a comfortable and conducive environment for optimization implementation, algorithm analysis, and further research and exploration.

References

1. "Nist pqc project." `https://csrc.nist.gov/Projects/post-quantum-cryptography`. Accessed : 2023-10-06.
2. R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber algorithm specifications and supporting documentation," *NIST PQC Round*, vol. 2, no. 4, 2019.
3. L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-Dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.

4. P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over NTRU," *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, 2018.
5. D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS+ signature framework," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 2129–2146, 2019.
6. R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, *et al.*, "Supersingular isogeny key encapsulation," *Submission to the NIST Post-Quantum Standardization project*, vol. 152, pp. 154–155, 2017.
7. N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, *et al.*, "Bike: bit flipping key encapsulation," 2022.
8. D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, *et al.*, "Classic McEliece: conservative code-based cryptography," *NIST submissions*, 2017.
9. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "Hamming quasi-cyclic (hqc)," *NIST PQC Round*, vol. 2, no. 4, p. 13, 2018.
10. "Kpqc competition." `https://kpqc.or.kr/competition.html`. Accessed: 2023-07-30.
11. D.-C. Kim, C.-Y. Jeon, Y. Kim, and M. Kim, "Paloma: binary separable goppa-based kem," in *Code-Based Cryptography Workshop*, pp. 144–173, Springer, 2023.
12. J.-L. Kim, J. Hong, T. S. C. Lau, Y. Lim, and B.-S. Won, "Redog and its performance analysis," *Cryptology ePrint Archive*, 2022.
13. J. Kim and J. H. Park, "Ntru+: compact construction of ntru using simple encoding method," *IEEE Transactions on Information Forensics and Security*, 2023.
14. "Smaug-t: the key exchange algorithm based on module-lwe and module-lwr." `https://kpqc.or.kr/images/pdf/SMAUG-T_Document.pdf`. Accessed : 2024-02-23.
15. J. H. Cheon, H. Choe, D. Hong, and M. Yi, "Smaug: pushing lattice-based key encapsulation mechanisms to the limits," in *International Conference on Selected Areas in Cryptography*, pp. 127–146, Springer, 2023.
16. S. Park, C.-G. Jung, A. Park, J. Choi, and H. Kang, "Tiger: tiny bandwidth key encapsulation mechanism for easy migration based on rlwe (r)," *Cryptology ePrint Archive*, 2022.
17. J. H. Cheon, H. Choe, J. Devevey, T. Güneysu, D. Hong, M. Krausz, G. Land, M. Möller, D. Stehlé, and M. Yi, "Haetae: Shorter lattice-based fiat-shamir signatures," *Cryptology ePrint Archive*, 2023.
18. K.-A. Shim, J. Kim, and H. Kwon, "Ncc-sign: A new lattice-based signature scheme using non-cyclotomic polynomials and trinomials," *KpqC Round*, vol. 1, 2024.
19. K.-A. Shim and H. Kwon, "Mq-sign: A new post-quantum signature scheme based on multivariate quadratic equations: Shorter and faster," *KpqC Round*, vol. 1, 2024.
20. S. Kim, J. Ha, and J. Lee, "Aim: Symmetric primitive for shorter signatures with stronger security," in *ACM CCS 2023: ACM Conference on Computer and Communications Security*, ACM (Association for Computing Machinery), 2023.
21. H. Kwon, M. Sim, G. Song, M. Lee, and H. Seo, "Evaluating kpqc algorithm submissions: Balanced and clean benchmarking approach," in *International Conference on Information Security Applications*, pp. 338–348, Springer, 2023.
22. "PQClean project." Available online: `https://github.com/PQClean/PQClean`. Accessed: 2022-07-29.

23. Y. Choi, M. Kim, Y. Kim, J. Song, J. Jin, H. Kim, and S. C. Seo, "Kpqbench: Performance and implementation security analysis of kpqc competition round 1 candidates," *IEEE Access*, 2024.
24. J. Viega, M. Messier, and P. Chandra, *Network security with openSSL: cryptography for secure communications.* " O'Reilly Media, Inc.", 2002.
25. D. T. Nguyen and K. Gaj, "Optimized software implementations of crystals-kyber, ntru, and saber using neon-based special instructions of armv8," in *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*, 2021.
26. "Valgrind." `https://valgrind.org/`. Accessed : 2024-07-23.

A Appendix-A

A.1 Performance Comparison of KEM Clean

Table 33: Performance Comparison of KEM Key Generation(clean)(unit:cc)

Key Generation								
Testing Environment1			Testing Environment2			Testing Environment3		
rank	scheme	avg	rank	scheme	avg	rank	scheme	avg
1	SMAUG-TiMER	69,234	1	SMAUG-TiMER	61,064	1	SMAUG-T1	39,278
2	SMAUG-T1	143,625	2	SMAUG-T1	128,685	2	SMAUG-TiMER	40,742
3	SMAUG-T3	193,935	3	SMAUG-T3	176,020	3	SMAUG-T3	66,770
4	NTRU+KEM576	253,031	4	NTRU+PKE576	207,953	4	SMAUG-T5	119,675
5	NTRU+PKE768	310,102	5	NTRU+KEM576	209,949	5	NTRU+KEM576	152,653
6	NTRU+PKE576	329,712	6	NTRU+PKE864	223,966	6	NTRU+PKE576	162,109
7	NTRU+PKE864	348,486	7	NTRU+KEM864	228,042	7	NTRU+KEM768	170,987
8	NTRU+KEM864	354,592	8	NTRU+KEM768	232,394	8	NTRU+PKE768	172,444
9	NTRU+KEM768	354,704	9	SMAUG-T5	233,829	9	NTRU+KEM864	175,592
10	NTRU+PKE1152	637,652	10	NTRU+PKE768	234,498	10	NTRU+PKE864	180,081
11	NTRU+KEM1152	659,905	11	NTRU+PKE1152	558,631	11	NTRU+PKE1152	420,607
12	SMAUG-T5	1,566,958	12	NTRU+KEM1152	558,996	12	NTRU+KEM1152	425,625
13	PALOMA128	131,189,151	13	PALOMA128	147,357,235	13	PALOMA128	119,572,480
14	PALOMA256	769,657,392	14	PALOMA192	700,965,299	14	PALOMA192	561,481,159
15	PALOMA192	650,967,499	15	PALOMA256	819,112,319	15	PALOMA256	662,511,849

Table 34: Performance Comparison of KEM Encapsulation(clean)(unit:cc)

Encapsulation								
Testing Environment1(clean)			Testing Environment2(clean)			Testing Environment3(clean)		
rank	scheme	avg	rank	scheme	avg	rank	scheme	avg
1	SMAUG-T1	66,403	1	SMAUG-TiMER	58,115	1	SMAUG-T1	39,889
2	SMAUG-TiMER	76,165	2	SMAUG-T1	58,942	2	SMAUG-TiMER	39,914
3	NTRU+PKE576	83,245	3	NTRU+PKE576	78,594	3	NTRU+PKE576	56,945
4	NTRU+KEM576	85,541	4	NTRU+KEM576	81,678	4	NTRU+KEM576	57,392
5	NTRU+KEM768	106,690	5	PALOMA128	87,441	5	PALOMA128	58,133
6	NTRU+PKE768	107,632	6	SMAUG-T3	98,497	6	SMAUG-T3	64,752
7	NTRU+PKE864	128,465	7	NTRU+KEM768	101,297	7	NTRU+PKE768	73,866
8	NTRU+KEM864	128,823	8	NTRU+PKE768	102,024	8	NTRU+KEM768	74,605
9	PALOMA128	133,345	9	NTRU+KEM864	109,860	9	PALOMA192	83,003
10	NTRU+PKE1152	164,153	10	NTRU+PKE864	110,043	10	NTRU+PKE864	83,385
11	NTRU+KEM1152	164,274	11	PALOMA192	136,963	11	NTRU+KEM864	83,862
12	PALOMA192	176,834	12	NTRU+PKE1152	151,692	12	PALOMA256	97,140
13	PALOMA256	215,487	13	NTRU+KEM1152	152,931	13	NTRU+PKE1152	108,907
14	SMAUG-T3	243,970	14	SMAUG-T5	165,370	14	NTRU+KEM1152	108,935
15	SMAUG-T5	1,663,364	15	PALOMA256	200,987	15	SMAUG-T5	114,363

Table 35: Performance Comparison of KEM Decapsulation(clean)(unit:cc)

Decapsulation								
Testing Environment1(clean)			Testing Environment2(clean)			Testing Environment3(clean)		
rank	scheme	avg	rank	scheme	avg	rank	scheme	avg
1	SMAUG-T1	104,474	1	SMAUG-TiMER	73,249	1	SMAUG-TiMER	48,089
2	NTRU+PKE576	107,302	2	SMAUG-T1	74,463	2	SMAUG-T1	49,270
3	NTRU+KEM576	110,140	3	NTRU+PKE576	98,820	3	NTRU+KEM576	64,897
4	SMAUG-TiMER	112,282	4	NTRU+KEM576	108,342	4	NTRU+PKE576	65,513
5	NTRU+PKE768	140,803	5	NTRU+KEM768	125,636	5	SMAUG-T3	78,442
6	NTRU+KEM768	141,220	6	SMAUG-T3	128,148	6	NTRU+PKE768	85,777
7	NTRU+PKE864	169,804	7	NTRU+PKE768	129,611	7	NTRU+KEM768	85,814
8	NTRU+KEM864	170,729	8	NTRU+PKE864	142,710	8	NTRU+KEM864	100,583
9	NTRU+KEM1152	219,767	9	NTRU+KEM864	145,575	9	NTRU+PKE864	100,620
10	NTRU+PKE1152	230,312	10	SMAUG-T5	184,943	10	SMAUG-T5	130,111
11	SMAUG-T3	421,956	11	NTRU+KEM1152	212,747	11	NTRU+PKE1152	137,555
12	SMAUG-T5	1,877,628	12	NTRU+PKE1152	212,804	12	NTRU+KEM1152	137,606
13	PALOMA128	8,424,379	13	PALOMA128	8,033,779	13	PALOMA128	8,091,350
14	PALOMA192	41,799,839	14	PALOMA192	41,802,948	14	PALOMA192	39,835,958
15	PALOMA256	43,735,841	15	PALOMA256	45,735,541	15	PALOMA256	41,789,539

A.2 Performance Comparison of DSA Clean

Table 36: Performance Comparison of DSA Key Generation(clean)(unit:cc)

Key Generation								
Testing Environment1(clean)			Testing Environment2(clean)			Testing Environment3(clean)		
rank	scheme	avg	rank	scheme	avg	rank	scheme	avg
1	AIMer128s	108,704	1	AIMer128f	90,968	1	AIMer128s	60,690
2	AIMer128f	159,134	2	AIMer128s	146,339	2	AIMer128f	62,198
3	AIMer192s	187,591	3	AIMer192f	175,871	3	AIMer192s	126,874
4	AIMer192f	201,497	4	NCC-Sign1	205,450	4	AIMer192f	132,077
5	NCC-Sign1	300,401	5	AIMer192s	223,472	5	NCC-Sign1	169,791
6	NCC-Sign3	352,126	6	NCC-Sign3	280,512	6	NCC-Sign3	218,513
7	AIMer256f	413,606	7	NCC-Sign5	416,020	7	AIMer256s	299,965
8	AIMer256s	454,476	8	AIMer256f	448,151	8	AIMer256f	302,295
9	NCC-Sign5	455,065	9	AIMer256s	488,242	9	NCC-Sign5	330,636
10	HAETAE2	1,160,435	10	HAETAE2	1,067,698	10	HAETAE2	951,707
11	HAETAE5	1,998,427	11	HAETAE3	2,155,146	11	HAETAE3	1,686,812
12	HAETAE3	2,103,397	12	HAETAE5	2,225,344	12	HAETAE5	1,993,473
13	MQ-Sign-MQLR-256-72-46	74,764,874	13	MQ-Sign-MQLR-256-72-46	70,289,113	13	MQ-Sign-MQLR-256-72-46	108,328,299
14	MQ-Sign-MQRR-256-72-46	100,975,612	14	MQ-Sign-MQRR-256-72-46	101,307,620	14	MQ-Sign-MQRR-256-72-46	137,903,106
15	MQ-Sign-MQLR-256-112-72	286,180,511	15	MQ-Sign-MQLR-256-112-72	270,156,039	15	MQ-Sign-MQLR-256-112-72	548,057,963
16	MQ-Sign-MQRR-256-112-72	387,732,599	16	MQ-Sign-MQRR-256-112-72	372,556,356	16	MQ-Sign-MQRR-256-112-72	650,807,901
17	MQ-Sign-MQLR-256-148-96	750,300,836	17	MQ-Sign-MQLR-256-148-96	682,362,344	17	MQ-Sign-MQLR-256-148-96	1,575,443,946
18	MQ-Sign-MQRR-256-148-96	997,169,889	18	MQ-Sign-MQRR-256-148-96	913,690,602	18	MQ-Sign-MQRR-256-148-96	1,822,096,906

Table 37: Performance Comparison of DSA Sign(clean)(unit:cc)

Sign								
Testing Environment1(clean)			Testing Environment2(clean)			Testing Environment3(clean)		
rank	scheme	avg	rank	scheme	avg	rank	scheme	avg
1	MQ-Sign-MQLR-256-72-46	463,964	1	MQ-Sign-MQLR-256-72-46	423,941	1	AIMer128s	60,690
2	NCC-Sign1	479,198	2	MQ-Sign-MQRR-256-72-46	753,662	2	AIMer128f	62,198
3	NCC-Sign3	600,143	3	NCC-Sign3	884,954	3	AIMer192s	126,874
4	MQ-Sign-MQRR-256-72-46	809,523	4	NCC-Sign5	1,049,172	4	AIMer192f	132,077
5	MQ-Sign-MQLR-256-112-72	1,289,973	5	MQ-Sign-MQLR-256-112-72	1,138,677	5	NCC-Sign1	169,791
6	NCC-Sign5	1,585,378	6	NCC-Sign1	1,161,522	6	NCC-Sign3	218,513
7	MQ-Sign-MQRR-256-112-72	2,057,398	7	MQ-Sign-MQRR-256-112-72	1,812,867	7	AIMer256s	299,965
8	HAETAE3	2,739,863	8	HAETAE2	1,866,070	8	AIMer256f	302,295
9	MQ-Sign-MQLR-256-148-96	2,762,744	9	MQ-Sign-MQLR-256-148-96	2,370,417	9	NCC-Sign5	330,636
10	HAETAE5	3,369,567	10	AIMer128f	3,288,188	10	HAETAE2	951,707
11	AIMer128f	3,742,915	11	HAETAE5	3,389,114	11	HAETAE3	1,686,812
12	MQ-Sign-MQRR-256-148-96	4,298,168	12	MQ-Sign-MQRR-256-148-96	3,660,222	12	HAETAE5	1,993,473
13	HAETAE2	5,032,357	13	HAETAE3	7,057,356	13	MQ-Sign-MQLR-256-72-46	108,328,299
14	AIMer192f	8,745,805	14	AIMer192f	7,292,073	14	MQ-Sign-MQRR-256-72-46	137,903,106
15	AIMer256f	16,706,302	15	AIMer256f	15,639,023	15	MQ-Sign-MQLR-256-112-72	548,057,963
16	AIMer128s	26,541,270	16	AIMer128s	23,324,902	16	MQ-Sign-MQRR-256-112-72	650,807,901
17	AIMer192s	65,094,493	17	AIMer192s	57,739,339	17	MQ-Sign-MQLR-256-148-96	1,575,443,946
18	AIMer256s	119,340,970	18	AIMer256s	116,361,684	18	MQ-Sign-MQRR-256-148-96	1,822,096,906

Table 38: Performance Comparison of DSA Verify(clean)(unit:cc)

Verify								
Testing Environment1(clean)			Testing Environment2(clean)			Testing Environment3(clean)		
rank	scheme	avg	rank	scheme	avg	rank	scheme	avg
1	HAETAE2	158,529	1	HAETAE2	170,103	1	HAETAE2	135,534
2	HAETAE3	275,419	2	HAETAE3	285,845	2	NCC-Sign1	220,085
3	NCC-Sign1	285,502	3	NCC-Sign1	288,198	3	HAETAE3	238,747
4	HAETAE5	342,870	4	HAETAE5	355,867	4	NCC-Sign3	279,032
5	NCC-Sign3	356,543	5	NCC-Sign3	391,048	5	HAETAE5	300,372
6	NCC-Sign5	591,947	6	NCC-Sign5	630,294	6	NCC-Sign5	447,622
7	MQ-Sign-MQRR-256-72-46	706,565	7	MQ-Sign-MQRR-256-72-46	643,609	7	MQ-Sign-MQLR-256-72-46	1,383,970
8	MQ-Sign-MQLR-256-72-46	733,587	8	MQ-Sign-MQLR-256-72-46	656,693	8	MQ-Sign-MQRR-256-72-46	1,449,188
9	MQ-Sign-MQRR-256-112-72	1,986,691	9	MQ-Sign-MQRR-256-112-72	1,709,939	9	MQ-Sign-MQLR-256-112-72	5,069,042
10	MQ-Sign-MQLR-256-112-72	2,016,272	10	MQ-Sign-MQLR-256-112-72	1,747,581	10	MQ-Sign-MQRR-256-112-72	5,294,775
11	AIMer128f	3,545,576	11	AIMer128f	2,961,062	11	AIMer128f	2,162,878
12	MQ-Sign-MQRR-256-148-96	4,272,864	12	MQ-Sign-MQRR-256-148-96	3,359,634	12	MQ-Sign-MQRR-256-148-96	6,023,046
13	MQ-Sign-MQLR-256-148-96	4,307,611	13	MQ-Sign-MQLR-256-148-96	3,454,074	13	MQ-Sign-MQLR-256-148-96	6,086,471
14	AIMer192f	7,800,523	14	AIMer192f	6,832,062	14	AIMer192f	5,503,737
15	AIMer256f	15,776,925	15	AIMer256f	14,820,473	15	AIMer256f	10,906,990
16	AIMer128s	27,013,406	16	AIMer128s	22,820,605	16	AIMer128s	13,628,278
17	AIMer192s	62,821,167	17	AIMer192s	56,700,184	17	AIMer192s	45,737,177
18	AIMer256s	119,625,648	18	AIMer256s	114,818,217	18	AIMer256s	88,643,000

A.3 Performance Comparison of KEM AVX2

Table 39: Performance Comparison of KEM Key Generation(AVX2)(unit:cc)

Key Generation					
Testing Environment2(avx2)			**Testing Environment4(avx2)**		
rank	**scheme**	**avg**	**rank**	**scheme**	**avg**
1	NTRU+PKE864	22,996	1	SMAUG-T1 (KEM90s)	72,566
2	NTRU+KEM864	26,132	2	NTRU+KEM576	73,798
3	NTRU+PKE576	28,451	3	NTRU+PKE576	79,520
4	NTRU+PKE768	29,094	4	NTRU+PKE864	87,100
5	NTRU+KEM768	34,551	5	NTRU+KEM864	94,406
6	SMAUG-T1 (KEM)	41,139	6	SMAUG-T1 (KEM)	131,751
7	NTRU+KEM1152	60,927	7	SMAUG-T3 (KEM)	132,416
8	SMAUG-T3 (KEM)	64,350	8	NTRU+KEM768	140,673
9	SMAUG-T1 (KEM90s)	70,489	9	NTRU+PKE768	141,808
10	NTRU+KEM576	74,524	10	SMAUG-T5 (KEM90s)	144,015
11	SMAUG-T3 (KEM90s)	91,033	11	NTRU+KEM1152	158,476
12	NTRU+PKE1152	97,123	12	NTRU+PKE1152	158,938
13	SMAUG-T5 (KEM90s)	108,447	13	SMAUG-T3 (KEM)	162,901
14	SMAUG-T5 (KEM)	146,222	14	SMAUG-T5 (KEM)	224,491

Table 40: Performance Comparison of KEM Encapsulation(AVX2)(unit:cc)

Encapsulation					
Testing Environment2(avx2)			**Testing Environment4(avx2)**		
rank	**scheme**	**avg**	**rank**	**scheme**	**avg**
1	NTRU+PKE576	21,394	1	NTRU+PKE768	26,999
2	NTRU+KEM768	29,121	2	NTRU+KEM768	37,193
3	NTRU+PKE768	29,486	3	NTRU+KEM1152	38,796
4	NTRU+PKE864	30,464	4	NTRU+PKE1152	44,797
5	NTRU+KEM864	32,559	5	SMAUG-T1 (KEM)	48,594
6	SMAUG-T1 (KEM)	33,704	6	NTRU+KEM864	51,431
7	SMAUG-T1 (KEM90s)	35,396	7	NTRU+KEM576	55,179
8	NTRU+KEM1152	40,212	8	NTRU+PKE576	61,034
9	NTRU+PKE1152	40,233	9	NTRU+PKE864	64,849
10	NTRU+KEM576	43,325	10	SMAUG-T3 (KEM90s)	69,773
11	SMAUG-T3 (KEM90s)	45,414	11	SMAUG-T1 (KEM90s)	74,438
12	SMAUG-T3 (KEM)	55,982	12	SMAUG-T3 (KEM)	74,948
13	SMAUG-T5 (KEM90s)	62,174	13	SMAUG-T5 (KEM90s)	76,148
14	SMAUG-T5 (KEM)	107,760	14	SMAUG-T5 (KEM)	135,314

Table 41: Performance Comparison of KEM Decapsulation(AVX2)

Decapsulation					
Testing Environment2(avx2)			Testing Environment4(avx2)		
rank	scheme	avg	rank	scheme	avg
1	NTRU+KEM576	12,933	1	NTRU+PKE576	14,639
2	NTRU+KEM768	16,449	2	NTRU+KEM768	17,330
3	NTRU+KEM864	17,272	3	NTRU+PKE768	17,558
4	NTRU+PKE768	18,613	4	NTRU+KEM864	19,302
5	NTRU+PKE864	19,384	5	NTRU+KEM576	19,814
6	NTRU+PKE1152	24,370	6	NTRU+PKE864	21,408
7	NTRU+KEM1152	26,813	7	NTRU+KEM1152	24,911
8	SMAUG-T1 (KEM90s)	39,227	8	NTRU+PKE1152	26,357
9	SMAUG-T3 (KEM90s)	57,594	9	SMAUG-T1 (KEM90s)	42,832
10	SMAUG-T3 (KEM)	63,344	10	SMAUG-T1 (KEM)	55,682
11	SMAUG-T1 (KEM)	64,401	11	SMAUG-T3 (KEM90s)	67,223
12	SMAUG-T5 (KEM90s)	85,226	12	SMAUG-T3 (KEM)	91,347
13	SMAUG-T5 (KEM)	109,709	13	SMAUG-T5 (KEM90s)	102,099
14	NTRU+PKE576	11,979	14	SMAUG-T5 (KEM)	156,083

A.4 Performance Comparison of DSA AVX2

Table 42: Performance Comparison of DSA Key Generation(AVX2)(unit:cc)

Key Generation					
Testing Environment2(avx2)			Testing Environment4(avx2)		
rank	scheme	avg	rank	scheme	avg
1	AIMer128f	40,172	1	AIMer128f	109,474
2	AIMer128s	93,037	2	AIMer128s	126,120
3	AIMer192s	97,972	3	AIMer192f	199,741
4	AIMer192f	99,173	4	AIMer192s	211,196
5	NCC-Sign1	137,139	5	NCC-Sign1	226,966
6	NCC-Sign3	187,859	6	NCC-Sign3	233,022
7	AIMer256f	236,956	7	AIMer256s	294,201
8	AIMer256s	242,895	8	AIMer256f	359,442
9	NCC-Sign5	265,387	9	NCC-Sign5	398,589
10	HAETAE2	834,651	10	HAETAE2	829,349
11	HAETAE3	1,423,068	11	HAETAE3	1,463,470
12	HAETAE5	1,924,879	12	HAETAE5	1,902,225
13	MQ-Sign-MQLR-256-72-46	4,947,896	13	MQ-Sign-MQLR-256-72-46	3,737,246
14	MQ-Sign-MQRR-256-72-46	7,538,634	14	MQ-Sign-MQRR-256-72-46	5,690,604
15	MQ-Sign-MQLR-256-112-72	23,971,764	15	MQ-Sign-MQLR-256-148-96	16,810,793
16	MQ-Sign-MQRR-256-112-72	32,740,982	16	MQ-Sign-MQLR-256-112-72	16,810,793
17	MQ-Sign-MQLR-256-148-96	57,884,657	17	MQ-Sign-MQRR-256-112-72	24,379,382
18	MQ-Sign-MQRR-256-148-96	81,811,186	18	MQ-Sign-MQRR-256-148-96	61,059,896

Table 43: Performance Comparison of DSA Sign(AVX2)(unit:cc)

Sign					
Testing Environment2(avx2)			**Testing Environment4(avx2)**		
rank	**scheme**	**avg**	**rank**	**scheme**	**avg**
1	MQ-Sign-MQLR-256-72-46	59,773	1	MQ-Sign-MQLR-256-72-46	46,147
2	MQ-Sign-MQRR-256-72-46	76,323	2	MQ-Sign-MQRR-256-72-46	63,070
3	MQ-Sign-MQLR-256-112-72	164,022	3	MQ-Sign-MQLR-256-112-72	126,525
4	MQ-Sign-MQRR-256-112-72	206,977	4	MQ-Sign-MQRR-256-112-72	179,025
5	NCC-Sign1	207,323	5	NCC-Sign1	211,273
6	MQ-Sign-MQLR-256-148-96	260,591	6	MQ-Sign-MQLR-256-148-96	248,743
7	HAETAE3	374,489	7	MQ-Sign-MQRR-256-148-96	300,749
8	NCC-Sign3	374,557	8	NCC-Sign3	327,080
9	MQ-Sign-MQRR-256-148-96	409,799	9	HAETAE3	391,344
10	NCC-Sign5	422,955	10	HAETAE5	439,080
11	AIMer128f	811,275	11	NCC-Sign5	509,618
12	HAETAE5	1,077,729	12	HAETAE2	689,815
13	AIMer192f	2,210,305	13	AIMer128f	988,392
14	AIMer256f	4,071,768	14	AIMer192f	2,693,700
15	HAETAE2	4,945,876	15	AIMer256f	5,216,095
16	AIMer128s	5,889,742	16	AIMer128s	7,384,692
17	AIMer192s	15,833,475	17	AIMer192s	20,005,582
18	AIMer256s	29,154,407	18	AIMer256s	42,617,762

Table 44: Performance Comparison of DSA Verify(AVX2)(unit:cc)

Verify					
Testing Environment2(avx2)			**Testing Environment4(avx2)**		
1	MQ-Sign-MQLR-256-72-46	59,773	1	MQ-Sign-MQLR-256-72-46	46,147
2	MQ-Sign-MQRR-256-72-46	76,323	2	MQ-Sign-MQRR-256-72-46	63,070
3	MQ-Sign-MQLR-256-112-72	164,022	3	MQ-Sign-MQLR-256-112-72	126,525
4	MQ-Sign-MQRR-256-112-72	206,977	4	MQ-Sign-MQRR-256-112-72	179,025
5	NCC-Sign1	207,323	5	NCC-Sign1	211,273
6	MQ-Sign-MQLR-256-148-96	260,591	6	MQ-Sign-MQLR-256-148-96	248,743
7	HAETAE3	374,489	7	MQ-Sign-MQRR-256-148-96	300,749
8	NCC-Sign3	374,557	8	NCC-Sign3	327,080
9	MQ-Sign-MQRR-256-148-96	409,799	9	HAETAE3	391,344
10	NCC-Sign5	422,955	10	HAETAE5	439,008
11	AIMer128f	811,275	11	NCC-Sign5	509,618
12	HAETAE5	1,077,729	12	HAETAE2	699,815
13	AIMer192f	2,210,305	13	AIMer128f	988,392
14	AIMer256f	4,071,768	14	AIMer192f	2,674,129
15	HAETAE2	4,946,575	15	AIMer256f	5,216,095
16	AIMer128s	5,889,742	16	AIMer128s	7,384,692
17	AIMer192s	15,833,475	17	AIMer192s	20,005,582
18	AIMer256s	29,154,407	18	AIMer256s	42,617,762