KOOKMIN UNIVERSITY

양자내성암호연구단

KpqC 연구단 9차 워크숍

KpqC 공모전 Round 2 알고리즘에 대한 경량프로세서 환경에서의 성능분석 연구

24. 10. 23.

국민대학교 사이버보안학과

서석충, 김영범, 최용렬

CSE CRYPTO & SECURITY ENGINEERING Lab

목차

참여 연구원

KPQM4 Framework 구축
KPQC 알고리즘 통합 및 테스트
PQC 알고리즘 구현 가이드문서 개발

김영범 박사 과정

- 임베디드 PQC 최적화 연구
  (AVR, MSP, 32-bit ARM, 64-bit ARM, RISC-V)
- 암호 SW/HW codesign 최적화 연구
- 국가암호공모전 최우수상 (2021, 2023, 2024)

최용렬 석사 과정

- 임베디드 PQC 최적화 및 코드 정확정테스트 연구
  (32-bit ARM, RISC-V)
- 암호 SW/HW codesign 최적화 연구
- 국가암호공모전 최우수상 (2023)

KOOKMIN UNIVERSITY

양자내성암호연구단

Phase 1

CSE CRYPTO & SECURITY ENGINEERING Lab

❖ KpqC 알고리즘의 Performance는 주요 평가기준 중 하나

➢ **Performance : Measured on a variety of classical platforms**

✓ **x86/x64, AVX2 :** 고성능 범용 CPU 장치

- 통합라이브러리 : KPQClean (https://github.com/kpqc-cryptocraft/KpqClean_ver2)
- 2023년에 시작된 프로젝트로, 현재 Round2 코드(ref, AVX2)가 KPQClean 통합

✓ **Cortex-M4 :** 저 사양 Embedded(임베디드) 장치

- Round 1 코드에 대한 벤치마크 존재 (KpqC 7차 워크숍, 스마트엠투엠, 김호원 교수님)
- **업데이트된 Round 2 코드에 대한 벤치마크가 필요**
- **포팅 이슈를 해결하여 벤치마크 통합이 필요**
- **➔ 새로운 kpqm4 프로젝트 제안이 필요**

2021	2022	2022	2023 - 2024
Announce	Round 0	Round 1	Round2
KpqC 알고리즘 제안 요청 및 제안서 접수	KpqC Round 0	KpqC Round 1	**KPQClean 프로젝트 시작** **kpqm4 프로젝트 시작**

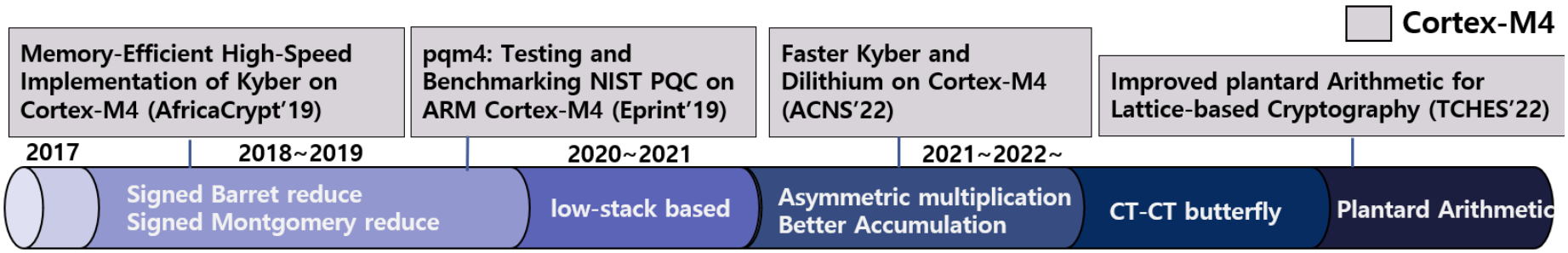
개요- Phase1 & 2 요약

❖ 본 연구과제 진행 내용

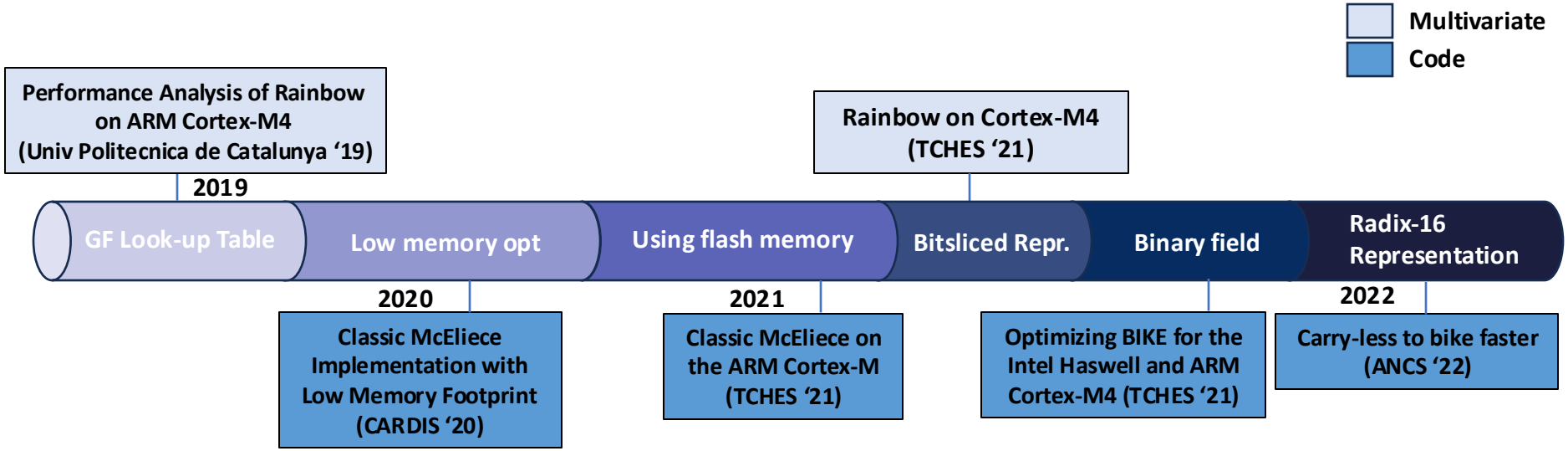
Phase 1 (4월 ~ 7월), 8차 워크숍	Phase 2 (7월 ~ 9월), 9차 워크숍
- 격자 / 코드 기반 PQC 최적 구현 동향 조사 - kpqm4 프레임워크 개발 - 알고리즘 포팅 - 통합 이슈 사항 정리 및 해결 - 알고리즘별 Speed / Stack 벤치마킹	- 다변수 기반 PQC 최적 구현 동향 조사 - 포팅 불가 알고리즘의 kpqm4 통합 방법론 - Flash memory 사용 방법론 - KpqC Round2 최신 업데이트 코드 반영 - 24.10.20.일까지 - 통합 이슈사항 정리 및 해결 - Cortex-M4 PQC 구현 가이드 문서 제작 - 프로그램 무결성 검증 스크립트 개발

구현최적화 동향

❖ 국외 Cortex-M4 Lattice 기반 암호 구현 최적화 연구 동향



❖ 국외 Cortex-M4 Code/Multivariate 기반 암호 구현 최적화 연구 동향



구현최적화 동향-Lattice

❖ 격자기반암호 최적화 방법론 요약

상세 방법론은 Appendix 참고

Modualr Arithmetic	개요	NIST PQC	KpqC 적용 (가능)
Signed improved Plantard	16-bit Modular Multiplication (2cc)	Kyber	NTRU+ SMAUG-T (Multimoduli)
Signed Montgomery	32-bit Modular Multiplication (3cc)	Dilithium	SMAUG-T (Unfriendly Ring) **HAETAE**, **NCC-Sign**
Signed Barrett	Modular Reduction - 두개의 packed 16-bit data에 대한 감산 (6cc) - 32-bit 단일 data에 대한 감산 (3cc)	Kyber	NTRU+
		Dilithium	**HAETAE**, **NCC-Sign**

Matrix-Vector Mul	개요	NIST PQC	KpqC 적용 (가능)
Streaming A and e	Module 구조의 메모리 최적화 방안	Kyber, Dilithium	SMAUG-T, HAETAE
Asymmetric Multiplication	Incomplete NTT의 PointMul 가속화 방안	Kyber	SMAUG-T (incomplete NTT)
Better Accumulation	Matrix-Vector Mul의 가속화 방안 (큰 누적 값)	Kyber, Dilithium	SMAUG-T, **HAETAE**

Others	개요	NIST PQC	KpqC 적용 (가능)
Merged NTT/iNTT	NTT/iNTT 어셈블리 메모리 최적화 방안	Kyber, Dilithium	SMAUG-T, NTRU+, **HAETAE**, **NCC-Sign**
Small NTT	작은 계수에 대한 half-word NTT (FNT, small NTT)	Dilithium	**HAETAE**
NTT for Unfriendly Ring	NTT Unfriendly Ring에 대한 NTT 적용 방안	Saber	SMAUG-T

구현최적화 동향-Code

❖ 코드기반암호 최적화 방법론 요약

- ➢ Classic McEliece와 Bike에 대한 Cortex-M4 구현 최적화 연구 존재
- ➢ HQC는 구현 최적화 연구 없음 → (국내) 최초 최적화 연구 수행

Classic-McEliece

- Classic McEliece Implementation with Low Memory Footprint, CARDIS 2020
- Classic McEliece on the ARM Cortex-M, TCHES 2021

BIKE

- Optimizing BIKE for the Intel Haswell and ARM Cortex-M4, TCHES 2021
- Carry-less to bike faster ACNS 2022

HQC

Cortex-M4 연구 X

상세 방법론은 Appendix 참고

메모리 최적화	개요	NIST PQC	KpqC 적용 (가능)
Streaming pk	LU 분해를 통한 공개키 Streaming 기법	Classic McEliece	PALOMA
Storing pk in Flash Memory	Streaming된 pk를 Flash Memory에 배치하는 기법		

이진체에서 다항식 곱셈	개요	NIST PQC	KpqC 적용 (가능)
Bernstein 5-Way	Intel Haswell을 타겟으로 적용된 recursive 곱셈	BIKE	-
Frobenius AFFT	Cortex-M4를 타겟으로 적용된 Addtive FFT 곱셈		

구현최적화 동향 – UOV

❖ 국외 Cortex-M4 UOV 기반 암호 구현 최적화 연구 동향

➢Rainbow에 대한 Cortex-M4 구현 최적화 연구 존재

- ✓ 대부분 Look-up table 기반 최적 구현
- ✓ Bitslicing 표현을 통한 곱셈 최적화 방법론이 제안됨

Performance Analysis of Rainbow on ARM Cortex-M4 – Evaluation of a Post-Quantum Multivariate Signature Scheme on a Resource-Constrained Device, Univ Politecnica de Catalunya 2019

- Karatsuba-Ofman 곱셈 최적화
- GF Look-up Table

Rainbow on Cortex-M4, TCHES '21

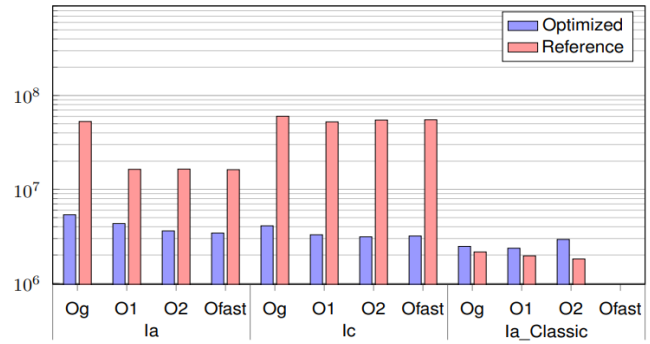
- Bitsliced Representation
- Tower field Arithmetic Optimization
- Constant-time Field Inversion
- Bitsliced Matrix Inversion

구현최적화 동향 – UOV

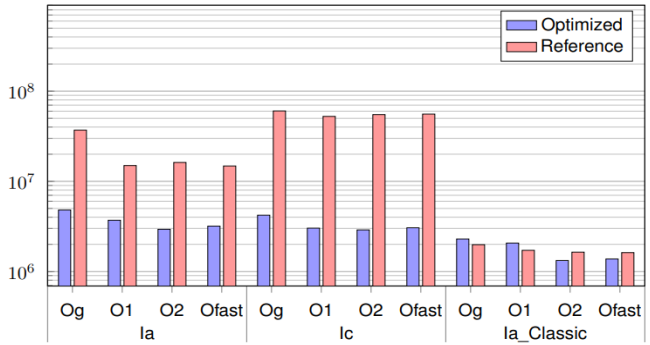
❖ "Performance Analysis of Rainbow on ARM Cortex-M4", 2019

- ➢ **Karatsuba-Ofman 곱셈에 대한 Look-up table 생성**
  - ✓ GF(16), GF(256)에 대한 곱셈 테이블 참조 구현
- ➢ GF(256) 곱셈에서 4번의 GF(16) 곱셈 발생
  - ✓ 크기가 큰 GF(256) 테이블보다는 GF(16) 4번 참조로 구현 가능

Sign



Verify



Algorithm 4 $GF(16)$Look-up Table Generation

Input: $GF(4)$ look-up table $gf4\_tab$
Output: $GF(16)$ look-up table $gf16\_tab$

1: $size \leftarrow 1 << 4$
2: **for** $i \leq size$ **do**
3: **for** $j \leq size$ **do**
4: $a_0b_0 \leftarrow$ `gf4_mul_lookupTable`$(x_0, y_0, gf4\_tab)$;
5: $a_1b_1 \leftarrow$ `gf4_mul_lookupTable`$(x_1, y_1, gf4\_tab)$;
6: $ab_x \leftarrow$ `gf4_mul_lookupTable`$(x_0 \oplus x_1, y_0 \oplus y_1, gf4\_tab) \oplus a_0b_0 \oplus a_1b_1$;
7: $a_1b_1\_2 \leftarrow$ `gf4_mul_lookupTable`$(a_1b_1, 2, gf4\_tab)$;
8: $gf16\_tab[16i + j] \leftarrow (ab_x \oplus a_1b_1) << 2) \oplus a_0b_0 + a_1b_1\_2$;
9: **return** $(gf16\_tab)$

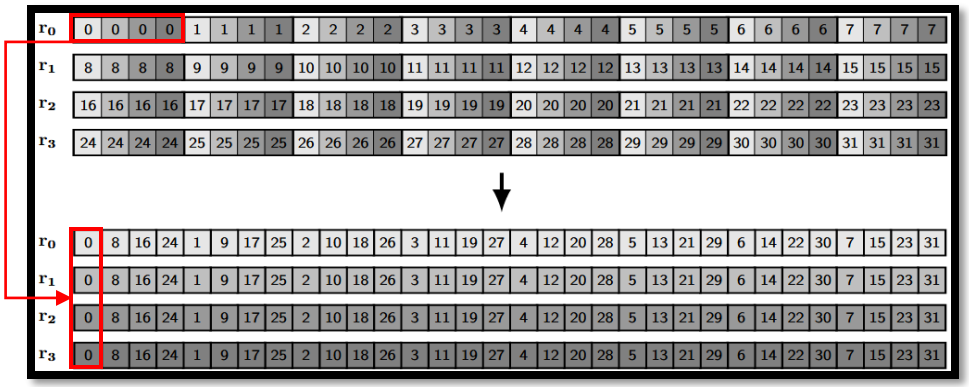
Algorithm 5 Look-up $GF(256)$ Table Generation

Input: $GF(16)$ look-up table $gf16\_tab$
Output: $GF(256)$ look-up table $gf256\_tab$

1: $size \leftarrow 1 << 8$
2: **for** $i \leq size$ **do**
3: **for** $j \leq size$ **do**
4: $a_0b_0 \leftarrow$ `gf16_mul_lookupTable`$(x_0, y_0, gf16\_tab)$;
5: $a_1b_1 \leftarrow$ `gf16_mul_lookupTable`$(x_1, y_1, gf16\_tab)$;
6: $ab_x \leftarrow$ `gf16_mul_lookupTable`$(x_0 \oplus x_1, y_0 \oplus y_1, gf16\_tab) \oplus a_0b_0 \oplus a_1b_1$;
7: $a_1b_1\_8 \leftarrow$ `gf16_mul_lookupTable`$(a_1b_1, 8, gf16\_tab)$;
8: $gf256\_tab[256i + j] \leftarrow (ab_x + a_1b_1) << 4) \oplus a_0b_0 \oplus a_1b_1\_8$;
9: **return** $(gf256\_tab)$

❖ "Rainbow on Cortex-M4", TCHES 2021

- ➢ $\mathbb{F}_{16} := \mathbb{F}_4[y]/(y^2+y+x)$ **의 유한체 연산을** $\mathbb{F}_4 := \mathbb{F}_2[x]/(x^2+x+1)$**로 표현**
  - ✓ Tower field representation으로 3개의 F_4 곱셈으로 F_{16} 곱셈 가능
- ➢ **Bitsliced representation 사용**
  - ✓ 32-bit 레지스터에 8개의 F_{16} 원소 packing 가능
  - ✓ 그러나 레지스터 4개에 32개의 원소를 packing하는 것이 더 빠른 곱셈 가능



- ➢ **Constant-time Field Inversion**
  - ✓ 16*4-bit 테이블의 constant-time 참조로 구현
  - ✓ F_{256}의 경우 플래시 메모리에 테이블 저장해야 함 (테이블 크기가 큼)

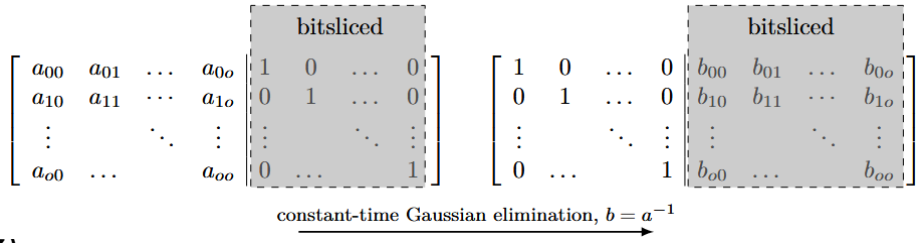
구현최적화 동향 – UOV

❖ "Rainbow on Cortex-M4", TCHES 2021

➢ **Matrix Inversion**

✓ 역행렬을 구할 때 bitsliced 표현을 유지하여 성능 향상

- 데이터 접근이 왼쪽 절반에만 발생
- 절반만 bitslice하여 곱셈할 때만 유지



➢ **성능 측정 결과** (EFM32GG11B, 16MHz)

✓ pqm4 프레임워크에 기반하여 성능 측정

Rainbow I	Key Gen	Sign	Verify	구분
classic	417 316k	5 433k	3 529k	ref
	134 354k	1 815k	1 619k	[MR19]
	98 431k	**957k**	**239k**	**This Work**
circumzenithal	462 322k	5 422k	27 965k	ref
	107 639k	**955k**	**12 903k**	**This Work**

Rainbow I의 두 개의 파라미터에 대한 Cortex-M4 구현 clock cycle 측정 결과

KOOKMIN UNIVERSITY

양자내성암호연구단

Phase 2

CSE CRYPTO & SECURITY ENGINEERING Lab

Release Code: https://github.com/COALA-5/kpqm4

❖ kpqm4 목표

➢ pqm4 프레임워크에 KpqC Round 2 알고리즘 최신 코드 통합

- ✓ 속도 및 메모리 벤치마크 기능 제공
- ✓ pqm4의 기본 테스트 기능 제공
  - hash clock cycles
  - 알고리즘 동작 테스트 (단순 동작, Canary Check, Negative Test)
- ✓ clean/m4/m4f 등 개발자가 제공한 kpqC 최신코드 통합

	Algorithm	최신 Code Update	pqm4 통합	kpqm4 통합
KEM	NTRU+	24.10.10.	-	√
	PALOMA	24.10.21.	-	√
	REDOG	Round 2	-	X
	SMAUG-T	24.10.14.	-	√
DSA	AIMer	24.10.11	√	√
	HAETAE	24.07.18.	√	√
	MQ-Sign	24.07.17.	-	√
	NCC-Sign	24.07.17.	-	√

kpqm4 – 개요

❖ pqm4 프레임워크에 KpqC Round 2 알고리즘 최신 코드 통합

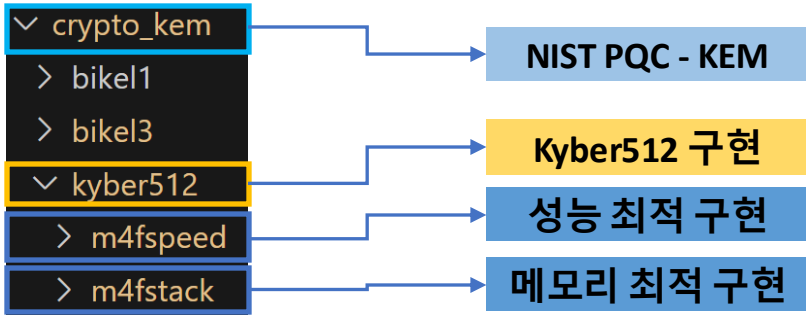
❖ pqm4와 동일하게 KpqC에 대한 벤치마크/검증 프로그램 자동 생성

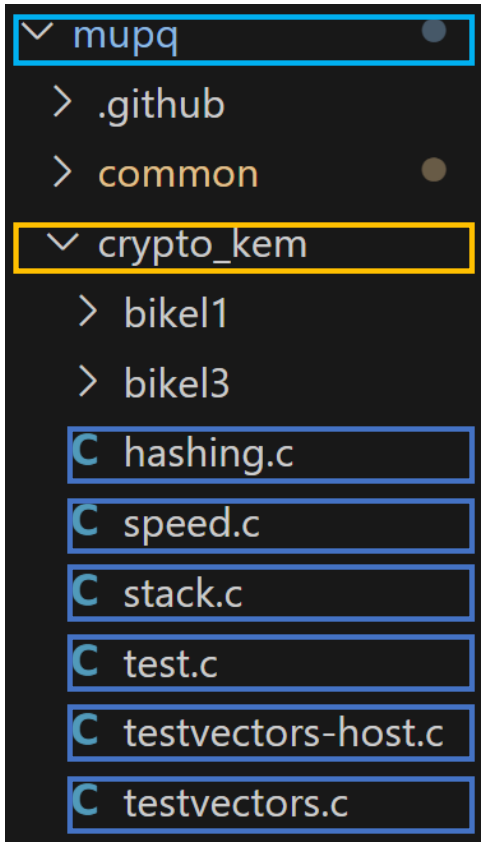
➢벤치마킹 대상: 스택 사용량, clock cycle, 해시 clock cycle

동작 주파수: 20MHz 고정

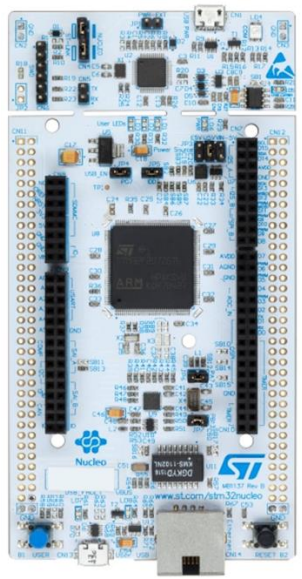
	설명
clean	- 외부 라이브러리 의존성 제거 - Cortex-M4에서 동작 가능하도록 수정
opt	- Reference 코드의 최적화된 C언어 구현
m4	- Reference 코드의 최적화된 어셈블리 구현 (Cortex-M4 대상)
m4f	- Reference 코드의 최적화된 어셈블리 구현 (Cortex-M4 대상, Floating Point Register 사용)

pqm4의 4가지 구현 레벨





❖ kpqm4 타겟 NUCLEO-144 개발보드 (STM32F429ZI MCU)



- ➢ **ARM Cortex-M4 내장** (최대 동작 클럭: 180 MHz)
  - ✓ ARMv7E-M 아키텍처
  - ✓ **코어 클럭 주파수 120MHz, 벤치마킹 주파수 20MHz 사용**
- ➢ **메모리 크기**
  - ✓ Flash Mem: 2 Mbytes, SRAM: 640 Kbytes
  - ✓ Flash: 4KB의 세그먼트가 512개 존재

0x08000000	4KB
0x08001000	4KB
0x08002000	4KB
⋮	
0x081FD000	4KB
0x081FE000	4KB
0x081FF000	4KB

STM32F429ZI의 플래시 메모리 모식도

```
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 640K   SRAM(O)
  RAM2   (xrw)    : ORIGIN = 0x10000000,   LENGTH = 64K    HW Parity Check(X)
  RAM3   (xrw)    : ORIGIN = 0x20040000,   LENGTH = 384K   CCM(X)
  FLASH  (rx)     : ORIGIN = 0x8000000,    LENGTH = 2048K  Flash(O)
}
```

STM32F429ZI의 메모리 영역 구분
O와 X는 메모리 사용 가능 여부

❖ 통합 상세

- ➢ 총 7개의 알고리즘에 대한 통합 완료
- ➢ √ : 모든 파라미터 통합
  △ : 부분 파라미터 통합

vuln : 알려진 취약점
pk : 큰 공개키 size
mem : 높은 메모리 footprint
not port : 완성되지 않은 코드
dyn mem : 동적 메모리 할당 실패

pk, sk, LUT **in Flash Memory**

	Algorithm	Kpqm4			reason(s) for exclusion				
		ref	m4f (mem)	params	vuln	pk	mem	not port	dyn mem
KEM	NTRU+	√	-	4/4					
	PALOMA	△	-	1/3		x	x		x
	REDOG	-	-	0/3				x	
	SMAUG-T	√	-	3/3					
DSA	AIMer	△	√ (mem)	5/6			x (ref5)		
	HAETAE	√	√ (m4f)	6/6					
	MQ-Sign	△	-	1/3		x	x		
	NCC-Sign	√	√ (m4f)	6/6					

❖ 알고리즘 올바른 통합을 위한 고려 사항

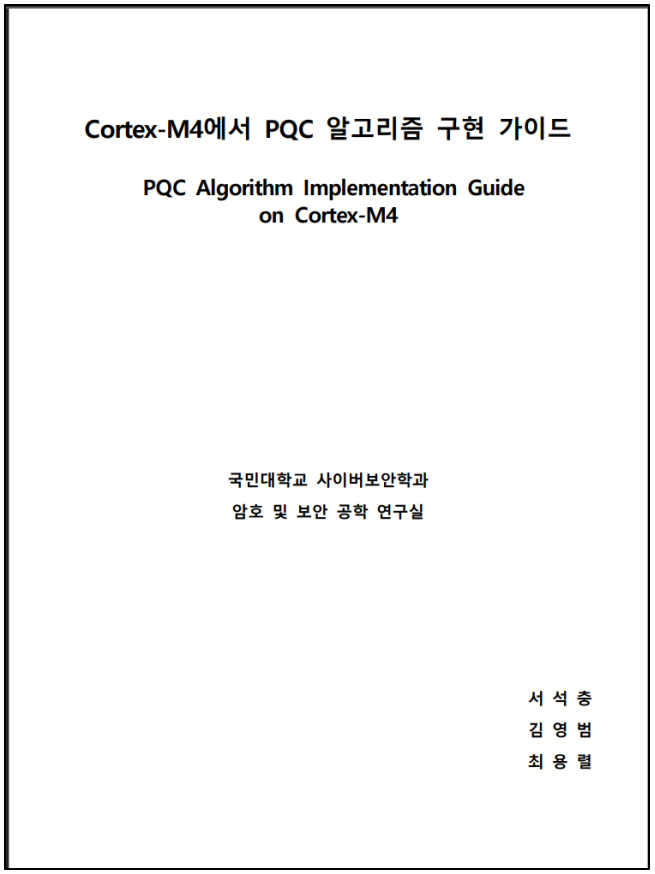
➢ 프로그램 오동작 및 통합에 이슈가 발생하는 부분을 해결

알고리즘 별 수정된 부분은 Appendix 참고

고려 사항	상세	해당 알고리즘
외부 라이브러리 종속성 제거	- OpenSSL 등 종속성 제거 - pqm4의 암호 알고리즘 사용	- SMAUG-T - NCC-Sign, MQ-Sign - NTRU+
동적할당 제거	- 메모리 파편화 유발 가능 - 동적할당 실패 가능성	- SMAUG-T - MQ-Sign - PALOMA
최상위 API 통일	- pqm4와 API 통일 - 통일 불가할 경우 대책 수립	~~- MQ-Sign (구)~~ ~~- PALOMA (구)~~ - SMAUG-T
디버깅 코드 제거	- stdio.h의 입출력 함수 제거	~~- PALOMA (구)~~

❖ Cortex-M4 환경에서 PQC 알고리즘 구현을 위한 가이드 문서 제공

- ➢ Cortex-M4 환경에서의 pqc 기술 및 구현 가이드 제공
- ➢ GitHub 레포지토리에 함께 제공

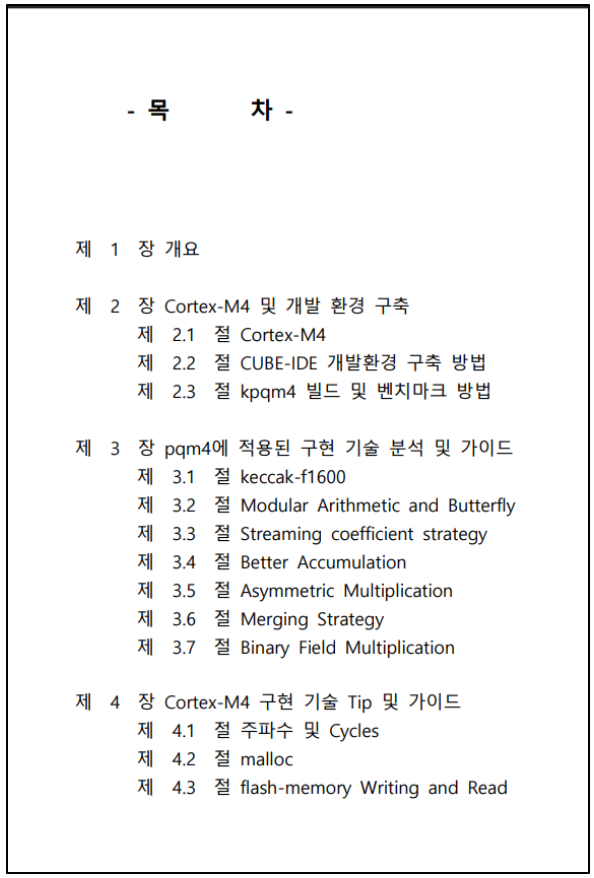


Cortex-M4에서 PQC 알고리즘 구현 가이드

PQC Algorithm Implementation Guide
on Cortex-M4

국민대학교 사이버보안학과
암호 및 보안 공학 연구실

서 석 충
김 영 범
최 용 렬

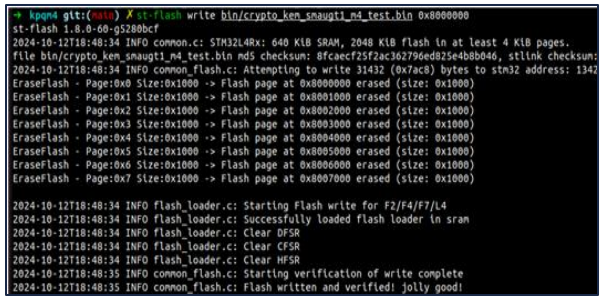


- 목 차 -

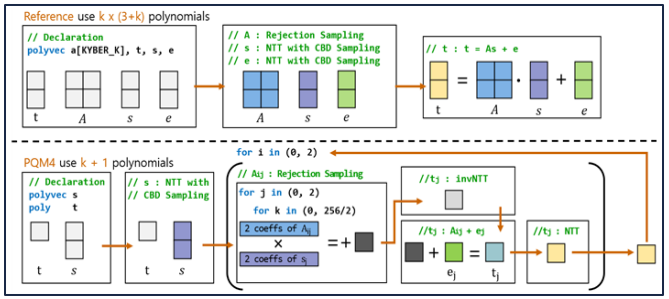
❖ Cortex-M4 환경에서 PQC 알고리즘 구현을 위한 가이드 문서 제공

- ➢ Cortex-M4 환경에서의 pqc 기술 및 구현 가이드 제공
- ➢ GitHub 레포지토리에 함께 제공

환경 설정 코드



실행 예시



기술 분석 모식도

포함 항목

- 개발 환경 구축 방법 제공

- CubeIDE 환경 구축
- UART 통신 설정
- kpqm4 빌드 / 벤치마크

- pqm4 코드 분석 및 적용 가이드

- Keccak f1600 permutation
- Multiplication Methods
- Streaming coefficients
- NTT Merging
- etc.

- Cortex-M4 구현 Tip 및 가이드

- 주파수 및 clock 설정
- 동적할당
- Flash memory write/read

❖ 가이드 문서 개요

➢ CubeIDE 및 kpqm4 환경 설정을 위한 코드 / 명령어 사용 가능

- ✓ CubeIDE : 프로젝트 생성 후 UART 설정 및 케이블 연결 방법
- ✓ kpqm4 : 기본적인 빌드 방법 및 테스트 방법

```
UART_HandleTypeDef UartHandle;

/* Private function prototypes
-----------------------------------------------*/
#ifdef __GNUC__
  /* With GCC, small printf (option LD Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */


/**
  * @brief  Retargets the C library printf function to the USART.
  * @param  None
  * @retval None
  */
PUTCHAR_PROTOTYPE {
  /* Place your implementation of fputc here */
  /* e.g. write a character to the EVAL_COM1 and Loop until the end of
transmission */
  HAL_UART_Transmit(&UartHandle, (uint8_t *)&ch, 1, 0xFFFF);

  return ch;
}
```

```
#ifdef __GNUC__
  /* With GCC, small printf (option LD Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

/**
  * @brief  Retargets the C library printf function to the USART.
  * @param  None
  * @retval None
  */
PUTCHAR_PROTOTYPE
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the EVAL_COM1 and Loop until the end of transmission */
  HAL_UART_Transmit(&hlpuart1, (uint8_t *)&ch, 1, 0xFFFF);

  return ch;
}
```

CubeIDE UART 설정 코드(좌)를 실제 CubeIDE에서 사용한 모습(우)

- st-flash write bin/<binary_file_name>.bin 0x8000000
- 예를 들어, SMAUG-T1의 벤치마크를 구동할 경우 st-flash write bin/crypto_kem_smaugt1_m4_test.bin.bin 0x8000000을 실행한다.
- 해당 명령어를 실행하면 [그림 5]와 같이 바이너리 파일이 플래시된다.

```
➜ kpqm4 git:(main) ✗ st-flash write bin/crypto_kem_smaugt1_m4_test.bin 0x8000000
st-flash 1.8.0-60-g5280bcf
2024-10-12T18:48:34 INFO common.c: STM32L4Rx: 640 KiB SRAM, 2048 KiB flash in at least 4 KiB pages.
file bin/crypto_kem_smaugt1_m4_test.bin md5 checksum: 8fcaecf25f2ac362796ed825e4b8b046, stlink checksum:
2024-10-12T18:48:34 INFO common_flash.c: Attempting to write 31432 (0x7ac8) bytes to stm32 address: 1342
EraseFlash - Page:0x0 Size:0x1000 -> Flash page at 0x8000000 erased (size: 0x1000)
EraseFlash - Page:0x1 Size:0x1000 -> Flash page at 0x8001000 erased (size: 0x1000)
EraseFlash - Page:0x2 Size:0x1000 -> Flash page at 0x8002000 erased (size: 0x1000)
EraseFlash - Page:0x3 Size:0x1000 -> Flash page at 0x8003000 erased (size: 0x1000)
EraseFlash - Page:0x4 Size:0x1000 -> Flash page at 0x8004000 erased (size: 0x1000)
EraseFlash - Page:0x5 Size:0x1000 -> Flash page at 0x8005000 erased (size: 0x1000)
EraseFlash - Page:0x6 Size:0x1000 -> Flash page at 0x8006000 erased (size: 0x1000)
EraseFlash - Page:0x7 Size:0x1000 -> Flash page at 0x8007000 erased (size: 0x1000)

2024-10-12T18:48:34 INFO flash_loader.c: Starting Flash write for F2/F4/F7/L4
2024-10-12T18:48:34 INFO flash_loader.c: Successfully loaded flash loader in sram
2024-10-12T18:48:34 INFO flash_loader.c: Clear DFSR
2024-10-12T18:48:34 INFO flash_loader.c: Clear CFSR
2024-10-12T18:48:34 INFO flash_loader.c: Clear HFSR
2024-10-12T18:48:35 INFO common_flash.c: Starting verification of write complete
2024-10-12T18:48:35 INFO common_flash.c: Flash written and verified! jolly good!
```

kpqm4 바이너리 플래시 명령어(좌)를 실제 터미널에서 실행한 모습(우)

❖ Cortex-M4 명령어 분석 및 Modular Multiplication 적용 코드 제공

Cortex-M4 명령어 세트 설명

명령어	설명
mov r0, r1	r0 ← r1
mov r0, #7	r0 ← 7 동일계열 : **movw**는 하위 16-bit, **movt**는 상위 16-bit
add r0, r1, r2	r0 ← r1 + r2, 동일계열 : **adds** (sets flags), **adc** (with carry), **adcs**
ror r0, r1, r2	r0 ← r1 >> r2 (rotate right by r2) 동일계열 : **lsl, lsr, asr**
mul r0, r1, r2	r0 ← (r1 × r2) mod 2^{32}
mla r0, r1, r2	r0 ← r0 + (r1 × r2) mod 2^{32}
mls r0, r1, r2	r0 ← r0 - (r1 × r2) mod 2^{32}
smull r0, r1, r2, r3	(r1 ‖ r0) ← r1 × r2, **smull** for signed, **umull** for unsigned
smlal r0, r1, r2, r3	(r1 ‖ r0) ← (r1 ‖ r0) + (r1 × r2), **smlal** for signed, **umlal** for unsigned
smulbb r0, r1, r2	r0 ← (r1의 하위 16-bit) × (r2의 하위 16-bit), 동일계열 : **smulbt, smultb, smultt** (t는 레지스터의 상위 16-bit를 의미)
smuad r0, r1, r2	r0 ← ((r1의 하위 16-bit) × (r2의 하위 16-bit)) +((r1의 상위 16-bit) × (r2의 상위 16-bit))
smulwb r0, r1, r2	r0 ← ((r1 × (r2의 하위 16-bit))의 상위 32-bit), 동일계열 : smulwt
smlabb r0, r1, r2	r0 ← r0 + ((r1의 하위 16-bit) × (r2의 하위 16-bit)), 동일계열 : **smlabt, smlatb,** smlatt (t는 레지스터의 상위 16-bit를 의미)
smlawb r0, r1, r2	r0 ← r0 + ((r1 × (r2의 하위 16-bit))의 상위 32-bit), 동일계열 : **smlawt**
pkhtb r0,r1,r2, asr #n	r0 ← ((r1의 상위 16-bit) << 16)) | (r2 >> n)
uadd16 r0, r1, r2	r0 ← (((r1의 상위 16-bit) + (r2의 상위 16-bit)) ‖ ((r1의 하위 16-bit) + (r2의 하위 16-bit))), 동일계열 : **usub16**

32-bit ModMul 예제

□ 32-bit Modular Multiplication (모듈러 곱셈)

■ 32-bit 모듈러 곱셈은 일반적으로 Montgomery 산술을 사용한다. 32-bit Improved Plantard 곱셈을 진행하기 위해서는 64-bit 사전 계산값이 필요하다. 따라서 Montgomery 곱셈을 사용하는 것이 최선이며, 32-bit 크기의 레지스터를 갖는 Cortex-M4에서는 하나의 레지스터에 하나의 계수를 처리한다.

■ Cortex-M4에서 32-bit Montgomery 곱셈에 대한 연구는 2020년에 제안된 이후에 크게 달라진 것은 없다. 하기 알고리즘은 참조 논문에서의 CT 및 GS butterfly에 대한 Cortex-M4 코드이다. CT butterfly의 Line 4~6번이 Montgomery 곱셈 수행 부분이다.

- $\mathrm{CT}(a, b) = a + \mathrm{Mont}(b \cdot \omega),\ a - \mathrm{Mont}(b \cdot \omega)$

```
.macro CT_butterfly p0, p1, twiddle
        ; q=8380417, qinv=4236238847
        ; Input: p0, p1, twiddle
        ; Output: p0, p1
        smull   tmp0, tmp1, p1, twiddle
        mul     p1, tmp0, qinv
        smlal   tmp0, tmp1, p1, q
        sub     p1, p0, tmp1
        add     p0, p0, tmp1
.endm
```

```
.macro GS_butterfly p0, p1, twiddle
        ; q=8380417, qinv=4236238847
        ; Input: p0, p1, twiddle
        ; Output: p0, p1
        sub     tmp0, p0, p1
        add     p0, p0, p1
        smull   tmp1, p1, tmp0, twiddle
        mul     tmp0, tmp1, qinv
        smlal   tmp1, p1, tmp0, q
.endm
```

[표 5] (좌) : Dilithium의 CT Butterfly, (우) Dilithium의 GS Butterfly
(https://github.com/mupq/pqm4/blob/master/crypto_sign/dilithium2/m4f/macros.i)

□ 32-bit Modular Reduction (모듈러 감산)

■ 32-bit Modular Reduction은 크게 두 가지 방식으로 구현한다. Dilithium 에서는 Barrett 감산의 변형을 사용하여, 32-bit로 표현되는 양/음수의 값에 대해서 $(-\frac{3}{4}q, \frac{3}{4}q)$ 범위내로 감산을 수행한다. 최종적으로 $(0, q)$ 범위내로 표현하고 싶으면 $(-\frac{3}{4}q, 0)$ 범위 내에 있는 값에다가 q를 더해주면 된다. 하기는 Dilithium의 reference C 코드의 pqm4 Asm 표현이다.

Dilithium reference C code

```
int32_t reduce32(int32_t a)
{
  int32_t t;

  t = (a + (1 << 22)) >> 23;
  t = a - t*Q;
  return t;
}
```

pqm4 Asm code

```
.macro redq a, tmp, q
        add     ₩tmp, ₩a,   #4194304
        asrs    ₩tmp, ₩tmp, #23
        mls     ₩a, ₩tmp, ₩q, ₩a
.endm
```

❖ NTT 구현을 위한 예제 코드 제공

Kyber Refrence NTT code (3 Layer)

```
// 3 Layer에 대한 NTT 계산
void ntt (int16_t r[256])
{
    unsigned int len, start, j, k;
    int16_t t, zeta;

    k = 1;
    // 0 Layer : len = 128
    // 1 Layer : len = 64
    // 2 Layer : len = 32

    // 매 Layer 계산 시 메모리 접근이 발생
    for (len = 128; len >= 32; len >>= 1) {
        for (start = 0; start < 256; start = j + len) {
            zeta = zetas[k++];
            for (j = start; j < start + len; j++) {
                t = fqmul(zeta, r[j + len]);
                r[j + len] = r[j] - t;
                r[j] = r[j] + t;
            }
        }
    }
}
```

ref NTT 코드 분석

Kyber NTT Merging 3 Layer code

```
void butfly(int16_t *a, int16_t *b, int16_t zeta)
{
    int16_t t = fqmul(*b, zeta);
    *b = *a - t;
    *a = *a + t;
}
// 3 Layer merging 기법이 적용된 NTT 코드
void merge3_ntt(int16_t r[256])
{
    int16_t __m_zetas[8] = {0};
    int16_t v[8] = { 0 };

    // Twiddle Factor Load
    for (int i = 0; i < 8; i++) __m_zetas[i] = m_zetas[i];

    for (int i = 0; i < 32; i++)
    {
    // 8개의 다항식 계수 Load
        for (int j = 0; j < 8; j++) v[j] = r[32 * j + i];

    // Load된 8개 계수에 대한 3 Layer butterfly를 수행
        butfly(&v[0], &v[4], __m_zetas[1]);
        butfly(&v[1], &v[5], __m_zetas[1]);
        butfly(&v[2], &v[6], __m_zetas[1]);
        butfly(&v[3], &v[7], __m_zetas[1]);

        butfly(&v[0], &v[2], __m_zetas[2]);
        butfly(&v[1], &v[3], __m_zetas[2]);
        butfly(&v[4], &v[6], __m_zetas[3]);
        butfly(&v[5], &v[7], __m_zetas[3]);

        butfly(&v[0], &v[1], __m_zetas[4]);
        butfly(&v[2], &v[3], __m_zetas[5]);
        butfly(&v[4], &v[5], __m_zetas[6]);
```

3 Layer Merging 예제 코드 제공

pqm4 Kyber speed version code

```
// 32-bit 레지스터 2개(a0, a1)에 저장된 총 4개의 계수에 대한 Butterfly, 즉 doublebutterfly는 연속된 두개의 계수 쌍에 대한 Butterlfy를 의미
.macro doublebutterfly_plant a0, a1, twiddle, tmp, q, qa
        smulwb ₩tmp, ₩twiddle, ₩a1
        smulwt ₩a1, ₩twiddle, ₩a1
        smlabt ₩tmp, ₩tmp, ₩q, ₩qa
        smlabt ₩a1, ₩a1, ₩q, ₩qa
        pkhtb ₩tmp, ₩a1, ₩tmp, asr#16
        usub16 ₩a1, ₩a0, ₩tmp
        uadd16 ₩a0, ₩a0, ₩tmp
.endm

// 4개의 레지스터 즉, 총 8개의 계수에 대한 Butterfly
.macro two_doublebutterfly_plant a0, a1, a2, a3, twiddle0, twiddle1, tmp, q, qa
        doublebutterfly_plant ₩a0, ₩a1, ₩twiddle0, ₩tmp, ₩q, ₩qa
        doublebutterfly_plant ₩a2, ₩a3, ₩twiddle1, ₩tmp, ₩q, ₩qa
.endm

// 3 Layer Merging을 위해서는 2^3개의 레지스터가 필요함 (c0, c1, ~, c7)
.macro _3_layer_double_CT_16_plant c0, c1, c2, c3, c4, c5, c6, c7, twiddle1, twiddle2, twiddle_ptr, q, qa, tmp
// 첫번째 레이어로 CT( (c0, c1, c2, c3), (c4, c5, c6, c7), t1 )에 대한 계산을 수행. twiddle factor(t1)은 한개 필요
  따라서, 세부적으로 동일한 twiddle에 대해 CT(c0,c4, t1), CT(c1,c5, t1), CT(c2,c6, t1), CT(c3,c7, t1)를 계산
        ldr.w ₩twiddle1, [₩twiddle_ptr], #4
        two_doublebutterfly_plant ₩c0, ₩c4, ₩c1, ₩c5, ₩twiddle1, ₩twiddle1, ₩tmp, ₩q, ₩qa
        two_doublebutterfly_plant ₩c2, ₩c6, ₩c3, ₩c7, ₩twiddle1, ₩twiddle1, ₩tmp, ₩q, ₩qa
        ----------
// 두번째 레이어로 twiddle factor는 두개가 필요 (t2, t3).
  CT( (c0, c1), (c2, c3), t2) 및 CT( (c4, c5), (c6, c7), t3),에 대한 계산을 수행.
  따라서, 세부적으로 두개의 twiddle에 대해 CT(c0, c2, t2), CT(c1, c3. t2), CT(c4, c6, t3), CT(c5, c7, t3)를 계산
        ldrd ₩twiddle1, ₩twiddle2, [₩twiddle_ptr], #8
        two_doublebutterfly_plant ₩c0, ₩c2, ₩c1, ₩c3, ₩twiddle1, ₩twiddle1, ₩tmp, ₩q, ₩qa
        two_doublebutterfly_plant ₩c4, ₩c6, ₩c5, ₩c7, ₩twiddle2, ₩twiddle2, ₩tmp, ₩q, ₩qa
        ----------
// 두번째 레이어로 twiddle factor는 4개가 필요 (t4, t5, t6, t7).
  4개의 twiddle에 대해 CT(c0, c1, t4), CT(c2, c3. t5), CT(c4, c5, t6), CT(c6, c7, t7)를 계산
        ldrd ₩twiddle1, ₩twiddle2, [₩twiddle_ptr], #8
        two_doublebutterfly_plant ₩c0, ₩c1, ₩c2, ₩c3, ₩twiddle1, ₩twiddle2, ₩tmp, ₩q, ₩qa
        ldrd ₩twiddle1, ₩twiddle2, [₩twiddle_ptr], #8
        two_doublebut
.endm
// Merging 전략을 사용하
```

pqm4 NTT 코드 분석

❖ Flash Memory Read/Write 방법론 제공

- ➢ CubeIDE에서 Flash Memory에 대한 사용설정 방법론 제공
- ➢ KpqC 알고리즘에 대한 적용 Example 제공

제 4.3 절 flash-memory Writing and Read

□ Flash-memory에 읽고 쓰기

■ STM 보드에서는 플래시 메모리 영역에 데이터를 읽고 쓸 수 있다. 플래시 메모리는 메모리 영역의 일부로, 프로그램이 동작하기 전에 저장된 데이터이며, 프로그램이 실행되면서 저장된 데이터를 읽거나 해당 영역에 다시 쓰는 것이 가능하다.

■ CubeIDE에서는 플래시 메모리에 데이터를 올리기 위해서 해당 변수에 다음과 같은 attribute를 부여한다.
- 예를 들어, unsigned char로 선언된 a 배열을 플래시 메모리에 올리기 위해서는 다음과 같이 작성한다.: __attribute__((section(".text"))) unsigned char a[] = { /* 데이터 */ };

```
__attribute__((section(".text"))) <type> <variable_name>
```

■ 이렇게 attribute가 부여된 변수는 CubeIDE에서 프로젝트를 빌드하고 실행할 때, 먼저 .text 영역에 데이터가 플래시 된다. .text 영역은 플래시 메모리의 일부로, 결과적으로 플래시 메모리에 데이터를 쓴 것이다.

■ KpqC 알고리즘을 통한 예시를 설명한다. 대상 알고리즘은 MQ-Sign으로, STM32429I-EVAL1 보드의 SRAM 크기를 초과하는 키 쌍 크기를 가진다. 그러나 해당 보드의 플래시 메모리는 2MB이기 때문에 키 쌍을 플래시 메모리에 저장한다면 충분히 구동이 가능하다.

■ MQ-Sign의 공개키와 개인키를 사전에 생성한다. 이는 PC에서 직접 생성하며, 키 생성 함수를 통해

MQ-Sign에 대한 Flash Memory 사용 방법론

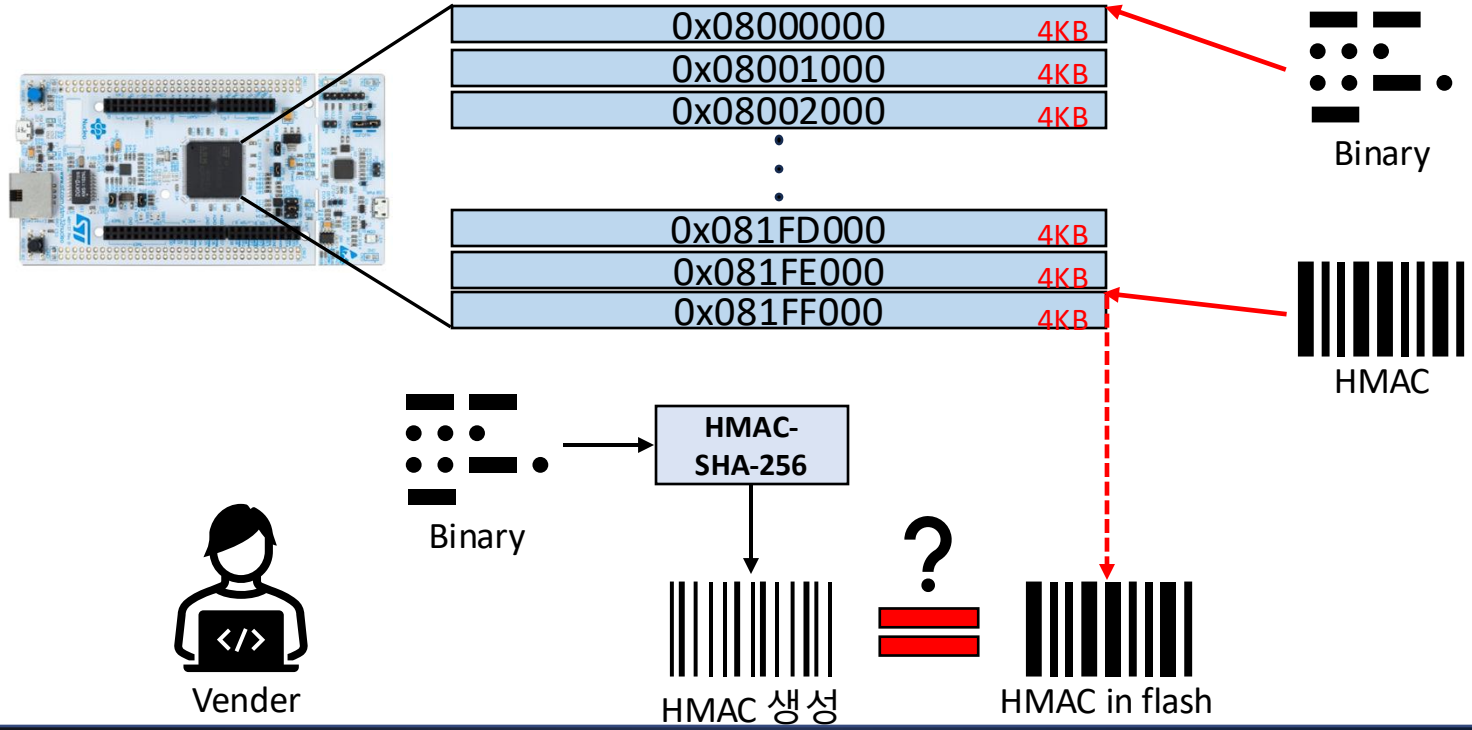
❖프로그램 무결성을 검증할 수 있는 스크립트 개발

➢바이너리(.bin)의 HMAC-SHA-256 값을 계산하여 Flash memory에 저장

✓저장 주소: 0x081FF000 (플래시 메모리의 마지막 섹터)

➢향후 KpqC 알고리즘이 암호모듈에 탑재될 경우 KCMVP 검증에 활용 가능

✓검증자가 HMAC 값 비교로 프로그램 무결성 검증



❖프로그램 무결성을 검증할 수 있는 스크립트 개발

- ➢바이너리 플래시 후 flash_hmac.sh 스크립트 실행
- ➢쉘 스크립트 실행 후 플래시한 바이너리의 파일명 입력
  - ✓입력 후 자동으로 HMAC 값 생성 후 플래시에 업로드 (현재 키는 고정됨)
- ➢STM32CubeProgrammer로 플래시된 HMAC 값 확인 가능

STM32CubeProgrammer CLI 설치 필요

```
→ kpqm4 git:(main) X integrity_test/flash_hmac.sh
Enter a filename without .bin:
crypto_kem_smaugt1_m4_test
Filepath: crypto_kem_smaugt1_m4_test
      -------------------------------------------------------------------
                        STM32CubeProgrammer v2.17.0
      -------------------------------------------------------------------

ST-LINK SN  : 0669FF363355473043224647
ST-LINK FW  : V2J45M31
Board       : NUCLEO-L4R5ZI
Voltage     : 3.23V
SWD freq    : 4000 KHz
Connect mode: Normal
Reset mode  : Software reset
Device ID   : 0x470
Revision ID : Rev V
Device name : STM32L4Rxxx/STM32L4Sxxx
Flash size  : 2 MBytes
Device type : MCU
Device CPU  : Cortex-M4
BL Version  : 0x95
Debug in Low Power mode enabled


Memory Programming ...
Opening and parsing file: hmac_crypto_kem_smaugt1_m4_test.bin
  File         : hmac_crypto_kem_smaugt1_m4_test.bin
  Size         : 32.00 B
  Address      : 0x081FF000

Erasing memory corresponding to segment 0:
Erasing internal memory sector 65535
Download in Progress:
[==================================================] 100%

File download complete
Time elapsed during download operation: 00:00:00.095


Verifying ...

Read progress:
[==================================================] 100%

Download verified successfully
```

Device memory | Open file | +

Address 0x081FF00 | Size 0x400 | Data wi... 32-bit | Find Data 0x

Address	0	4	8	C	
0x081FF000	CAE8821F	0679618C	92AE8F56	7DEAD81F	..èÊ.ay.V.®..Øê}
0x081FF010	EDF76884	58922145	0AFE7EC1	EE50AF71	.h÷íE!.XÁ~þ.q¯Pî
0x081FF020	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ

STM32CubeProgrammer로 확인한 플래시 결과

hmac_crypto_kem_smaugt1_m4_test.txt ×

hmac_crypto_kem_smaugt1_m4_test.txt

1 1f82e8ca8c617906568fae921fd8ea7d8468f7ed45219258c17efe0a71af50ee

실제로 생성된 HMAC 값

Endian 때문에 4-byte 단위로 반전되어서 플래시됨

❖ Cortex-M4 Speed 성능측정 방법론 (pqm4)

- ➢ System Control Space의 SysTick을 읽어서 현재 clock cycles를 측정
  - ✓ hal_get_time 함수를 구현하여 사용

```
typedef struct
{
  __IOM uint32_t CTRL;                  /*!< Offset: 0x000 (R/W)  SysTick Control and Status Register */
  __IOM uint32_t LOAD;                  /*!< Offset: 0x004 (R/W)  SysTick Reload Value Register */
  __IOM uint32_t VAL;                   /*!< Offset: 0x008 (R/W)  SysTick Current Value Register */
  __IM  uint32_t CALIB;                 /*!< Offset: 0x00C (R/ )  SysTick Calibration Register */
} SysTick_Type;
```

```
uint64_t hal_get_time()
{
  while (1) {
    unsigned long long before = overflowcnt;
    unsigned long long result = (before + 1) * 16777216llu - SysTick->VAL;
    if (overflowcnt == before) {
      return result;
    }
  }
}
```

```
// Encapsulation
t0 = hal_get_time();
MUPQ_crypto_kem_enc(ct, key_a, pk);
t1 = hal_get_time();
printcycles("encaps cycles:", t1-t0);
```

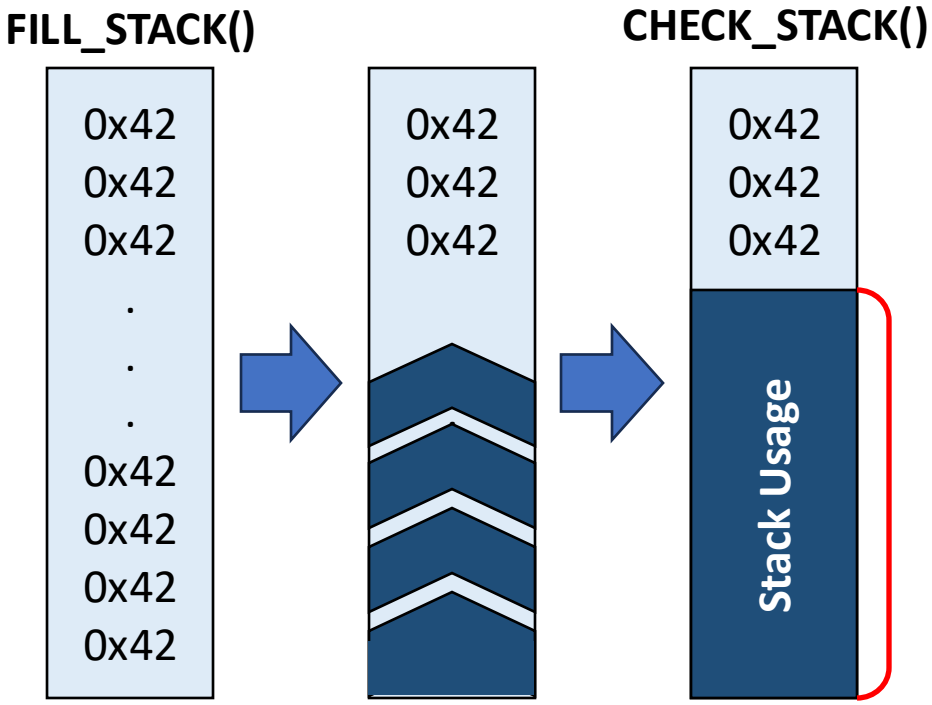
hal_get_time 함수의 구성

❖ Cortex-M4 Stack 성능측정 방법론 (pqm4)

- ➢Stack을 특정 값으로 모두 채운 뒤에 **값이 바뀐 범위를 확인** (canary 이용)
  - ✓Canary가 바뀐 부분만큼 Stack을 사용한 것
- ➢프로그램을 실행하며 **동적으로 확인**

```
uint8_t canary = 0x42;
#define FILL_STACK()
  p = &a;
  while (p > &a - canary_size)
    *(p--) = canary;
#define CHECK_STACK()
  c = canary_size;
  p = &a - canary_size + 1;
  while (*p == canary && p < &a) {
    p++;
    c--;
  }
```

pqm4의 Stack 측정 관련 코드



pqm4의 Stack 측정 과정의 모식도

kpqm4 벤치마크 결과 - KEM

❖ **(NUCLEO) KEM 성능 벤치마크 대상**

- **KpqC Algorithm:** SMAUG-T, NTRU+, PALOMA, ~~REDOG~~
- **Comparison Algorithm:** ML-KEM, Classic McEliece

특이사항

* **SMAUG-T**: clean
* **NTRU+** : clean
* **ML-KEM** : clean, m4f (speed)
* **McEliece**: m4 (pk streaming)
* **PALOMA**: clean (pk FLASH)

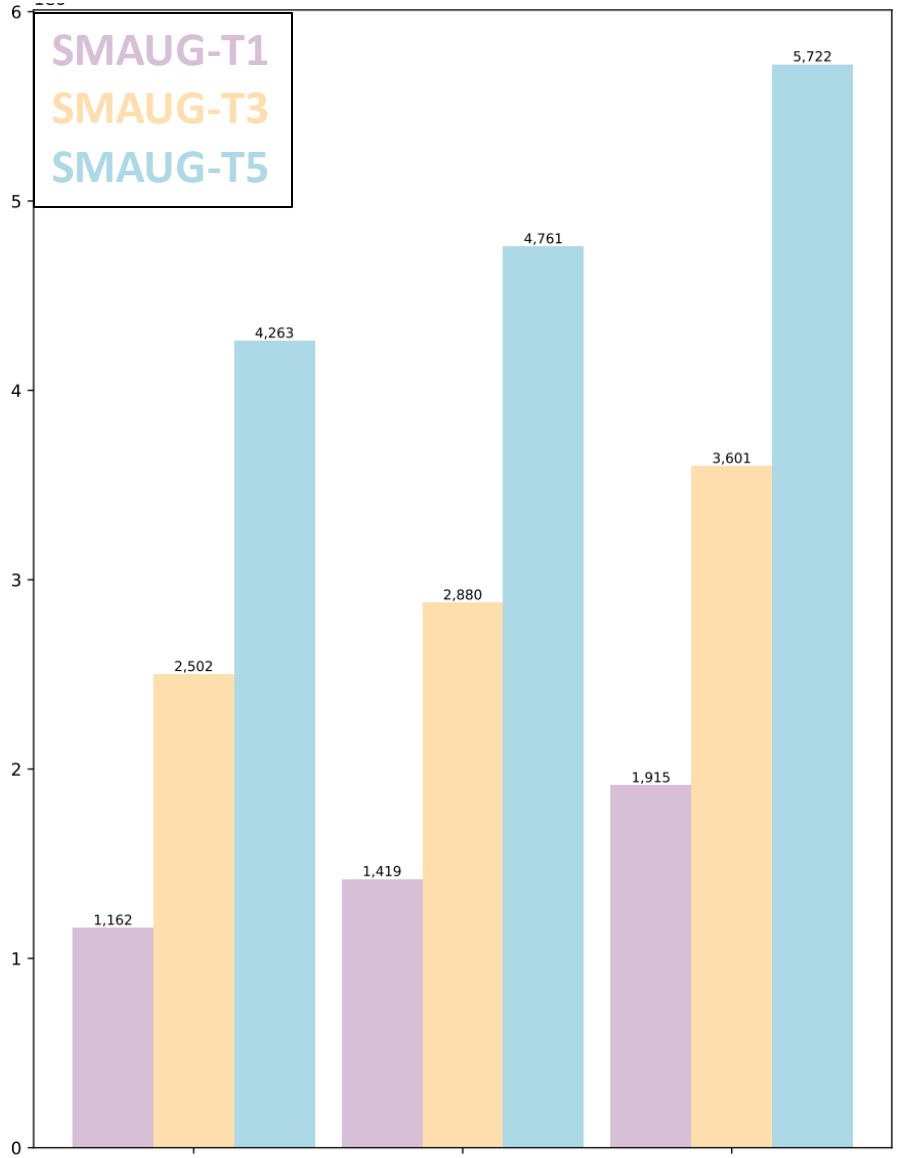
Category	Step	(high speed)				(low speed)
Speed (Security Level 1)	KeyGen	ML-KEM (m4f)	NTRU+ (clean)	ML-KEM (clean)	SMAUG-T (clean)	McEliece (m4)
	Encap	ML-KEM (m4f)	NTRU+ (clean)	McEliece (m4)	PALOMA (clean)	ML-KEM (clean)	SMAUG-T (clean)
	Decap	ML-KEM (m4f)	NTRU+ (clean)	ML-KEM (clean)	SMAUG-T (clean)	McEliece (m4)	PALOMA (clean)

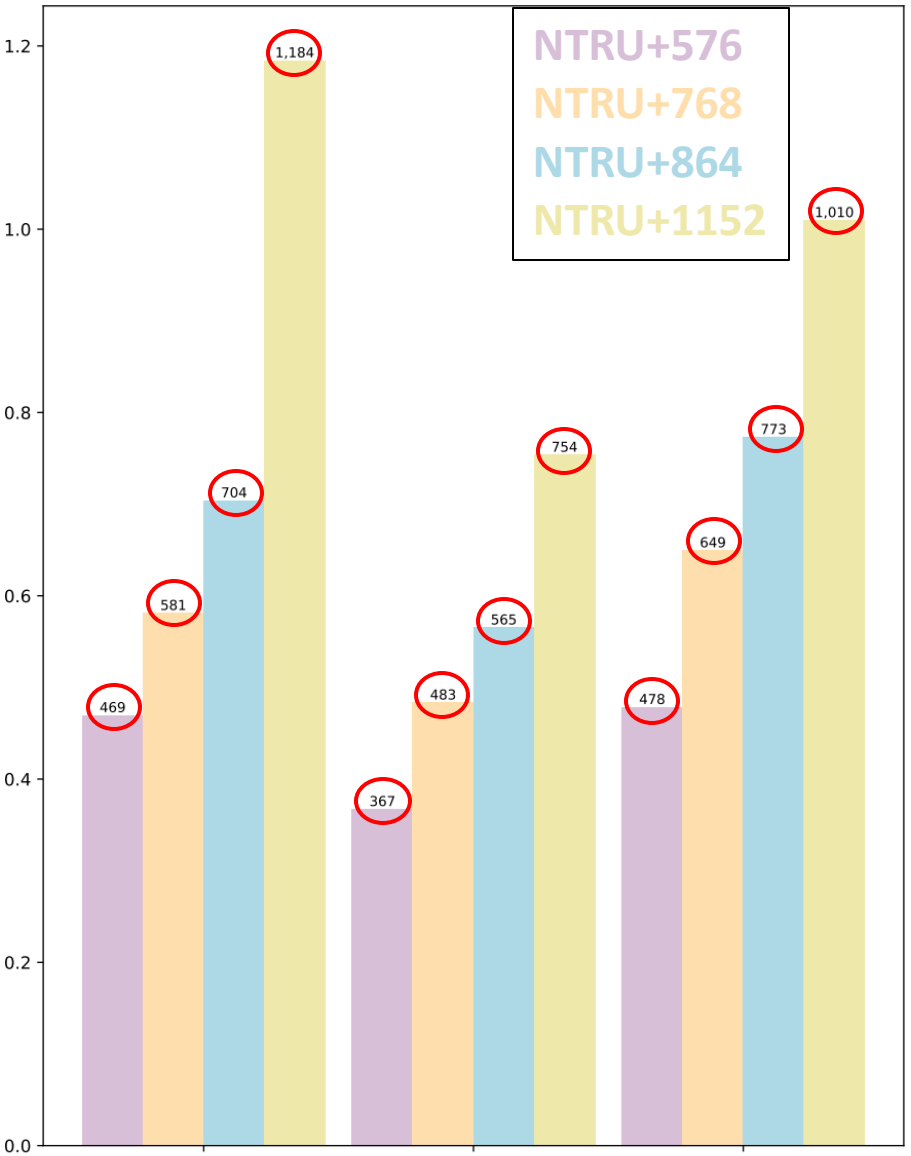
Category	Step	(low stack)				(high stack)
Stack Usage (Security Level 1)	KeyGen	ML-KEM (m4f)	ML-KEM (clean)	NTRU+ (clean)	SMAUG-T (clean)	McEliece (m4)
	Encap	McEliece (m4)	ML-KEM (m4f)	ML-KEM (clean)	NTRU+ (clean)	SMAUG-T (clean)	PALOMA (clean)
	Decap	ML-KEM (m4f)	ML-KEM (clean)	SMAUG-T (clean)	NTRU+ (clean)	PALOMA (clean)	McEliece (m4)

kpqm4 벤치마크 결과 - KEM

CSE CRYPTO & SECURITY ENGINEERING Lab

SMAUG-T / NTRU+의 Speed(clean) 벤치마크 (단위: kilo clock cycles)

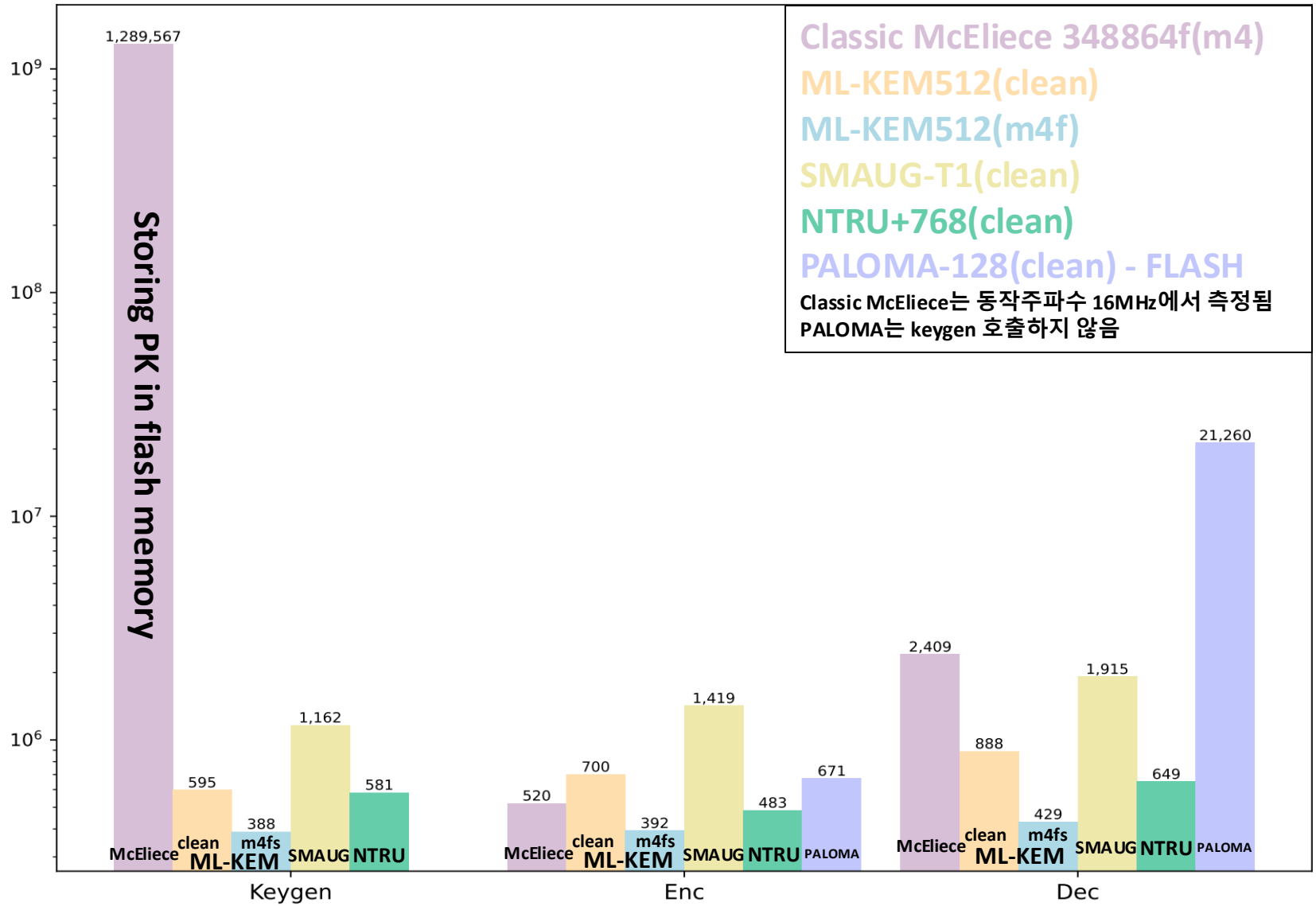




kpqm4 벤치마크 결과 - KEM

CSE CRYPTO & SECURITY ENGINEERING Lab

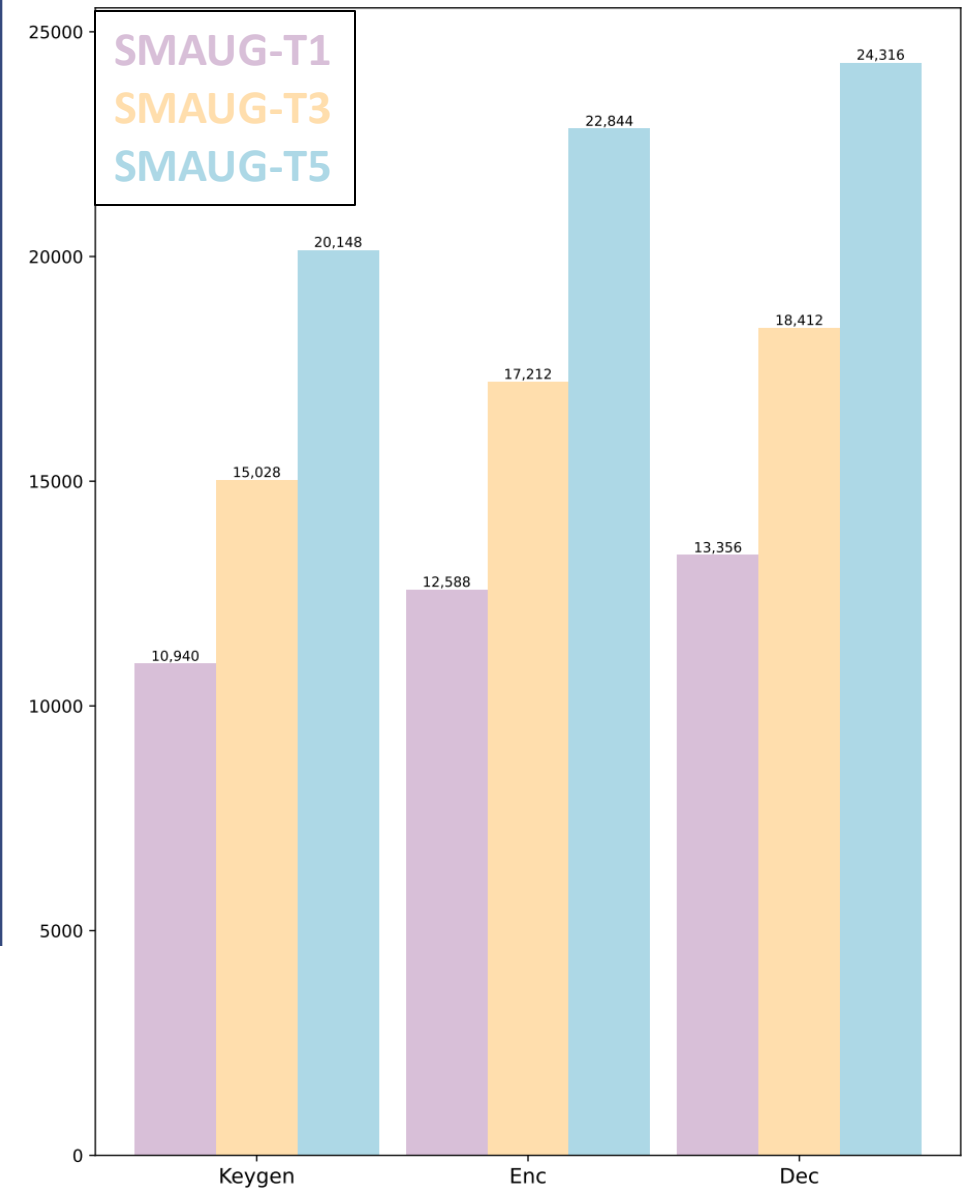
SMAUG-T, NTRU+, PALOMA, 의 Speed(clean) 벤치마크(단위: kilo clock cycles)
ML-KEM, Classic McEliece

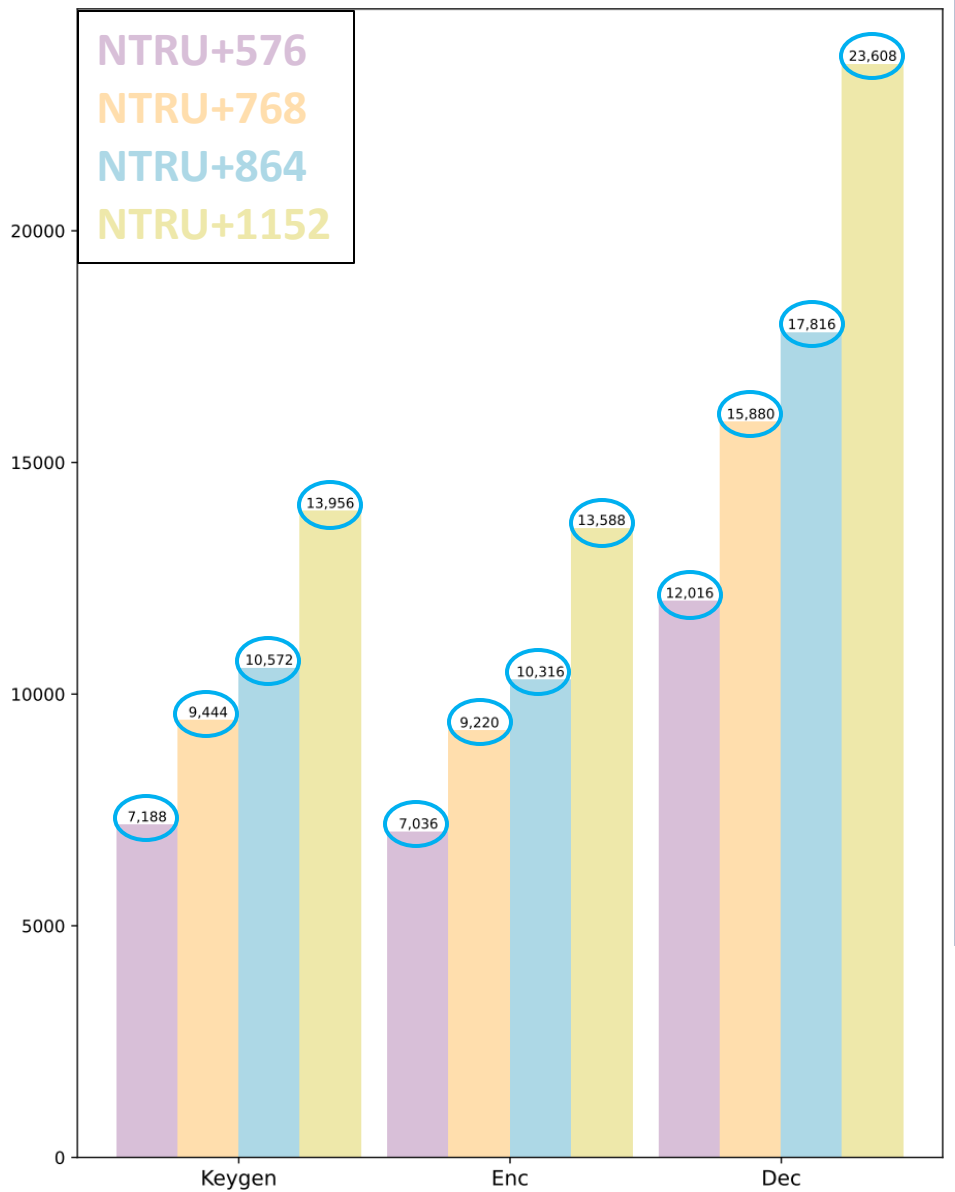


kpqm4 벤치마크 결과 - KEM

CSE CRYPTO & SECURITY ENGINEERING Lab

SMAUG-T / NTRU+의 Stack(clean) 벤치마크(단위: bytes)

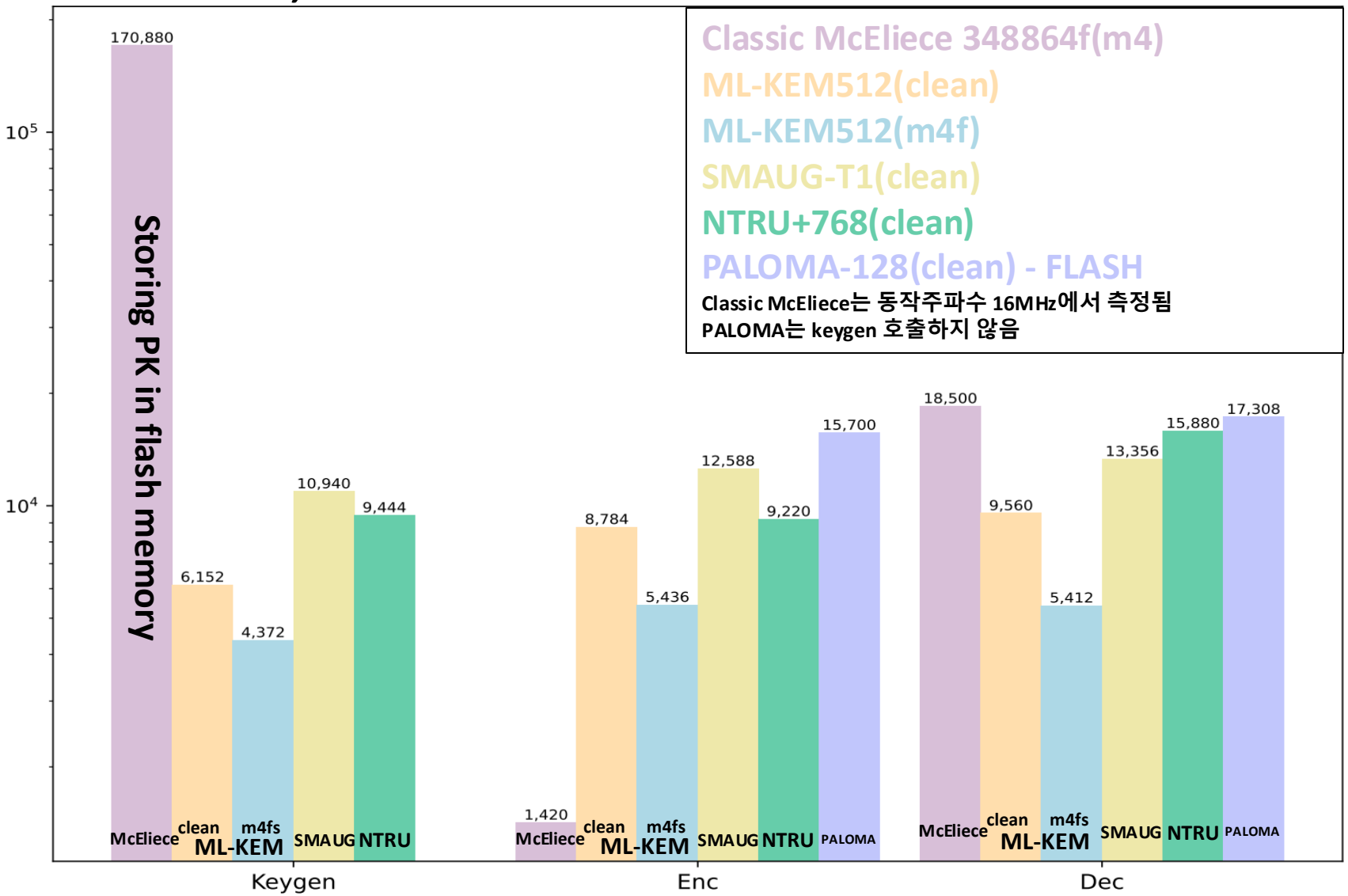




kpqm4 벤치마크 결과 - KEM

CSE CRYPTO & SECURITY ENGINEERING Lab

SMAUG-T, NTRU+, PALOMA, 의 Stack(clean) 벤치마크(단위 : bytes)
ML-KEM, Classic McEliece



kpqm4 벤치마크 결과 – DSA

❖ DSA 성능 벤치마크 대상

- **KpqC Algorithm :** AIMer, HAETAE, NCC-Sign, MQ-Sign
- **Comparison Algorithm :** Falcon, CRYSTALS-Dilithium

특이사항

* **AIMer** : m4speed, m4stack
* **HAETAE** : clean, m4f
* **NCC-Sign** : clean, m4f
* **Dilithium**: clean, m4f (speed)
* **Falcon** : clean, m4-ct
* **MQ-Sign** : clean (pk FLASH)

Category	Step	(high speed)									(low speed)
Speed (Security Level 1)	Key Gen	AIMER	NCC-Sign	Dilithium	NCC-Sign	HAETAE	Dilithium	HAETAE	Falcon	Falcon	
		m4spd/stk	m4f	m4f	clean	m4f	clean	clean	m4-ct	clean	
	Sign	NCC-Sign	Dilithium	HAETAE	NCC-Sign	Dilithium	MQ-Sign	AIMER	Falcon	HAETAE	AIMER	Falcon
		m4f	m4f	m4f	clean	clean	clean	m4speed	m4-ct	clean	m4stack	clean
	Verify	Falcon	Falcon	HAETAE	HAETAE	NCC-Sign	Dilithium	Dilithium	NCC-Sign	MQ-Sign	AIMER
		m4-ct	clean	m4f	clean	m4f	m4f	clean	clean	clean	m4spd/stk

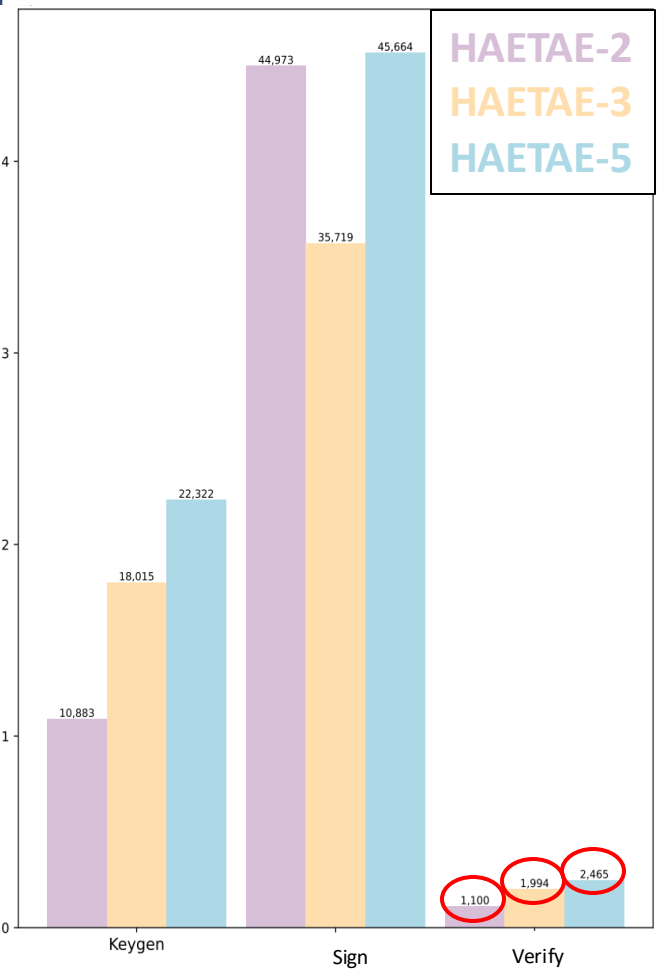
Category	Step	(low stack)									(high stack)
Stack Usage (Security Level 1)	Key Gen	Falcon	AIMER	Falcon	HAETAE	HAETAE	NCC-Sign	NCC-Sign	Dilithium	Dilithium	
		m4-ct	m4spd/stk	clean	m4f	clean	clean	m4f	m4f	clean	
	Sign	Falcon	AIMER	MQ-Sign	Falcon	Dilithium	NCC-Sign	NCC-Sign	Dilithium	HAETAE	HAETAE	AIMER
		m4-ct	m4stack	clean	clean	m4f	clean	m4f	clean	m4f	clean	m4speed
	Verify	Falcon	MQ-Sign	Falcon	Dilithium	AIMER	HAETAE	HAETAE	NCC-Sign	Dilithium	NCC-Sign
		m4-ct	clean	clean	m4f	m4spd/stk	m4f	clean	clean	clean	m4f

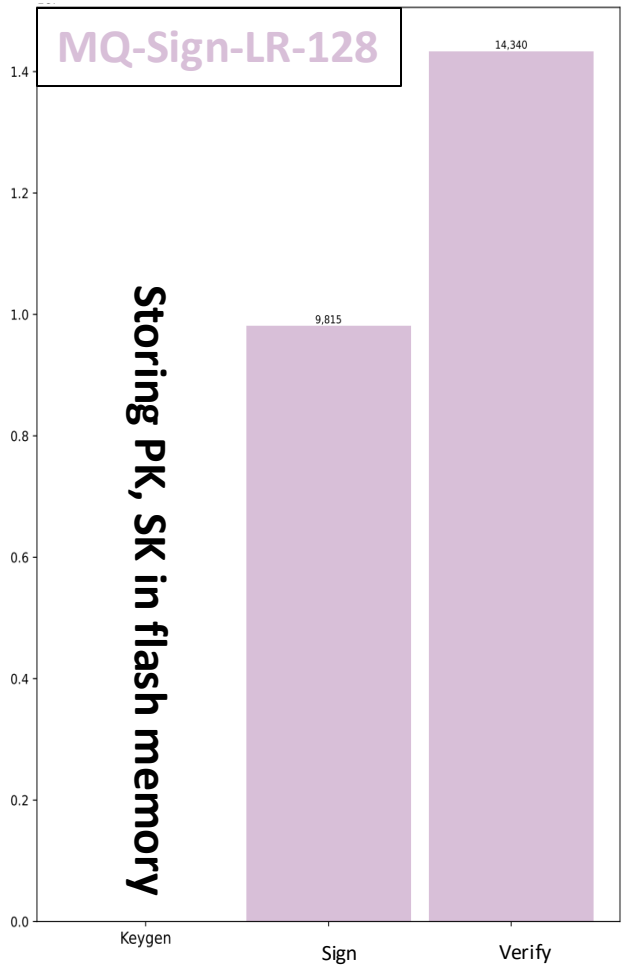
kpqm4 벤치마크 결과 - DSA

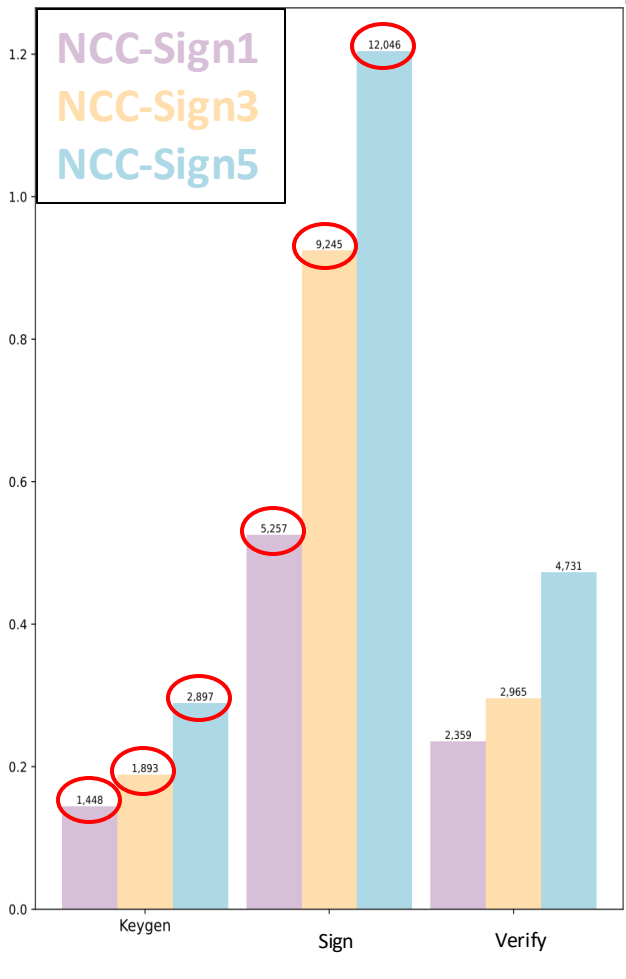
CSE CRYPTO & SECURITY ENGINEERING Lab

HAETAE / MQ-Sign / NCC-Sign의 Speed(clean) 벤치마크

(단위: kilo clock cycles)



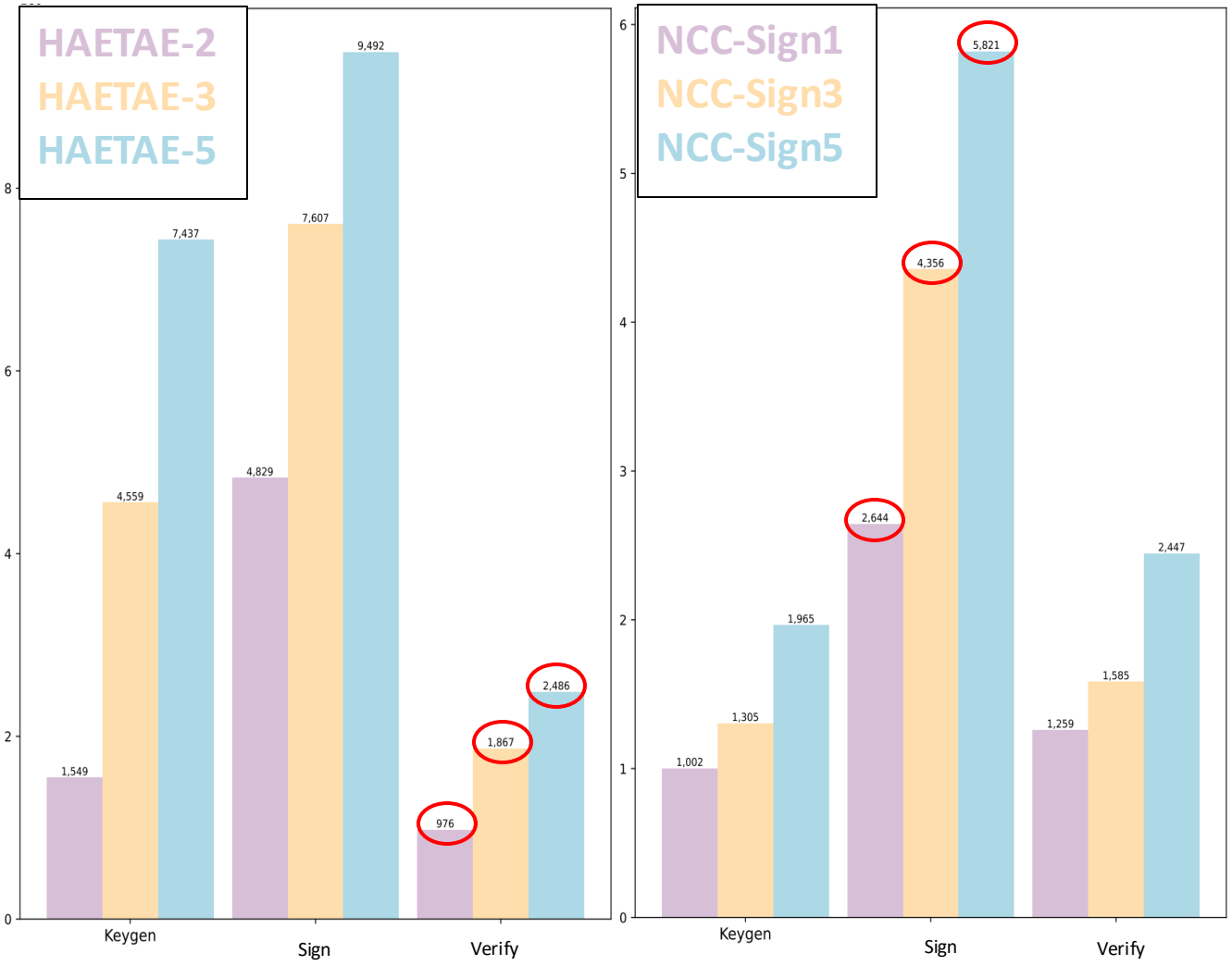




kpqm4 벤치마크 결과 - DSA

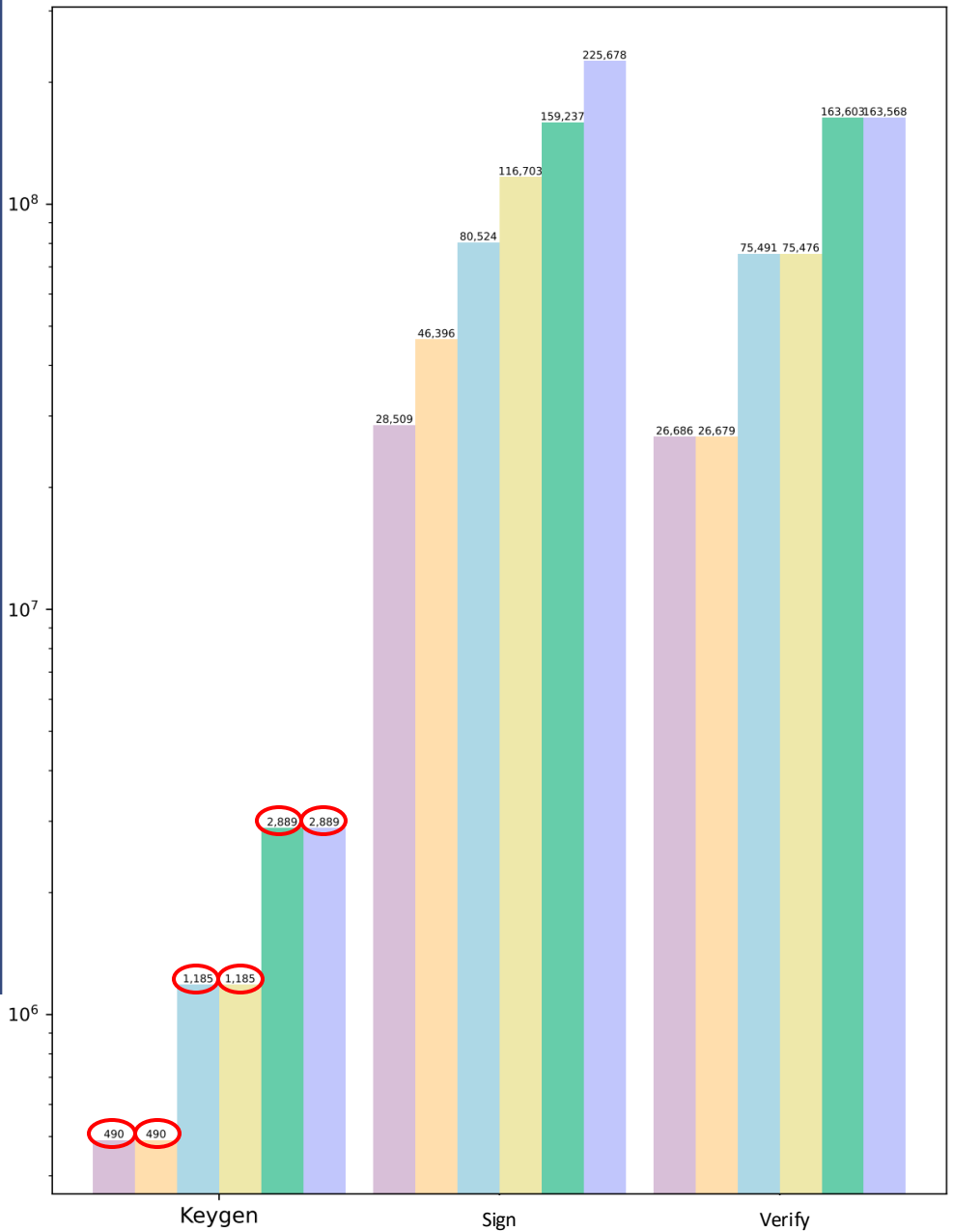
CSE CRYPTO & SECURITY ENGINEERING Lab

HAETAE / NCC-Sign 의 Speed(m4) 벤치마크(단위: kilo clock cycles)



kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab



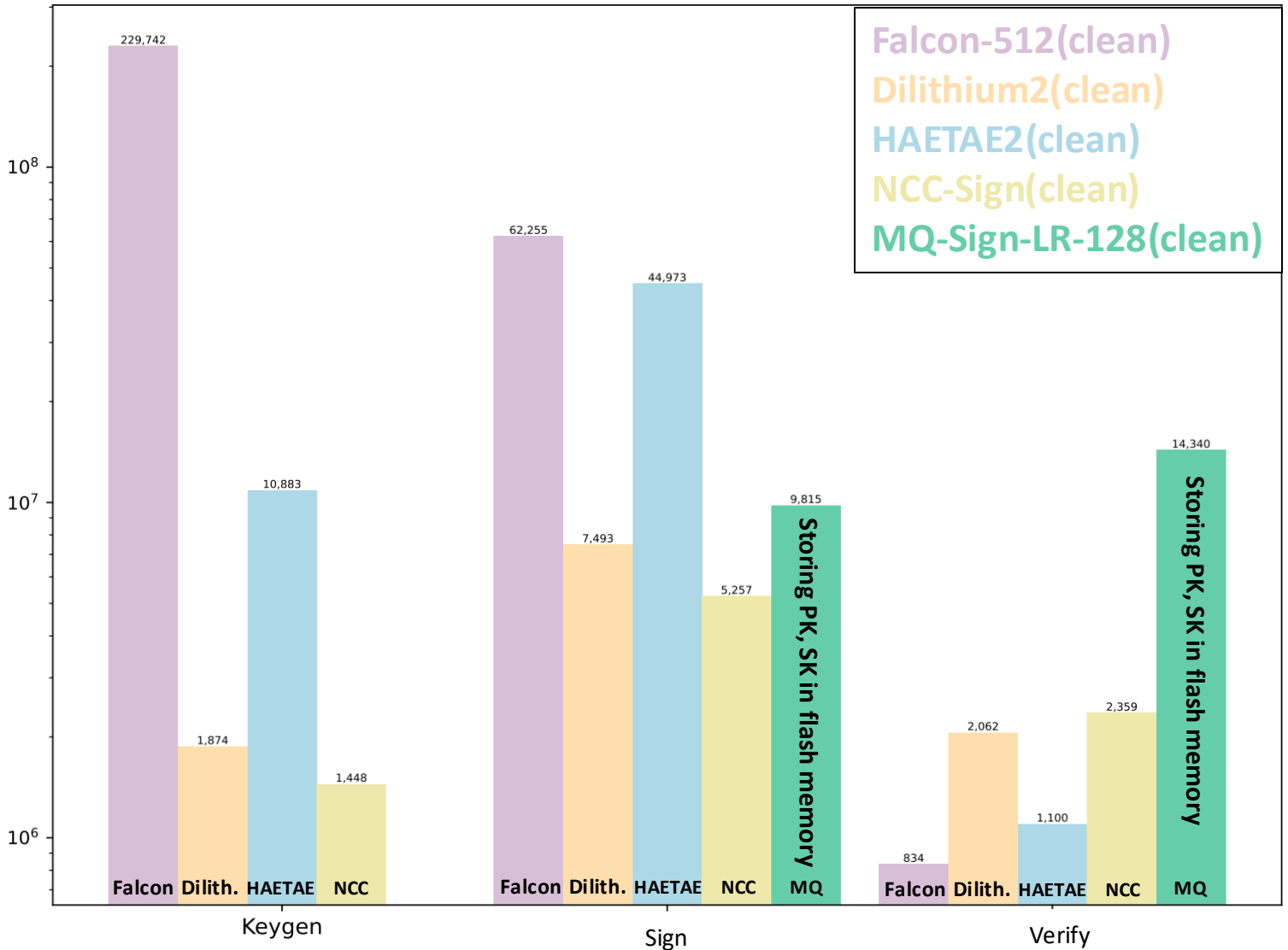
AIMer의 Speed(m4) 벤치마크

(단위: kilo clock cycles)

kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab

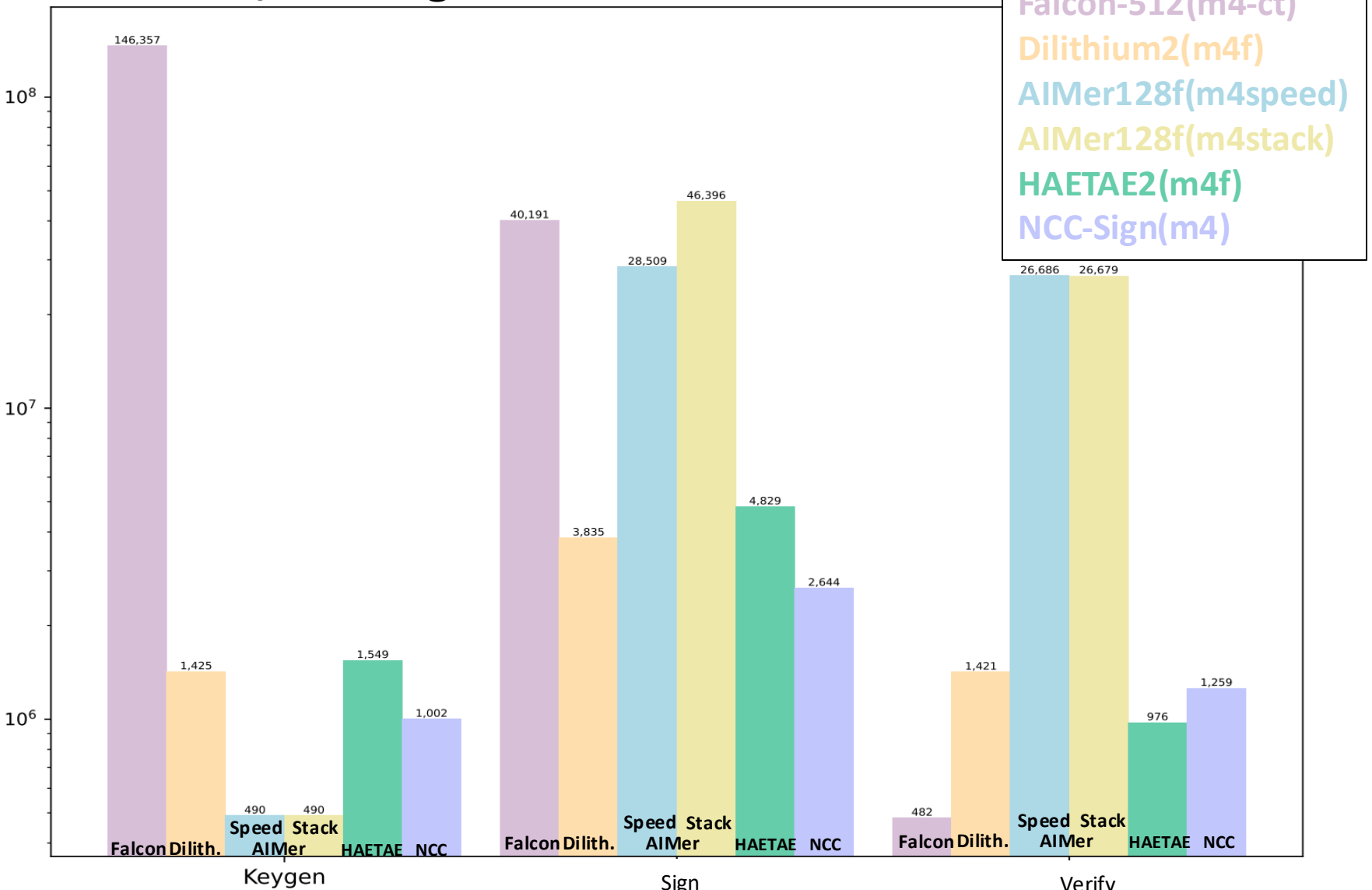
Falcon / Dilithium
HAETAE / NCC-Sign / MQ-Sign 의 Speed(clean) 벤치마크 (단위: kilo clock cycles)



kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab

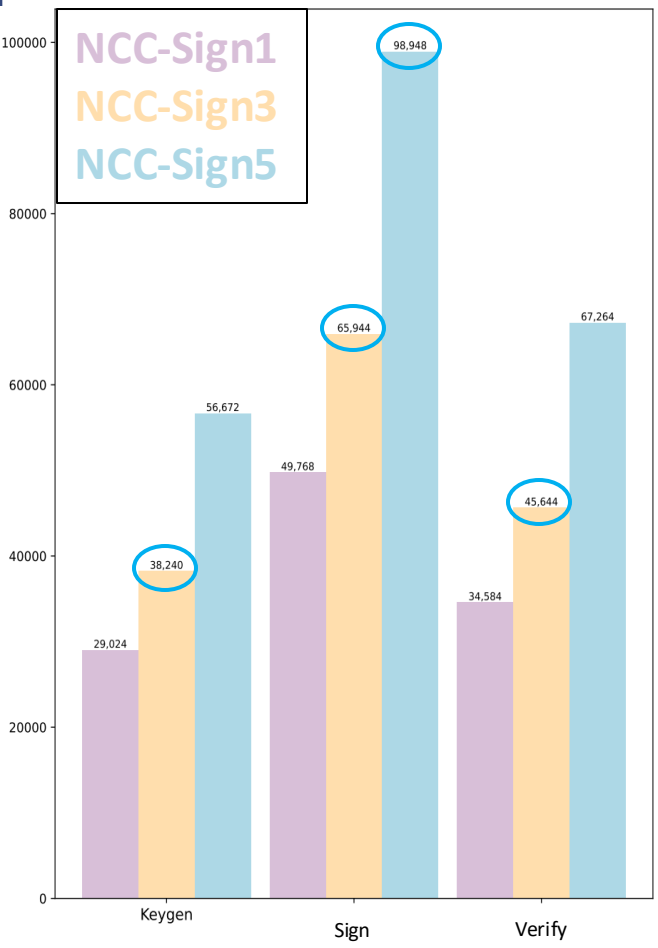
Falcon / Dilithium / AIMer HAETAE / NCC-Sign 의 Speed(m4f) 벤치마크(단위: kilo clock cycles)

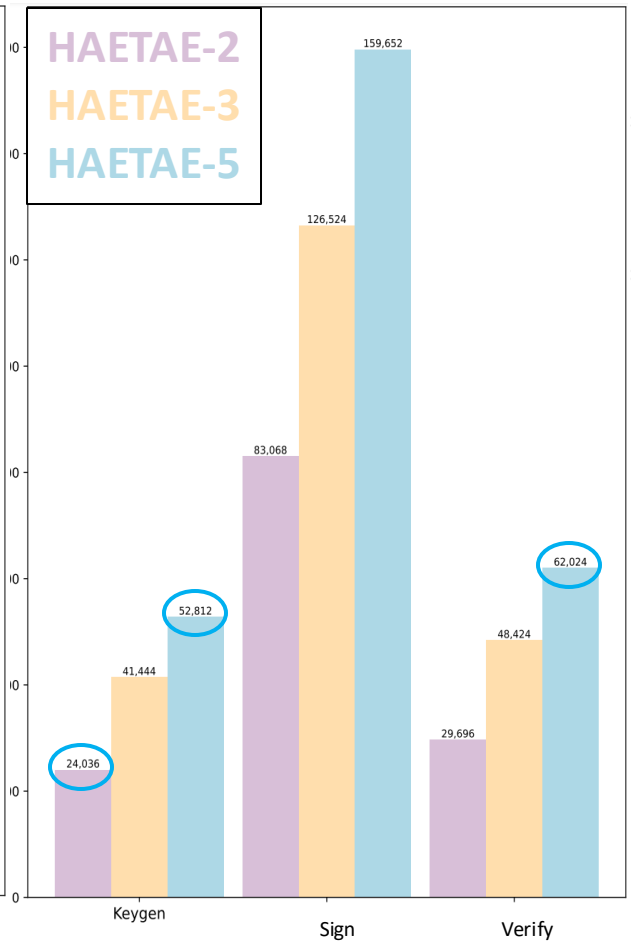


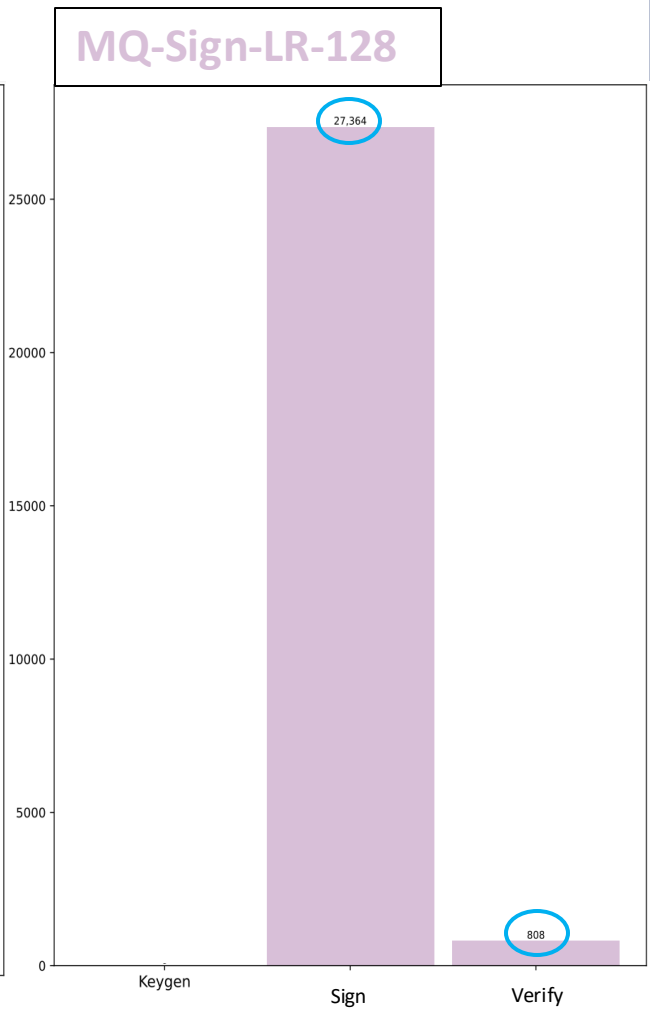
kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab

HAETAE / MQ-Sign / NCC-Sign 의 Stack(clean) 벤치마크(단위: bytes)



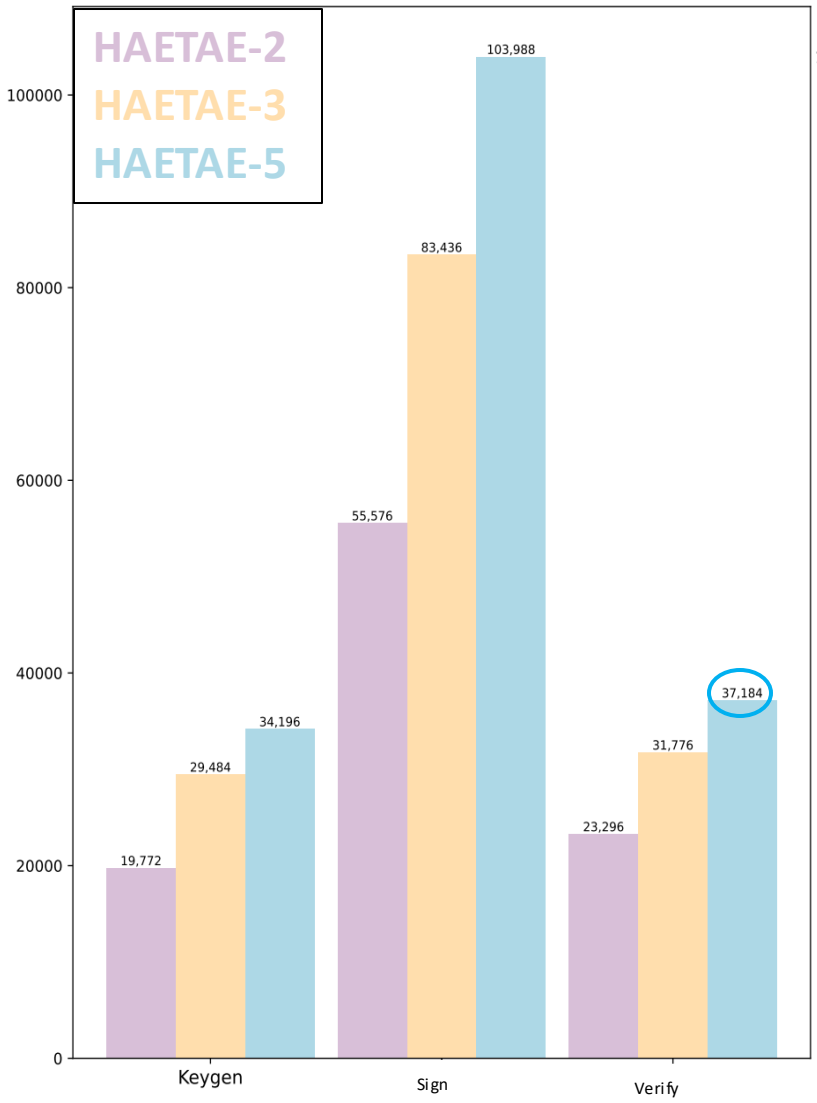


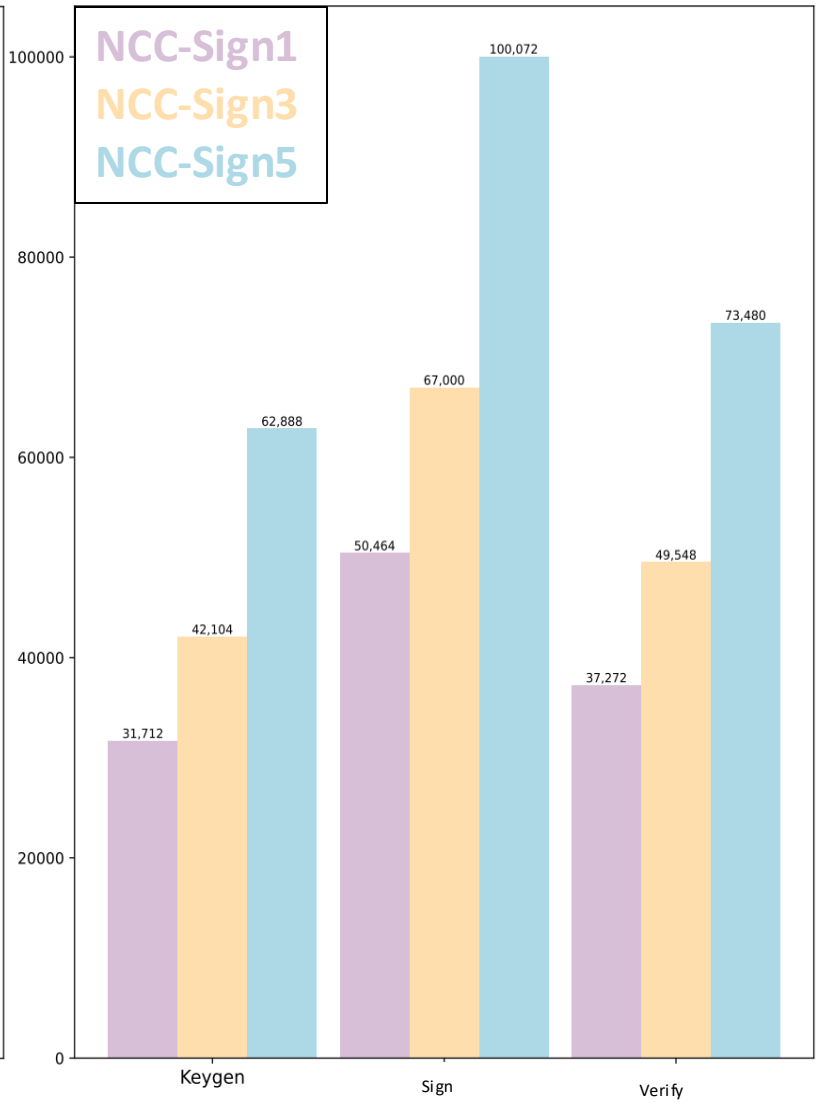


kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab

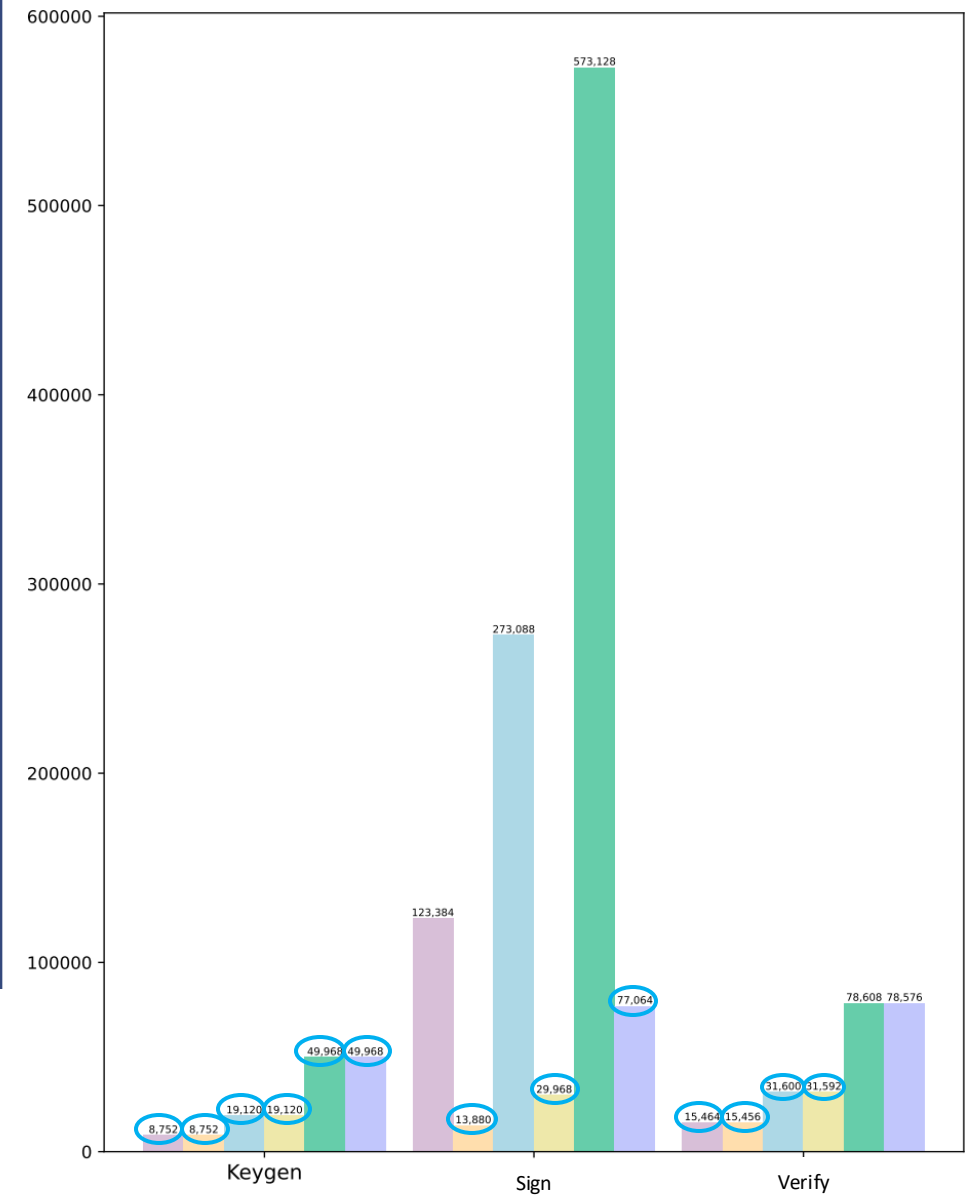
HAETAE / NCC-Sign의 Stack(m4) 벤치마크(단위:bytes)





kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab



AIMer128f speed
AIMer128f stack
AIMer192f speed
AIMer192f stack
AIMer256f speed
AIMer256f stack

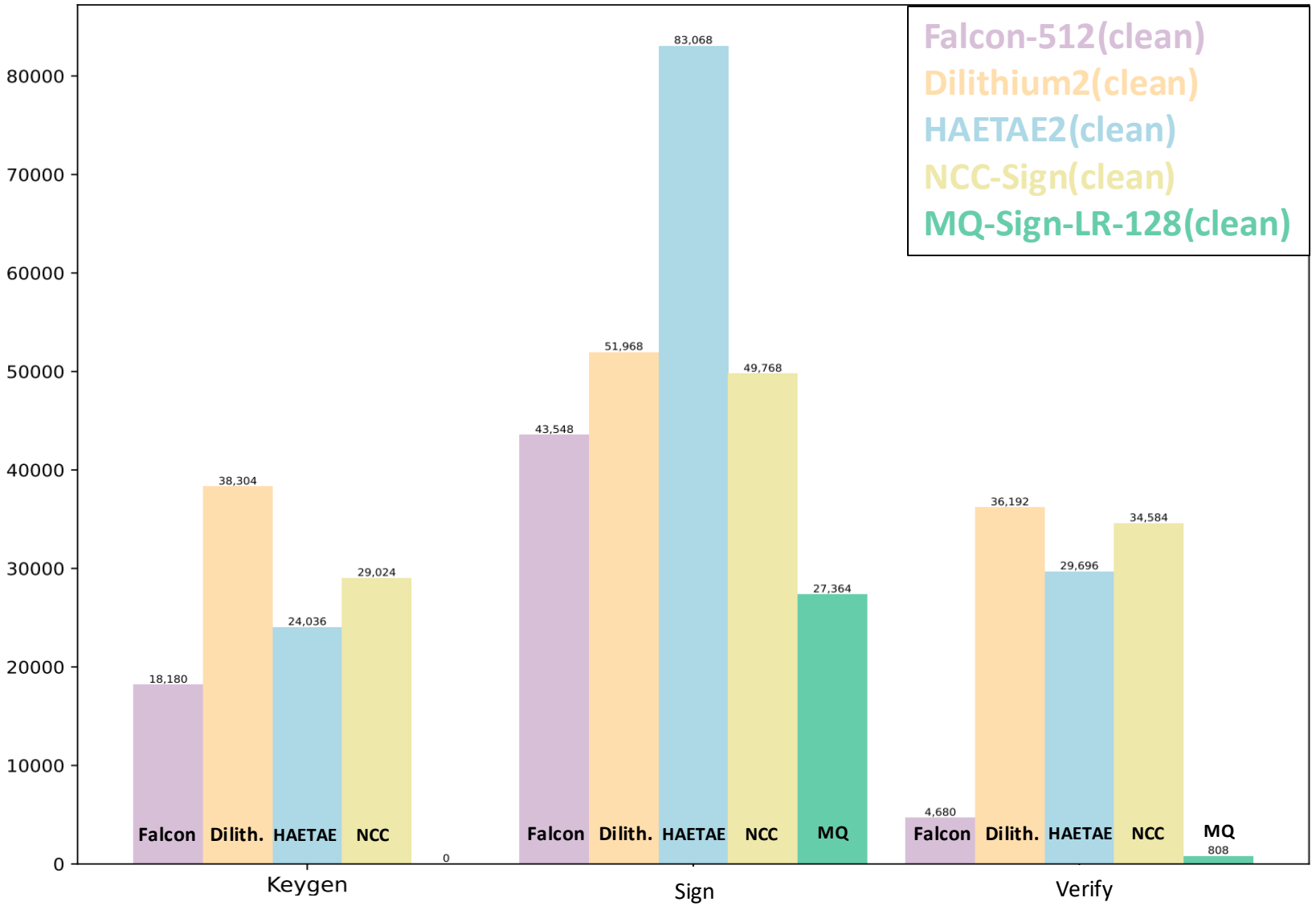
AIMer의 Stack(m4) 벤치마크

(단위: bytes)

kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab

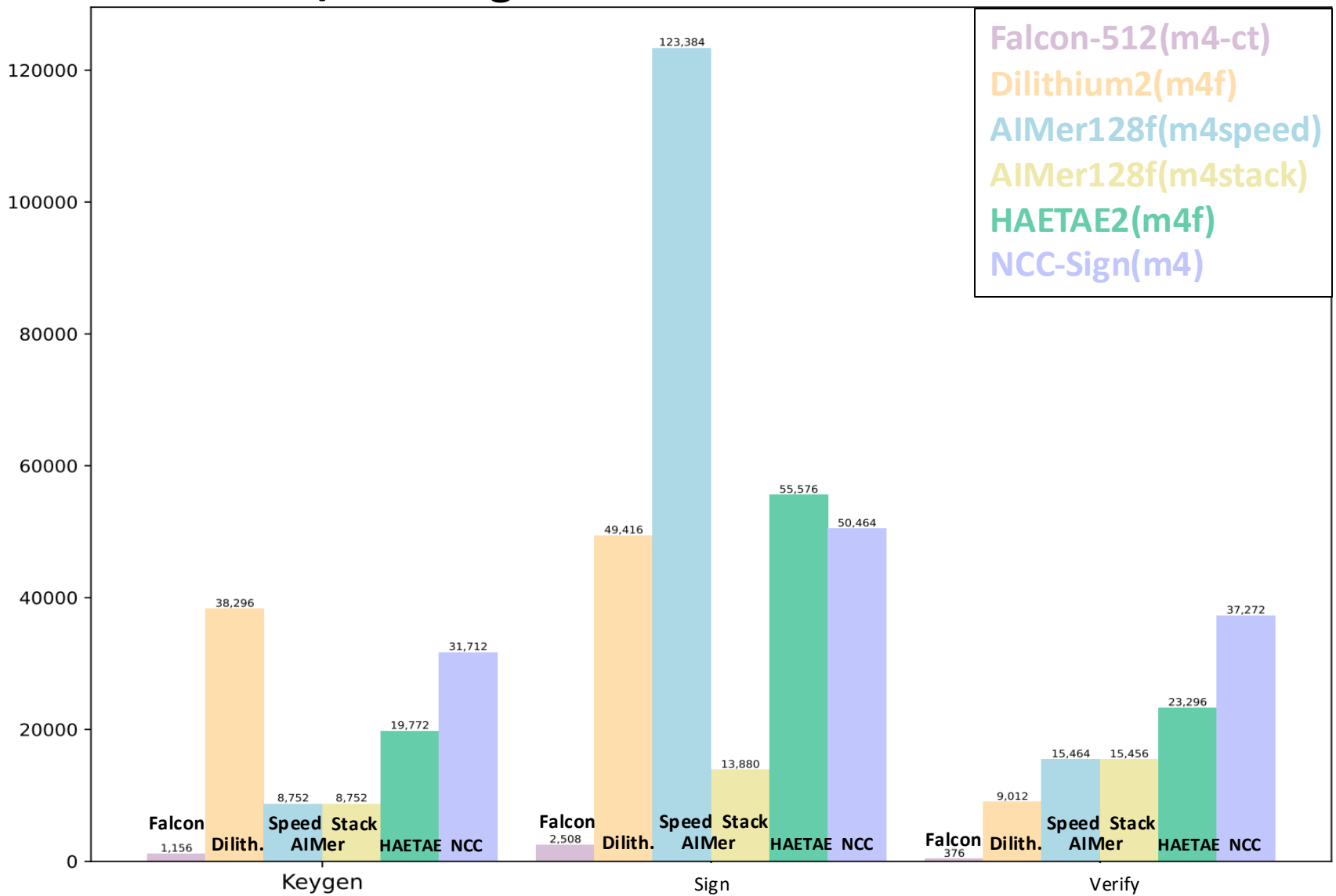
Falcon / Dilithium
HAETAE / NCC-Sign / MQ-Sign 의 Stack(clean) 벤치마크(단위: bytes)



kpqm4 벤치마크 결과 - DSA

CSE CRYPTO & SECURITY ENGINEERING Lab

Falcon / Dilithium / AIMer HAETAE / NCC-Sign 의 Stack(m4f) 벤치마크(단위: bytes)



알고리즘별 총평 - KEM

NTRU+

❖ NTRU+

- KpqC Round 2 KEM 알고리즘 중 **가장 속도가 빠르고, 적은 Stack 사용**
- **ML-KEM과 비교하여 경쟁력이 있음**
  - ✓ Cortex-M4에서 Reference 구현은 NTRU+가 더 빠름
  - ✓ Cortex-M4용 최적화 코드 개발이 필요
    - 최적화가 완료되면, ML-KEM의 m4f 구현과 경쟁력이 있을 것으로 기대
- 핵심 연산 **다항식 곱셈 (NTT)에 대한 구현 최적화**가 가능
  - ✓ Trinomial NTT (radix-3, radix-2)에 대한 Merging 기법 적용 가능
  - ✓ Cortex-M4 Assembly를 활용한 효율적인 Modular Arithmetic 활용가능
    - Improved Plantard Arithmetic for Lattice-based Cryptography, TCHES 2022
    - Faster Kyber and Dilithium on the Cortex-M4, ACNS 2022

알고리즘별 총평 - KEM

❖ SMAUG-T

- ML-KEM 및 Saber와 동일한 Module LWE/LWR구조
- Saber의 구현 최적화 방법론은 아직 모두 적용되지 않음

- 구현 최적화 가능성이 존재
  - ✓ **Ref 구현에서 NTT를 사용 해야함**
    - (TC4 + Kara)가 단일 다항식 곱셈에서는 빠를 수 있으나, 행렬 곱셈 관점에서는 NTT가 우세
    - x86/64(i7-13700K)에서 해당 방법론 적용 후 속도향상을 확인

	SMAUG-T 1 (TC4 → NTT)	SMAUG-T 3 (TC4 → NTT)	SMAUG-T 5 (TC4 → NTT)
Keypair	56,163 → **53,612**	109,267 → **99,121**	187,642 → **151,171**
Encaps	52,920 → **48,965**	110,780 → **89,725**	175,636 → **139,113**
Decaps	74,466 → **68,703**	132,623 → **116,376**	217,519 → **173,478**

- **Intel® Core™ i7-13700K + ~~Hyper-Threading~~ + ~~Turbo~~ (2022~)**
- **Median of 10,000 trying**

알고리즘별 총평 - KEM

❖ **SMAUG-T**

- ML-KEM 및 Saber와 동일한 Module LWE/LWR구조
- ML-KEM 및 Saber의 구현 최적화 방법론은 아직 적용되지 않음

- 구현 최적화 가능성이 존재
  - ✓ **Ref 구현에서 NTT를 사용 해야함**
    - TC4가 단일 다항식 곱셈에서는 빠를 수 있으나, 행렬 곱셈 관점에서는 NTT가 우세
    - x86/64(i7-13700K)에서 해당 방법론 적용 후 속도향상을 확인
  - ✓ **M4에서 ML-KEM 및 Saber가 사용한 구현 최적화 방법론을 적용 해야함**
    - **Better Accumulation** : Faster Kyber and Dilithium on the Cortex-M4, ACNS 2022
    - **Asymmetric Multiplication** : Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1, TCHES 2022
    - **multimodule NTT, MatrixMul Strategy** : Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4, TCHES 2022

알고리즘별 총평 - KEM

CSE CRYPTO & SECURITY ENGINEERING Lab

❖ PALOMA

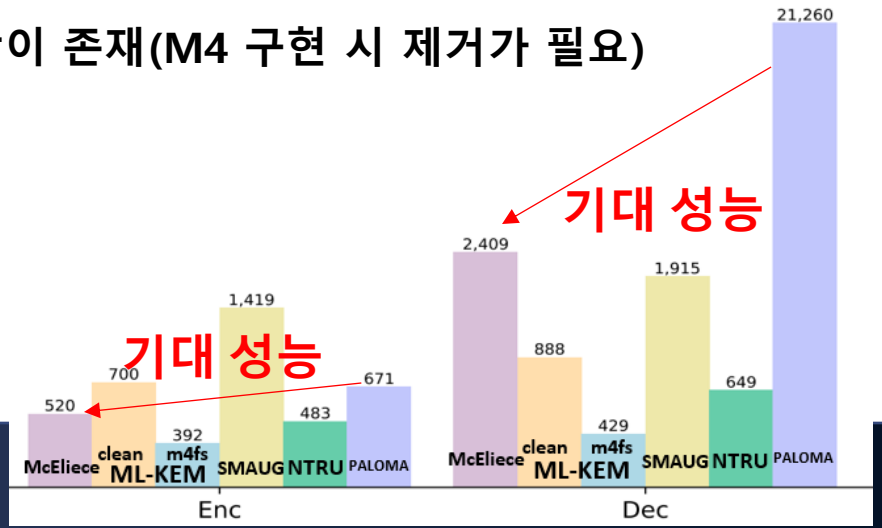
- ➢ pk in Flash Memory 및 KeyGen Process 가속화
  - ✓ Classic McEliece Implementation with Low Memory Footprint, CARDIS 2020
  - ✓ Classic McEliece on the ARM Cortex-M4, TCHES 2021
  - ✓ LU 분해, 곱셈 방법론 등을 적용가능

- ➢ 곱셈 테이블을 사용하지 않는 방법론을 고려 해야함
  - ✓ 부채널 분석을 고려하여, LUT를 사용하지 않는 곱셈 방법론 적용이 필요
    - https://github.com/pqcryptotw/mceliece-arm-m4

- ➢ 활발한 코드 업데이트 중 → 동적메모리 할당이 존재(M4 구현 시 제거가 필요)

Table 4.5: Comparison between PALOMA and Classic McEliece in Plat. 2 (in milliseconds(cycles))

		GenKeyPair	Encap	Decap
128-bit	PALOMA-128	27.81 (1,021,606)	0.042 (1,014)	1.75 (40,848)
	mceliece348864f	28.24 (651,804)	0.038 (940)	15.03 (353,312)
192-bit	PALOMA-192	119.77 (2,939,786)	0.051 (1,279)	8.97 (212,087)
	mceliece460896f	79.67 (1,922,618)	0.073 (1,817)	37.16 (898,408)
256-bit	PALOMA-256	150.03 (3,665,877)	0.060 (1,368)	9.43 (225,231)
	mceliece6688128f	139.61 (3,331,050)	0.124 (2,920)	71.94 (1,726,978)



알고리즘별 총평 - DSA

❖ HAETAE

HAETAE
HEAAN CRYPTO LAB

- HAETAE: Shorter Lattice-Based Fiat-Shamir Signatures, TCHES 2024
- **Cortex-M4의 고도의 최적화 구현이 존재**
  - ✓ pqm4에 통합됨 → kpqm4는 그대로 이용
  - ✓ 짧은 서명 및 검증 키 크기
  - ✓ 밸런스 있는 성능
  - ✓ 스택 최적화 구현이 없어 추가 개발 가능성 존재

❖ AIMer

- mupq에 통합되었음
  - ✓ 메모리 최적화 구현 제공
- **매우 빠른 keygen**
- 최근 Cortex-M4 Assembly 최적화 구현이 통합되었음

알고리즘별 총평 - DSA

❖ **NCC-Sign**

- ➢ **Cortex-M4의 고도의 최적화 구현이 존재**
  - ✓ https://github.com/NIMS-Cryptography/NCC-Sign-MQ-Sign
  - ✓ 모든 보안레벨에서 ML-DSA 보다 높은 성능
- ➢ Stack 구현 최적화가 필요
  - ✓ Clean 코드는 ML-DSA보다 적은 stack을 사용
  - ✓ ML-DSA stack 최적화 코드가 NCC-Sign m4f 코드보다 더 적은 stack을 사용

❖ **MQ-Sign**

- ➢ 현재 NUCLEO 보드에서 keygen 불가능
  - ✓ kpqm4는 flash-memory에 pk, sk를 저장하고 Sign, Verify를 수행
- ➢ Sign, Verify에서 낮은 stack 사용량

결론

❖ ARM Cortex-M4 기반 임베디드 시스템용 벤치마크 프레임워크 개발

- ➢ KpqC Round 2 알고리즘 대상 모든 최신코드 업데이트
- ➢ NUCELO-L4R5ZI 보드에서의 벤치마킹 코드 제공

Release Code: https://github.com/COALA-5/kpqm4

❖향후 업데이트 계획

- ➢ 업데이트 되는 KpqC Round 2 알고리즘 코드 최신화
- ➢ 더욱 상세한 벤치마킹 결과 제공
  - ✓ 최대/최소 및 파라미터 크기 등

KOOKMIN UNIVERSITY

Q & A

(scseo;darania) @kookmin.ac.kr

CSE CRYPTO & SECURITY ENGINEERING Lab

KOOKMIN UNIVERSITY

양자내성암호연구단

Appendix

1 : Lattice/Code 동향

2 : pqm4 project

3 : kpqm4 포팅 이슈사항 해결 방안

CSE CRYPTO & SECURITY ENGINEERING Lab

Appendix – Lattice 동향

❖ **(성능 : Mod_Mul) 감산 방법 최적화 (32-bit Montgomery 곱셈)**

- 32-bit Cortex-M4의 Assembly 명령어를 활용
- Dilithium에는 하기와 같은 Montgomery 곱셈 구현에 대한 최적화가 적용됨
  - ✓ **HAETAE 및 NCC-Sign의 m4f 구현에 해당 최적화 방안이 적용되어 있음**
  - ✓ **SMAUG-T가 NTT를 적용하는 경우 32-bit Montgomery 최적화 적용가능**

32-bit Signed Montgomery multiplication ($R = 2^{32}$)

```
smull a0, a1, a, b      // 곱셈
mul   tmp, a0, Qinv     → t = (int64_t)(int32_t)a*QINV;
smlal a0, a1, tmp, Q    → t = (a - (int64_t)t*Q) >> 32;
(Result in a1)
```

```
int32_t montgomery_reduce(int64_t a) {
  int32_t t;

  t = (int64_t)(int32_t)a*QINV;
  t = (a - (int64_t)t*Q) >> 32;
  return t;
}
```

$aR^{-1} \bmod Q$

Cortex-M4 optimizations for {R, M}LWE schemes, TCHES 2020
Compact Dilithium Implementation on Cortex-M4 and Cortex-M4, TCHES 2021

❖ (성능 : Mod_Mul) 감산 방법 최적화 (16-bit Montgomery 곱셈)

- Kyber의 ref는 Montgomery 곱셈을 사용
  - ✓ **NTRU+의 ref도 Montgomery 곱셈을 사용**
- 초기에 M4에서 Kyber의 Montgomery 곱셈에 대한 구현 최적화가 진행 됨
- 총 3개의 곱셈 명령어로 구현 가능

16-bit Signed Montgomery multiplication ($R = 2^{16}$)

```
smulbb  t, a, b          // 곱셈
smulbb  tmp, t, Qinv     → t = (int16_t)a*QINV;
smlabb  tmp, tmp, Q, t   → t = (a - (int32_t)t*KYBER_Q)
asr  tmp, tmp, #16       → t = (a - (int32_t)t*KYBER_Q) >> 16;
```

(**asr** can be eliminated by merging with packing **pkhbt**)

```
int16_t montgomery_reduce(int32_t a)
{
  int16_t t;

  t = (int16_t)a*QINV;
  t = (a - (int32_t)t*KYBER_Q) >> 16;
  return t;
}
```

Cortex-M4 optimizations for {R, M}LWE schemes, TCHES 2020
Compact Dilithium Implementation on Cortex-M4 and Cortex-M4, TCHES 2021

❖ (성능 : Mod_Mul) 감산 방법 최적화 (16-bit Plantard 곱셈)

➢ 2021 TCHES에서 16-bit Mod_Mul에 대한 구현 최적화 방안이 제안됨

✓ Montgomery 곱셈 3 cycles → Plantard 곱셈 2 cycles (Kyber에 적용됨)

- *l*×*2l* 곱셈 명령어가 활용 가능한 경우에만 적합 (예: 16×32 곱셈 연산)

✓ **NTRU+의 Cortex-M4 구현에서 Plantard 곱셈을 적용가능**

✓ **SMAUG-T가 Multi-moduli NTT를 사용하면 Plantard 곱셈 적용가능**

Algorithm The 2-cycle improved Plantard multiplication by a constant on Cortex-M4

Input: An l-bit signed integer $a \in [-2^{l-1}, 2^{l-1})$, a precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r_{top} = ab(-2^{-2l}) \bmod^{\pm} q, r_{top} \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **smulwb** r, bq', a ▷ $r \leftarrow [[abq']_{2l}]^l$
3: **smlabb** $r, r, q, q2^{\alpha}$ ▷ $r_{\text{top}} \leftarrow [q[r]_l + q2^{\alpha}]^l$
4: **return** r_{top}

Algorithm The 3-cycle signed Montgomery multiplication on Cortex-M4 [ABCG20]

Input: Two l-bit signed integers a, b such that $ab \in [-q2^{l-1}, q2^{l-1})$

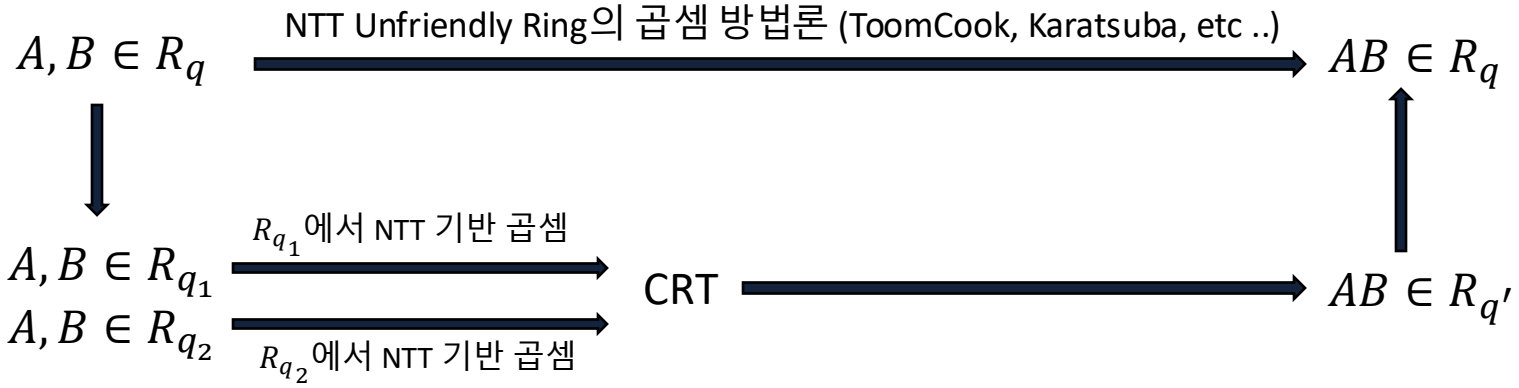
Output: $r_{top} = ab2^{-l} \bmod^{\pm} q, r_{top} \in (-q, q)$

1: **mul** c, a, b
2: **smulbb** $r, c, -q^{-1}$ ▷ $r \leftarrow [c]_l \cdot (-q^{-1})$
3: **smlabb** r, r, q, c ▷ $r_{\text{top}} \leftarrow [[r]_l \cdot q]^l + [c]^l$
4: **return** r_{top}

Improved Plantard Arithmetic for Lattice-based Cryptography, TCHES 2021

❖ (성능 : 다항식 곱셈) NTT 적용 및 최적화

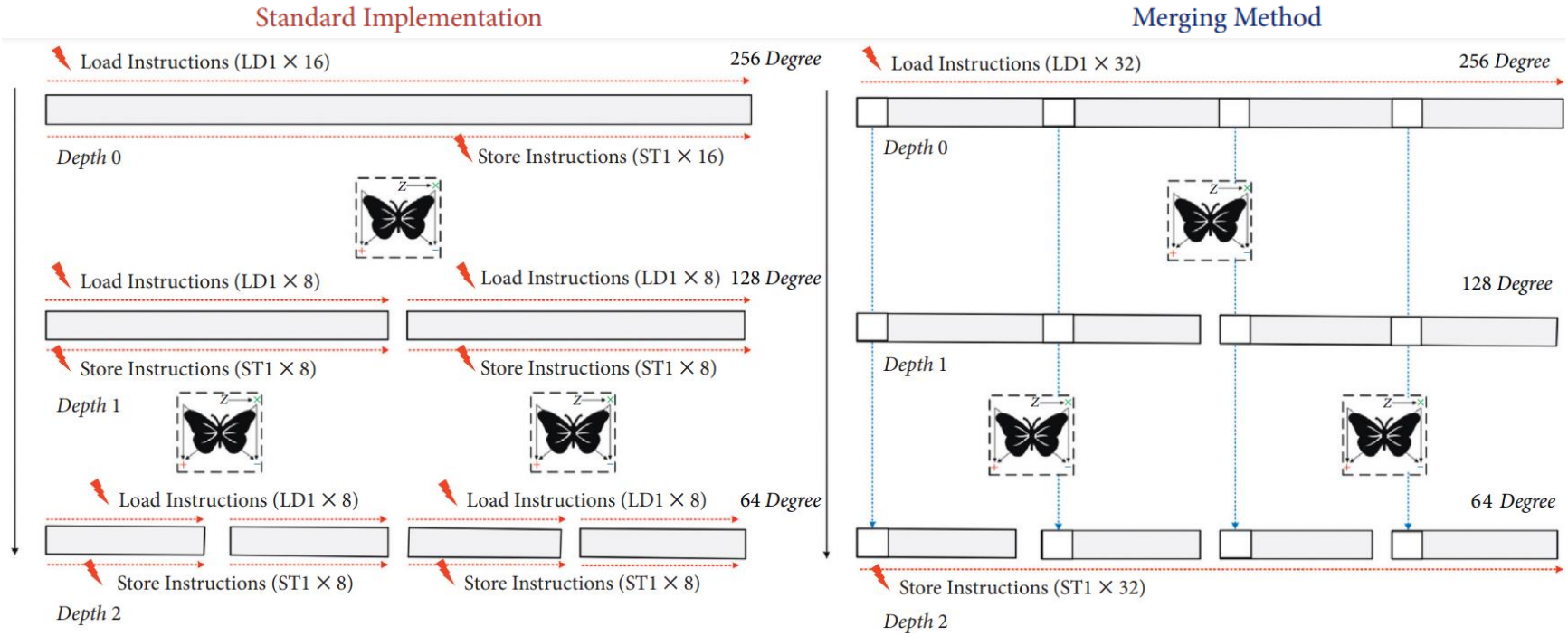
- ➢ NTT Unfriendly한 다항링 R_q에 대하여 NTT를 적용할 수 있는 방법론
  - ✓ 기존 알고리즘의 Matrix-Vector 곱셈에서의 최댓값 m을 계산
  - ✓ $q' = q_1 q_2 > m$를 만족하는 NTT friendly q_1, q_2에 대하여 $modulus\ q_1, q_2$에서의 NTT/iNTT 를 적용할 수 있음 → **SMAUG-T, NCC-Sign(NC)에 적용가능**
    - $\begin{cases} modulus\ q_1 \colon NTT/iNTT \\ modulus\ q_2 \colon NTT/iNTT \end{cases} \rightarrow CRT$를 통해 $modulus\ q_1 q_2 \rightarrow modulus\ q$



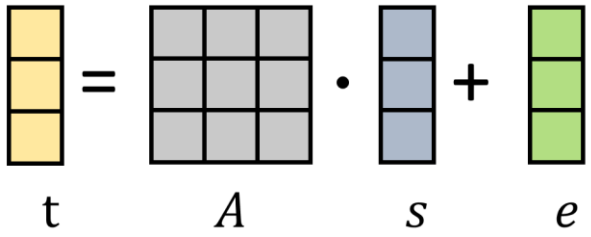
NTT Multiplication for NTT-unfriendly Rings, TCHES 2021
Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4, TCHES 2022

❖ **(성능 : 다항식 곱셈) NTT Layer Merging**

- ➢ 제한된 레지스터 공간을 효율적으로 이용하기 위해 NTT 연산 계층을 통합함
- ➢ 연산결과가 의존적인 부분을 식별하여, 해당 부분들을 레지스터에 로드
- ➢ **HAETAE, NCC-Sign의 m4f에 해당 부분이 구현되어 있음**
- ➢ **NTRU+, SMAUG-T(NTT 사용하는 경우) Cortex-M4 구현에 적용 가능**



❖ (성능 : 행렬 곱셈) Asymmetric Multiplication



➢ Kyber의 $\boldsymbol{A}, \boldsymbol{s}$ 간의 PointMul 계산 예시

✓ for any element in $Z_q[x]/< X^2 - \zeta^{2i+1} >$

$$(a_0 + a_1 x) \cdot (s_0 + s_1 x) = \left(a_0 s_0 + a_1 s_1 \zeta^{2i+1}\right) + (a_0 s_1 + a_1 s_0)x$$

✓ 비밀 벡터 s는 행렬 곱셈에서 반복적으로 사용됨

✓ $s_1 \zeta^{2i+1}$ 을 미리 계산

✓ (k - 1) x k x 128 번 몽고메리 곱셈 절약

✓ k x 16 x 128 / 8 추가스택 필요

```
void basemul
(int16_t r[2], int16_t a[2], int16_t s[2], int16_t zeta)
{
  r[0]  = montgomery_reduce(s[1] * zeta); // 생략가능
  r[0]  = montgomery_reduce(r[0] * a[1]);
  r[0] += montgomery_reduce(a[0] * s[0]);
  r[1]  = montgomery_reduce(a[0] * s[1]);
  r[1] += montgomery_reduce(a[1] * s[0]);
}
```

✓ SMAUG-T가 incomplete NTT를 사용하는 경우에 적용가능

❖ (성능 : 행렬 곱셈) Better Accumulation

➢ Kyber에서 $A \cdot s$ 계산 시 몽고메리 곱셈을 사용하지 않고 32-bit로 누적

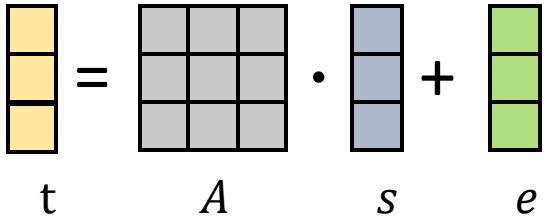
✓ (k - 1) x (k - 1) x 256 번의 몽고메리 곱셈 절약

✓단, 32 x 256/8 bytes의 추가 스택 필요

✓ **HAETAE, SMAUG-T에 적용가능**

```
[Reference]
t0 += A00 x s0 : montmul(16-bit x 16-bit) = 16-bit acc
t0 += A01 x s1 : montmul(16-bit x 16-bit) = 16-bit acc
t0 += A02 x s2 : montmul(16-bit x 16-bit) = 16-bit acc
barrett_reduce(t0)


[Better Accumulation]
t0 += A00 x s0 : 16-bit x 16-bit = 32-bit acc
t0 += A01 x s1 : 16-bit x 16-bit = 32-bit acc
t0 += A02 x s2 : 16-bit x 16-bit = 32-bit acc
montgomery_reduce(t0)
barrett_reduce(t0)
```



// A_{ij} : Rejection Sampling

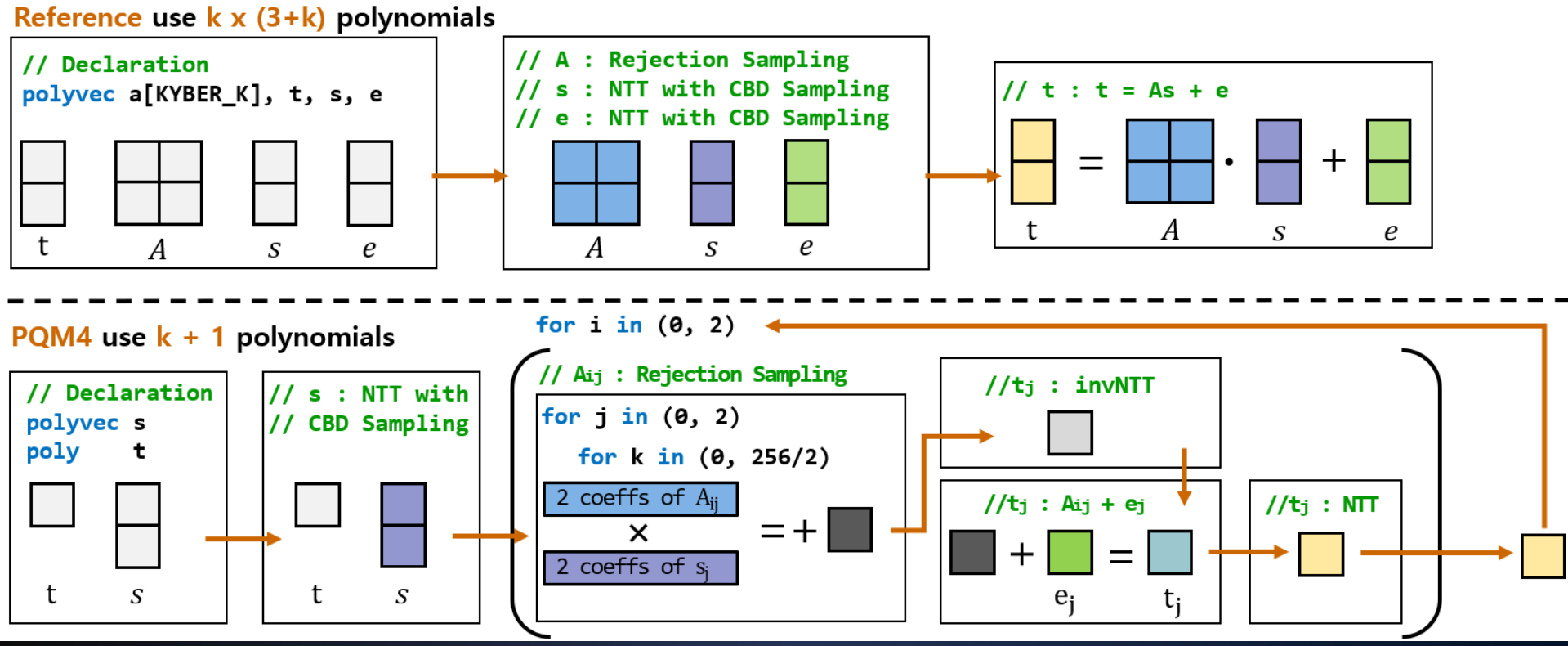
```
for j in (0, 2)
   for k in (0, 256/2)
```

2 coeffs of A_{ij} × 2 coeffs of s_j =+ ■

Fig : Better Accumulation presented in Kyber

❖ (메모리) stack 최적화 방안

- ➢ Kyber의 공개행렬 A 및 비밀벡터 s, e에 대한 메모리 footprint 감소
- ➢ 공개행렬 A를 즉석에서 생성 후 $s' = \mathrm{NTT}(s)$와 곱셈 후 누적
- ➢ **HAETAE, SMAUG-T의 구현에 적용 가능**



❖ 국외 Cortex-M4 Code 기반 암호 구현 최적화 연구 동향

- Classic McEliece와 Bike에 대한 Cortex-M4 구현 최적화 연구 존재
- HQC는 구현 최적화 연구 없음 → (국내) 최초 최적화 연구 수행

Classic-McEliece

- Classic McEliece Implementation with Low Memory Footprint, CARDIS 2020
- Classic McEliece on the ARM Cortex-M, TCHES 2021

BIKE

- Optimizing BIKE for the Intel Haswell and ARM Cortex-M4, TCHES 2021
- Carry-less to bike faster ACNS 2022

HQC

Cortex-M4 연구 X

Code-based Cryptography			
Algorithm	**(KEM) Classic-McEliece**	**(KEM) BIKE**	**(KEM) HQC**
동일 계열 KpqC	**PALOMA**	-	-
공통 최적화	Optimized FIPS202 (Keccak, SHA-3, SHAKE) (pqm4 present)		
성능 최적화 (다항식 곱셈)		Bernstein 5-way, Frobenius AFFT	-
메모리 최적화	Streaming pk Storing pk in flash memory	-	-

❖ 국외 Cortex-M4 Code 기반 암호 구현 최적화 연구 동향

- ➢ Classic McEliece와 Bike에 대한 Cortex-M4 구현 최적화 연구 존재
- ➢ HQC는 구현 최적화 연구 없음

Classic McEliece 주요 최적화 논문

Classic McEliece Implementation with Low Memory Footprint, CARDIS 2020

→

Classic McEliece on the ARM Cortex-M, TCHES 2021

Classic-McEliece의 문제점

- pk size(255/512/1326 KB) → Cortex-M4 실행 불가

(메모리) 최적화 방안

- 핵심 전략 1 : **Storing pk in Flash-Memory**
- 핵심 전략 2 : **Streaming pk**

Bike 주요 최적화 논문

Optimizing BIKE for the Intel Haswell and ARM Cortex-M4, TCHES 2021

- 각 환경에 적합한 알고리즘 활용
- 이진 필드 상에서의 연산 최적화

↓

Carry-less to bike faster ACNS 2022

- Radix-16 Representation 고도화

❖ (메모리) Cortex-M4 Classic-McEliece 구현 최적화 연구 동향

- 키 생성 알고리즘은 먼저 페리티 체크 행렬 $\hat{\boldsymbol{H}}$를 생성
  - ✓ $\hat{\boldsymbol{H}} = (\boldsymbol{M} \mid \boldsymbol{T}) \in \mathrm{F}_2^{(n-k)\times n}$를 생성 후 $\boldsymbol{H} = (\boldsymbol{I} \mid \hat{\boldsymbol{T}}) \in \mathrm{F}_2^{(n-k)\times n}$ 를 연산
  - ✓ $\hat{\boldsymbol{T}}$은 가우스-조던 소거법을 사용 $\hat{\boldsymbol{T}} \rightarrow pk$

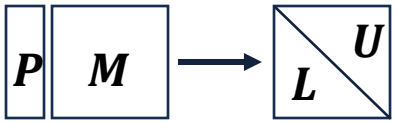
- **Reference Implementation**
  - ✓ 가우스-조던 소거법을 사용하여 계산
  - ✓ $\hat{\boldsymbol{H}} = (\boldsymbol{M} \mid \boldsymbol{T}) \in \mathrm{F}_2^{(n-k)\times n}$를 메모리에 유지 해야함 (최소 약 335 KB)
  - ✓ 높은 메모리 footprint 때문에 Cortex-M4에서 구현이 어려움

$k = n - mt$	m	n	t	level	public key	secret key	ciphertext
mceliece348864*	12	3488	64	1	261 120	6492	128
mceliece460896*	13	4608	96	3	524 160	13 608	188
mceliece6688128*	13	6688	128	5	1 044 992	13 932	240
mceliece6960119*	13	6960	119	5	1 047 319	13 948	226
mceliece8192128*	13	8192	128	5	1 357 824	14 120	240

❖ (메모리) Cortex-M4 Classic-McEliece 구현 최적화 연구 동향

➢ Optimization(메모리) Method

- ✓ $\hat{H} = (M \mid T)$ 에서 가우스-조던 방식이 아닌, M^{-1}를 생성
- ✓ LU 분해를 이용하여 $H = (I \mid M^{-1}\hat{T})$을 생성
- ✓ $PM = LU \rightarrow M^{-1} = U^{-1}L^{-1}P$



// U^{-1}, L^{-1}: in-place manner (전방/후방 대치)

➢ 특징

- ✓ $pk_i = M^{-1} T_i$로 streaming이 가능 (flash memory에 저장)
  - $i \in [0, k), T_i\ (row - representation\ of\ T)$
- ✓ M 에 L 및 U를 저장 가능
- ✓ **ref 메모리 요구량**
  - $(n-k) \times n$
- ✓ **최적화 메모리 요구량**
  - $(n-k) \times (n-k)$

Param	n	k	$(n-k) \times n$: $\hat{H}$	$(n-k)^2$: M^{-1}
348864	3,488	2,720	334,848 bytes	75,264 bytes
460896	4,608	3,360	718,848 bytes	197,184 bytes
6688128	6,688	5,024	1,391,104 bytes	349,440 bytes

❖ (메모리) Cortex-M4 Classic-McEliece 구현 최적화 연구 동향

➢ **Classic-McEliece**

✓ binary Goppa 코드를 페리티 체크행렬 $\hat{H} = \mathbf{BC}$로 사용

✓ Berlekamp-Massey decoding 활용

➢ **Paloma**

✓ binary Goppa 코드를 페리티 체크행렬 $\hat{H} = \mathbf{ABC}$로 사용

✓ extended Patterson decoding 활용

✓ **Classic-McEliece의 메모리 최적화 방안 적용 가능 (구조의 유사성)**

- Streaming pk
- Storing pk in flash memory

구조의 유사성 존재

$$\mathbf{A} := \begin{pmatrix} g_1 & g_2 & \cdots & g_t \\ g_2 & g_3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_t & 0 & \cdots & 0 \end{pmatrix} \mathbf{B} := \begin{pmatrix} \alpha_0^0 & \alpha_1^0 & \cdots & \alpha_{n-1}^0 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{t-2} & \alpha_1^{t-2} & \cdots & \alpha_{n-1}^{t-2} \\ \alpha_0^{t-1} & \alpha_1^{t-1} & \cdots & \alpha_{n-1}^{t-1} \end{pmatrix} \text{ and } \mathbf{C} := \begin{pmatrix} g(\alpha_0)^{-1} & 0 & \cdots & 0 \\ 0 & g(\alpha_1)^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g(\alpha_{n-1})^{-1} \end{pmatrix}$$

PALOMA: Binary Separable Goppa-based KEM, KpqC Round2 Proposal

❖ (메모리) Cortex-M4 Classic-McEliece 구현 최적화 연구 동향

➢ Optimization(메모리) Method

- ✓ pk_i를 Flash memory로 streaming(write) 하므로 많은 부하가 존재
- ✓ 하기 표는 STM32L4R5 (640KB)에서 동작 주파수별 Classic-McEliece의 cycles

CPU Clock : 16MHz	keypair cycles	keypair (real time)	encaps cycles	decaps cycles
mceliece348864f	1,289,567 k	(80.60sec)	520k	2,409 k
mceliece460896f	4,216,263 k	(263.52sec)	919k	5,787 k
mceliece6688128f	7,435,609 k	(464.73sec)	2,123k	6,696 k
mceliece8192128f	8,473,271 k	(529.58sec)	3,445k	6,762 k

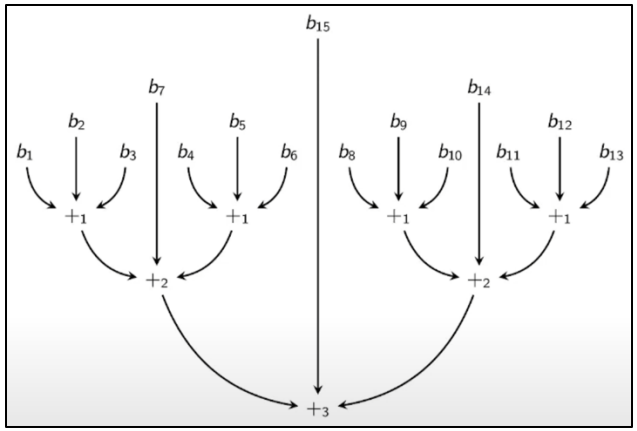
CPU Clock : 120MHz	keypair cycles	keypair (real time)	encaps cycles	decaps cycles
mceliece348864f	1,298,845 k	(10.82sec)	627 k	4,213 k
mceliece460896f	5,109,945 k	(42.58sec)	1,484 k	10,315 k
mceliece6688128f	9,210,630 k	(76.76sec)	2,931 k	11,327 k
mceliece8192128f	10,773,637 k	(89.78sec)	4,545 k	11,400 k

❖ (성능) Cortex-M4 BIKE 구현 최적화 연구 동향

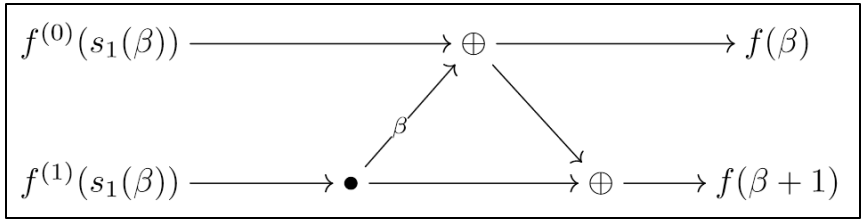
➢ Optimizing Bike for the Intel Haswell and ARM Cortex-M4 (TCHES 2021)

BIKE 최적화 핵심 전략

- 각 환경에 적합한 Shifter 설계
  - Intel Haswell : Matrix transpositions 활용
  - **ARM Cortex-M4 : Barrel shifter 활용**
- 덧셈 연산에 대한 Full adders 적용
  - **Boyar-Peralta 알고리즘 활용**
- 각 환경 별 최적 곱셈 적용
  - Intel Haswell : Bernstein 5-way recursive 알고리즘
  - **ARM Cortex-M4 : Frobenius addtive FFT 알고리즘**



[Boyar-Peralta 알고리즘 모식도]



[FAFFT Butterfly 연산 모식도]

Cotex-M4	Key Gen.	Encap.	Decap.	구분
BIKE1	65,414,337	4,824,059	114,592,442	Reference
	24,935,033	**3,253,379**	**49,911,673**	**최적화 구현**
BIKE3	212,999,628	15,041,356	374,777,003	Reference
	59,820,502	**8,376,212**	**139,234,176**	**최적화 구현**

[Cortex-M4 상에서의 Reference 대비 성능 비교 (CC)]

❖ (성능) Cortex-M4 HQC 구현 최적화 연구 동향

- ➢ 현재까지 Cortex-M4 상에서의 HQC 연구 결과 부재
- ➢ PQM4 역시 PQClean을 통한 기본적인 테스트만을 제공
  - ✓ HQC의 최대 연산 부하 지점은 이진 필드에서의 다항식 곱셈
  - ✓ HQC Reference의 경우 곱셈 최적화를 위해 NTL, GMP와 같은 외부라이브러리 사용

- ➢ (국내) Cortex-M4 상에서의 최초 HQC 외부 종속성 해제 및 최적화 곱셈(NTL) 적용

곱셈 방법	HQC128	HQC192	HQC256
PQClean	51.50 M	158.25 M	289.32 M
NTL	**4.67 M**	**14.13 M**	**27.39 M**
성능 향상	**x11.02**	**x11.20**	**x10.56**

[Cotex-M4 상에서의 HQC 보안레벨별 이진 필드 곱셈의 성능 비교 (CC)]

- ➢ 기존 BIKE 최적화 곱셈 연구 방안(FAFFT) HQC 적용

알고리즘	PQClean	NTL	FAFFT
HQC128	51,505,609	**4,676,551**	**4,206,319**

[Cotex-M4 상에서의 HQC128 곱셈 성능 비교 (CC)]

❖ NIST PQC 공모전의 "Evaluation Criteria"

- ➢ Security : Against BOTH classical and quantum attacks
- ➢ **Performance : Measured on a variety of classical platforms**
  - ✓ **x86/x64, AVX2 :** 고성능 범용 CPU 장치
    - 통합라이브러리 : **PQClean** (https://github.com/PQClean/PQClean)
    - 제출된 코드(ref, AVX2)가 PQClean 프로젝트에 통합
  - ✓ **Cortex-M4 :** 저 사양 Embedded(임베디드) 장치
    - 통합라이브러리 : **pqm4** (https://github.com/mupq/pqm4)
    - TCHES, IEEE/ACM 등 저명 학회/저널에서의 최적화 구현물들이 pqm4 프로젝트에 통합
- ➢ Other properties : Drop-in replacements, Side-channel-attack , etc …

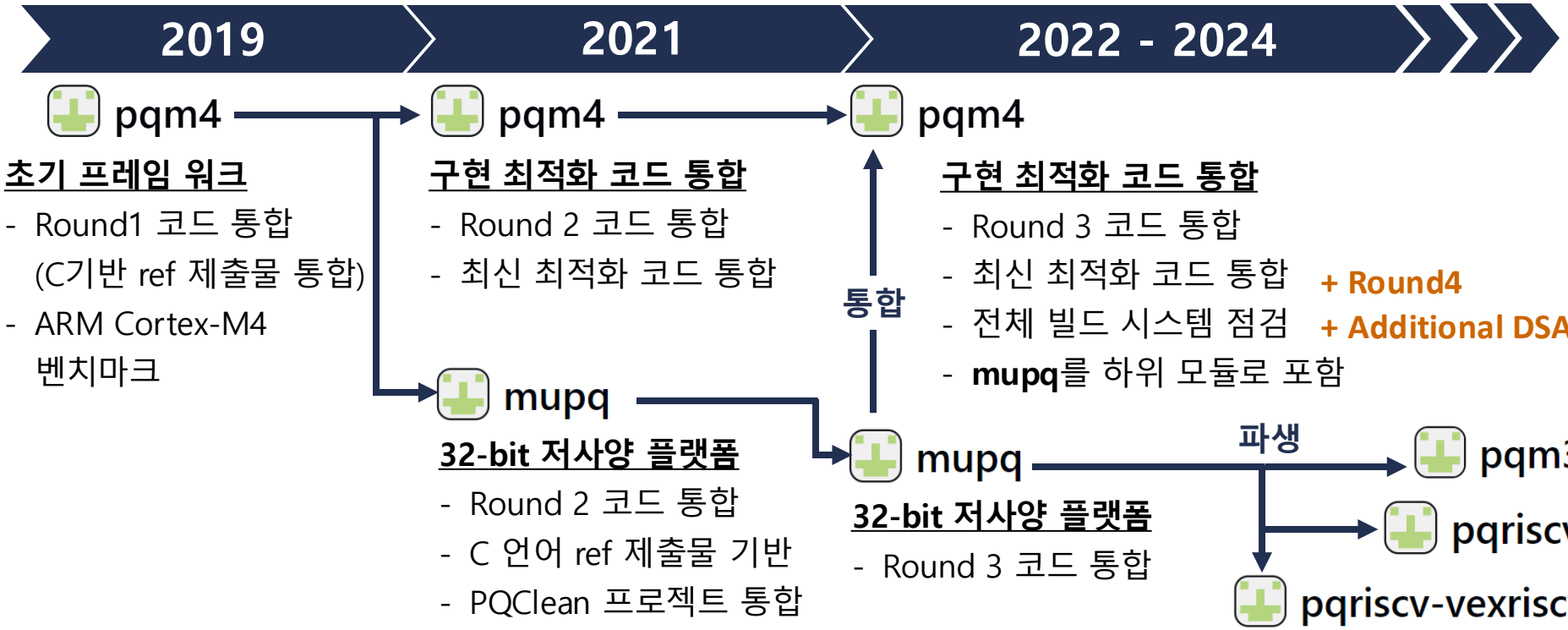
NIST National Institute of Standards and Technology

2017	2019	2020	2022 - 2024
Round 1	Round 2	Round 3	Round 4/ Additional
NIST 알고리즘 제안 요청 및 제안서 접수	(국외) **PQClean, pqm4 프로젝트 시작**	프로젝트 내 알고리즘 업데이트	ML-KEM/DSA 반영 및 **Round4/Additional 알고리즘 업데이트**

❖ ARM Cortex-M4 기반 임베디드 시스템용 벤치마크 프레임워크

- ➢ 2019년부터 NIST PQC 공모전 알고리즘 통합
- ➢ 최신 Round4 KEM / Additional DSA 알고리즘 통합

❖ 각 스킴에 대한 벤치마크/검증 프로그램 자동 생성



Appendix - PQM4

❖ ARM Cortex-M4 기반 임베디드 시스템용 벤치마크 프레임워크

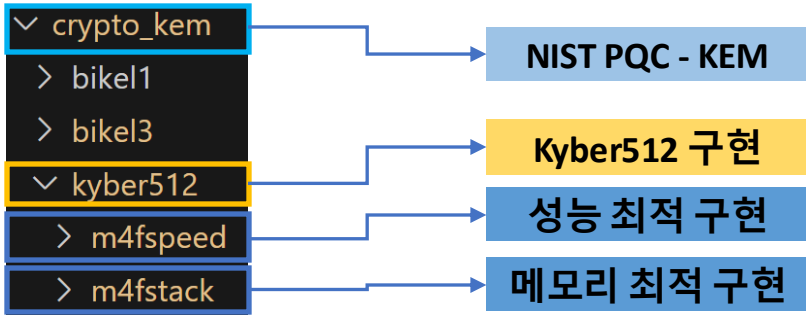
❖ 각 스킴에 대한 벤치마크/검증 프로그램 자동 생성

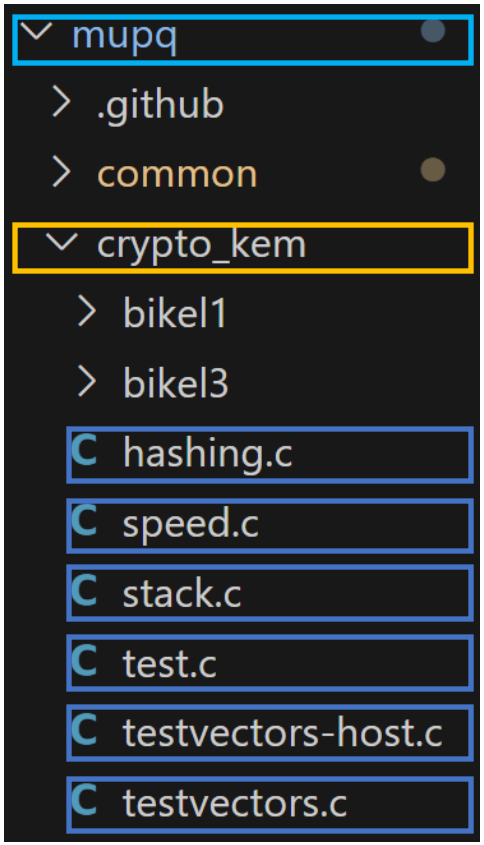
➢벤치마킹 대상: <u>스택 사용량, clock cycle, 해시 clock cycle</u>

동작 주파수: 20MHz 고정

	설명
clean	- 외부 라이브러리 의존성 제거 - Cortex-M4에서 동작 가능하도록 수정
opt	- Reference 코드의 최적화된 c언어 구현
m4	- Reference 코드의 최적화된 어셈블리 구현 (Cortex-M4 대상)
m4f	- Reference 코드의 최적화된 어셈블리 구현 (Cortex-M4 대상, Floating Point Register 사용)

pqm4의 4가지 구현 레벨





Appendix - PQM4

- **Additional DSA 통합**
  - 총 40건의 새로운 알고리즘 통합 시도
  - **15개의 구현에 대한 통합완료**
- **25개의 구현에 대한 통합 이슈 존재**
  - 공개적으로 알려진 취약점
    - ✓ shake256에 대한 잘못된 호출방식
    - ✓ 안전성 이슈 등
  - RAM 크기(640KB)이상의 공개키
  - 이식 불가능한 코드, 동적할당 사용
    - ✓ __int128 사용
    - ✓ 광범위한 동적할당 사용
  - 외부 종속성 라이브러리 사용

				pqm4		reason(s) for exclusion						
		issue	PR	ref	m4f	vuln	pk	mem	not port	ext lib	dyn mem	params
CROSS	[BBB+23b]	#265	#309	✓								12/24
Enhanced pqsigRM	[CNL+23]	#270					✗				✗	0/1
FuLeeca	[RBK+23]	#272				✗						0/3
LESS	[BBB+23a]	#278						✗			✗	0/7
MEDS	[CNP+23]	#280	#324	✓								2/6
Wave	[BCC+23a]	#298					✗				✗	0/3
SQIsign	[CSSF+23]	#293								✗	✗	0/3
EagleSign	[SHDS23]	#267				✗						0/4
EHTv3 and EHTv4	[SF23]	#268				✗					✗	0/5
HAETAE	[CCD+23b]	#273	#313	✓	✓							3/3
HAWK	[BBD+23]	#274	#305	✓								3/3
HuFu	[YJL+23]	#276					✗				✗	0/3
Raccoon	[dPEK+23]	#288							✗			0/18
SQUIRRELS	[ENST23]	#294					✗			✗		0/5
Biscuit	[BKPV23]	#264	#314	✓								3/6
MIRA	[ABB+23c]	#281									✗	0/6
MiRitH	[ARZV+23]	#282	#315	✓	✓							16/32
MQOM	[FR23]	#283	#322	✓							(✗)	2/12
PERK	[ABB+23a]	#284	#318	✓	✓							12/12
RYDE	[ABB+23b]	#289									✗	0/6
SDitH	[MFG+23]	#290						✗			✗	0/12
3WISE	[Rod23a]	#260				✗				✗		0/3
DME-Sign	[LA23]	#266				✗						0/3
HPPC	[Rod23b]	#275								✗		0/3
MAYO	[BCC+23b]	#279	#302	✓	✓							3/4
PROV	[GCF+23]	#286						✗			✗	0/3
QR-UOV	[FIH+23]	#287						✗			✗	0/12
SNOVA	[WCD+23]	#291	#311	✓								7/18
TUOV	[DGG+23]	#295	#327	✓				✗			✗	0/12
UOV	[BCD+23]	#296	#300	✓	✓							3/12
VOX	[PCF+23]	#297				✗					✗	0/3
AIMer	[KCC+23]	#261	#323	✓							(✗)	3/12
Ascon-Sign	[SGJ+23]	#263	#308	✓								8/8
FAEST	[BBdSG+23]	#271									✗	0/12
SPHINCS-alpha	[YCZ23]	#292	#312	✓								6/24
ALTEQ	[BDN+23]	#262						✗			✗	0/6
eMLE-Sig 2.0	[LZ23]	#269				✗					✗	0/3
KAZ-SIGN	[AAC+23]	#277				✗				✗	✗	0/3
Preon	[CCC+23]	#285						✗			✗	0/9
Xifrat1-Sign.I	[NP23]	#299				✗						0/1
				15	5	9	4	7	1	5	20	83/325

❖ 이슈 1 : pqm4 암호 알고리즘 미사용

- ➢일부 알고리즘은 **pqm4와 다른 형식의 AES, SHA2, Keccak 함수 사용**
- ➢통합을 위해 pqm4의 형식을 따르거나, 새로운 함수를 정의해야 함

KPQC

```
#define sha256 SHA2_NAMESPACE(sha256)
void sha256(uint8_t out[32], const uint8_t *in, size_t inlen);
#define sha512 SHA2_NAMESPACE(sha512)
void sha512(uint8_t out[64], const uint8_t *in, size_t inlen);
```

pqm4

```
void sha256_inc_init(sha256ctx *state);

void sha256_inc_ctx_clone(sha256ctx *stateout, const sha256ctx *statein);

void sha256_inc_blocks(sha256ctx *state, const uint8_t *in, size_t inblocks);

void sha256_inc_finalize(uint8_t *out, sha256ctx *state, const uint8_t *in, size_t inlen);

void sha256_inc_ctx_release(sha256ctx *state);
```

pqm4(아래)와 다른 SHA2 함수를 사용하는 경우(위)

```
void aes256ctr_prf(uint8_t *out, size_t outlen,
    const uint8_t key[32], const uint8_t nonce[12]) {
  uint64_t sk_exp[120];

  br_aes_ct64_ctr_init(sk_exp, key);
  br_aes_ct64_ctr_run(sk_exp, nonce, 0, out, outlen);
}
```

pqm4에서 사용하지 않는
AES 함수를 사용하는 경우

❖ 이슈 해결 방안 1 : pqm4 암호 알고리즘 미사용

➢NTRU+

✓pqm4에서 지원하지 않는 aes256ctr_prf 함수를 사용

NTRU+

```
void aes256ctr_prf(uint8_t *out, size_t outlen,
    const uint8_t key[32], const uint8_t nonce[12]) {
  uint64_t sk_exp[120];

  br_aes_ct64_ctr_init(sk_exp, key);
  br_aes_ct64_ctr_run(sk_exp, nonce, 0, out, outlen);
}
```

pqm4의 AES 코드에 삽입 및 코드 수정

pqm4

```
void aes256ctr_prf(uint8_t *out, size_t outlen,
    const uint8_t key[32], const uint8_t nonce[12]) {
    aes256ctx ctx;

    aes256_ctr_keyexp(&ctx, key);
    aes256_ctr(out, outlen, nonce, &ctx);
}
```

➢NCC-Sign

✓Keccak 함수 포맷이 다름

NCC-Sign

```
typedef struct {
  uint64_t s[25];
  unsigned int pos;
} keccak_state;

void shake256_init(keccak_state *state);
void shake256_absorb(keccak_state *state, const uint8_t *in, size_t inlen);
void shake256_finalize(keccak_state *state);
void shake256_squeezeblocks(uint8_t *out, size_t nblocks,  keccak_state *state);
void shake256_squeeze(uint8_t *out, size_t outlen, keccak_state *state);
```

```
typedef struct {
  uint64_t ctx[26];
} shake256incctx;

void shake128_squeezeblocks(uint8_t *output, size_t nblocks, shake128ctx *state);
void shake128_ctx_release(shake128ctx *state);
void shake128_ctx_clone(shake128ctx *dest, const shake128ctx *src);
```

pqm4

pqm4의 포맷에 따라 코드 수정

❖ 이슈 2 : pqm4 API와 다른 API를 사용하는 경우

- ➢pqm4 테스트 실행 시 api.h의 함수를 호출하여 사용
- ➢api.h에서 제공하는 함수 signature가 다를 경우 파라미터 에러 발생

❖해결 방안

- ➢**테스트 함수 수정**
- ➢KpqC 알고리즘의 함수 signature 수정

pqm4

```
int crypto_kem_keypair( unsigned char *pk,  unsigned char *sk);

int crypto_kem_enc( unsigned char *ct,
                    unsigned char *ss,
                    const unsigned char *pk);

int crypto_kem_dec( unsigned char *ss,
                    const unsigned char *ct,
                    const unsigned char *sk);
```

테스트하면서 keypair 함수를 호출

```
t0 = hal_get_time();
MUPQ_crypto_kem_keypair(pk, sk);
t1 = hal_get_time();
printcycles("keypair cycles:", t1-t0);
```

KpqC

```
int crypto_kem_keypair(
                    unsigned char* pk,
                    unsigned char* sk,
                    const gf2m_tab* gf2m_tables);
```

컴파일 실패

```
t0 = hal_get_time();
MUPQ_crypto_kem_keypair(pk, sk);
t1 = hal_get_time();
printcycles("keypair cycles:", t1-t0);
```

❖ 이슈 해결 방안 2 : pqm4 API와 다른 API를 사용하는 경우

➢SMAUG-T

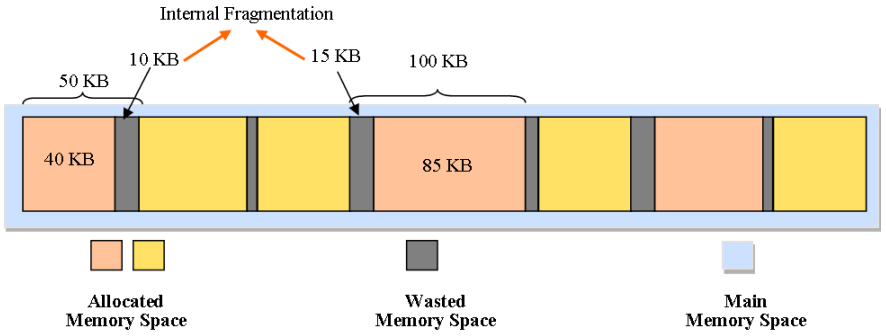
- ✓Decap에서 공개키를 인자로 받음
- ✓SMAUG-T의 api.h에 KPQM4 정의 후 ifdef로 분기
- ✓SMAUG-T가 실행될 때 적절한 함수 파라미터를 찾을 수 있게 함

```
int crypto_kem_dec(uint8_t *ss, const uint8_t *sk,
    SMAUG-T      const uint8_t *pk, const uint8_t *ctxt);
int crypto_kem_dec(unsigned char *ss, const unsigned char *ct,
      pqm4       const unsigned char *sk);
```

```
#elif defined (KPQM4)
    MUPQ_crypto_kem_dec(key_a, sk_a, pk, sendb);
#else
    MUPQ_crypto_kem_dec(key_a, sendb, sk_a);
#endif
```

❖ 이슈 3 및 해결방안 : 동적할당 변경

- **임베디드 환경에서 잦은 동적 할당은 오동작을 일으킬 수 있음**
  - ✓Alloc/Free 호출 과정에서 오류 발생 가능 (**Memory Fragmentation**)
  - ✓pqm4에서도 동적할당을 모두 제거하고 stack으로만 구현
- **사용하는 최대 크기만큼 정적할당하도록 코드 수정**



Memory Fragmentation의 모식도

```
typedef struct {
    uint8_t* sx;
    uint8_t neg_start;
    uint8_t cnt;
} sppoly; // sparse poly
```

→

```
typedef struct {
    uint8_t sx[97];
    uint8_t neg_start;
    uint8_t cnt;
} sppoly; // sparse poly
```

```
for (unsigned long long i = 0; i < MODULE_RANK; ++i) {
    r[i].cnt = cnt_arr[i];
    r[i].sx = (uint8_t *)malloc(cnt_arr[i] * sizeof(uint8_t));
    r[i].neg_start = convToIdx(r[i].sx, r[i].cnt, res + (i * LWE_N), LWE_N);
}
```

↓

```
for (unsigned long long i = 0; i < MODULE_RANK; ++i) {
    r[i].cnt = cnt_arr[i];
    r[i].neg_start = convToIdx(r[i].sx, r[i].cnt, res + (i * LWE_N), LWE_N);
}
```

동적할당을 제거하는 예시(SMAUG-T)

❖큰 키 크기를 가진 PALOMA, MQ-Sign은 키 쌍을 플래시 메모리에 저장

➢keypair 함수로 사전 생성한 키를 플래시 메모리에 저장

✓NUCLEO-L4R5ZI의 플래시 메모리 크기 : 2MB

➢키를 사전 생성하므로 sign, verify / encap, decap만을 수행 후 성능 측정

Security	Algorithm	Public key	Secret key	Ciphertext	Key
128	hqc-128	2,249	40	4,481	64
	BIKE	1,541	281	1,573	32
	mceliece348864	261,120	6,492	96	32
	PALOMA-128	319,488	94,528	136	32
192	hqc-192	4,522	40	9,026	64
	BIKE	3,083	419	3,115	32
	mceliece460896	524,160	13,608	156	32
	PALOMA-192	812,032	357,568	240	32
256	hqc-256	7,245	40	14,469	64
	BIKE	5,122	580	5,154	32
	mceliece6688128	1,044,992	13,932	208	32
	PALOMA-256	1,025,024	359,616	240	32

PALOMA의 데이터 크기(byte)

Scheme	Security Level	1	3	5
MQ-Sign-RR	Public Key	328,505	1,238,825	2,893,025
	Secret Key	276,649	1,044,385	2,436,769
	Signature	150	216	276
MQ-Sign-LR	Public Key	328,441	1,238,761	2,892,961
	Secret Key	160,881	601,249	1,400,113
	Signature	150	216	276

MQ-Sign의 데이터 크기(byte)

❖큰 키 크기를 가진 PALOMA, MQ-Sign은 키 쌍을 플래시 메모리에 저장

➢__attribute__((section(".text")))로 플래시 메모리에 할당 가능

✓플래시 메모리는 프로그램 실행 중 write가 어려우므로 **키를 미리 생성**

➢테스트 코드에서는 key.h가 없을 때만 pk, sk를 정의하도록 수정

✓key.h가 있을 경우(플래시할 경우) key.h를 include

```
3   __attribute__((section(".text"))) unsigned char pk[PK_BYTES] = {0x34,
```

key.c

```
3   extern unsigned char pk[PK_BYTES];
4   extern unsigned char sk[SK_BYTES];
```

key.h

```
133  #ifdef KPQM4_PALOMA
134  #include "key.h"
135  #else
136      unsigned char pk[MUPQ_CRYPTO_PUBLICKEYBYTES];
137      unsigned char sk[MUPQ_CRYPTO_SECRETKEYBYTES];
138      MUPQ_crypto_kem_keypair(pk, sk);
139  #endif
```

test.c

❖ **REDOG의 경우 코드가 아직 미완성**

➢완벽한 C Reference 코드가 업데이트 되지 않음

➢**추후에 코드 업데이트 이후에 kpqm4 프로젝트에 포팅 예정**

✓미완성 코드 포팅 시 실패할 가능성 있음

✓REDOG 팀에서도 특정 상황에서 실패할 가능성이 있음을 알림

* REDOG: My understanding is that the submission team is still working on their initial C code.

D. J. Bernstein의 KpqC Bulletin Board 게시물 인용

```
REDOG
April 3, 2024

The implementation package may fail to run under certain circumstances.
We are still working on implementation and will post the updated version on the REDOG webpage soon.

https://sites.google.com/view/code-redog
```

REDOG 구현물의 README 파일

KOOKMIN UNIVERSITY

양자내성암호연구단

KpqC 연구단
9차 워크숍

Thank You

CSE CRYPTO & SECURITY ENGINEERING Lab