ARMv8 상에서 HAETAE v1.0 구현

https://youtu.be/oMxPzOmG9BA

KpqC 공모전

- 2021년 말부터 국내 양자내성암호 표준 선정을 위한 KpqC 공모전이 진행 중
- 2022년 12월 1라운드 후보 알고리즘 선정
  - 7개의 공개키와 9개의 전자서명

KpqC 일정	
2021.11.25	공모전 공지
2022.02.18	공모전 제출 마감
2022.03.15	1차 평가
2023.12	1 라운드 결과 발표 2 라운드 후보 목록 공개
2024.03	2라운드 결과 발표
2024.09	KpqC 표준 알고리즘 발표

Type	Code-based	Lattice-based	Multivariate Quadratic-based	Hash-based	Zero-knowledge
Encryption/Key-establishment	IPCC	NTRU+	-	-	-
	Layered ROLLO	SMAUG			
	PALOMA	TiGER			
	REDOG				
Digital Signature	Enhanced pqsigRM	GCKSign	MQ-Sign	FIBS	AIMer
		HAETAE			
		NCC-Sign			
		Peregrine			
		SOLMAE			

KpqC 공모전

- 2021년 말부터 국내 양자내성암호 표준 선정을 위한 KpqC 공모전이 진행 중
- 2023년 12월 2라운드 후보 알고리즘 선정

KpqC 일정	
2021.11.25	공모전 공지
2022.02.18	공모전 제출 마감
2022.03.15	1차 평가
2023.12	1 라운드 결과 발표 2 라운드 후보 목록 공개
2024.03	2라운드 결과 발표
2024.09	KpqC 표준 알고리즘 발표

A. **전자서명**
- AIMer
- HAETAE
- MQ-Sign
- NCC-Sign

B. **공개키암호/키설정**
- NTRU+
- PALOMA
- REDOG
- SMAUG + TiGER (merged)

HAETAE

https://kpqc.cryptolab.co.kr/haetae

- 격자기반 전자서명 알고리즘
- Learning With Error(LWE)와 Short Integer Solution(SIS)의 어려움에 기반
- NIST PQC 표준으로 선정된 CRYSTALS-Dilithium에서 영감을 받아 설계
  - Rejection sampling에 대해 Bimodal distribution을 사용
  - Hyperball Uniform Distributions을 사용
- CRYSTALS-Dilithium보다 구현이 복잡하지만, 서명 크기가 작음
  - Dilithium과 동일한 보안 수준에서 ~~30~40%~~ 39% 작은 서명 크기와 ~~20%~~ 25% 작은 검증 크기를 가짐

Parameters sets Target security	HAETAE120 120	HAETAE180 180	HAETAE260 260
n	256	256	256
q	64513	64513	64513
(k,ℓ)	(2,4)	(3,6)	(4,7)
η	1	1	1
τ	39	49	60
S	293.51	385	457.01
B	9779.3329	15709.1546	20614.9815
B'	9774.9271	15704.4361	20609.9152
B''	11197.4229	17973.1740	23740.4482
α	256	324	384
a	8	8	9
N	6161	7045	7254

HAETAE1.0

Table 1: HAETAE parameters sets. Hardness is measured with the Core-SVP methodology and a refined analysis is given for LWE. The numbers in parenthesis for SIS are for the strong unforgeability property.

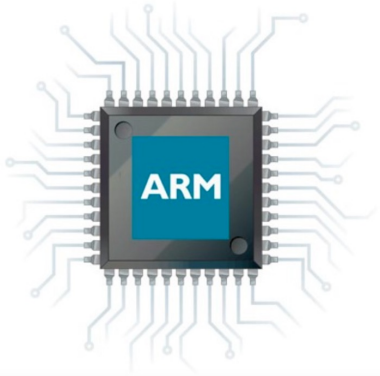
Security		120	180	260
n	Degree of $\mathcal{R}$ (2.1)	256	256	256
(k,ℓ)	Dimensions of $\mathbf{z}_2$, $\mathbf{z}_1$ (4.2)	(2,4)	(3,6)	(4,7)
q	Modulus for MLWE & MSIS (2.3)	64513	64513	64513
η	Range of sk coefficients (2.1)	1	1	1
τ	Weight of c (3.3)	58	80	128
γ	sk rejection parameter (3.1)	48.858	57.707	55.13
	Resulting key acceptance rate (3.1)	0.1	0.1	0.1
d	Truncated bits of vk (3.1)	1	1	0
M	Expected # of repetitions (3.4)	6.0	5.0	6.0
B	$\mathbf{y}$ radius (3.4)	9846.02	18314.98	22343.66
B'	Rejection radius (3.4)	9838.99	18307.70	22334.95
B''	Verify radius (4.2)	12777.52	21906.65	24441.49
α	$\mathbf{z}_1$ compression factor (3.5)	256	256	256
α_h	$\mathbf{h}$ compression factor (3.5)	512	512	256

HAETAE2.0

ARMv8프로세서

- ARM(**A**dvanced **R**ISC **M**achine)
  - ISA(Instruction Set Architecture) 고성능 임베디드 프로세서
  - 1958년 Acorn사에서 개발한 ARM1에서 시작
    추후 ARM Holdings를 설립하여 개발 시작

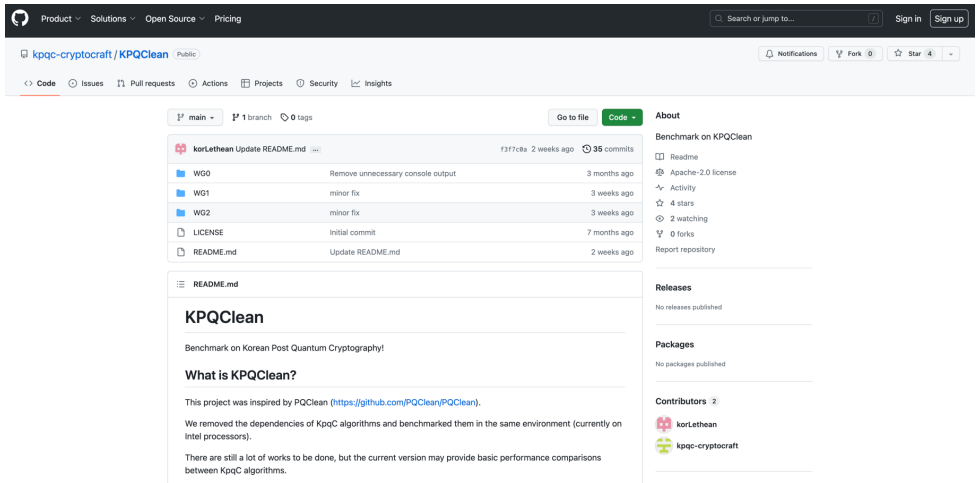
- ARMv8 프로세서
  31개의 64비트 general 레지스터(x0~x30)와
  128-bit 32개 벡터 레지스터(v0~v31) 지원



ASM	Description	Operation
MOV	Move	Xd←Xn
MVN	Bitwise NOT (shifted register)	Xd←Xn<<#shift
ASR	Arithmetic shift right (immediate)	Xd←Xn>>#shift
ORR	Bitwise OR (shifted register)	Xd←Xn|(Xm <<#amount or Xm >>#amount)
AND	Bitwise AND (shifted register)	Xd←Xn&(Xm <<#amount or Xm >>#amount)
ADD	Add	Xd←Xn+Xm
SUB	Subtract	Xd←Xn-Xm
RET	Return from subroutine	Return

제안 기법

- ARMv8 상에서 HAETAE를 구현하기 위해 레퍼런스의 **OpenSSL 의존성 제거한 KPQClean 사용**
- HAETAE의 **곱셈 연산의 일부 하위 모듈** 구현
  - General 레지스터를 사용하여 곱셈 연산을 ARM 명령어를 활용하여 고속 구현
- ARM에서 제공하는 **AES 암호화 가속 명령어** 사용한 고속 구현

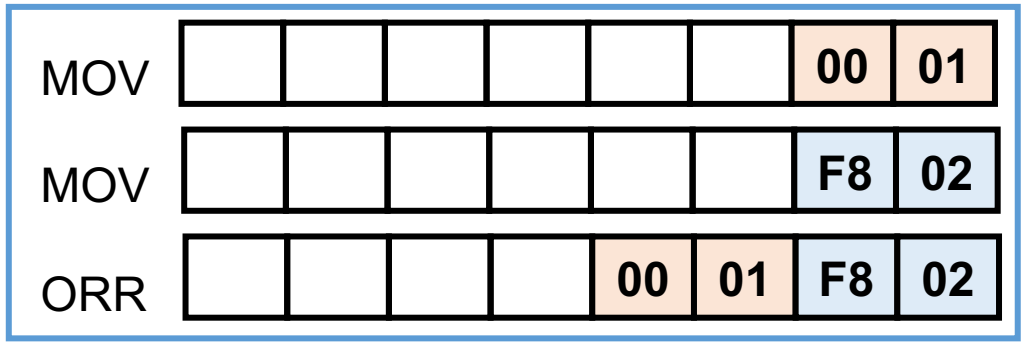


제안 기법

```
int32_t reduce32_2q(int32_t a) {
    int64_t t = (int64_t)a * DQREC;
    t >>= 32;
    t = a - t * DQ;              // -4Q <  t < 4Q
    t += (t >> 31) & (DQ * 2);   //   0 <= t < 4Q
    t -= ~((t - DQ) >> 31) & DQ; //   0 <= t < Q
    t -= ~((t - Q) >> 31) & DQ;  // centered representatior
    return (int32_t)t;
}
```

- **32-bit Reduce 연산 구현**
- 모듈러 2Q값을 하나에 위치하는 연산 수행
  - 하나의 레지스터에 MOV 명령어를 사용하여

  레지스터에 값을 지정할 수 있는 범위는 고정적



- Reduce 모듈을 MUL, ASR, SUB, ADD 등의 명령어로 효율적으로 구현
- NOT 연산이 수행되기 때문에, MVN 을 사용하여 구현

```
.macro asm_reduce32_2q
    mk_DQREC
    mk_DQ
    mk_Q

    mul x5, x0, x4
    asr x5, x5, #32
    mul x6, x5, x7
    sub x5, x0, x6

    asr x9, x5, #31
    mul x10, x7, x3
    and x11, x9, x10
    add x5, x5, x12

    sub x8, x5, x7
    asr x8, x8, #31
    mvn x9, x8
    and x10, x9, x7
    sub x5, x5, x10

    sub x8, x5, x3
    asr x8, x8, #31
    mvn x9, x8
    and x10, x9, x3
    sub x5, x5, x10
.endm
```

[표 3] Source code of reduction with double Q implementation.

제안 기법

- Finite field 요소에 대한 standard representative r을 위한 **freeze 연산 구현**
- Reduce 연산과 유사하게 freeze 모듈을 MUL, ASR, SUB, ADD, MVN의 명령어로 효율적으로 구현

```
.macro freeze
    mk_Q_rec
    mk_DQ
    mk_Q

    mul x5, x0, x4
    asr x5, x5, #32
    mul x6, x5, x3
    sub x5, x0, x5

    and x8, x7, x5, asr #31
    add x5, x5, x8

    sub x8, x5, x3
    asr x8, x8, #31
    mvn x9, x8
    and x10, x9, x3
    sub x5, x5, x10
.endm
```

[표 4] Source code of computation standard representative implementation.

제안 기법

- **AES 암호화 구현**
- ARM에서는 AES 가속기 명령어 지원
- HAETAE의 난수 생성 프로세서에서 사용

→ ARMv8에서 지원하는 AES 암호화 명령어를 활용하여 AES 암호화 고속화

ASM	Description
AESE	AES single round encryption
AESMC	AES mix columns

[표 4] AES encryption instructions supported by ARMv8.

성능평가

- 성능 측정 환경
  - ARMv8 아키텍처를 사용하는 Apple M1chip이 탑재된 MacBook Pro 13(2020)에서 성능 측정
  - 반복횟수 : 10,000
  - 최적화 옵션 : -O3 Level(i.e. fastest)

- 알고리즘별 성능 비교
  - 키 생성, 서명 생성, 검증 알고리즘에 적용한 결과
  - **키 생성 알고리즘의 경우, 1.00배 성능 향상**
  - **서명 생성 알고리즘의 경우, 1.50배 성능 향상**
  - **서명 검증 알고리즘의 경우, 1.22배 성능 향상**

HAETAE -120		Keygen	Sign	Verify
Ref	ms	2,705	13,546	413
	cc	865,600	4,334,720	132,160
This work	**ms**	**2,702**	**9,030**	**339**
	cc	**864,640**	**2,889,600**	**108,480**

[표 5] Performance evaluation result of HAETAE algorithm.

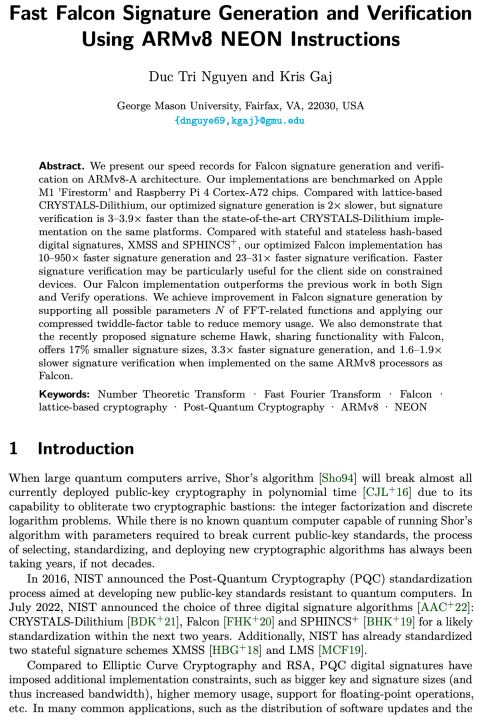
향후 진행 계획

- (현재)2023 정보보호학회 충청지부 발표 결과
  - 1라운드 제출된 HAETAE에 대한 구현
  - NEON 명령어가 아닌 ARM 명령어 사용
  - Ramdombyes.h도 이전 pqclean 코드 사용하기 때문에 aes 가속기 사용

- 진행 계획
  - Arm_neon.h에서 제공하는 Apple_neon 명령어 분석 중
  - 2라운드 제출된 update된 HAETAE에 대한 구현
  - NEON 명령어로 병렬 구현
  - 최근 pqclean에 업로드된 Keccak2x 포팅
  - pqclean의 Ramdombyes의 경우 aes 가속기 X

```c
#if defined(BSD)
static int randombytes_bsd_randombytes(void *buf, size_t n) {
    arc4random_buf(buf, n);
    return 0;
}
```

https://github.com/PQClean/PQClean/blob/master/common/randombytes.c



Fast Falcon Signature Generation and Verification Using ARMv8 NEON Instructions

Duc Tri Nguyen and Kris Gaj

George Mason University, Fairfax, VA, 22030, USA
{dnguye69,kgaj}@gmu.edu

Abstract. We present our speed records for Falcon signature generation and verification on ARMv8-A architecture. Our implementations are benchmarked on Apple M1 'Firestorm' and Raspberry Pi 4 Cortex-A72 chips. Compared with lattice-based CRYSTALS-Dilithium, our optimized signature generation is 2× slower, but signature verification is 3–3.9× faster than the state-of-the-art CRYSTALS-Dilithium implementation on the same platforms. Compared with stateful and stateless hash-based digital signatures, XMSS and SPHINCS$^+$, our optimized Falcon implementation has 10–950× faster signature generation and 23–31× faster signature verification. Faster signature verification may be particularly useful for the client side on constrained devices. Our Falcon implementation outperforms the previous work in both Sign and Verify operations. We achieve improvement in Falcon signature generation by supporting all possible parameters N of FFT-related functions and applying our compressed twiddle-factor table to reduce memory usage. We also demonstrate that the recently proposed signature scheme Hawk, sharing functionality with Falcon, offers 17% smaller signature sizes, 3.3× faster signature generation, and 1.6–1.9× slower signature verification when implemented on the same ARMv8 processors as Falcon.

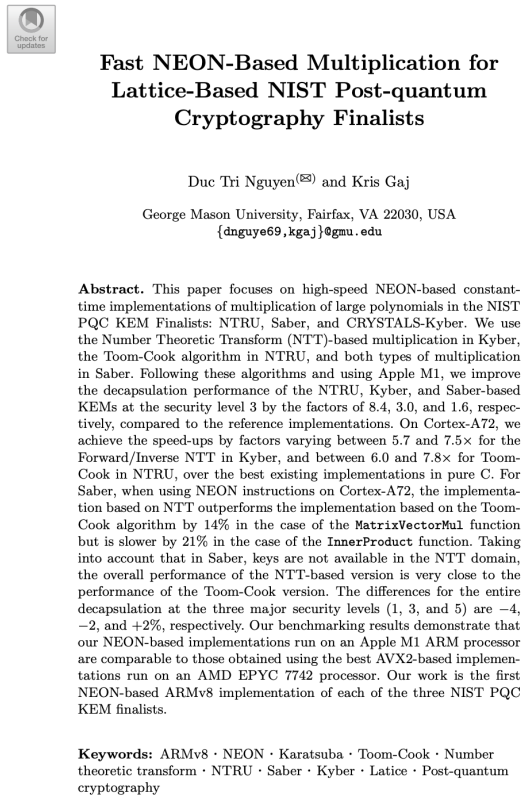
Keywords: Number Theoretic Transform · Fast Fourier Transform · Falcon · lattice-based cryptography · Post-Quantum Cryptography · ARMv8 · NEON

1 Introduction

When large quantum computers arrive, Shor's algorithm [Sho94] will break almost all currently deployed public-key cryptography in polynomial time [CJL+16] due to its capability to obliterate two cryptographic bastions: the integer factorization and discrete logarithm problems. While there is no known quantum computer capable of running Shor's algorithm with parameters required to break current public-key standards, the process of selecting, standardizing, and deploying new cryptographic algorithms has always been taking years, if not decades.

In 2016, NIST announced the Post-Quantum Cryptography (PQC) standardization process aimed at developing new public-key standards resistant to quantum computers. In July 2022, NIST announced the choice of three digital signature algorithms [AAC+22]: CRYSTALS-Dilithium [BDK+21], Falcon [FHK+20] and SPHINCS$^+$ [BHK+19] for a likely standardization within the next two years. Additionally, NIST has already standardized two stateful signature schemes XMSS [HBG+18] and LMS [MCF19].

Compared to Elliptic Curve Cryptography and RSA, PQC digital signatures have imposed additional implementation constraints, such as bigger key and signature sizes (and thus increased bandwidth), higher memory usage, support for floating-point operations, etc. In many common applications, such as the distribution of software updates and the



Fast NEON-Based Multiplication for Lattice-Based NIST Post-quantum Cryptography Finalists

Duc Tri Nguyen(✉) and Kris Gaj

George Mason University, Fairfax, VA 22030, USA
{dnguye69,kgaj}@gmu.edu

Abstract. This paper focuses on high-speed NEON-based constant-time implementations of multiplication of large polynomials in the NIST PQC KEM Finalists: NTRU, Saber, and CRYSTALS-Kyber. We use the Number Theoretic Transform (NTT)-based multiplication in Kyber, the Toom-Cook algorithm in NTRU, and both types of multiplication in Saber. Following these algorithms and using Apple M1, we improve the decapsulation performance of the NTRU, Kyber, and Saber-based KEMs at the security level 3 by the factors of 8.4, 3.0, and 1.6, respectively, compared to the reference implementations. On Cortex-A72, we achieve the speed-ups by factors varying between 5.7 and 7.5× for the Forward/Inverse NTT in Kyber, and between 6.0 and 7.8× for Toom-Cook in NTRU, over the best existing implementations in pure C. For Saber, when using NEON instructions on Cortex-A72, the implementation based on NTT outperforms the implementation based on the Toom-Cook algorithm by 14% in the case of the `MatrixVectorMul` function but is slower by 21% in the case of the `InnerProduct` function. Taking into account that in Saber, keys are not available in the NTT domain, the overall performance of the NTT-based version is very close to the performance of the Toom-Cook version. The differences for the entire decapsulation at the three major security levels (1, 3, and 5) are −4, −2, and +2%, respectively. Our benchmarking results demonstrate that our NEON-based implementations run on an Apple M1 ARM processor are comparable to those obtained using the best AVX2-based implementations run on an AMD EPYC 7742 processor. Our work is the first NEON-based ARMv8 implementation of each of the three NIST PQC KEM finalists.

Keywords: ARMv8 · NEON · Karatsuba · Toom-Cook · Number theoretic transform · NTRU · Saber · Kyber · Latice · Post-quantum cryptography

성능 측정 문제...

```
gettimeofday(&start, NULL);
for (int i = 0; i < TEST_LOOP; i++)
    crypto_sign_keypair(pk, sk);
gettimeofday(&end, NULL);

seconds  = end.tv_sec  - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
mtime = ((seconds) * 1000 + useconds/1000.0) + 0.5;
```

HAETAE v1.0	Keygen	Sign	Verify
Ref-c	2,898	12,400	411
Ref-c (update randombyte)	2,927	8,826	411
asm	3,015	9,991	349
asm (update randombyte)	2,968	6,083	345

```
BENCHMARK ENVIRONMENTS  ============================
CRYPTO_PUBLICKEYBYTES: 992
CRYPTO_SECRETKEYBYTES: 1408
CRYPTO_BYTES: 1474
Number of loop: 10000
KeyGen //////////////////////////////////////////////
  KeyGen runs in .................     2939 cycles
Sign ////////////////////////////////////////////////
  Sign runs in .................     9886 cycles
Verify //////////////////////////////////////////////
  Verify runs in .................      432 cycles
=====================================================
```

HAETAE 2.0(레퍼런스)의 경우
Sign의 성능이 측정할때마다 이상하게 튐
(update randombyte 사용)
pqm4 에서 공개한 HAETAE 콘

```
BENCHMARK ENVIRONMENTS  ============================
CRYPTO_PUBLICKEYBYTES: 992
CRYPTO_SECRETKEYBYTES: 1408
CRYPTO_BYTES: 1474
Number of loop: 10000
KeyGen //////////////////////////////////////////////
  KeyGen runs in .................     2904 cycles
Sign ////////////////////////////////////////////////
  Sign runs in .................     3095 cycles
Verify //////////////////////////////////////////////
  Verify runs in .................      431 cycles
=====================================================
Program ended with exit code: 0
```

성능 측정 문제...

```c
int64_t cpucycles(void)
{
    struct timespec time;

    clock_gettime(CLOCK_REALTIME, &time);
    return (int64_t)(time.tv_sec*1e9 + time.tv_nsec);
}
```

```c
int crypto_sign_keypair(uint8_t *pk, uint8_t *sk) {
    uint8_t seedbuf[2 * SEEDBYTES + CRHBYTES] = {0};
    uint16_t nonce = 0;
    const uint8_t *rhoprime, *sigma, *key;
    polyvecm A[K], s1, s1hat;
    polyveck b, s2;
#if D > 0
    polyveck a, b0;
#else
    polyveck s2hat;
#endif
    xof256_state state;

    // Get entropy \rho
    randombytes(seedbuf, SEEDBYTES);
```

HEATAE 1.0	Keygen	Sign	Verify
Ref-c (update randombyte)	287,168	850,656	46,427
asm (update randombyte)	297,000	845,293	39.410

HEATAE 2.0	Keygen	Sign	Verify
Ref-c (update randombyte)	293,063	862,740	45,561
asm (update randombyte)	294,271	864,177	38,128

```
BENCHMARK ENVIRONMENTS  ============================
CRYPTO_PUBLICKEYBYTES: 992
CRYPTO_SECRETKEYBYTES: 1408
CRYPTO_BYTES: 1463
Number of loop: 10000
  KeyGen runs in ................. 853720042862967 nsec
  Sign runs in ................... 1562422841852875 nsec
  Verify runs in ................. 1844674407295393 nsec
Program ended with exit code: 0
```

HAETAE 1.0(ref/asm)의 경우
성능 측정이이상하게 튐
(이전 pqclean
randombyte 사용+ AES 가속기 구현 사용)

성능 측정 문제...

PQC_NEON / neon / kyber / **m1cycles.c**

cothan init

Code | Blame | 177 lines (153 loc) · 4.79 KB | Raw

```
/*
 * Duc Tri Nguyen (CERG GMU)
 * Modified from M1:
 * https://gist.github.com/dougallj/5bafb113492047c865c0c8cfbc930155#file-m1_robsize-c-L390
 */

#include <dlfcn.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include "m1cycles.h"

#define KPERF_LIST                              \
    /*  ret, name, params */                    \
    F(int, kpc_get_counting, void)              \
    F(int, kpc_force_all_ctrs_set, int)         \
    F(int, kpc_set_counting, uint32_t)          \
    F(int, kpc_set_thread_counting, uint32_t)   \
    F(int, kpc_set_config, uint32_t, void *)    \
    F(int, kpc_get_config, uint32_t, void *)    \
    F(int, kpc_set_period, uint32_t, void *)    \
    F(int, kpc_get_period, uint32_t, void *)    \
    F(uint32_t, kpc_get_counter_count, uint32_t) \
    F(uint32_t, kpc_get_config_count, uint32_t)  \
    F(int, kperf_sample_get, int *)             \
    F(int, kpc_get_thread_counters, int, unsigned int, void *)

#define F(ret, name, ...)               \
    typedef ret name##proc(__VA_ARGS__); \
    static name##proc *name;
KPERF_LIST
#undef F
```

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

Q & A