

Evaluating KpqC Algorithm Submissions: Balanced and Clean Benchmarking Approach

HyeokDong Kwon¹, MinJoo Sim¹, GyeongJu Song¹, Minwoo Lee², and Hwajeong Seo²

¹Department of Information Computer Engineering, Hansung University,

²Department of Convergence Security, Hansung University

23. 08. 25.

Introduction

KpqC: Korea Post-Quantum Cryptography Standardization

Background

Study details

Conclusion

Introduction

- With the development of quantum computers, classic cryptography algorithms are threatened. (e.g. RSA, ECC)
- In preparation for the quantum computer era, the Post-Quantum cryptography (PQC) became necessary.
 - NIST selected four PQC algorithms through PQC standardization competition.
- KpqC, a Korean Post-Quantum Cryptography competition was held.
 - Round 1 is currently in progress.
 - 16 candidates are in competition.

Introduction

- Our main contributions.

1. Benchmark all candidates in a unified environment.

- All KpqC algorithms provide its own benchmark results.
- However, these benchmarks were run in different environments.
- **We tested all candidates in the same environment.**
- This facilitates performance comparison between algorithms.

2. Eliminate dependencies in candidates.

- Dependencies must be secured for the operation of the KpqC candidate.
 - Mainly for external library dependencies such as OpenSSL.
- **We removed the external dependencies of the algorithm.**
- This reduces the difficulty of setting up the environment and eases the barrier to entry.

Background: KpqC standardization competition

- Contest for selection of standard for Korean Post-Quantum Cryptography.
 - First notice in November 2021.
 - **16 algorithms** advanced to Round 1.
 - Round 2 will be announced in September 2023.



Main schedule of KpqC		Scheme	Code	Lattice	Multivariate Quadratic	Isogeny	Zero-knowledge
2021.11.	KpqC Competition notice	Key Encapsulation Mechanism	IPCC Layered ROLLO PALOMA REDOG	NTRU+ SMAUG TiGER	-	-	-
2022.02.	Submission deadline						
2022.12.	Announcement of Round 1 candidates						
2023.09	Announcement of Round 2 candidates	Digital Signature	Enhanced pqsigRM	GCKSign HAETAE NCC-Sign Peregrine SOLMAE	MQ-Sign	FIBS	AIMer
2024.12	Announcement of KpqC Competition						

Background: Main objective

- KpqC candidate algorithms are provided in white paper.
- Performance measurement results are provided in each white paper.
- **However, every performance measurement environment is different.**
 - Some specification (OS, Memory, etc.) was not provided.

The benchmark is performed in Intel Xeon E5-1650 v3 @ 3.50GHz CPU with 128GB memory with GNU C 7.5.0 compiler on the Ubuntu 18.04.1 operating system. For the instantiation of the XOF, we use SHAKE in the KpqC library. We use SHAKE128 for AImmer-1, and SHAKE256 for AImmer-111 and AImmer-V. All the implementations used in the experiments are compiled at the -O3 optimization level. Our experiments are measured by the average clock cycles executed 10^3 times. For a fair comparison, we measure the execution time for signature scheme on the same CPU using the `taskset` command with Hyper-Threading and Turbo Boost features disabled.

6.2 Performance of AImmer.

In this section, we provide the performance of C standalone implementation of the AImmer signature scheme in Table 5. The performance of signature scheme is represented as milliseconds (ms) and CPU clock cycles (cc) and the size of public key, secret key, and signature is represented as bytes (B). For the implementation results of AVX2 version and comparison to other signature schemes, we refer to Appendix B.

4 Performance analysis

In this section, we report the performance results of C reference implementation.

4.1 Description of platform

All benchmarks are obtained on one core of an Intel(R) Core(TM) i7-10700K CPU processor with clock speed 3.80GHz. The benchmarking machine has 64 GB of RAM and runs Debian GNU/Linux with Linux kernel version 5.4.0. The implementation is compiled with gcc version 9.4.0, and the compiler flags as indicated in the CMakeLists included in the submission package.

4.2 Performance of reference implementation

We instantiate the hash functions G, H and the extendable function XOF with the following symmetric primitives:

Intel Xeon E3-1650(@ 3.50GHz)

128 GB memory

Ubuntu 18.04

Intel Core i7-8700k (@ 3.70GHz)

??? memory

Ubuntu 20.04

Intel Core i7-10700K(@ 3.80GHz)

64 GB memory

Debian Linux kernel 5.4.0

Intel Core i7-8700k (@ 3.70GHz)

16 GB memory

??? Operation system

4 Performance Analysis

In Table 4, we evaluate the performance of our implementations on a 3.7GHz Intel Core i7-8700k running Ubuntu 20.04 LTS. The table shows that the key generation, signing, and verification algorithms of our scheme are faster for every security level, compared to those of Dilithium. We choose the modulus q satisfying the condition $q \equiv 1 \pmod{2n}$, which enables the “fully-splitting” NTT algorithm for the multiplication operation. Even though our modulus q is much larger than the modulus of Dilithium ($q \approx 2^{23}$), our scheme is faster than Dilithium because of the simplicity of our scheme.

9.2 Implementation Results

All benchmarks were obtained on single core of an Intel Core i7-8700K (Coffee Lake) processor clocked at 3700 MHz. The benchmarking machine has 16 GB of RAM. Implementations were compiled with gcc version 9.4.0. The cycles listed below are the average of the cycle counts of 100,000 executions of the respective algorithms. Table 8 reports performance results of the reference and AVX2 implementation of NTRU+, along with the sizes of secret keys, public keys, and ciphertexts. The source code of NTRU+ is available for download on Github: <https://github.com/ntruplus/ntruplus>.

NTRU+576			
Size (Bytes)		Cycles (ref)	Cycles (AVX2)
sk:	1,728	gen: 321,405	gen: 17,440
pk:	864	encap: 110,754	encap: 14,307
ct:	864	decap: 163,277	decap: 12,445

Background: Main objective

- KpqC candidate algorithms are provided in white paper
- Performance measurement results are provided in each white paper
- **However, every performance measurement environment is different**
 - Some specification (OS, Memory, etc.) was not provided.

Comparison of performance is difficult

because each algorithm has a different measurement background

The benchmark is performed in Intel Xeon E5-1650 v3 @ 3.50GHz CPU with 128GB memory, with GNU C 7.5.0 compiler on the Ubuntu 16.04 LTS. The results are shown in Table 1. We use the `gcc -O3` compiler and the `libcrypto` library. We use the `openssl` library for the signature scheme. The results are shown in Table 1. The results used in the experiments are compared at the `-O3` optimization level. Our experiments are measured by the average clock cycles executed 10⁵ times. For a fair comparison, we measure the execution time for signature scheme on the same CPU using the `taskset` command with Hyper-Threading and Turbo Boost features disabled.

6.2 Performance of AIMER.

In this section, we provide the performance of C standalone implementation of the AIMer signature scheme in Table 5. The performance of signature scheme is represented as milliseconds (ms) and CPU clock cycles (cc) and the size of public key, secret key, and signature is represented as bytes (B). For the implementation results of AVX2 version and comparison to other signature schemes, we refer to Appendix B.

4 Performance analysis

4.1 Description of platform

All benchmarks are obtained on one core of an Intel(R) Core(TM) i7-10700K CPU processor with clock speed 3.80GHz. The benchmarking machine has 64 GB of RAM and runs Debian GNU/Linux with Linux kernel version 5.4.0. The implementation is compiled with gcc version 9.4.0, and the compiler flags as indicated in the CMakeLists included in the submission package.

4.2 Performance of reference implementation

We instantiate the hash functions G, H and the extendable function XOF with the following symmetric primitives:

thm has a diff

128 GB memory

Ubuntu 18.04

Intel Core i7-8700k (@ 3.70GHz)

??? memory

Ubuntu 20.04

Intel Core i7-10700K(@ 3.80GHz)

64 GB memory

Debian Linux kernel 5.4.0

Intel Core i7-8700k (@ 3.70GHz)

16 GB memory

??? Operation system

4 Performance Analysis

In Table 4, we evaluate the performance of our implementations on a 3.7GHz Intel Core i7-8700k running Ubuntu 20.04 LTS. The table shows that the key generation, signing, and verification algorithms of our scheme are faster for every security level, compared to those of Dilithium. We choose the modulus q satisfying the condition $q \equiv 1 \pmod{2n}$, which enables the “fully-splitting” NTT algorithm for the multiplication operation. Even though our modulus q is much larger than the modulus of Dilithium ($q \approx 2^{23}$), our scheme is faster than Dilithium because of the simplicity of our scheme.

9.2 Implementation Results

All benchmarks were obtained on single core of an Intel Core i7-8700K (Coffee Lake) processor clocked at 3700 MHz. The benchmarking machine has 16 GB of RAM. Implementations were compiled with gcc version 9.4.0. The cycles listed below are the average of the cycle counts of 100,000 executions of the respective algorithms. Table 8 reports performance results of the reference and NTRU+ implementation of NTRU+, along with the sizes of secret keys, public keys, and ciphertexts. The source code of NTRU+ is available for download on GitHub: <https://github.com/ntruplus/ntruplus>.

NTRU+576			
Size (Bytes)		Cycles (ref)	Cycles (AVX2)
sk:	1,728	gen: 321,405	gen: 17,440
pk:	864	encap: 110,754	encap: 14,307
ct:	864	decap: 163,277	decap: 12,445

Background: Additional objective

- Existence of external library dependencies.
 - Most algorithms use OpenSSL for random number generation.
- The use of libraries brings convenience when developing algorithms.
- However, there are also some disadvantages.
 - **Performance may vary depending on the library version.**
 - Acts as an error in measuring algorithm performance.
 - **It need to set the environment to run the algorithm.**
 - An entry barrier for users unfamiliar with the environment setup.

Testing environment

• Environment 1

- Ryzen 7 4800H(@ 2.90Ghz), RTX 3060, 16GB RAM
- **Our first experimental environment.**
- However, **most KpqC algorithms are measured on Intel processors.**

• Environment 2

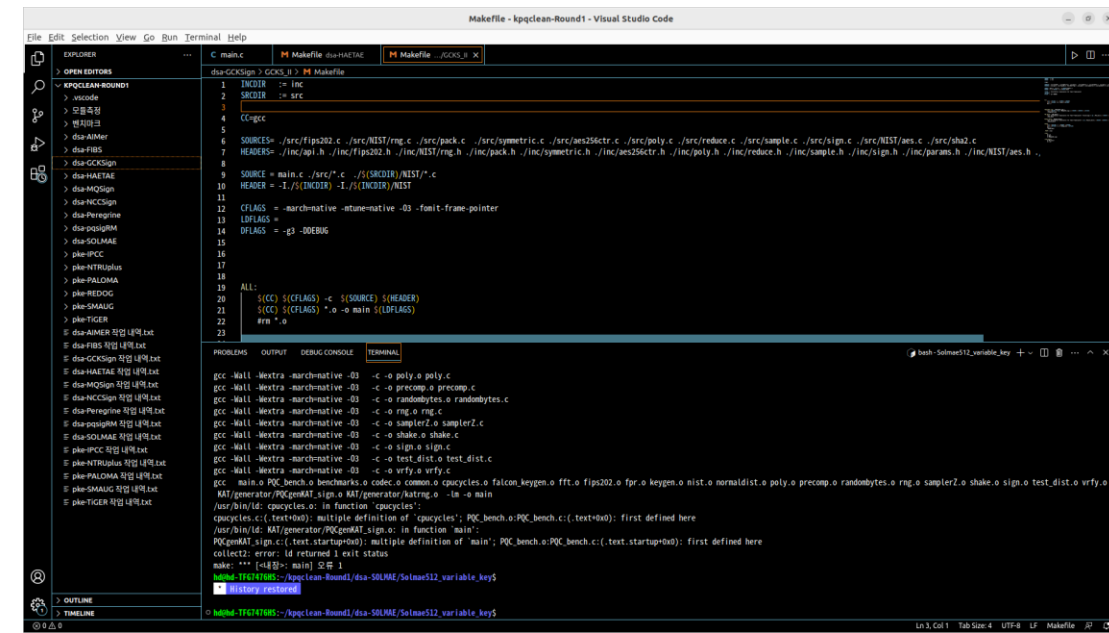
- Intel i5-8259U(@ 3.80GHz), Coffee Lake GT3e, 16GB RAM
- **Additional experimental environment.**

• Common settings

- OS: Ubuntu 22.04
- Compiler: gcc 11.33.0
- IDE: Visual Studio Code
- Optimization level: **-O2**, **-O3**

Our first option

Options for most KpqC algorithms



```
Makefile - kpclean-Round1 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C main.c Makefile - kpclean-Round1 Makefile - kpclean-Round1
kpclean-Round1: Makefile
1 INCDIR := inc
2 SRC := src
3
4 CC := gcc
5
6 SOURCES := ./src/fips202.c ./src/NIST_rng.c ./src/pack.c ./src/symmetric.c ./src/aes256ctr.c ./src/poly.c ./src/reduce.c ./src/sample.c ./src/sign.c ./src/NIST_aes.c ./src/sha2.c
7 HEADERS := ./inc/api.h ./inc/fips202.h ./inc/NIST_rng.h ./inc/pack.h ./inc/symmetric.h ./inc/aes256ctr.h ./inc/poly.h ./inc/reduce.h ./inc/sample.h ./inc/sign.h ./inc/params.h ./inc/NIST_aes.h
8
9 SOURCE = main.c ./src/* ./inc/*
10 HEADER = -I./$(INCDIR) -I./$(SRC)
11
12 CFLAGS = -march=native -mno-mmx -D3 -fomit-frame-pointer
13 LDFLAGS =
14
15 ALL:
16 $(CC) $(CFLAGS) -c $(SOURCES) $(HEADERS)
17 $(CC) $(CFLAGS) *.o -o main $(LDFLAGS)
18
19 gcc -Wall -Wextra -march=native -O3 -o poly.o poly.c
20 gcc -Wall -Wextra -march=native -O3 -o precomp.o precomp.c
21 gcc -Wall -Wextra -march=native -O3 -o randombytes.o randombytes.c
22 gcc -Wall -Wextra -march=native -O3 -o rng.o rng.c
23 gcc -Wall -Wextra -march=native -O3 -o sampler2.o sampler2.c
24 gcc -Wall -Wextra -march=native -O3 -o shake.o shake.c
25 gcc -Wall -Wextra -march=native -O3 -o sign.o sign.c
26 gcc -Wall -Wextra -march=native -O3 -o test_dist.o test_dist.c
27 gcc -Wall -Wextra -march=native -O3 -o verify.o verify.c
28 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
29 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
30 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
31 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
32 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
33 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
34 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
35 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
36 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
37 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
38 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
39 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
40 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
41 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
42 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
43 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
44 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
45 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
46 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
47 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
48 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
49 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
50 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
51 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
52 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
53 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
54 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
55 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
56 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
57 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
58 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
59 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
60 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
61 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
62 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
63 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
64 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
65 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
66 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
67 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
68 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
69 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
70 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
71 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
72 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
73 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
74 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
75 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
76 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
77 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
78 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
79 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
80 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
81 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
82 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
83 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
84 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
85 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
86 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
87 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
88 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
89 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
90 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
91 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
92 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
93 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
94 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
95 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
96 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
97 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
98 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
99 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
100 gcc -Wall -Wextra -march=native -O3 -o main.o main.c
```

Testing environment

- Differences by optimization level.
 - **Higher levels are more efficient (but less reliable).**
- We prioritized reliable behavior, so we initially used the -O2 option.
 - Later, -O3 was additionally measured, respecting the intentions of the developers.

-O2

: Support for all optimizations that do not utilize the space-speed tradeoff

```
-falign-functions -falign-jumps  
-falign-labels -falign-loops  
-fcaller-saves  
-fcode-hoisting  
-fcrossjumping  
-fcse-follow-jumps -fcse-skip-blocks  
-fdelete-null-pointer-checks  
-fdevirtualize -fdevirtualize-speculatively  
-fexpensive-optimizations  
-ffinite-loops  
-fgcse -fgcse-lm  
-fhoist-adjacent-loads  
-finline-functions  
-finline-small-functions  
-findirect-inlining  
-fipa-bit-cp -fipa-cp -fipa-icf  
-fipa-ra -fipa-sra -fipa-vrp  
-fisolte-erroneous-paths-dereference
```

```
-flra-remat  
-foptimize-sibling-calls  
-foptimize-strlen  
-fpartial-inlining  
-fpeephole2  
-freorder-blocks-algorithm=stc  
-freorder-blocks-and-partition -freorder-functions  
-frerun-cse-after-loop  
-fschedule-insns -fschedule-insns2  
-fsched-interblock -fsched-spec  
-fstore-merging  
-fstrict-aliasing  
-fthread-jumps  
-ftree-builtin-call-dce  
-ftree-loop-vectorize  
-ftree-pre  
-ftree-slp-vectorize  
-ftree-switch-conversion -ftree-tail-merge  
-ftree-vrp  
-fvect-cost-model=very-cheap
```

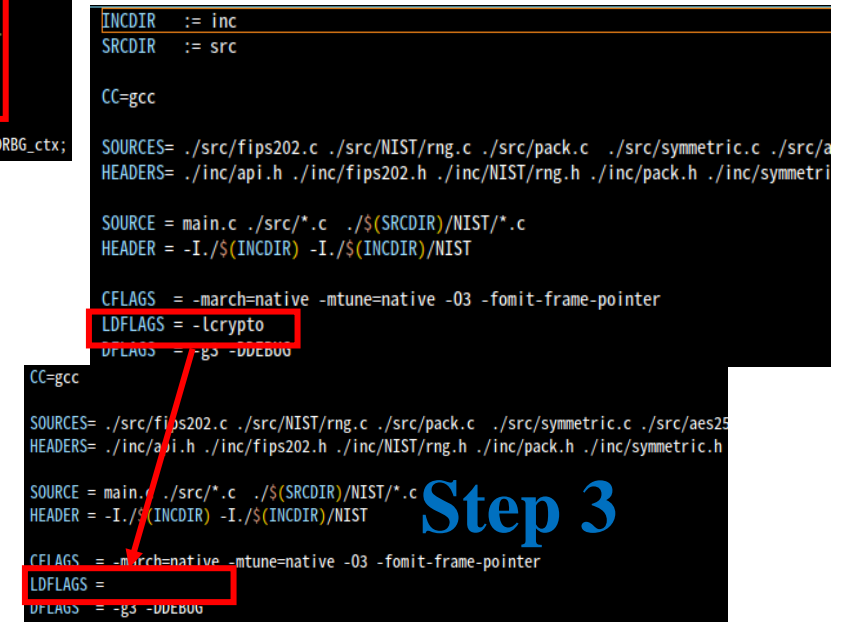
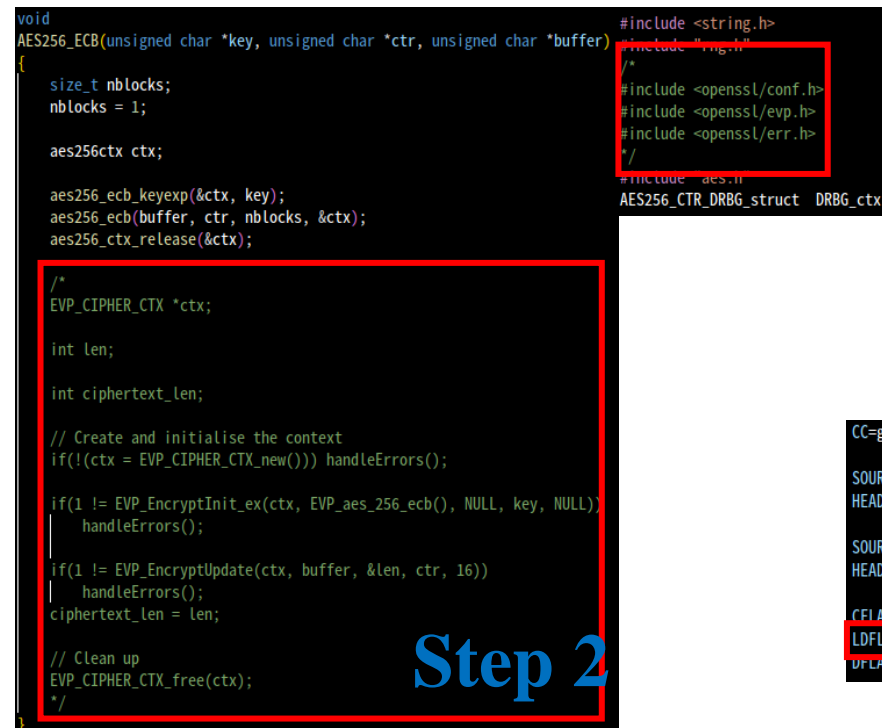
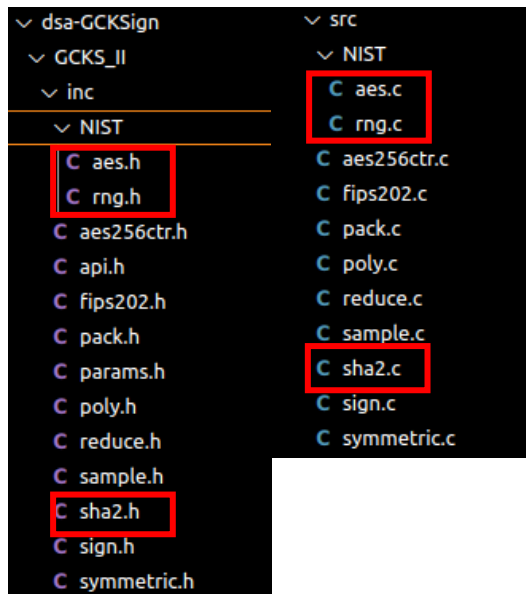
-O3

: Apply optimization as much as possible

```
-fgcse-after-reload  
-fipa-cp-clone  
-floop-interchange  
-floop-unroll-and-jam  
-fpeel-loops  
-fpredictive-commoning  
-fsplit-loops  
-fsplit-paths  
-ftree-loop-distribution  
-ftree-partial-pre  
-funswitch-loops  
-fvect-cost-model=dynamic  
-fversion-loops-for-strides
```

Work history

- We removed external library dependencies.
 1. Add the required algorithm source code (AES, SHA, etc.).
 2. Remove OpenSSL from the source code (include headers, function call, etc.).
 3. Modify Makefile rules.
- Each algorithm may have slightly different methods.



Work history

- Writing source code for evaluation.
 - It measured CPU clock cycles.
 - **Median/Average values are measured.**
- Initial performance measurements use average values only.
 - Most KpqC algorithms use the median.
 - So, we used both the median and average.

```
BENCHMARK ENVIRONMENTS =====
CRYPTO_PUBLICKEYBYTES: 1760
CRYPTO_SECRETKEYBYTES: 288
CRYPTO_BYTES: 1952
Number of loop: 10000
KeyGen //////////////////////////////////////////////////
KeyGen runs in ..... 191048 cycles
Sign //////////////////////////////////////////////////
Sign runs in ..... 812492 cycles
Verify //////////////////////////////////////////////////
Verify runs in ..... 175823 cycles
=====
hd@hd-TFG7476HS:~/kqclea-Round1/dsa-GCKSign/GCKS_II$
```

Result screen

Parameters size

Signature: key generation, signature generation, verification time

KEM: key generation, encryption, decryption time

```
#define TEST_LOOP 10000

uint64_t t[TEST_LOOP];

int64_t cpucycles(void)
{
    unsigned int hi, lo;

    __asm__ __volatile__ ("rdtsc\n\t" : "=a" (lo), "=d" (hi));

    return ((int64_t)lo) | (((int64_t)hi) << 32);
}

int PQc_bench(void)
{
    int r;
    double rejw=.0, rejyz=.0, rejctr=.0, rejctrkg=.0;
    unsigned long long i, j;

    unsigned char pk[CRYPTO_PUBLICKEYBYTES];
    unsigned char sk[CRYPTO_SECRETKEYBYTES];

    unsigned char m[100] = "kpqc benchmark system";
    unsigned char sm[CRYPTO_BYTES + 200];
```

Work history

- Write Makefile.
 - We modified Makefile by **adding only the necessary parts.**
 - Remove external dependencies.
 - Add new code targets.
 - Add new build rules.

```
PQCgenKAT_sign: ./PQCgenKAT_sign.c
    $(CC) $(CFLAGS) -o $@ ./PQCgenKAT_sign.c $(HEADER) $(SOURCES) $(LDFLAGS)
    ./PQCgenKAT_sign

PQC_bench: ./PQC_bench.c
    $(CC) -march=native -mtune=native -O2 -fomit-frame-pointer -fstack-usage -o $@ ./PQC_bench.c $(HEADER) $(SOURCES) $(LDFLAGS)
    ./PQC_bench

Module_bench: ./Module_bench.c
    $(CC) -march=native -mtune=native -O2 -fomit-frame-pointer -o $@ ./Module_bench.c $(HEADER) $(SOURCES) $(LDFLAGS)
    ./Module_bench
```

Strikethrough: need to re-measured		: Lattice based algorithm				Ryzen –O2 Unit: clock cycles
Green colored name: AVX applied		: Code based algorithm				
Algorithm	Keygen (Med.)	Encapsulation (Med.)	Decapsulation (Med.)	Keygen (Avr.)	Encapsulation (Avr.)	Decapsulation (Avr.)
NTRUplus-576	208,742	111,998	128,093	286,443	112,614	128,670
NTRUplus-768	279,386	148,480	181,250	298,193	154,346	185,298
NTRUplus-864	304,819	179,858	224,953	306,436	180,793	225,978
NTRUplus-1152	444,744	223,619	278,690	801,975	224,602	279,606
SMAUG-128 (revised)	171,477	154,483	178,205	181,007	156,512	181,950
SMAUG-192 (revised)	250,096	229,999	277,298	260,837	230,994	279,080
SMAUG-256 (revised)	479,138	385,178	438,364	490,702	387,420	439,345
TiGER-128	273,470	466,755	628,778	286,082	471,567	632,066
TiGER-192	288,550	518,491	674,192	293,000	524,467	691,838
TiGER-256	536,152	1,088,747	1,477,318	541,935	1,092,191	1,276,848
PALOMA-128	125,800,419	510,922	35,496	125,630,139	513,098	36,061
PALOMA-192	125,360,779	514,228	34,220	125,242,970	516,579	34,419
PALOMA-256	125,294,065	510,284	34,713	125,174,502	512,685	34,978
IPCC_f1	14,362,627	164,892,550	2,484,981	14,376,006	239,300,607	2,501,514
IPCC_f3	14,170,647	898,710	2,619,570	14,178,752	941,012	2,633,907
IPCC_f4	14,209,594	1,075,059	2,904,524	14,245,778	1,135,740	2,935,161

Strikethrough: need to re-measured		<div></div> : Lattice based algorithm		Ryzen –O3 Unit: clock cycles		
Green colored name: AVX applied		<div></div> : Code based algorithm				
Algorithm	Keygen (Med.)	Encapsulation (Med.)	Decapsulation (Med.)	Keygen (Avr.)	Encapsulation (Avr.)	Decapsulation (Avr.)
NTRUplus-576	202,652	110,026	121,742	287,810	110,910	123,929
NTRUplus-768	270,512	146,566	174,435	281,685	147,174	175,018
NTRUplus-864	297,192	168,113	204,537	302,876	168,857	206,046
NTRUplus-1152	435,305	222,459	266,626	772,442	223,429	268,110
SMAUG-128 (revised)	72,790	57,246	50,460	82,292	57,466	50,708
SMAUG-192 (revised)	105,966	82,940	80,475	108,491	83,648	81,029
SMAUG-256 (revised)	158,021	139,925	135,749	161,110	141,573	138,010
TiGER-128	65,482	48,749	51,214	68,866	49,105	51,589
TiGER-192	69,426	63,510	57,739	79,105	63,805	59,383
TiGER-256	81,316	87,551	93,090	90,989	88,218	93,436
PALOMA-128	122,325,408	498,365	34,307	122,253,994	500,446	34,484
PALOMA-192	122,290,738	503,266	34,278	122,173,457	506,366	34,468
PALOMA-256	122,321,957	497,959	34,249	122,254,172	500,026	34,420
IPCC_f1	13,940,097	160,111,204	16,360,164	13,969,607	232,561,407	2,408,173
IPCC_f3	13,996,024	926,492	2,512,836	14,036,005	967,543	2,532,438
IPCC_f4	13,989,832	1,106,031	2,714,531	14,007,544	1,165,274	2,732,139
Layered ROLLO I-128	285,940	83,346	788,104	296,880	84,198	805,790
Layered ROLLO I-192	320,958	136,503	1,014,203	345,689	149,843	1,110,378
Layered ROLLO I-256	687,721	201,913	1,945,871	700,284	207,030	1,948,662

Strikethrough: need to re-measured		: Lattice based algorithm		Intel –O2		
Green colored name: AVX applied		: Code based algorithm		Unit: clock cycles		
Algorithm	Keygen (Med.)	Encapsulation (Med.)	Decapsulation (Med.)	Keygen (Avr.)	Encapsulation (Avr.)	Decapsulation (Avr.)
NTRUplus-576	186,944	105,686	120,194	271,460	121,722	132,428
NTRUplus-768	246,616	139,310	166,938	265,516	154,868	174,788
NTRUplus-864	270,494	160,789	200,702	288,025	180,014	206,856
NTRUplus-1152	698,490	202,678	257,114	744,381	212,073	267,046
SMAUG-128 (revised)	158,149	164,598	196,470	165,414	169,648	203,288
SMAUG-192 (revised)	244,736	225,490	272,132	265,142	236,227	285,366
SMAUG-256 (revised)	435,790	411,917	465,572	448,654	422,602	486,263
TiGER-128	163,856	209,168	311,924	182,794	217,723	325,532
TiGER-192	171,578	214,126	312,702	181,774	221,613	324,412
TiGER-256	444,558	433,462	673,105	461,623	448,364	714,429
PALOMA-128	118,204,341	499,914	39,724	118,365,137	511,837	41,693
PALOMA-192	118,310,371	499,302	38,846	118,490,933	514,299	41,016
PALOMA-256	118,366,206	503,814	43,174	118,507,160	518,903	45,385
IPCC_f1	13,792,887	159,126,951	1,196,157	13,896,694	231,010,613	1,259,215
IPCC_f3	13,754,219	870,059	1,235,991	13,864,988	922,755	1,307,538
IPCC_f4	13,754,687	1,050,451	1,318,173	13,851,205	1,151,306	1,380,740

Strikethrough: need to re-measured		: Lattice based algorithm				Intel -O3 Unit: clock cycles
Green colored name: AVX applied		: Code based algorithm				
Algorithm	Keygen (Med.)	Encapsulation (Med.)	Decapsulation (Med.)	Keygen (Avr.)	Encapsulation (Avr.)	Decapsulation (Avr.)
NTRUplus-576	177,748	102,296	111,820	258,761	117,949	124,783
NTRUplus-768	239,546	137,135	161,970	257,560	165,057	177,077
NTRUplus-864	260,672	153,481	186,386	272,001	163,794	197,686
NTRUplus-1152	568,556	201,226	246,050	698,764	209,569	256,800
SMAUG-128 (revised)	63,020	49,324	39,196	65,919	55,873	42,528
SMAUG-192 (revised)	92,658	69,739	67,691	95,436	74,836	70,950
SMAUG-256 (revised)	135,202	122,766	115,096	142,842	128,734	118,789
TiGER-128	62,490	45,398	53,248	66,987	48,285	56,591
TiGER-192	66,512	60,238	58,572	71,626	71,973	71,967
TiGER-256	78,772	82,776	89,902	83,770	90,129	98,287
PALOMA-128	108,402,198	459,846	40,838	108,597,537	473,532	42,840
PALOMA-192	108,206,652	460,374	40,688	108,344,570	472,432	42,798
PALOMA-256	108,216,713	459,880	40,886	108,461,853	465,766	41,780
IPCC_f1	12,643,392	145,233,220	1,159,273	12,712,124	210,977,105	1,185,580
IPCC_f3	12,795,377	874,663	1,206,585	12,874,291	922,533	1,267,783
IPCC_f4	13,078,917	1,037,485	1,310,503	13,250,237	1,107,017	1,368,035
Layered ROLLO I-128	203,181	66,529	558,503	231,523	77,774	602,966
Layered ROLLO I-192	227,813	102,758	671,605	255,243	125,567	761,739
Layered ROLLO I-256	375,056	136,052	1,245,346	455,911	146,919	1,337,504

Evaluation

- Performance measurement of Key encapsulation mechanisms.
 - Metric: Median/Average time of **10,000 iterations** (-O2, -O3 both).
- Overall performance summary (Based on Encap, Decap, -O3).
 - **TiGER > SMAUG > NTRU+ > PALOMA > ROLLO > IPCC** (Normal)
 - **TiGER > SMAUG > PALOMA > NTRU+ > IPCC > ROLLO** (AVX considered)
- Analysis
 - NTRU+: Moderate performance even considering AVX, good keygen performance.
 - SMAUG: **The best performance when applying -O2.**
 - TiGER: Big difference in performance between -O2 and -O3.
 - PALOMA: The performance of keygen is slow, **but decapsulation is very fast.**
 - IPCC: IPCC-f1 encapsulation need to re-measured.
 - Layered ROLLO: The performance of decapsulation is slow, **but encapsulation is fast.**

Strikethrough: need to re-measured		<div></div> : Lattice based algorithm	<div></div> : Zero-knowledge based algorithm		<div></div> : Code based algorithm		<div></div> : Multivariate Quadratic based algorithm		Unit: clock cycles		Ryzen –O2
Green colored name: AVX applied											
Algorithm	Keygen (Med.)	Sign (Med.)	Verify (Med.)	Keygen (Avr.)	Sign (Avr.)	Verify (Avr.)					
GCKSign-II	179,771	601,707	176,987	181,822	848,504	178,229					
GCKSign-III	186,673	649,049	183,367	198,852	899,646	185,793					
GCKSign-V	252,822	917,415	277,733	255,206	1,099,271	284,217					
HAETAE-II (revised)	798,312	4,605,461	147,494	1,091,637	5,704,780	148,078					
HAETAE-III (revised)	1,533,941	11,474,155	257,926	2,127,683	12,068,749	259,846					
HAETAE-V (revised)	846,713	3,902,298	305,428	1,104,472	5,214,861	306,973					
NCCSign-II(con)	2,650,542	10,404,301	5,232,079	2,670,083	10,419,012	5,244,741					
NCCSign-III(con)	4,477,513	17,657,839	8,867,243	4,497,436	17,666,605	8,869,094					
NCCSign-V(con)	7,240,343	64,377,767	14,358,074	7,257,655	64,387,183	14,375,040					
NCCSign-II(ori)	1,869,079	23,762,252	3,681,057	1,882,892	23,763,293	3,684,640					
NCCSign-III(ori)	3,655,334	39,587,190	7,241,808	3,675,996	39,635,337	7,246,465					
NCCSign-V(ori)	6,263,739	179,281,596	12,418,902	6,268,503	179,337,534	12,422,702					
Peregrine-512	12,401,256	329,933	37,294	12,609,569	332,600	37,505					
Peregrine-1024	39,405,505	709,848	80,243	42,160,344	722,426	81,200					
SOLMAE-512	23,848,774	378,392	43,935	29,181,985	385,719	44,109					
SOLMAE-1024	55,350,546	760,380	141,375	70,141,847	764,304	142,357					
Enhanced pqsigRM-613	6,013,112,315	7,210,560	2,223,401	5,970,970,554	9,823,994	2,303,399					
Enhanced pqsigRM-612	58,238,108,879	1,864,512	1,053,034	58,669,322,672	2,650,133	1,064,763					
AIMER-I	145,058	3,912,361	3,669,834	156,204	3,966,517	3,701,498					
AIMER-III	296,496	8,001,274	7,550,063	315,810	8,033,590	7,548,322					
AIMER-V	710,442	18,068,276	17,415,022	730,960	18,077,211	17,421,527					
MQSign-72/46	94,788,559	516,954	1,461,281	94,829,257	518,651	1,465,923					
MQSign-112/72	488,913,828	1,493,703	5,211,909	490,448,324	1,513,132	5,258,218					
MQSign-148/96	1,488,480,956	3,162,943	12,036,827	1,488,377,972	3,164,654	12,041,118					

Strikethrough: need to re-measured		<div></div> : Lattice based algorithm	<div></div> : Zero-knowledge based algorithm		<div></div> : Code based algorithm		<div></div> : Multivariate Quadratic based algorithm		Unit: clock cycles		Ryzen –O3
Green colored name: AVX applied											
Algorithm	Keygen (Med.)	Sign (Med.)	Verify (Med.)	Keygen (Avr.)	Sign (Avr.)	Verify (Avr.)					
GCKSign-II	164,836	537,675	159,674	175,216	765,476	160,447					
GCKSign-III	166,199	581,189	161,646	180,908	806,260	162,452					
GCKSign-V	231,797	895,549	279,009	242,850	1,068,798	280,118					
HAETAE-II (revised)	688,083	3,429,265	131,805	957,268	4,247,185	132,462					
HAETAE-III (revised)	1,329,157	8,734,670	228,578	1,843,459	9,183,604	229,703					
HAETAE-V (revised)	723,318	2,790,612	272,542	946,202	3,700,449	273,840					
NCCSign-II(con)	2,619,295	10,301,902	5,171,686	2,639,164	10,308,375	5,175,958					
NCCSign-III(con)	4,379,261	86,475,941	8,685,877	4,405,049	86,515,726	8,685,125					
NCCSign-V(con)	7,178,921	42,637,366	14,245,148	7,194,274	42,681,718	14,247,358					
NCCSign-II(ori)	1,843,356	50,520,712	3,636,803	1,860,128	50,540,655	3,643,639					
NCCSign-III(ori)	3,618,997	21,416,384	7,170,903	3,646,841	21,437,009	7,169,406					
NCCSign-V(ori)	6,149,059	151,973,282	12,196,791	6,162,746	152,011,122	12,213,326					
Peregrine-512	11,953,307	253,402	25,462	12,146,320	254,228	25,634					
Peregrine-1024	38,366,232	535,920	53,621	41,014,591	538,260	53,946					
SOLMAE-512	23,053,028	349,566	40,513	28,233,370	355,950	40,812					
SOLMAE-1024	53,966,332	698,581	135,256	68,603,714	702,006	136,193					
Enhanced pqsigRM-613	6,139,551,981	4,610,319	2,278,806	6,144,274,759	6,276,554	2,376,095					
Enhanced pqsigRM-612	54,994,439,928	714,647	225,577	55,073,661,751	967,439	234,553					
AIMER-I	145,986	3,878,272	3,672,923	156,213	4,077,840	4,384,331					
AIMER-III	296,032	8,087,462	7,678,098	307,498	8,364,809	7,740,701					
AIMER-V	713,922	17,983,857	17,361,691	817,056	18,096,797	17,472,521					
MQSign-72/46	39,040,917	311,112	512,227	39,057,616	312,293	514,042					
MQSign-112/72	115,942,827	669,465	1,143,296	116,040,569	672,499	1,147,066					
MQSign-148/96	235,289,035	1,186,622	1,943,667	235,425,321	1,190,984	1,952,355					

Strikethrough: need to re-measured		<div></div> : Lattice based algorithm	<div></div> : Zero-knowledge based algorithm			Intel –O2
Green colored name: AVX applied		<div></div> : Code based algorithm	<div></div> : Multivariate Quadratic based algorithm			Unit: clock cycles
Algorithm	Keygen (Med.)	Sign (Med.)	Verify (Med.)	Keygen (Avr.)	Sign (Avr.)	Verify (Avr.)
GCKSign-II	171,176	640,093	167,116	190,739	845,502	173,676
GCKSign-III	173,252	698,964	168,824	185,265	943,376	176,768
GCKSign-V	248,629	945,815	273,631	264,811	1,151,316	282,979
HAETAE-II (revised)	700,875	4,173,002	142,584	979,130	5,274,158	150,759
HAETAE-III (revised)	1,352,577	10,615,663	250,534	1,940,364	11,445,286	262,470
HAETAE-V (revised)	752,413	3,418,728	311,986	983,382	4,622,966	328,866
NCCSign-II(con)	2,296,351	15,914,954	4,519,308	2,412,156	16,002,740	4,622,316
NCCSign-III(con)	4,009,717	16,015,734	7,996,462	4,169,703	16,116,734	8,085,000
NCCSign-V(con)	6,561,582	26,019,063	13,005,536	6,639,348	26,080,187	13,084,234
NCCSign-II(ori)	1,704,190	27,083,021	3,344,228	1,799,742	27,354,886	3,460,388
NCCSign-III(ori)	3,271,119	65,455,745	6,533,931	3,402,118	65,582,525	6,586,857
NCCSign-V(ori)	5,723,169	39,565,842	11,290,884	6,088,747	39,658,546	11,384,040
Peregrine-512	12,073,005	295,128	33,114	12,299,755	305,264	35,943
Peregrine-1024	38,493,479	640,132	71,246	41,112,188	652,620	74,891
SOLMAE-512	22,494,902	351,311	64,526	27,556,843	366,508	68,880
SOLMAE-1024	52,388,360	706,028	152,984	65,688,581	729,400	158,540
Enhanced pqsigRM-613	4,961,556,899	7,505,040	2,125,125	4,973,260,518	10,823,438	2,645,728
Enhanced pqsigRM-612	74,021,054,015	2,113,913	1,126,131	73,941,690,821	2,765,068	1,295,161
AIMER-I	145,566	3,691,256	3,713,173	159,391	3,845,843	4,018,047
AIMER-III	274,358	7,771,108	7,366,672	304,919	7,863,536	7,438,953
AIMER-V	790,456	18,394,069	17,662,359	899,383	18,802,192	18,080,181
MQSign-72/46	87,038,447	509,630	1,377,392	87,156,508	527,234	1,411,202
MQSign-112/72	448,271,119	1,472,032	4,808,216	448,141,266	1,500,297	4,875,532
MQSign-148/96	1,326,638,494	3,128,536	11,091,036	1,328,649,536	3,150,219	11,143,601

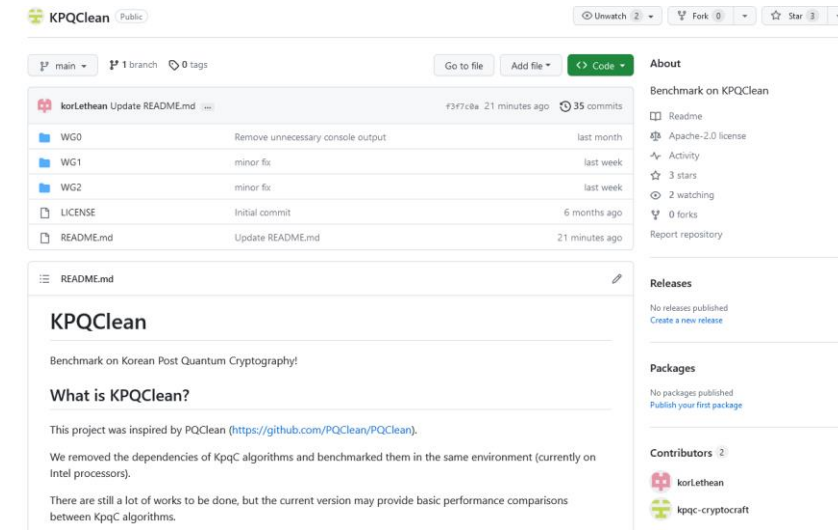
Strikethrough: need to re-measured		<div></div> : Lattice based algorithm	<div></div> : Zero-knowledge based algorithm			Intel –O3
Green colored name: AVX applied		<div></div> : Code based algorithm	<div></div> : Multivariate Quadratic based algorithm			Unit: clock cycles
Algorithm	Keygen (Med.)	Sign (Med.)	Verify (Med.)	Keygen (Avr.)	Sign (Avr.)	Verify (Avr.)
GCKSign-II	175,993	597,712	172,893	188,999	869,677	182,127
GCKSign-III	183,987	698,941	179,608	223,689	976,483	186,837
GCKSign-V	238,884	928,251	262,868	259,401	1,228,167	293,133
HAETAE-II (revised)	672,901	3,334,242	126,972	944,910	4,200,552	132,300
HAETAE-III (revised)	1,291,292	8,261,232	227,780	1,828,235	8,769,910	238,478
HAETAE-V (revised)	719,708	2,627,334	270,600	973,865	3,546,813	280,493
NCCSign-II(con)	2,317,555	13,776,448	4,568,006	2,393,641	13,868,809	4,647,302
NCCSign-III(con)	3,981,551	83,521,123	7,935,382	4,209,101	83,634,184	8,001,129
NCCSign-V(con)	6,333,006	25,183,392	12,555,623	6,470,472	25,269,299	12,680,799
NCCSign-II(ori)	1,666,543	16,352,341	3,248,162	1,846,947	16,530,887	3,321,373
NCCSign-III(ori)	3,141,974	34,454,252	6,234,249	3,227,617	34,523,301	6,288,505
NCCSign-V(ori)	5,613,303	167,158,023	11,155,020	5,851,360	167,337,719	11,307,818
Peregrine-512	11,783,005	260,328	26,262	12,032,320	269,678	28,484
Peregrine-1024	37,875,534	551,168	55,654	40,364,494	569,794	58,474
SOLMAE-512	22,627,042	332,848	64,838	27,866,035	348,841	67,662
SOLMAE-1024	53,245,753	668,103	149,168	67,369,725	686,523	154,073
Enhanced pqsigRM-613	4,702,612,115	4,732,706	2,064,731	4,703,836,987	6,667,564	2,458,625
Enhanced pqsigRM-612	71,111,088,778	923,513	417,658	71,168,430,985	1,166,665	502,448
AIMER-I	133,130	3,960,345	3,747,101	143,746	4,070,924	3,834,717
AIMER-III	272,484	8,440,184	7,968,982	282,896	8,530,553	8,041,509
AIMER-V	643,253	17,998,305	17,373,174	662,744	18,202,241	17,455,874
MQSign-72/46	38,474,591	298,952	533,676	38,612,360	308,203	547,680
MQSign-112/72	117,049,542	650,928	1,120,124	117,234,338	667,681	1,147,333
MQSign-148/96	236,124,011	1,165,706	1,897,664	236,332,422	1,173,558	1,908,458

Evaluation

- Performance measurement of Digital signatures.
 - Metric: Median/Average time of **10,000 iterations** (-O2, -O3 both).
 - Enhanced pqsigRM iterated 100 times.
- Overall performance summary (Based on Sign, Verify, -O3).
 - Peregrine > **SOLMAE** > pqsigRM > GCKSign > **MQSign** > HAETAE > AIMer > NCCSign (Normal)
 - Peregrine > pqsigRM > GCKSign > **SOLMAE** > HAETAE > **MQSign** > AIMer > NCCSign (AVX considered)
- Analysis
 - GCKSign: Fast verify speed.
 - HAETAE: HAETAE-III should be re-measured, overall performance is moderate.
 - NCCSign: Some performance re-measurement required.
 - Peregrine: The **fastest verify speed**, and **overall performance is good**.
 - SOLMAE: Even considering AVX, it has good performance.
 - Enhanced pqsigRM: Keygen performance is bad compared to its sign and verify performance.
 - AIMer: **Fast keygen speed, and has the smallest parameters size**.
 - MQSign: Has the smallest signature size, but difference in performance between -O2 and -O3.

Open source

- We published all data through online.
 - <https://github.com/kpqc-cryptocraft/KPQClean>
 - These are updated periodically.
- The following information is being released.
 - Source code for performance measurement.
 - Performance measurement results.
 - Source Code Compilation Guide.
- Also posted on the KpqC bulletin board.
 - Anyone who interested in KpqC can be participated this board.
- This Project will continue for the duration of the KpqC Competition.



(23.08) KpqC Benchmark Results 조회수 30회



HD Kwon

발는사람 KpqC-bulletin

Dear all,

our team updated the previous KpqC benchmark results.

Please refer to the following github link.
: <https://github.com/kpqc-cryptocraft/KPQClean>

The updated contents include the following items.

- Benchmark results for Intel processors have been added.
- Compiled with optimization level -O3 also tested.
- The compile method for some algorithms has been rewritten.

The following are ongoing issues.

- Benchmark of FIBS(All processors) and Layered-ROLLO(Ryzen with -O2).
- Re-measurement of an oddly measured algorithm.
- Performance measurements using revised versions of some algorithms.

Thank you.

Best regards,
Hyeokdong Kwon.

Conclusion

- We proceed to **measure the performance of the KpqC algorithms.**
 - **Performance measurements all in the same environment.**
 - Some development environments are also accommodated (Intel processor, -O3 option).
 - Remove external library dependencies → **Easing Research Entry Barriers.**
- Limitations
 - Security is the most important part of cryptographic algorithms.
 - **Evaluation only based on performance indicators should be avoided.**
 - **Difference in performance depending on the algorithm base**, so this should be considered.
- Future works
 - **Performance measurements on unmeasured algorithms (Layered ROLLO with -O2, FIBS).**
 - **Performance measurements for the revised algorithm.**
 - Remeasurement for some algorithms.

Q & A