KpqC 공모전 알고리즘 기본 구현에 대한 성능 비교 분석

2023. 11. 14. @연세대학교

권혁동, 송경주, 심민주, 이민우, 김상원, 서화정

KpqC 선정과 알고리즘 효율성

연산 속도 관점에서의 분석

메모리 소모량 관점에서의 분석

부채널 관점에서의 분석

Scheme	KEM	Digital Signature
Code	Layered-ROLLO PALOMA REDOG	Enhanced pqsigRM
Lattice	NTRU+ SMAUG TiGER	GCKSign HAETAE NCC-Sign Peregrine SOLMAE
Multivariate	-	MQ-Sign
Isogeny	-	FIBS
Symmetric	-	AIMER
Graph	IPCC	-

PQC 공모전은 전세계에서 **미국, 중국, 한국**이 유일하게 개최!!! → 국가의 암호에 대한 높은 관심도!!!

KpqC 공모전

- 양자컴퓨터의 등장으로 인한 **양자알고리즘 해킹에 대비**하기 위한 국산 양자내성암호 선발 공모전 **(2024년 선정 목표)**
- NIST 공모전과 같이 **Lattice, Code에 대한 제안이 주**를 이루고 있음

신규 암호알고리즘 선정에 있어서의 주안점

- **암호알고리즘의 선정 기준**은 크게 **안전성과 효율성**으로 나눌 수 있음
- NIST에서는 현재 **4개의 암호군에 대해 표준화**를 진행 중에 있음

Public-Key Encryption/KEMs	Digital Signatures
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS+

보안성!

CRYSTALS-KYBER (key-establishment) and CRYSTALS-Dilithium (digital signatures) were both selected for their strong security and excellent performance, and NIST expects them to work well in most applications.

연산 효율성!

FALCON will also be standardized by NIST since there may be use cases for which CRYSTALS-Dilithium signatures are too large.

메모리 효율성!

SPHINCS+ will also be standardized to avoid relying only on the security of lattices for signatures. NIST asks for public feedback on a version of SPHINCS+ with a lower number of maximum signatures.

기반 문제 보안성!

효율성 → 구현의 용이성???

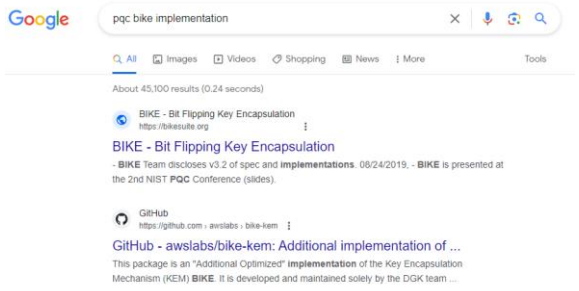
누구나 큰 노력 없이도 구현이 가능해야 함!!! (약간 노력은 필요해 보임)

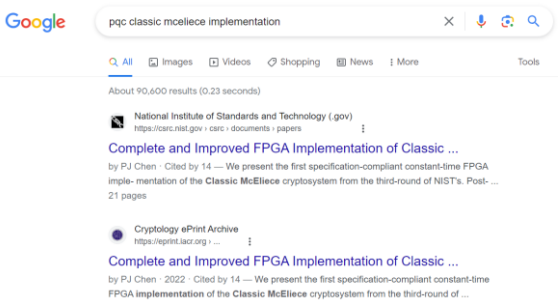
- C언어의 일반적인 자료형을 통해 특수 모듈 (DSP / 가속기) 없이 구현 가능 (알고리즘 설계 사상에 크게 영향을 줌)
- 외부 라이브러리에 대한 의존성없이 쉽게 구현이 가능해야 함 (다양한 레퍼런스 코드 개발 결과 필요)
- 많은 사람들이 알고리즘에 대한 구현결과물을 제시하고 그에 대한 상세한 설명 정리 (충분한 시간과 관심이 필요)

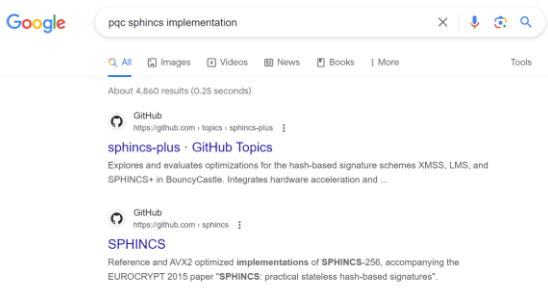
NIST PQC 암호 구현 관련 검색 결과 (구글 검색어: PQC 암호명 implementation)

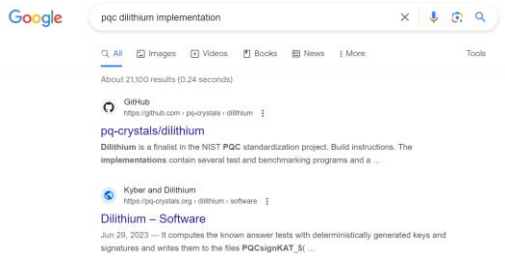
- **오래된 암호일수록 많은 연구 결과물 검색**
- **선정된 암호에 대한 관심도가 높음**
- **SPHINCS는 생각보다 구현 관심도가 낮음**
  - Hash 기반이여서 구현 성능 개선에서 한계점 존재

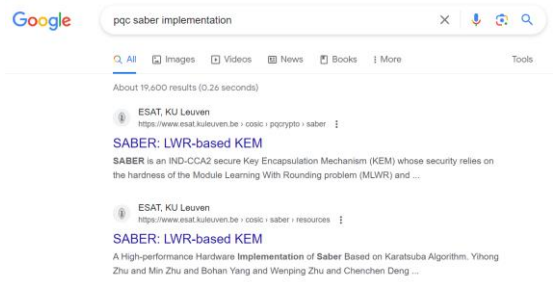
암호	검색결과 수
Classic McEliece	90,600
Kyber	46,900
BIKE	45,100
HQC	25,100
FALCON	22,500
Dilithium	21,100
SPHINCS	4,860

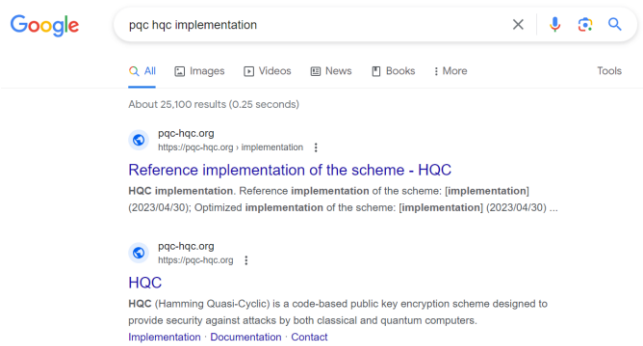


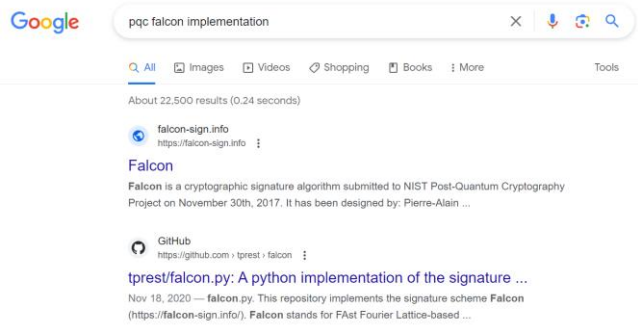


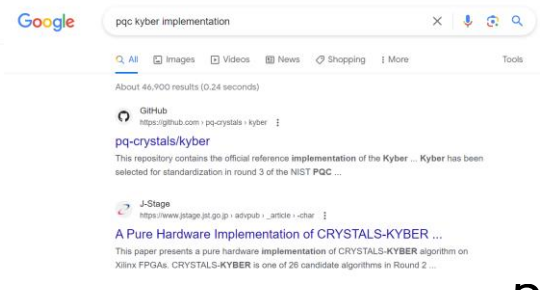












KpqC 공모전 암호들의 보안성 검토

WISA'23
Prof. Bo-Ying Yang 발표자료 캡처

- KpqC 공모전 제출 암호들에 대한 **보안 취약점 보고 및 보완**이 이루어짐
- **보완과정과 효율성**
  - Security를 높이기 위해 보수적인 파라미터 선정 → **연산 효율성 감소**

Meanwhile: the KpqC Competition ii

What Happened to the KEMs?

~~IPCC~~	Attack script, a big revision was submitted
Layered ROLLO-I†	parameters wrong, revision submitted
NTRU+	CCA2 conversion (and implementation) faulty — not a real attack
~~REDOG~~	parameters are too small (and the system doesn't work as described)
SMAUG‡	Error distributions is narrow, missed that [A. May 2021] works.
TIGER‡	same as SMAUG except that decryption failures made the attack worse.

† groups.google.com/g/kpqc-bulletin/c/GejJ_lp3GLI
‡ groups.google.com/g/kpqc-bulletin/c/8nOd28f2K7k

Meanwhile: the KpqC Competition iii

What Happened to the Digital Signatures?

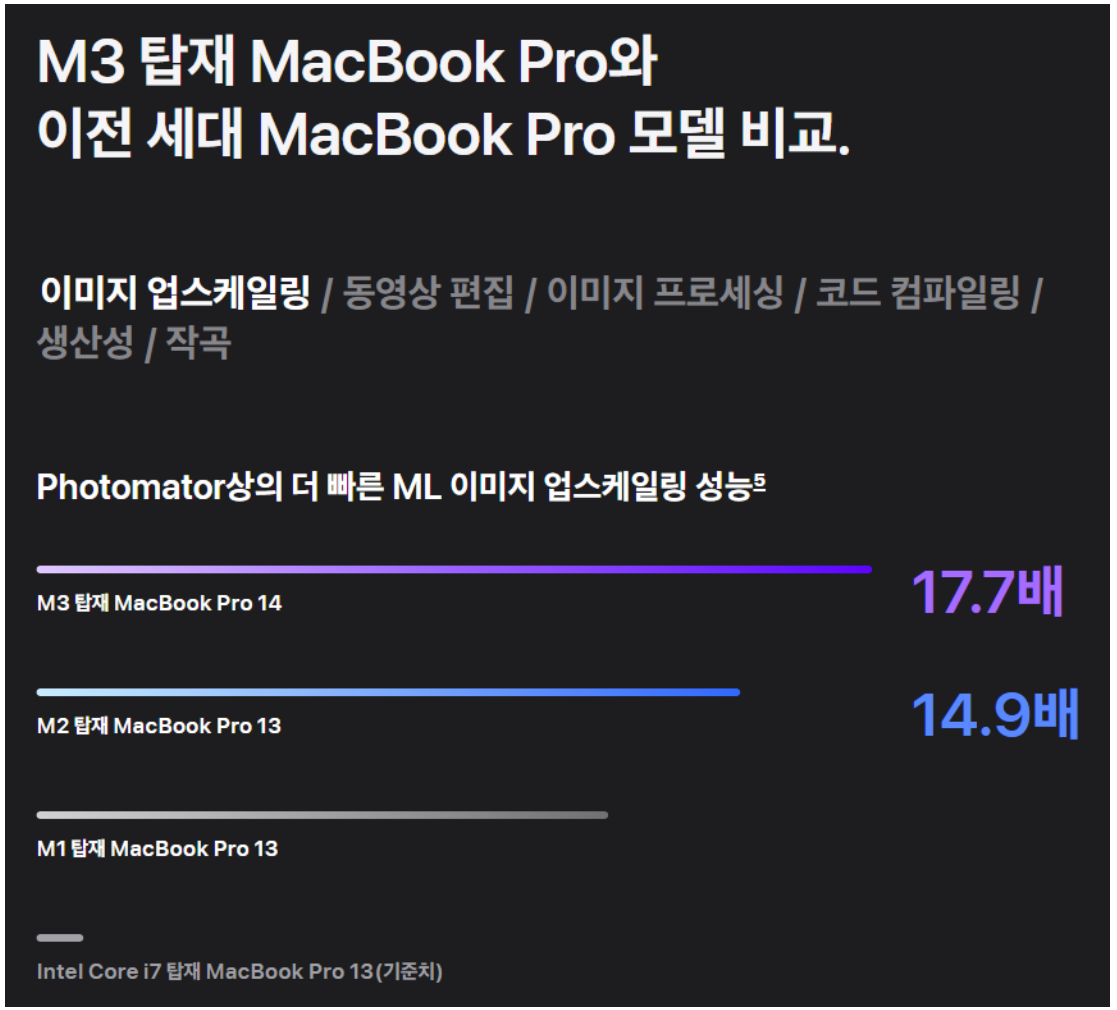
~~Enhanced pqsigRM~~	3h attack script using 10k signatures
FIBS	Slow; CGL hash not guaranteeing SHA3's expected properties
GCKSign	Authors acknowledged an attack and will revise
~~MQSign-II~~	The UOV part of MQSign seems okay, the other part not
Peregrine	a GGH (sampled parallelpiped) attack, #signatures required still unclear

First Elimination Coming Up Nov. 2023
Feels more serious than the CACR competition.
(The 1-year CACR competition was won by Aigis-Sign, Aigis-Enc and LAC.)

암호 효율성이 높을 때 장점

- 가용성 증가
- 전력 소모 감소
- 기존 인프라 성능 업그레이드에 대한 투자 감소
- 암호에 대한 부정적 인식 (?) 감소

연산 효율성이란?
IT 시스템에 있어 가장 피부에 와닿는 성능 수치



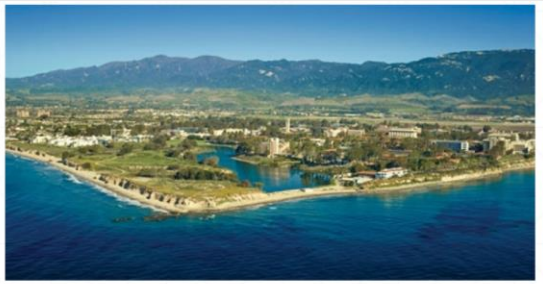
본 발표에서 다루는 내용 → 암.호.효.율.성

- **NIST에서는 2라운드부터 효율성을 강조**
  - 최종 표준안인 **Dilithium의 경우 매우 높은 효율성을 가짐**

Signature Algorithm	Local Machine (ms)				Cloud Instance (ms)			
	Sign		Verify		Sign		Verify	
	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
RSA 3072	3.19	0.023	0.06	0.001	2.39	0.010	0.04	0.002
ECDSA 384	1.32	0.012	1.05	0.020	1.28	0.015	0.93	0.004
Dilithium II	0.82	0.021	0.16	0.005	0.41	0.018	0.12	0.008
Falcon 512	5.22	0.054	0.05	0.004	6.50	0.091	0.07	0.003
MQDSS 48	10.30	0.147	7.25	0.100	10.25	0.181	7.40	0.110
Picnic $L1FS$	4.09	0.050	3.25	0.049	3.17	0.051	2.39	0.044
SPHINCS$^+$ SHA256-128f-simple	93.37	0.654	3.92	0.043	62.7	0.548	2.50	0.037
Rainbow Ia	0.34	0.015	0.83	0.036	0.25	0.020	0.48	0.044
Dilithium IV	1.25	0.021	0.30	0.012	0.46	0.019	0.23	0.010
Falcon 1024	11.37	0.102	0.11	0.005	14.20	0.156	0.14	0.005

What NIST wants

- Performance (hardware+software) will play more of a role
  - More benchmarks
  - For hardware, NIST asks to focus on Cortex M4 (with all options) and Artix-7
    - pqc-hardware-forum
  - How do schemes perform on constrained devices?
  - Side-channel analysis (concrete attacks, protection, etc…)
- Continued research and analysis on **ALL** of the 2nd round candidates
- See how submissions fit into applications/procotols. Any constraints?



The 2nd Round
of the NIST PQC Standardization Process

Dustin Moody

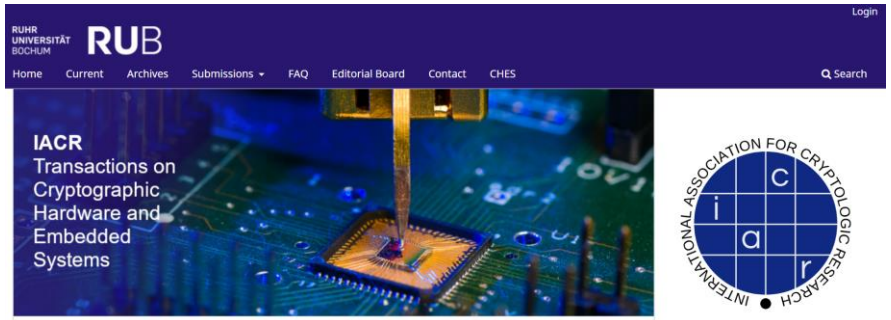
NIST National Institute of Standards and Technology U.S. Department of Commerce

Performance

- We have internal numbers, based on implementations sent to us
  - We strongly prefer code that is constant time
- Performance will play a larger role in the 2nd Round
  - We encourage benchmarking on a variety of platforms
  - We are looking for mature schemes – beyond just proof of concept
- Implementations can always be updated
  - We won't change the implementations on our Round 2 webpage
  - Teams should feel free to advertise results on the pqc-forum, and on their own websites

NIST 양자내성암호 공모전에서의 효율성 확인 수단

- **CHES**
  - 암호 구현 분야 최고 워크샵
  - NIST PQC 시작이후 **PQC에 대한 구현 논문 급증**



CHES'18
- FPGA-based Accelerator for Post-Quantum Signature Scheme SPHINCS-256
- Practical CCA2-Secure and Masked Ring-LWE Implementation
- Rhythmic Keccak: SCA Security and Low Latency in HW
- **SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange**
- Saber on ARM
- Standard Lattice-Based Key Encapsulation on Embedded Devices

CHES'19
- Implementing RLWE-based Schemes Using an RSA Co-Processor
- NTTRU: Truly Fast NTRU Using NTT
- Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols

CHES'20
- Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT
- Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography
- A Compact and Scalable Hardware/Software Co-design of SIKE
- ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V
- Parameterized Hardware Accelerators for Lattice-Based Cryptography and Their Application to the HW/SW Co-Design of qTESLA
- Cortex-M4 optimizations for {R,M} LWE schemes
- Improving the Performance of the Picnic Signature Scheme
- RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography
- High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware

CHES'21
- Compact Dilithium Implementations on Cortex-M3 and Cortex-M4
- Rapidly Verifiable XMSS Signatures
- Polynomial Multiplication in NTRU Prime
- NTT Multiplication for NTT-unfriendly Rings
- A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA
- Optimizing BIKE for the Intel Haswell and ARM Cortex-M4
- Classic McEliece on the ARM Cortex-M4
- Batching CSIDH Group Actions using AVX-512
- Rainbow on Cortex-M4

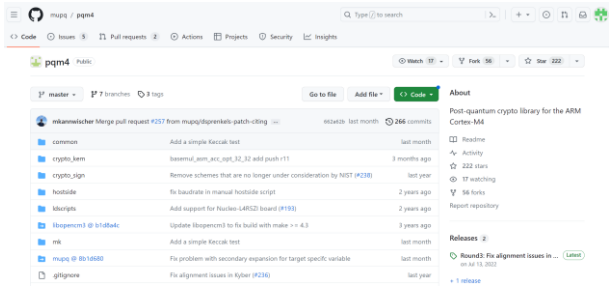
CHES'22
- Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4
- A Constant-time AVX2 Implementation of a Variant of ROLLO
- Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1
- Efficient Implementations of Rainbow and UOV using AVX2
- A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium
- Polynomial multiplication on embedded vector architectures
- Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware
- Highly Vectorized SIKE for AVX-512
- Complete and Improved FPGA Implementation of Classic McEliece
- Faster Constant-Time Decoder for MDPC Codes and Applications to BIKE KEM
- Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4
- Improved Plantard Arithmetic for Lattice-based Cryptography

CHES'23
- Enabling FrodoKEM on Embedded Devices
- Oil and Vinegar: Modern Parameters and Implementations

NIST 양자내성암호 공모전에서의 효율성 확인 수단

- **Open benchmark platform (저전력)**
  - pqm4: ARM Cortex-M4 상에서의 성능 분석
  - Speed, Stack, Code Size, #Hashing 관점에서 분석
  - M4를 선택한 이유?

최소 사양	구 분	권장 사양
64-Bit Windows 7 이상의 운영체제	운영체제	Windows 7/8/10 운영체제(64비트)
Intel Core i3(4세대), AMD FX-6300	프로세서	Intel Core i5(3.5GHz), 라이젠 5 1600X
6GB RAM	메모리	8 / 16GB RAM
GeForce GTX 660 2GB, Radeon HD 7850 2GB	그래픽카드	GeForce GTX 1060 6GB, Radeon R9 4GB
30GB (HDD or SSD)	저장용량	40GB (HDD or SSD)



https://github.com/mupq/pqm4

Key Encapsulation Schemes

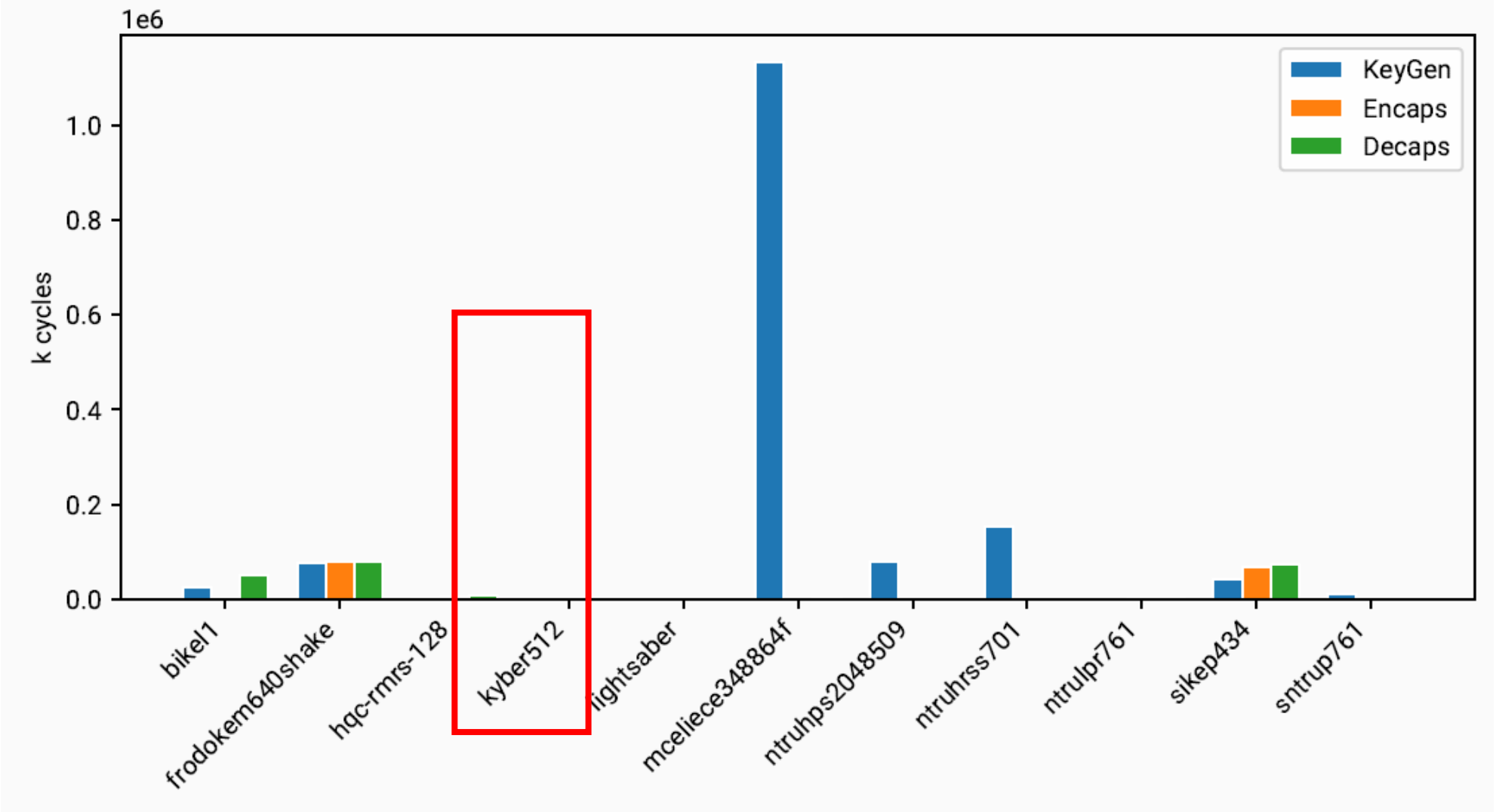
Scheme	Type	Key Generation	Encapsulation	Decapsulation
firesaber	m4	1 448 776	1 786 930	1 853 339
frodokem640aes	m4	47 050 559	45 883 334	45 366 065
frodokem640shake	m4	79 331 373	79 745 404	79 231 474
kyber1024	m4	1 575 052	1 779 848	1 709 348
kyber512	m4	514 291	652 769	621 245

The **pqm4** library, benchmarking and testing framework started as a result of the PQCRYPTO project funded by the European Commission in the H2020 program. It currently contains implementations post-quantum key-encapsulation mechanisms and post-quantum signature schemes targeting the ARM Cortex-M4 family of microcontrollers. The design goals of the library are to offer

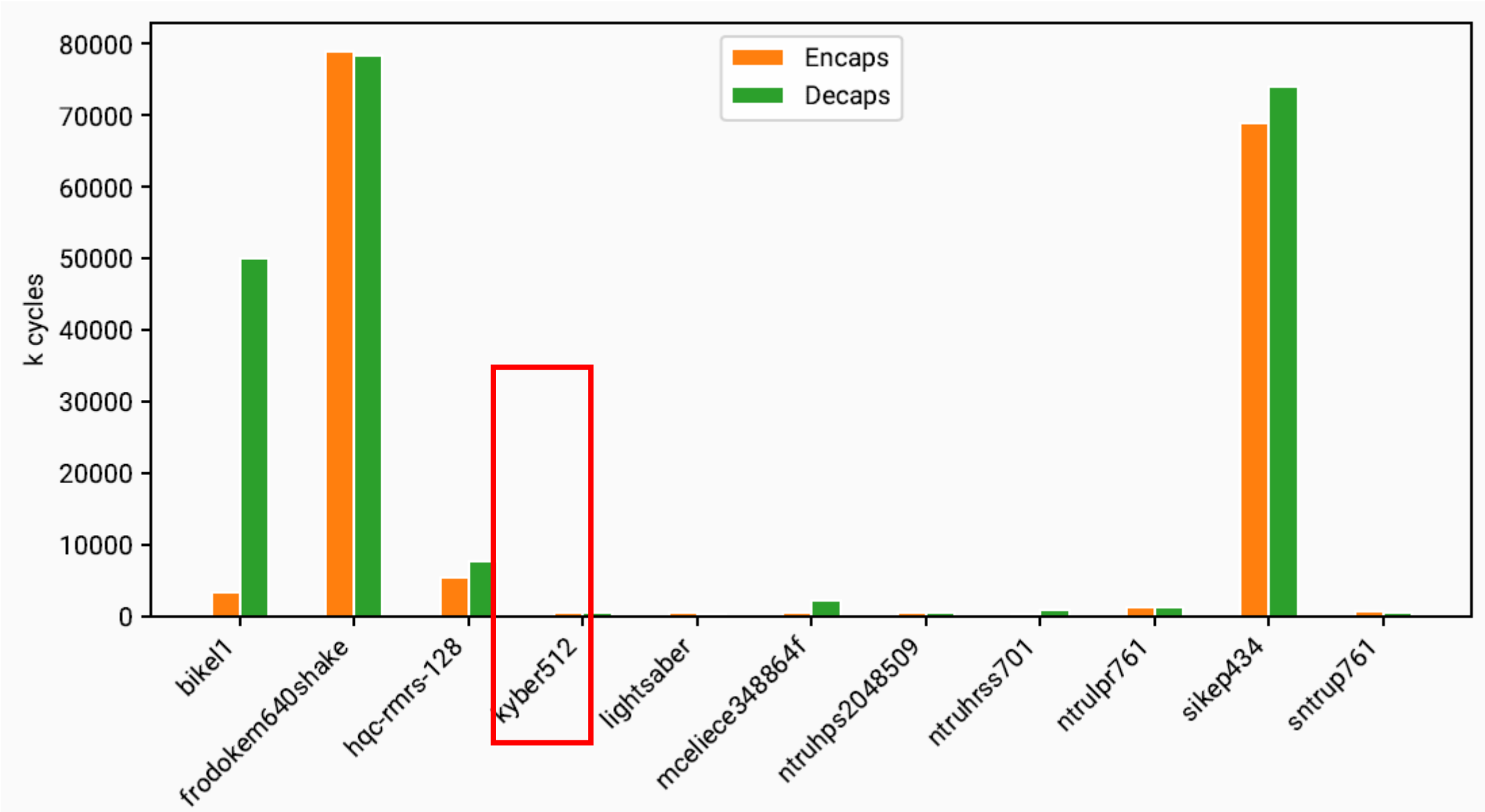
- automated functional testing on a widely available development board;
- automated generation of test vectors and comparison against output of a reference implementation running host-side (i.e., on the computer the development board is connected to);
- automated benchmarking for speed, stack usage, and code-size;
- automated profiling of cycles spent in symmetric primitives (SHA-2, SHA-3, AES);
- integration of clean implementations from PQClean; and
- easy integration of new schemes and implementations into the framework.

비고	Cortex-M4
Architecture	ARMv7E-M (Harvard)
Instruction Set	**Thumb, Thumb-2, DSP, SIMD, FPU**
Pipeline Stage	3
Instruction/Data Cache	X
Interrupt Priority	8-256
Single Cycle Multiply	**전체 제공**
Hardware Divide	O
발표일자	2010

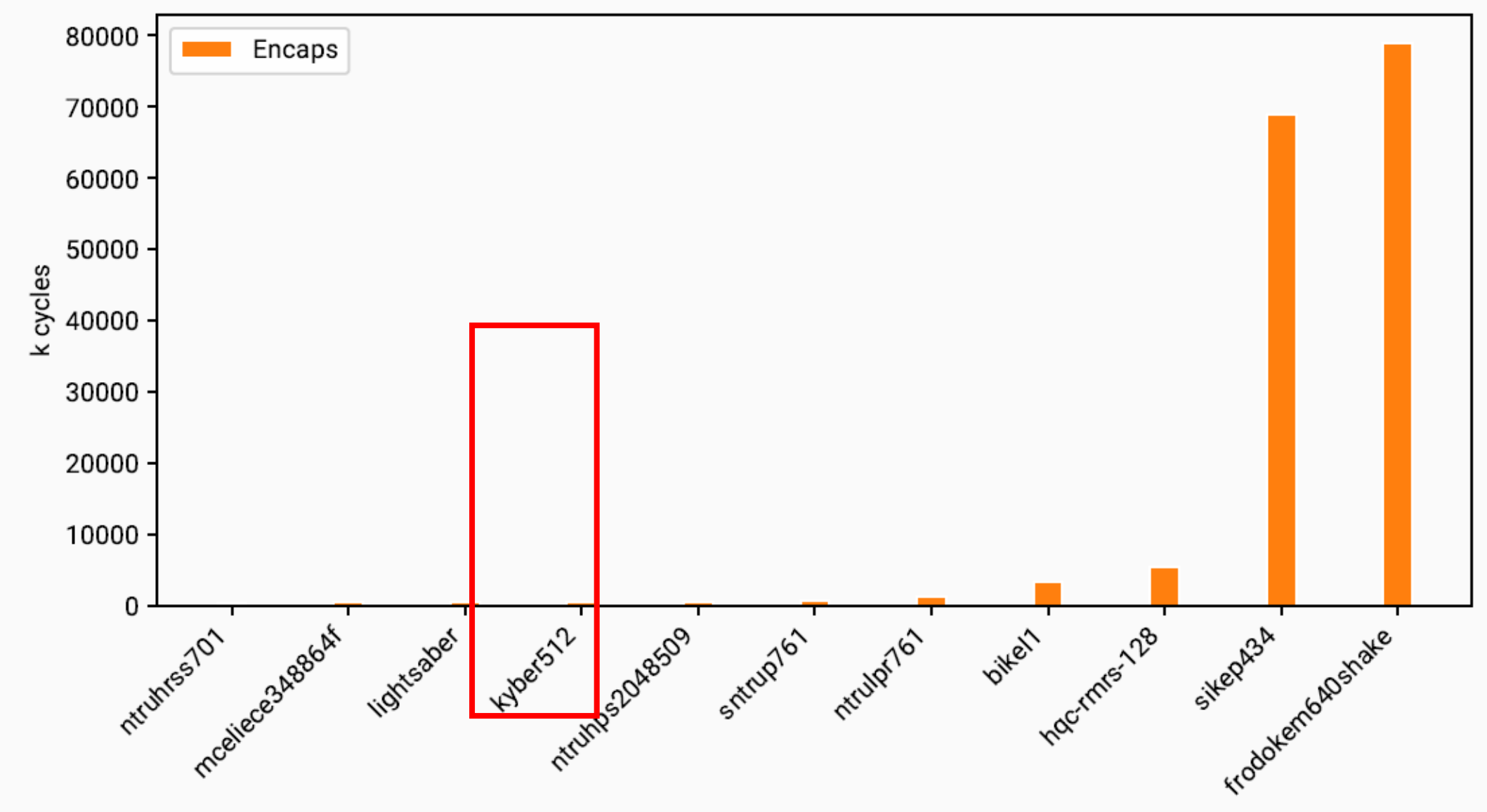
pqm4 결과를 중심으로 KEM Round 3 연산 성능 요약



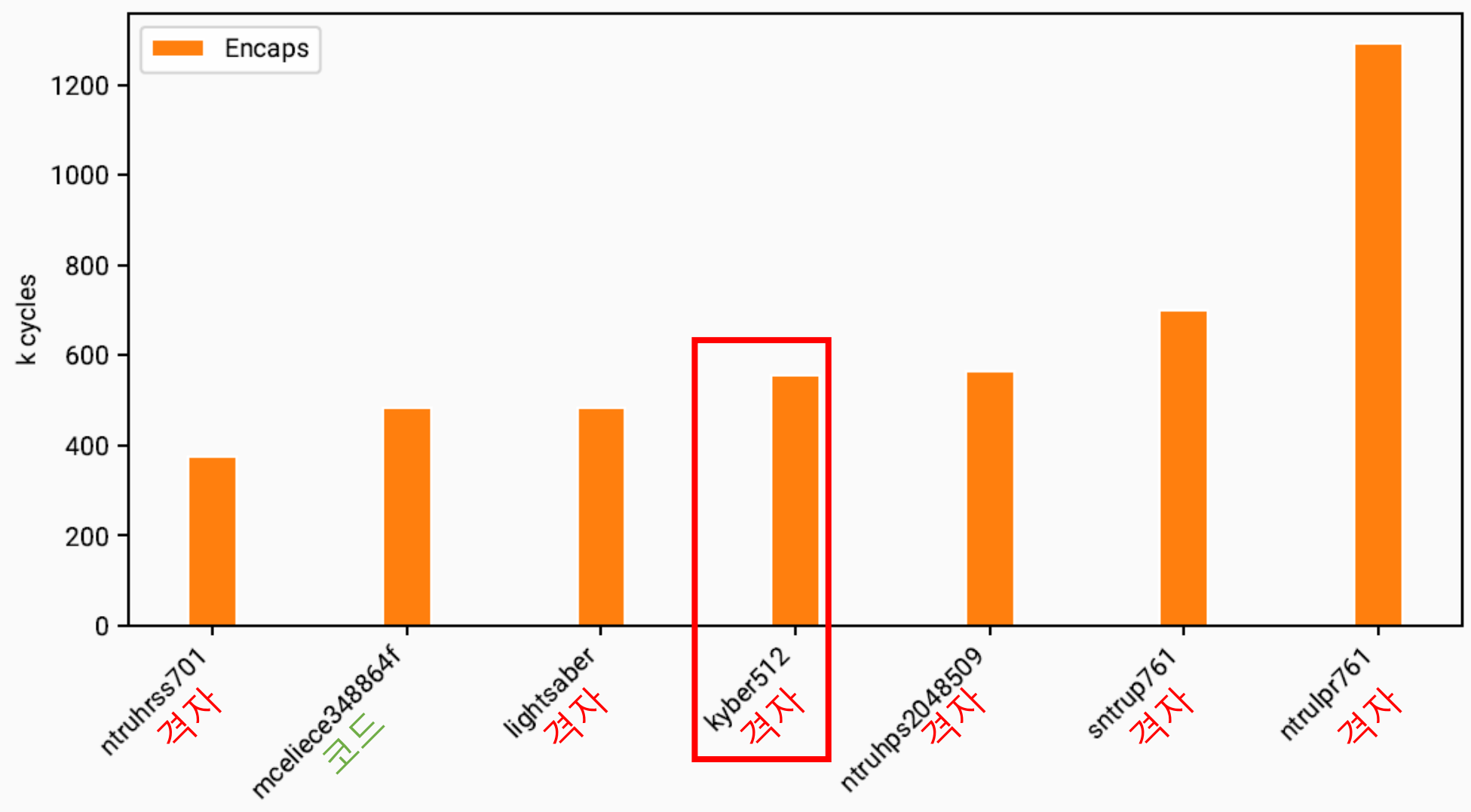
pqm4 결과를 중심으로 KEM Round 3 연산 성능 요약



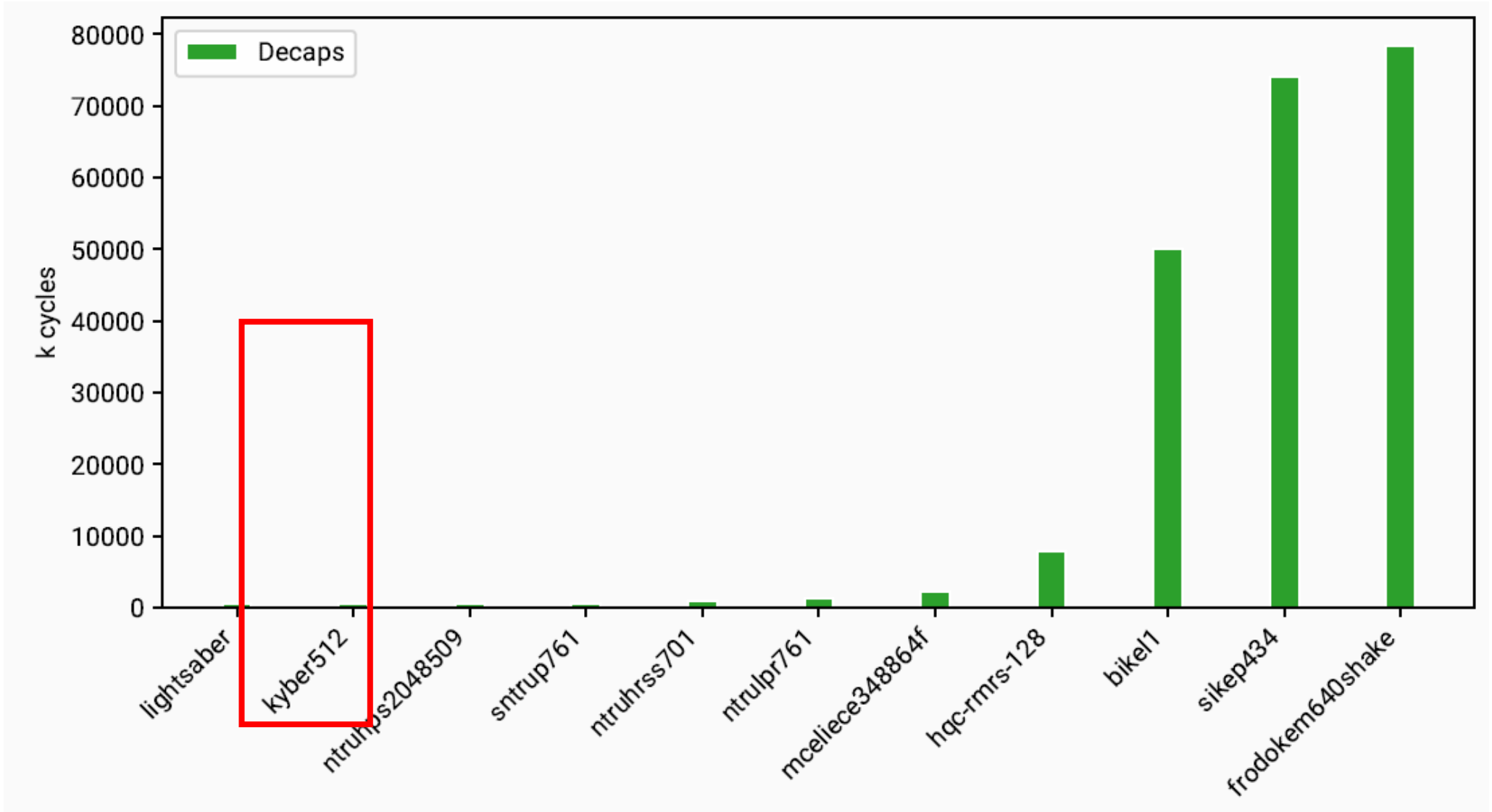
pqm4 결과를 중심으로 KEM Round 3 연산 성능 요약



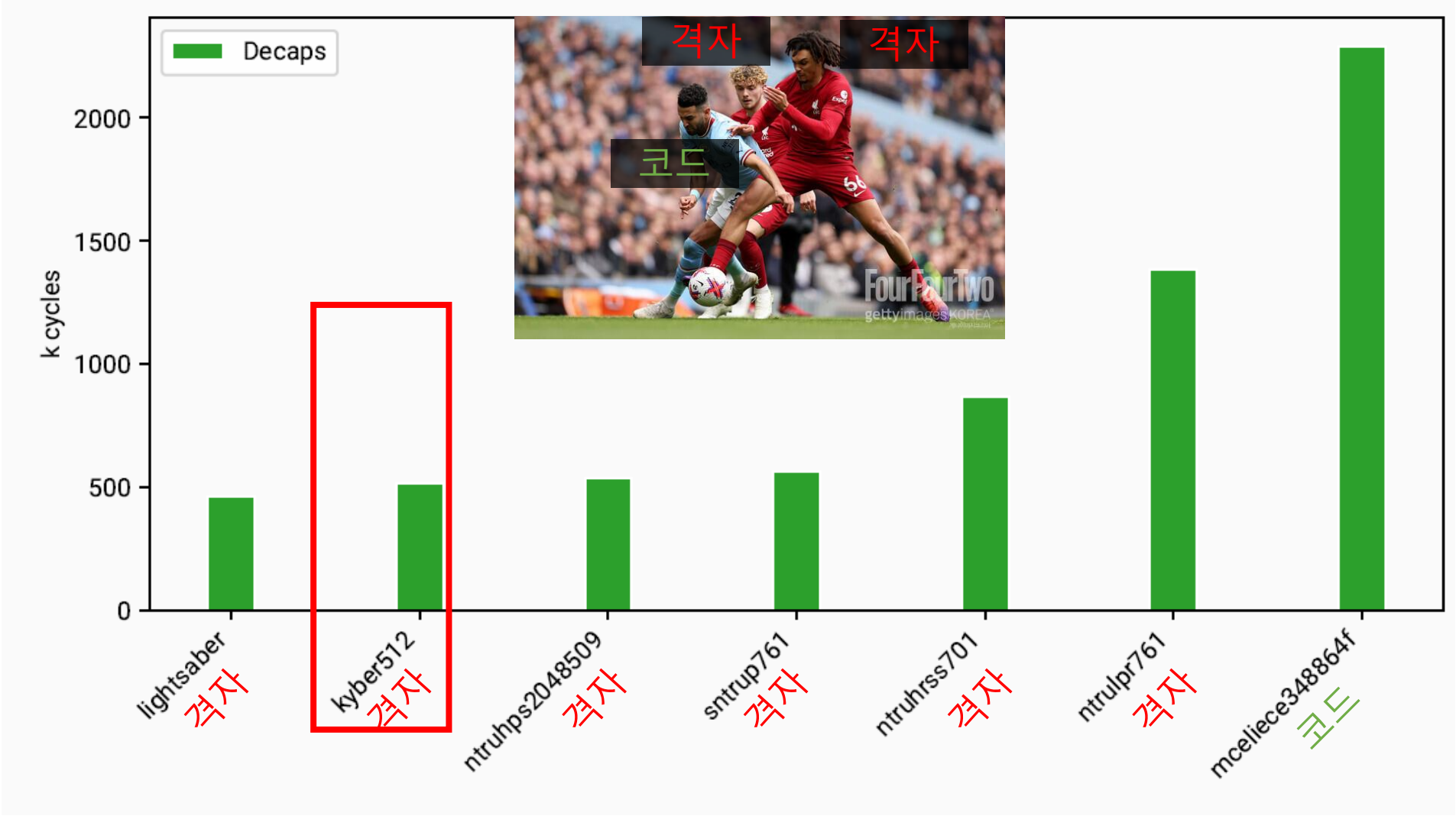
pqm4 결과를 중심으로 KEM Round 3 연산 성능 요약



pqm4 결과를 중심으로 KEM Round 3 연산 성능 요약



pqm4 결과를 중심으로 KEM Round 3 연산 성능 요약



여기서 잠깐 Lattice와 Code (+ 기타) 기반 암호를 연산 효율성 관점에서 바로 비교하는 것이 타당한가?

- Lattice는 NIST와 KpqC공모전에서 확인해 본 바와 같이 연산 효율성은 가장 우수함

- Lattice와 Code (+ 기타)는 특징과 용도가 다름
  - **마치 Sedan과 SUV가 다르듯이**

- 따라서 Lattice와 Code 및 기타는 기반 문제 별로 나누어 생각하는 것이 좋을 것 같음 (이러한 방향성이 NIST PQC에 적용)

- 뒤에 나오는 예제에서는 색깔별로 기반 문제들을 엮어서 나타냄
  - 빨강색: 격자, 초록색: 코드, 보라색: 다변수, 검은색: 기타

NIST 양자내성암호 공모전에서의 효율성 확인 수단

- **Open benchmark platform (일반 컴퓨터 환경)**
  - 벤치마크보다는 실제 활용성 관점에서 접근
  - 다만 C언어를 비롯하여 병렬화 언어 (AVX, ARM64)를 통한 구현 최적화
  - 이를 통해 OQS를 이용한 다양한 application 벤치마크에 활용

PQClean

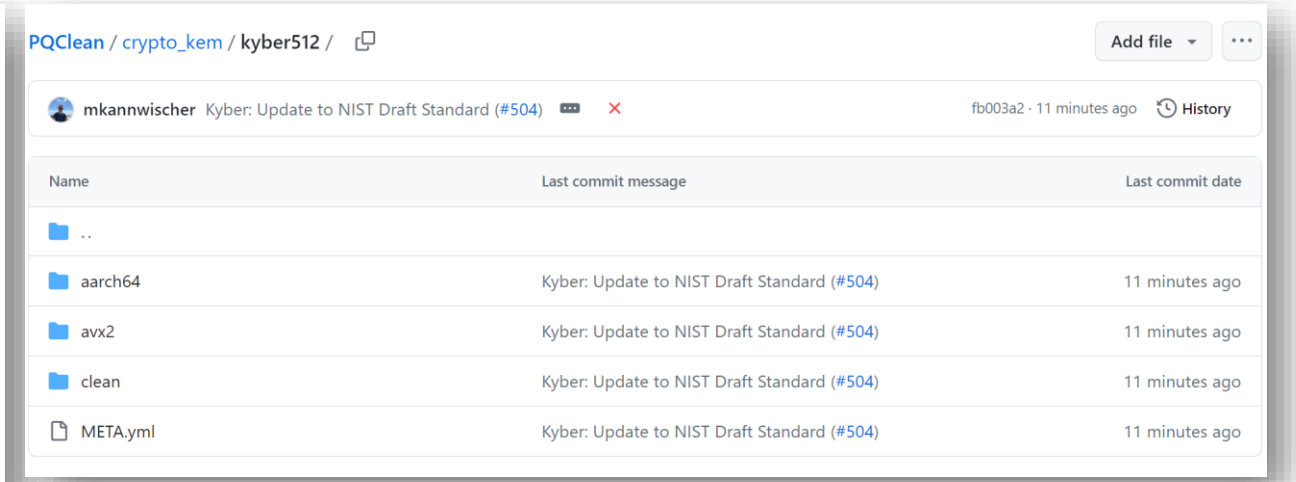
See the build status for each component here

PQClean, in short, is an effort to collect **clean** implementations of the post-quantum schemes that are in the NIST post-quantum project. The goal of PQClean is to provide *standalone implementations* that

- can easily be integrated into libraries such as liboqs.
- can efficiently upstream into higher-level protocol integration efforts such as Open Quantum Safe;
- can easily be integrated into benchmarking frameworks such as SUPERCOP;
- can easily be integrated into frameworks targeting embedded platforms such as pqm4;
- are suitable starting points for architecture-specific optimized implementations;
- are suitable starting points for evaluation of implementation security; and
- are suitable targets for formal verification.

What PQClean is **not** aiming for is

- a build system producing an integrated library of all schemes;
- including benchmarking of implementations; and
- including integration into higher-level applications or protocols.



KpqC에서 진행한 벤치마크

	KpqC	NIST PQC
공모전 기간	2022~2024	2017~2022
참여연구자	한국인 + 제한적인 외국인	전 세계인
벤치마크 플랫폼	고성능	고성능/저전력
벤치마크 언어	C	C, 어셈블리

- 제한적인 시간과 인력으로 인해
  NIST PQC와 **동일한 레벨에서의 비교분석에는 어려움**이 있음

- KpqClean을 통해서
  대략적인 성능 지표 도출을 통해 **상대적인 차이를 도출**하는 것에 집중
  - **절대적인 성능 지표 도출**은 알고리즘 선정이후에 지속적으로 추진 필요
    (해당 부분에 대한 연구는 암호가 사용되는 한 계속 개선가능)

벤치마크 언어에 따른 비교 예시

- 어셈블리 적용 시 성능은 일반적으로 2~3배 빨라지게 됨
- 단 **상대적인 성능 지표는 유지되는 경향**이 있음 (모든 암호가 다같이 좋아짐으로)

대한민국
양자내성암호에
대한 연산 성능 비교

KpqC에 대한 벤치마크 KpqClean

- **KpqC Round 1 후보 알고리즘들에 대한 공정한 벤치마킹**
  - 현재 각 알고리즘이 제시하는 벤치마크 결과는 **각기 다른 컴퓨터 환경에서 측정**
  - **동일한 환경에서 알고리즘들을 벤치마크**하여 객관성을 확보하는 것이 목표
  - 내부 모듈 (AES, SHA)을 통일하여 모듈에 따른 성능 차이를 최소화

- **KpqC 후보 알고리즘의 의존성 제거**
  - NIST 양자내성암호와 관련된 PQClean 프로젝트에서 추구한 방향성
  - 의존성 제거 시 **플랫폼에 구애받지 않고 알고리즘 가동 용이**
  - 개발 환경 설정을 용이하도록 하여 양자내성 알고리즘 벤치마크 진입장벽 완화

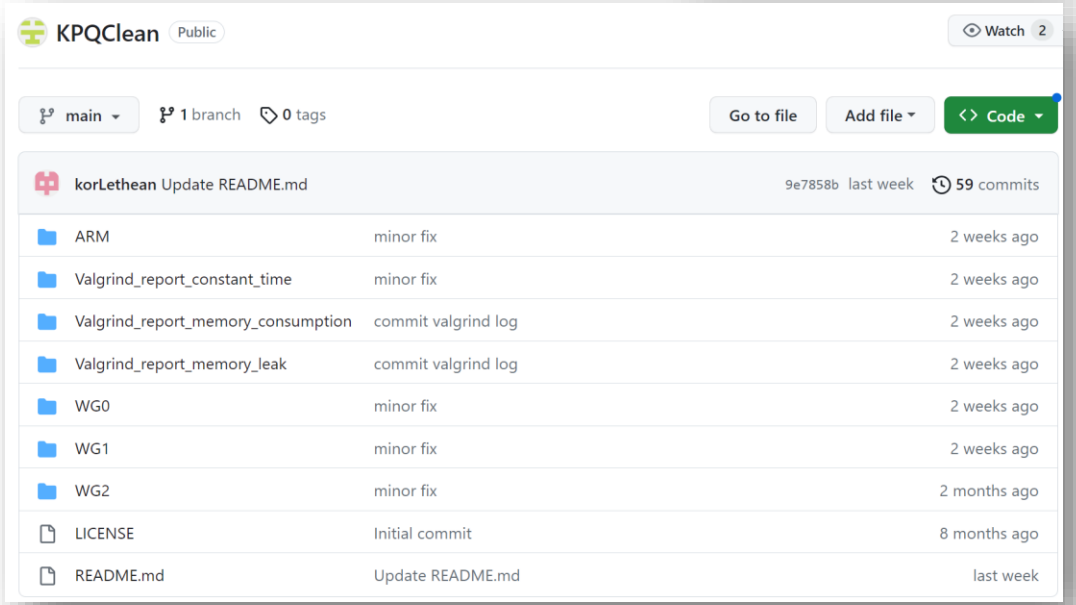
벤치마크 관련 내용 상세 (1 / 2)

https://github.com/kpqc-cryptocraft/KPQClean/tree/main

프로젝트 코드 주소

- **OpenSSL 의존성 제거**
  - 모든 알고리즘에서 **OpenSSL 의존성 제거**
  - gcc 컴파일러를 사용가능한 환경이면 소스코드 실행이 설정없이 가능
  - **추가적인 외부 라이브러리 의존성 제거 진행**

- **벤치마크 수행 사항**
  - Ryzen/Intel (Ubuntu OS) 상에서 벤치마크 진행
  - **ARM (MacOS) 상에서 추가적인 벤치마크 진행**
  - STACK 사용량, Constant time, 그리고 메모리 누수 확인
    - 해당 metric과 효율성의 상관관계는 뒷장에 이어서 설명



벤치마크 관련 내용 상세 (2 / 2)

- **STACK 사용량과 효율성**
  - STACK은 접근이 **빠른 휘발성 저장공간**
    → 전체 STACK 사용량은 임베디드 장비 상에서의 **동작 가능성 좌우 (불가능은 없고 불편함은 있고)**
  - 해당 저장공간은 연산 중간값 혹은 사전연산값 저장에 활용
    → **사전연산값은 연산/메모리 효율성에 영향을 미침**

- **Constant time과 효율성**
  - Constant time을 만족한다는 것은 경우에 따른 선택을 막는다는 것을 의미
  - 즉 어떤 선택을 할 때 **비밀값의 조건에 따르는 것이 아니라 항상 일정한 선택을 함**
  - 따라서 constant time을 만족하는 구현은 variable time보다 느릴 확률이 큼 (예외: bitslicing)

- **메모리 누수와 효율성**
  - 메모리 할당과 비할당 문제로 인해 **메모리가 낭비되거나 잘못된 참조를 수행**
  - 메모리 낭비가 될 경우 **메모리 스캐쥴링 (e.g. 페이징)으로 인한 비효율성 발생**가능
  - 잘못된 참조를 할 경우 연산 수행 시간이 **비정상적이 될 수도 있음 (e.g. program counter 조작)**

KpqC의 상대적인 효율성 평가

KpqC	
2021.11.25	공모전 공지
2022.02.18	제출마감
2022.03.15	1차 평가
2022.12	1라운드 결과 발표 2라운드 후보 목록 공개
2023.12	**2라운드 알고리즘 발표 예정**
2024.10	KpqC 공모전 최종결과 발표 예정

현재 제출된 KpqC 알고리즘 분류

- KEM: 7종류
- Digital Signature: 9종류

공개키 암호화 암호문 크기

암호문과 공개키의 크기는 네트워크 트래픽에 영향을 미침
e.g. 낮은 네트워크 가용성은 DDOS 공격에 보다 취약

- **Classic McEliece (NIST Round 4 후보)가 가장 작은 암호문 크기를 지님**
- **KpqC 알고리즘 내에서는 PALOMA 알고리즘이 가장 작은 암호문 크기를 가짐**
  - 국제 표준인 Kyber를 기준으로 한다면, 2,048 바이트 정도 까지가 안정적인 암호문 크기라고 할 수 있음

KpqC standards/candidates

NIST standards/candidates

Rank	PKE Scheme	Security level	Cipher size(Bytes)	Rank	PKE Scheme	Security level	Cipher size(Bytes)
1	mceliece348864	1	96	21	KYBER-768	3	1,088
2	PALOMA-128	1	136	22	NTRU+-768	1	1,152
3	mceliece460896	3	156	23	Layered ROLLO-I-256	5	1,286
4	mceliece6960119	5	194	24	NTRU+-864	3	1,296
5	mceliece6688128	5	208	25	REDOG-II	3	1,440
	mceliece8192128	5	208	26	SMAUG-256 (revised)	5	1,472
7	PALOMA-192	2	240	27	KYBER-1024	5	1,568
	PALOMA-256	3	240	28	NTRU+-1152	5	1,728
9	~~SIKE434~~	1	374	29	TIGER-192	3	2,048
10	~~SIKE503~~	2	434		TIGER-256	5	2,048
11	~~SIKE610~~	3	524	31	REDOG-III	5	2,230
11	Layered ROLLO-I-128	1	620	32	HQC-128	1	4,497
13	~~SIKE751~~	5	644	33	HQC-192	3	9,042
14	SMAUG-128 (revised)	1	672	34	BIKE-I	1	12,579
15	KYBER-512	1	768	35	HQC-256	5	14,485
16	REDOG-1	1	830	36	BIKE-III	3	24,915
17	Layered ROLLO-I-192	3	850	37	BIKE-V	5	41,229
18	NTRU+-576	1	864	38	IPCC-f1	1	322,000
19	TIGER-128	1	1,024		IPCC-f3	3	322,000
	SMAUG-192 (revised)	3	1,024		IPCC-f4	4	322,000

공개키 암호화 공개키 크기

암호문과 공개키의 크기는 네트워크 트래픽에 영향을 미침

- ~~SIKE (NIST Round 4 후보)~~가 가장 작은 공개키 크기를 지님
- **KpqC 알고리즘 내에서는 TiGER 알고리즘이 가장 작은 공개키 크기를 가짐**
  - Tiger는 KpqC 후보군 중에서 가장 작은 공개키 크기를 가지고 있음 (Kyber보다 작은 크기를 가지고 있음)
  - PALOMA는 KpqC 후보군 중에서 가장 큰 공개키 크기를 가지고 있으며, 이는 4라운드 후보인 McEliece와 비슷함

KpqC standards/candidates

NIST standards/candidates

Rank	PKE Scheme	Security level	Public key(byte)	Rank	PKE Scheme	Security level	Public key(byte)
1	~~SIKE434~~	1	330	21	Layered-ROLLO-I-256	5	2,571
2	~~SIKE503~~	2	378	22	IPCC-f1	1	3,600
3	~~SIKE610~~	3	462		IPCC-f3	3	3,600
4	TiGER-128	1	544		IPCC-f4	4	3,600
5	~~SIKE751~~	5	564	25	HQC-192	3	4,522
6	SMAUG-128 (revised)	1	672	26	HQC-256	5	7,245
7	KYBER-512	1	800	27	BIKE-I	1	12,323
8	NTRU+-576	1	864	28	REDOG-I	1	14,250
9	TiGER-192	3	1,056	29	BIKE-III	3	24,659
	TiGER-256	5	1,056	30	REDOG-II	3	32,840
11	SMAUG-192 (revised)	3	1,088	31	BIKE-V	5	40,973
12	NTRU+-768	1	1,152	32	REDOG-III	5	62,980
13	KYBER-768	3	1,184	33	mceliece348864	1	261,120
14	Layered-ROLLO-I-128	1	1,240	34	PALOMA-128	1	319,488
15	NTRU+-864	3	1,296	35	mceliece460896	3	524,160
16	KYBER-1024	5	1,568	36	PALOMA-192	2	812,032
17	Layered-ROLLO-I-192	3	1,699	37	PALOMA-256	3	1,025,024
18	NTRU+-1152	5	1,728	38	mceliece6688128	5	1,044,992
19	SMAUG-256 (revised)	5	1,792	39	mceliece6960119	5	1,047,319
20	HQC-128	1	2,249	40	mceliece8192128	5	1,357,824

공개키 암호화 비밀키 크기

- **KpqC 내에서는 Layered ROLLO가 가장 작은 비밀키 크기**를 지님
  - ROLLO의 암호문과 공개키 크기는 약간 작거나 중간 정도
- NIST 표준인 Kyber를 기준으로 한다면, **대부분의 알고리즘이 이상적인 비밀키** 크기를 지닌다고 할 수 있음
  - 예외로 PALOMA는 매우 큰 비밀키를 가지며, 특히 보안강도 1에 비해 2, 3의 비밀키가 매우 큼

KpqC standards/candidates

NIST standards/candidates

Rank	PKE Scheme	Security level	Private key(byte)	Rank	PKE Scheme	Security level	Private key(byte)
1	Layered-ROLLO-I-128	1	120	21	HQC-128	1	2,289
	Layered-ROLLO-I-192	3	120	22	NTRU+-768	1	2,304
	Layered-ROLLO-I-256	5	120	**23**	**KYBER-768**	**3**	**2,400**
4	SMAUG-128 (revised)	1	174	24	REDOG-II	3	2,520
5	SMAUG-256 (revised)	5	208	25	NTRU+-864	3	2,592
6	SMAUG-192 (revised)	3	230	**26**	**KYBER-1024**	**5**	**3,168**
7	~~SIKE434~~	1	374	27	BIKE-III	3	3,346
8	IPCC-f1	1	400	28	NTRU+-1152	5	3,456
	IPCC-f3	3	400	29	REDOG-III	5	3,890
	IPCC-f4	4	400	30	HQC-192	3	4,562
11	~~SIKE503~~	2	434	31	BIKE-V	5	4,640
12	~~SIKE610~~	3	524	32	mceliece348864	1	6,492
13	TiGER-128	1	528	33	HQC-256	5	7,285
14	~~SIKE751~~	5	644	34	mceliece460896	3	13,608
15	TiGER-192	3	1,056	35	mceliece6688128	5	13,932
	TiGER-256	5	1,056	36	mceliece6960119	5	13,948
17	REDOG-I	1	1,450	37	mceliece8192128	5	14,120
18	**KYBER-512**	**1**	**1,632**	38	PALOMA-128	1	93,008
19	NTRU+-576	1	1,728	39	PALOMA-192	2	355,400
20	BIKE-I	1	2,244	40	PALOMA-256	3	357,064

전자서명 서명 크기

서명과 공개키의 크기는 네트워크 트래픽에 영향을 미침

- **MQSign이 가장 작은 서명 크기**를 지님
  - 모든 서명 스킴이 다른 알고리즘에 비해서 가장 작은 크기를 가짐
- **NIST Standard 중에서 Dilithium이 서명 크기가 큰 편에 속함**
  - **KpqC 알고리즘은 대체로 Dilithium보다 작은 서명 크기를 가짐**
  - 따라서 KpqC 서명 후보군의 대부분 적당한 서명 크기를 지닌다고 할 수 있음

KpqC standards/candidates

NIST standards/candidates

Rank	Digital Signature Scheme	Security level	Sig size(Bytes)	Rank	Digital Signature Scheme	Security level	Sig size(Bytes)
1	MQSign-72-46	1	134	19	GCKSign-V	5	3,104
2	MQSign-112-72	3	200	20	NCCSign-I (conserparam)	1	3,186
3	MQSign-148-96	5	260	**21**	Dilithium-III	**3**	**3,293**
4	pqsigRM-613	5	528	22	NCCSign-III (original)	3	3,605
5	Falcon-512	1	666	23	NCCSign-III (conserparam)	3	4,251
	SOLMAE-512	1	666	**24**	Dilithium-V	**5**	**4,595**
	Peregrine-512	1	666	25	NCCSign-V (original)	5	5,055
8	pqsigRM-612	1	1,040	26	NCCSign-V (conserparam)	5	5,385
9	Falcon-1024	5	1,280	27	AlMer-I (revised, PARAM1)	1	5,904
	Peregrine-1024	5	1,280	28	SPHINCS+ 128s	1	7,856
11	SOLMAE-1024	5	1,375	29	AlMer-III (revised, PARAM1)	3	13,080
12	HAETAE-II (revised)	2	1,463	30	SPHINCS+ 192s	3	16,224
13	GCKSign-II	2	1,952	31	SPHINCS+ 128f	1	17,088
14	GCKSign-III	3	2,080		FIPS	1	17,088
15	HAETAE-III (revised)	3	2,337	32	AlMer-V (revised, PARAM1)	5	25,152
16	Dilithium-II	**2**	**2,420**	33	SPHINCS+ 256s	5	29,792
17	NCCSign-I (original)	1	2,458	34	SPHINCS+ 192f	3	35,664
18	HAETAE-V (revised)	5	2,908	35	SPHINCS+ 256f	5	49,856

전자서명 공개키 크기

서명과 공개키의 크기는 네트워크 트래픽에 영향을 미침

- SPHINCS (NIST Round 4 후보)와 **FIPS가 가장 작은 공개키 크기를 지님**
- **AIMer 알고리즘도 거의 비슷한 크기를 가짐**
  - AIMer는 KpqC 후보군 중에서 **가장 큰 서명 크기를 가지지만, 공개키 크기는 가장 작음**
  - **FIPS도 매우 큰 서명 크기를 가짐**
  - 파라미터가 크다고 평가되는 Dilithium을 고려한다면, 2,048~4,096 바이트의 공개키가 적절한 크기로 생각할 수 있음
  - **MQSign은 서명 크기는 가장 작았으나 공개키 크기는 가장 큰 편**에 속함

KpqC standards/candidates

NIST standards/candidates

Rank	Digital Signature Scheme	Security level	Public key(byte)	Rank	Digital Signature Scheme	Security level	Public key(byte)
1	SPHINCS+128	1	32	18	FALCON-1024	5	1,793
	FIPS	1	32	**19**	Dilithium-III	**3**	**1,952**
	AIMer-I (revised)	1	32	20	GCKSign-III	3	1,952
4	SPHINCS+192	3	48	21	NCCSign-I (conserparam)	1	1,984
	AIMer-III (revised)	3	48	22	NCCSign-III (original)	3	1,997
6	SPHINCS+256	5	64	23	HAETAE-V (revised)	5	2,080
	AIMer-V (revised)	5	64	24	NCCSign-III (conserparam)	3	2,443
8	SOLMAE-512	1	896	**25**	Dilithium-V	**5**	**2,592**
9	Peregrine-512	1	897	26	NCCSign-V (original)	5	2,663
	FALCON-512	1	897	27	GCKSign-V	5	3,040
11	HAETAE-II (revised)	2	992	28	NCCSign-V (conserparam)	5	3,091
12	Dilithium-II	**2**	**1,312**	29	MQSign-72-46	1	328,411
13	HAETAE-III (revised)	3	1,472	30	MQSign-112-72	3	1,238,761
14	NCCSign-I (original)	1	1,564	31	pqsigRM-613	5	1,285,120
15	GCKSign-II	2	1,760	32	MQSign-148-96	5	2,892,961
16	SOLMAE-1024	5	1,792	33	pqsigRM-612	1	4,194,304
17	Peregrine-1024	1	1,793				

전자서명 비밀키 크기

- **AIMer가 가장 작은 비밀키 크기**를 지님
  - AIMer는 모든 KpqC 후보군 중에서도 가장 작은 공개키 크기를 가짐
  - **MQSign은 서명 크기가 가장 작으나, 비밀키 크기는 가장 큰 편**에 속함
- **파라미터가 크다고 알려진 Dilithium을 기준으로 보면 대부분의 KpqC 알고리즘은 비밀키 크기가 이상적임**
  - 두 알고리즘 (MQSign, pqsigRM)만 비밀키 크기가 큼

KpqC standards/candidates

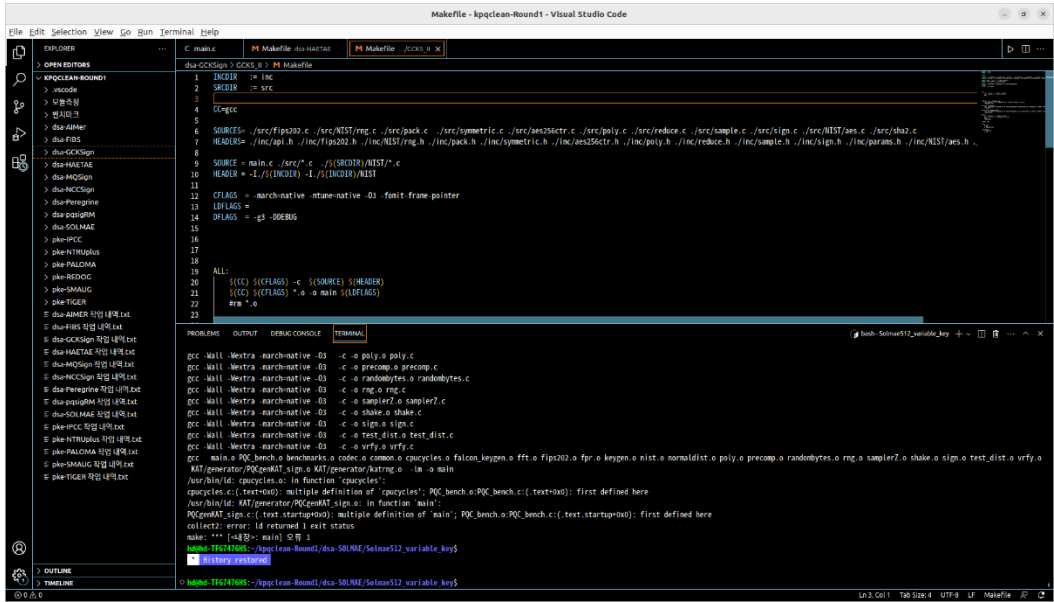
NIST standards/candidates

Rank	Digital Signature Scheme	Security level	Private key(byte)	Rank	Digital Signature Scheme	Security level	Private key(byte)
1	AIMer-I (revised)	1	16	18	Peregrine-1024	5	2,305
2	AIMer-III (revised)	3	24		FALCON-1024	5	2,305
3	AIMer-V (revised)	5	32	20	Dilithium-II	2	2,528
4	SPHINCS+128	1	64	21	HAETAE-V (revised)	5	2,720
	FIPS	1	64	22	NCCSign-I (conserparam)	1	2,800
6	SPHINCS+192	3	96	23	NCCSign-III (original)	3	3,312
7	SPHINCS+256	5	128	24	NCCSign-III (conserparam)	3	3,914
8	GCKSign-II	2	288	25	Dilithium-III	3	4,000
	GCKSign-III	3	288	26	NCCSign-V (original)	5	4,402
10	GCKSign-V	5	544	27	Dilithium-V	5	4,864
11	SOLMAE-512	1	1,281	28	NCCSign-V(conserparam)	5	4,940
	Peregrine-512	1	1,281	29	MQSign-72-46	1	15,561
	FALCON-512	1	1,281	30	pqsigRM-612	1	24,592
14	HAETAE-II (revised)	2	1,376	31	MQSign-112-72	3	37,729
15	HAETAE-III (revised)	3	2,080	32	MQSign-148-96	5	66,421
16	NCCSign-I (original)	1	2,266	33	pqsigRM-613	5	331,074
17	SOLMAE-1024	5	2,305				

알고리즘 성능 측정 결과

- **벤치마크 테스트 환경**
  - **Ubuntu 22.04, Ryzen 7 4800H, RTX 3060, 16GB RAM**
  - **Ubuntu 22.04, Intel i5-8259U, Coffee Lake GT3e, 16GB RAM**
  - **macOS Ventura 13.5.2, Apple M1 Pro, 16GB RAM**

- **IDE: Visual Studio Code**
- **Compiler: gcc 11.3.0, Apple clang 14.0.3**
- **Optimization Level: O3 에서 테스트**



벤치마크 수행 (Intel 그리고 ARM)

- **Intel 상에서의 성능과 ARM 상에서의 성능은 비슷한 경향을 보임**
  - **Intel에서 빠르게 동작한 알고리즘은 ARM에서도 빠르게 동작**
  - Intel/ARM의 ALU가 유사함을 의미
- **동작 과정에서 AVX가 포함된 알고리즘은 ARM에서 구동 불가**
  - **ARM은 AVX를 지원하지 않음**
  - AVX를 비활성화 할 수 있는 알고리즘에 한해 측정을 진행

    →일부 알고리즘은 AVX가 비활성화 될 경우 오동작

PKE/KEM 인텔(-O3) 성능 측정 표

Unit: clock cycles

Type	Algorithm	Keygen(Med.)	Encapsulation (Med.)	Decapsulation (Med.)	Keygen(Avr.)	Encapsulation (Avr.)	Decapsulation (Avr.)
Lattice	NTRUplus-576	162,956	94,981	93,298	241,793	101,155	99,479
Lattice	NTRUplus-768	250,592	119,006	122,437	271,454	127,308	132,202
Lattice	NTRUplus-864	240,904	138,104	149,058	261,313	147,497	160,356
Lattice	NTRUplus-1152	517,578	172,660	198,334	617,132	184,218	212,901
Lattice	SMAUG-128	63,020	49,324	39,196	65,919	55,873	42,528
Lattice	SMAUG-192	92,658	69,739	67,691	95,436	74,836	70,950
Lattice	SMAUG-256	135,202	122,766	115,096	142,842	128,734	118,789
Lattice	TiGER-128	55,092	39,850	45,977	60,904	44,451	51,154
Lattice	TiGER-192	65,530	58,926	55,800	73,066	66,244	66,563
Lattice	TiGER-256	73,902	74,756	73,548	80,087	88,262	84,296
Code	PALOMA-128	108,402,198	459,846	40,838	108,597,537	473,532	42,840
Code	PALOMA-192	108,206,652	460,374	40,688	108,344,570	472,432	42,798
Code	PALOMA-256	108,216,713	459,880	40,886	108,461,853	465,766	41,780
Code	IPCC-1	12,643,392	145,233,220	1,159,273	12,712,124	210,977,105	1,185,580
Code	IPCC-3	12,795,377	874,663	1,206,585	12,874,291	922,533	1,267,783
Code	IPCC-4	13,078,917	1,037,485	1,310,503	13,250,237	1,107,017	1,368,035
Code	Layered ROLLO I-128	203,181	66,529	558,503	231,523	77,774	602,966
Code	Layered ROLLO I-192	227,813	102,758	671,605	255,243	125,567	761,739
Code	Layered ROLLO I-256	375,056	136,052	1,245,346	455,911	146,919	1,337,504

PKE/KEM ARM(-O3) 성능 측정 표

Unit: Nano Seconds

Type	Algorithm	Keygen(Med.)	Encapsulation(Med.)	Decapsulation (Med.)	Keygen(Avr.)	Encapsulation (Avr.)	Decapsulation (Avr.)
Lattice	NTRUplus-576	119,040	46,848	36,864	124,979	47,508	36,682
Lattice	NTRUplus-768	87,040	46,080	38,912	76,460	46,359	38,684
Lattice	NTRUplus-864	59,136	37,888	33,024	59,668	38,484	32,640
Lattice	NTRUplus-1152	273,024	73,984	61,952	297,897	71,583	61,501
Lattice	SMAUG-128	51,200	46,080	38,912	61,448	49,454	38,013
Lattice	SMAUG-192	82,944	72,960	46,848	91,505	73,039	47,158
Lattice	SMAUG-256	64,000	70,912	79,104	76,370	69,309	77,965
Lattice	TiGER-128	54,016	56,064	44,032	54,369	56,271	44,413
Lattice	TiGER-192	64,000	88,960	75,008	62,359	94,866	94,733
Lattice	TiGER-256	70,912	125,952	99,968	68,004	128,740	103,688
Code	PALOMA-128	34,776,064	155,008	8,960	34,915,917	153,480	9,236
Code	PALOMA-192	34,742,912	146,176	8,960	35,071,649	147,387	9,359
Code	PALOMA-256	34,691,968	155,008	8,960	34,935,721	154,355	9,062
Code	IPCC-1	4,171,904	7,326,464	230,144	4,445,222	11,629,652	243,666
Code	IPCC-3	4,205,952	191,207,040	246,400	4,293,573	277,514,522	277,967
Code	IPCC-4	4,153,984	233,392,384	234,624	4,281,823	317,245,691	255,708

벤치마크 수행 (PKE/KEM)

- **키 생성 성능 순위 (-O3 기준)**
  - **Intel:** TiGER > SMAUG > NTRU+ > Layered ROLLO I > IPCC > PALOMA
  - **ARM:** SMAUG > TiGER > NTRU+ > IPCC > PALOMA

키생성, 암호화, 복호화가 활용되는 영역은 프로토콜에 따라 상이함

따라서 성능은 프로토콜 디펜던시가 존재함

Rank	Algorithm(Intel)	Keygen(Med., cc)	Rank	Algorithm(ARM)	Keygen(Med., nsec)
1	TiGER-128	55,092	1	SMAUG-128	51,200
2	SMAUG-128	63,020	2	TiGER-128	54,016
3	TiGER-192	65,530	3	NTRUplus-864	59,136
4	TiGER-256	73,902	4	SMAUG-256	64,000
5	SMAUG-192	92,658	5	TiGER-192	64,000
6	SMAUG-256	135,202	6	TiGER-256	70,912
7	NTRUplus-576	162,956	7	SMAUG-192	82,944
8	Layered ROLLO I-128	203,181	8	NTRUplus-768	87,040
9	Layered ROLLO I-192	227,813	9	NTRUplus-576	119,040
10	NTRUplus-864	240,904	10	NTRUplus-1152	273,024
11	NTRUplus-768	250,592	11	IPCC-4	4,153,984
12	Layered ROLLO I-256	375,056	12	IPCC-1	4,171,904
13	NTRUplus-1152	517,578	13	IPCC-3	4,205,952
14	IPCC-1	12,643,392	14	PALOMA-256	34,691,968
15	IPCC-3	12,795,377	15	PALOMA-192	34,742,912
16	IPCC-4	13,078,917	16	PALOMA-128	34,776,064
17	PALOMA-192	108,206,652			
18	PALOMA-256	108,216,713			
19	PALOMA-128	108,402,198			

벤치마크 수행 (PKE/KEM)

- **암호화 성능 순위 (-O3 기준)**
  - **Intel:** TiGER > SMAUG > Layered ROLLO I > NTRU+ > PALOMA > IPCC
  - **ARM:** NTRU+ > SMAUG > TiGER > PALOMA > IPCC

Rank	Algorithm(Intel)	Enc(Med., cc)	Rank	Algorithm(ARM)	Enc(Med., nsec)
1	TiGER-128	39,850	1	NTRUplus-864	37,888
2	SMAUG-128	49,324	2	NTRUplus-768	46,080
3	TiGER-192	58,926	3	SMAUG-128	46,080
4	Layered ROLLO I-128	66,529	4	NTRUplus-576	46,848
5	SMAUG-192	69,739	5	TiGER-128	56,064
6	TiGER-256	74,756	6	SMAUG-256	70,912
7	NTRUplus-576	94,981	7	SMAUG-192	72,960
8	Layered ROLLO I-192	102,758	8	NTRUplus-1152	73,984
9	NTRUplus-768	119,006	9	TiGER-192	88,960
10	SMAUG-256	122,766	10	TiGER-256	125,952
11	Layered ROLLO I-256	136,052	11	PALOMA-192	146,176
12	NTRUplus-864	138,104	12	PALOMA-128	155,008
13	NTRUplus-1152	172,660	13	PALOMA-256	155,008
14	PALOMA-128	459,846	14	IPCC-1	7,326,464
15	PALOMA-256	459,880	15	IPCC-3	191,207,040
16	PALOMA-192	460,374	16	IPCC-4	233,392,384
17	IPCC-3	874,663			
18	IPCC-4	1,037,485			
19	IPCC-1	145,233,220			

벤치마크 수행 (PKE/KEM)

- **복호화 성능 순위 (-O3 기준)**
  - **Intel:** SMAUG >= PALOMA > TiGER > NTRU+ > Layered ROLLO I > IPCC
  - **ARM:** PALOMA > NTRU+ > SMAUG > TiGER > IPCC

Rank	Algorithm(Intel)	Dec(Med., cc)
1	SMAUG-128	39,196
2	PALOMA-192	40,688
3	PALOMA-128	40,838
4	PALOMA-256	40,886
5	TiGER-128	45,977
6	TiGER-192	55,800
7	SMAUG-192	67,691
8	TiGER-256	73,548
9	NTRUplus-576	93,298
10	SMAUG-256	115,096
11	NTRUplus-768	122,437
12	NTRUplus-864	149,058
13	NTRUplus-1152	198,334
14	Layered ROLLO I-128	558,503
15	Layered ROLLO I-192	671,605
16	IPCC-1	1,159,273
17	IPCC-3	1,206,585
18	Layered ROLLO I-256	1,245,346
19	IPCC-4	1,310,503

Rank	Algorithm(ARM)	Dec(Med., nsec)
1	PALOMA-128	8,960
2	PALOMA-192	8,960
3	PALOMA-256	8,960
4	NTRUplus-864	33,024
5	NTRUplus-576	36,864
6	NTRUplus-768	38,912
7	SMAUG-128	38,912
8	TiGER-128	44,032
9	SMAUG-192	46,848
10	NTRUplus-1152	61,952
11	TiGER-192	75,008
12	SMAUG-256	79,104
13	TiGER-256	99,968
14	IPCC-1	230,144
15	IPCC-4	234,624
16	IPCC-3	246,400

Digital Signature 인텔(-O3) 성능 측정 표 (파트1)

Unit: clock cycles

Type	Algorithm	Keygen(Med.)	Signature(Med.)	Verification(Med.)	Keygen(Avr.)	Signature(Avr.)	Verification(Avr.)
Zero-knowledge	AIMer-I-param1	78,212	3,799,913	3,125,105	94,251	4,000,548	3,318,324
Zero-knowledge	AIMer-I-param2	62,013	8,202,991	7,719,642	67,545	8,623,032	8,299,636
Zero-knowledge	AIMer-I-param3	59,949	28,891,133	26,007,825	73,077	29,338,289	25,865,036
Zero-knowledge	AIMer-I-param4	59,132	125,595,827	111,966,098	73,953	127,933,918	118,860,248
Zero-knowledge	AIMer-III-param1	150,830	9,550,533	8,247,989	167,334	9,563,875	8,456,415
Zero-knowledge	AIMer-III-param2	111,755	22,535,406	22,149,494	131,077	22,879,148	22,107,888
Zero-knowledge	AIMer-III-param3	109,995	67,033,442	54,668,451	136,612	66,512,279	57,154,915
Zero-knowledge	AIMer-III-param4	141,443	271,749,297	266,189,006	165,637	284,501,178	267,296,086
Zero-knowledge	AIMer-V-param1	255,129	14,687,409	13,724,601	295,481	16,357,037	15,402,041
Zero-knowledge	AIMer-V-param2	285,212	43,671,802	34,428,290	294,113	43,211,961	35,975,358
Zero-knowledge	AIMer-V-param3	290,019	115,523,707	109,326,832	292,918	122,646,177	109,994,189
Zero-knowledge	AIMer-V-param4	243,628	557,052,072	521,735,231	313,602	581,221,074	526,630,666
Multivariate Quadratic	MQSign-72/46	38,474,591	298,952	533,676	38,612,360	308,203	547,680
Multivariate Quadratic	MQSign-112/72	117,049,542	650,928	1,120,124	117,234,338	667,681	1,147,333
Multivariate Quadratic	MQSign-148/96	236,124,011	1,165,706	1,897,664	236,332,422	1,173,558	1,908,458

Digital Signature 인텔(-O3) 성능 측정 표 (파트2)

Unit: clock cycles

Type	Algorithm	Keygen(Med.)	Signature(Med.)	Verification(Med.)	Keygen(Avr.)	Signature(Avr.)	Verification(Avr.)
Lattice	GCKSign-II	175,993	597,712	172,893	188,999	869,677	182,127
Lattice	GCKSign-III	183,987	698,941	179,608	223,689	976,483	186,837
Lattice	GCKSign-V	238,884	928,251	262,868	259,401	1,228,167	293,133
Lattice	HAETAE-II	672,901	3,334,242	126,972	944,910	4,200,552	132,300
Lattice	HAETAE-III	1,291,292	8,261,232	227,780	1,828,235	8,769,910	238,478
Lattice	HAETAE-V	719,708	2,627,334	270,600	973,865	3,546,813	280,493
Lattice	NCCSign-II(original)	1,666,543	16,352,341	3,248,162	1,846,947	16,530,887	3,321,373
Lattice	NCCSign-III(original)	3,141,974	34,454,252	6,234,249	3,227,617	34,523,301	6,288,505
Lattice	NCCSign-V(original)	5,613,303	167,158,023	11,155,020	5,851,360	167,337,719	11,307,818
Lattice	NCCSign-II(conserparam)	2,317,555	13,776,448	4,568,006	2,393,641	13,868,809	4,647,302
Lattice	NCCSign-III(conserparam)	3,981,551	83,521,123	7,935,382	4,209,101	83,634,184	8,001,129
Lattice	NCCSign-V(conserparam)	6,333,006	25,183,392	12,555,623	6,470,472	25,269,299	12,680,799
Lattice	Peregrine-512	11,783,005	260,328	26,262	12,032,320	269,678	28,484
Lattice	Peregrine-1024	37,875,534	551,168	55,654	40,364,494	569,794	58,474
Lattice	SOLMAE-512	22,627,042	332,848	64,838	27,866,035	348,841	67,662
Lattice	SOLMAE-1024	53,245,753	668,103	149,168	67,369,725	686,523	154,073
Code	pqsigRM-613	4,702,612,115	4,732,706	2,064,731	4,703,836,987	6,667,564	2,458,625
Code	pqsigRM-612	71,111,088,778	923,513	417,658	71,168,430,985	1,166,665	502,448

Digital Signature ARM(-O3) 성능 측정 표

* NCCSign은 AVX가 없지만 ARM에서 무한루프가 발생하여 측정 불가

Unit: Nano Seconds

Type	Algorithm	Keygen(Med.)	Signature(Med.)	Verification(Med.)	Keygen(Avr.)	Signature(Avr.)	Verification(Avr.)
Zero-knowledge	AIMer-I-param1	44,032	899,968	647,040	48,655	1,047,603	651,395
Zero-knowledge	AIMer-I-param2	52,992	1,698,048	1,593,472	54,157	1,909,775	1,605,245
Zero-knowledge	AIMer-I-param3	52,992	5,171,584	5,123,072	54,638	5,427,658	5,137,475
Zero-knowledge	AIMer-I-param4	52,992	25,836,800	25,750,528	54,533	26,125,240	25,846,420
Zero-knowledge	AIMer-III-param1	94,976	1,921,536	1,597,952	90,442	2,120,712	1,611,597
Zero-knowledge	AIMer-III-param2	94,976	4,444,160	4,254,080	91,876	4,682,583	4,271,025
Zero-knowledge	AIMer-III-param3	90,496	12,902,016	12,858,624	88,392	13,133,614	12,907,110
Zero-knowledge	AIMer-III-param4	77,056	63,625,472	63,475,584	85,435	63,923,018	63,566,275
Zero-knowledge	AIMer-V-param1	150,016	3,278,976	2,849,408	156,132	3,461,350	2,870,072
Zero-knowledge	AIMer-V-param2	151,040	7,632,128	7,357,952	165,619	7,812,086	7,382,221
Zero-knowledge	AIMer-V-param3	80,128	22,604,416	22,692,992	87,229	22,887,040	22,764,360
Zero-knowledge	AIMer-V-param4	150,016	110,281,984	110,957,440	168,602	110,695,219	111,235,878
Multivariate Quadratic	MQSign-72/46	25,404,416	140,032	431,104	25,694,354	140,605	436,836
Multivariate Quadratic	MQSign-112/72	135,705,472	380,160	1,565,952	136,245,245	385,275	1,575,908
Multivariate Quadratic	MQSign-148/96	411,995,904	795,136	3,660,416	412,772,841	802,437	3,674,153
Lattice	GCKSign-II	102,912	254,976	50,944	108,541	320,015	51,244
Lattice	GCKSign-III	108,032	238,080	54,016	118,446	315,105	54,879
Lattice	GCKSign-V	142,080	264,576	75,008	144,274	352,781	75,077
Lattice	HAETAE-II	357,120	208,384	40,960	485,315	367,772	43,341
Lattice	HAETAE-III	428,544	450,432	72,960	718,948	656,292	75,410
Lattice	HAETAE-V	403,968	432,000	91,136	486,986	618,040	94,843
Lattice	Peregrine-512	3,891,456	45,056	5,120	4,175,012	45,507	5,179
Lattice	Peregrine-1024	13,422,464	93,952	9,984	14,907,274	95,708	10,150
Code	pqsigRM-613	793,679,488	2,152,960	379,392	793,098,163	2,555,494	405,683
Code	pqsigRM-612	10,549,565,568	329,600	72,064	10,564,255,590	271,821	76,672

벤치마크 수행 (Digital Signature)

- **키 생성 성능 순위 (-O3 기준)**
  - **Intel:** AIMer > GCKSign > HAETAE > NCCSign > Peregrine >= SOLMAE > MQSign > Enhanced pqsigRM
  - **ARM:** AIMer > GCKSign > HAETAE > Peregrine > MQSign > Enhanced pqsigRM
    - Intel과 ARM의 성능 순위가 동일함

Rank	Algorithm(Intel)	Keygen(Med., cc)	Rank	Algorithm(Intel)	Keygen(Med., cc)
1	AIMer-I-param2	63,375	18	HAETAE-III	1,352,577
2	AIMer-I-param1	63,933	19	NCCSign-II(ori.)	1,704,190
3	AIMer-I-param4	84,224	20	NCCSign-II(con.)	2,296,351
4	AIMer-III-param3	113,296	21	NCCSign-III(ori.)	3,271,119
5	AIMer-III-param1	114,714	22	NCCSign-III(con.)	4,009,717
6	AIMer-I-param3	120,915	23	NCCSign-V(ori.)	5,723,169
7	AIMer-III-param2	128,547	24	NCCSign-V(con.)	6,561,582
8	AIMer-III-param4	149,336	25	Peregrine-512	12,073,005
9	GCKSign-II	171,176	26	SOLMAE-512	22,494,902
10	GCKSign-III	173,252	27	Peregrine-1024	38,493,479
11	AIMer-V-param1	238,347	28	SOLMAE-1024	52,388,360
12	AIMer-V-param2	241,737	29	MQSign-72/46	87,038,447
13	GCKSign-V	248,629	30	MQSign-112/72	448,271,119
14	AIMer-V-param4	283,645	31	MQSign-148/96	1,326,638,494
15	AIMer-V-param3	299,868	32	pqsigRM-613	4,961,556,899
16	HAETAE-II	700,875	33	pqsigRM-612	74,021,054,015
17	HAETAE-V	752,413			

Rank	Algorithm(ARM)	Keygen(Med., nsec)	Rank	Algorithm(ARM)	Keygen(Med., nsec)
1	AIMer-I-param1	44,032	14	AIMer-V-param4	150,016
2	AIMer-I-param2	52,992	15	AIMer-V-param2	151,040
3	AIMer-I-param3	52,992	16	HAETAE-II	357,120
4	AIMer-I-param4	52,992	17	HAETAE-V	403,968
5	AIMer-III-param4	77,056	18	HAETAE-III	428,544
6	AIMer-V-param3	80,128	19	Peregrine-512	3,891,456
7	AIMer-III-param3	90,496	20	Peregrine-1024	13,422,464
8	AIMer-III-param1	94,976	21	MQSign-72/46	25,404,416
9	AIMer-III-param2	94,976	22	MQSign-112/72	135,705,472
10	GCKSign-II	102,912	23	MQSign-148/96	411,995,904
11	GCKSign-III	108,032	24	pqsigRM-613	793,679,488
12	GCKSign-V	142,080	25	pqsigRM-612	10,549,565,568
13	AIMer-V-param1	150,016			

벤치마크 수행 (Digital Signature)

- **서명 생성 성능 순위 (-O3 기준)**
  - **Intel:** Peregrine >= MQSign >= SOLMAE > GCKSign >= Enhanced pqsigRM >= HAETAE > AIMer >= NCCSign
  - **ARM:** Peregrine > GCKSign >= HAETAE > MQSign >= Enhanced pqsigRM > AIMer
    - 특정 알고리즘이 모든 파라미터에서 성능을 압도하는 경우가 적음

Rank	Algorithm(Intel)	Sign(Med., cc)	Rank	Algorithm(Intel)	Sign(Med., cc)
1	Peregrine-512	260,328	18	AIMer-III-param1	9,550,533
2	MQSign-72/46	298,952	19	NCCSign-II(con.)	13,776,448
3	SOLMAE-512	332,848	20	AIMer-V-param1	14,687,409
4	Peregrine-1024	551,168	21	NCCSign-II(ori.)	16,352,341
5	GCKSign-II	597,712	22	AIMer-III-param2	22,535,406
6	MQSign-112/72	650,928	23	NCCSign-V(con.)	25,183,392
7	SOLMAE-1024	668,103	24	AIMer-I-param3	28,891,133
8	GCKSign-III	698,941	25	NCCSign-III(ori.)	34,454,252
9	pqsigRM-612	923,513	26	AIMer-V-param2	43,671,802
10	GCKSign-V	928,251	27	AIMer-III-param3	67,033,442
11	MQSign-148/96	1,165,706	28	NCCSign-III(con.)	83,521,123
12	HAETAE-V	2,627,334	29	AIMer-V-param3	115,523,707
13	HAETAE-II	3,334,242	30	AIMer-I-param4	125,595,827
14	AIMer-I-param1	3,799,913	31	NCCSign-V(ori.)	167,158,023
15	pqsigRM-613	4,732,706	32	AIMer-III-param4	271,749,297
16	AIMer-I-param2	8,202,991	33	AIMer-V-param4	557,052,072
17	HAETAE-III	8,261,232			

Rank	Algorithm(ARM)	Sign(Med., nsec)	Rank	Algorithm(ARM)	Sign(Med., nsec)
1	Peregrine-512	45,056	14	AIMer-I-param2	1,698,048
2	Peregrine-1024	93,952	15	AIMer-III-param1	1,921,536
3	MQSign-72/46	140,032	16	pqsigRM-613	2,152,960
4	HAETAE-II	208,384	17	AIMer-V-param1	3,278,976
5	GCKSign-III	238,080	18	AIMer-III-param2	4,444,160
6	GCKSign-II	254,976	19	AIMer-I-param3	5,171,584
7	GCKSign-V	264,576	20	AIMer-V-param2	7,632,128
8	pqsigRM-612	329,600	21	AIMer-III-param3	12,902,016
9	MQSign-112/72	380,160	22	AIMer-V-param3	22,604,416
10	HAETAE-V	432,000	23	AIMer-I-param4	25,836,800
11	HAETAE-III	450,432	24	AIMer-III-param4	63,625,472
12	MQSign-148/96	795,136	25	AIMer-V-param4	110,281,984
13	AIMer-I-param1	899,968			

벤치마크 수행 (Digital Signature)

- **서명 검증 성능 순위 (-O3 기준)**
  - **Intel:** Peregrine > SOLMAE >= HAETAE > GCKSign > Enhanced pqsigRM >= MQSign > NCCSign > AIMer
  - **ARM:** Peregrine > HAETAE >= GCKSign >= Enhanced pqsigRM > MQSign > AIMer
    - Intel과 ARM에서의 성능 순위가 거의 비슷하게 유지

Rank	Algorithm(Intel)	Verify(Med., cc)	Rank	Algorithm(Intel)	Verify(Med., cc)
1	Peregrine-512	26,262	18	NCCSign-II(con.)	4,568,006
2	Peregrine-1024	55,654	19	NCCSign-III(ori.)	6,234,249
3	SOLMAE-512	64,838	20	AIMer-I-param2	7,719,642
4	HAETAE-II	126,972	21	NCCSign-III(con.)	7,935,382
5	SOLMAE-1024	149,168	22	AIMer-III-param1	8,247,989
6	GCKSign-II	172,893	23	NCCSign-V(ori.)	11,155,020
7	GCKSign-III	179,608	24	NCCSign-V(con.)	12,555,623
8	HAETAE-III	227,780	25	AIMer-V-param1	13,724,601
9	GCKSign-V	262,868	26	AIMer-III-param2	22,149,494
10	HAETAE-V	270,600	27	AIMer-I-param3	26,007,825
11	pqsigRM-612	417,658	28	AIMer-V-param2	34,428,290
12	MQSign-72/46	533,676	29	AIMer-III-param3	54,668,451
13	MQSign-112/72	1,120,124	30	AIMer-V-param3	109,326,832
14	MQSign-148/96	1,897,664	31	AIMer-I-param4	111,966,098
15	pqsigRM-613	2,064,731	32	AIMer-III-param4	266,189,006
16	AIMer-I-param1	3,125,105	33	AIMer-V-param4	521,735,231
17	NCCSign-II(ori.)	3,248,162			

Rank	Algorithm(ARM)	Verify(Med., nsec)	Rank	Algorithm(ARM)	Verify(Med., nsec)
1	Peregrine-512	5,120	14	AIMer-I-param2	1,593,472
2	Peregrine-1024	9,984	15	AIMer-III-param1	1,597,952
3	HAETAE-II	40,960	16	AIMer-V-param1	2,849,408
4	GCKSign-II	50,944	17	MQSign-148/96	3,660,416
5	GCKSign-III	54,016	18	AIMer-III-param2	4,254,080
6	pqsigRM-612	72,064	19	AIMer-I-param3	5,123,072
7	HAETAE-III	72,960	20	AIMer-V-param2	7,357,952
8	GCKSign-V	75,008	21	AIMer-III-param3	12,858,624
9	HAETAE-V	91,136	22	AIMer-V-param3	22,692,992
10	pqsigRM-613	379,392	23	AIMer-I-param4	25,750,528
11	MQSign-72/46	431,104	24	AIMer-III-param4	63,475,584
12	AIMer-I-param1	647,040	25	AIMer-V-param4	110,957,440
13	MQSign-112/72	1,565,952			

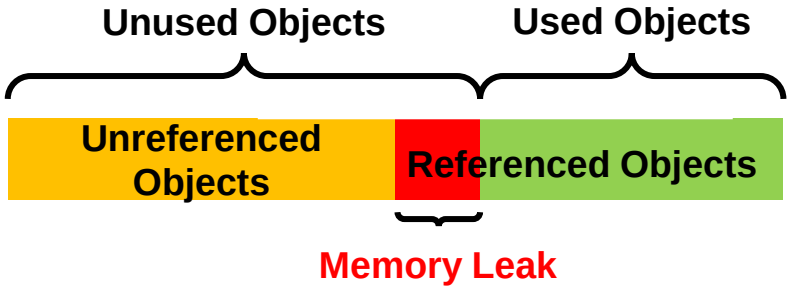

SAMSUNG
Controller
SAMSUNG
DRAM
SAMSUNG
V-NAND
SAMSUNG
V-NAND

메모리와 암호구현

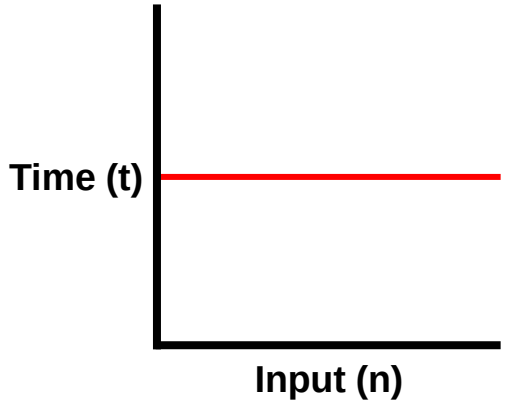
본 발표에서 다루는 내용

- **Memory consumption (메모리 사용량) 측정**
  - 알고리즘 가동에 필요한 메모리 양을 측정
  - 최대 Stack과 Heap을 합산

- **Memory leak (메모리 누수) 점검**
  - 동적할당이 제대로 해제되었는지 확인



- **Constant time (상수 시간) 구현 확인**
  - 입력 값에 따라 연산 시간이 달라진다면 정보 유출이 가능
  - 동일한 연산 시간을 확보하여 비밀 값 유출을 방지



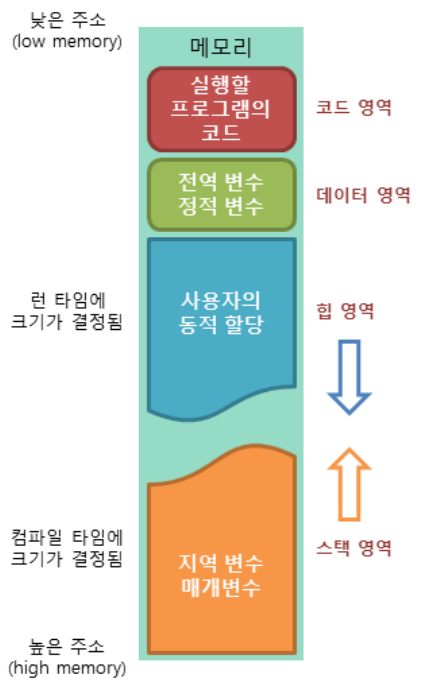
메모리 사용량 측정

- **Valgrind**를 사용한 메모리 사용량 측정

장점	단점
Stack 사용량 뿐만 아니라 Heap 사용량도 정확하게 측정	프로그램을 실행하면서 측정하기에 시간이 소요 출력 로그를 따로 분석해야 함

- **측정 방법**
  1. 메모리 사용량을 측정할 소스코드를 컴파일
     - **이때 반복 횟수는 1회로 고정, Valgrind는 동적할당이 발생할 때마다 이를 메모리 소모로 기록함**
  2. Valgrind 명령어를 통해서 바이너리 파일을 실행
     - valgrind --tool=massif --stacks=yes ./(측정파일이름)
  3. 출력된 로그를 확인하여 Stack양과 Heap양을 합산

메모리 사용량 측정



- valgrind 명령어를 사용하여 프로그램을 실행
- 로그의 양이 방대하여 직접 확인하는 것은 많은 시간이 소요
- **분석에 사용할 Python 프로그램을 따로 작성하여 확인**
- 가장 큰 Stack 과 Heap 크기를 합산

Valgrind 실행

```
$ valgrind --tool=massif --stacks=yes ./PQC_bench
info
```

AIMer-l3-param1.massif.out.215080.txt 56.7 kB

로그 파일 생성

```
desc: --stacks=yes
cmd: ./PQC_bench
time_unit: i
#-----------
snapshot=0
#-----------
time=0
mem_heap_B=0
mem_heap_extra_B=0
mem_stacks_B=0
heap_tree=empty
#-----------
snapshot=1
#-----------
time=1442649
mem_heap_B=30048
mem_heap_extra_B=192
mem_stacks_B=15168
heap_tree=empty
#-----------
snapshot=2
#-----------
time=2318511
mem_heap_B=149296
mem_heap_extra_B=240
mem_stacks_B=16128
heap_tree=empty
#-----------
snapshot=3
#-----------
time=3575103
mem_heap_B=151752
mem_heap_extra_B=400
mem_stacks_B=16312
heap_tree=empty
```

로그 파일 내용

```
import re

f = open(r'AIMer-l3-param3.massif.out.215121.txt')

heap = 0
stack = 0

max_heap = 0
max_stack = 0

while True:
    line = f.readline()

    if not line:
        break

    if "mem_heap_extra_B=" in line:
        heap = int(re.sub(r'[^0-9]', '', line))

    if "mem_stacks_B=" in line:
        stack = int(re.sub(r'[^0-9]', '', line))
        if stack > max_stack:
            max_heap = heap
            max_stack = stack

memory = max_heap + max_stack

print("heap {}".format(max_heap))
print("stack {}".format(max_stack))
print("memory consumption {}".format(memory))
```

```
hd@hd-NUC8i5BEH:~/kpqclean-Round_valgrind/valgrind_stackheap$ python3 log_analyz
er.py
heap 584
stack 16488
memory consumption 17072
hd@hd-NUC8i5BEH:~/kpqclean-Round_valgrind/valgrind_stackheap$
```

메모리 사용량

분석 프로그램을 작성하여 로그 분석

메모리 사용량 측정 (PKE/KEM) 수십 KB가 저전력 장비에 적합한 메모리 사용량

메모리 사용량이 작은 순서: Layered ROLLO, SMAUG, TiGER, NTRU+, IPCC, PALOMA

Algorithm	Stack+Heap (bytes)	Stack (bytes)	Heap (bytes)
Layered ROLLO-128	8,012	7,720	292
Layered ROLLO-192	9,780	9,536	244
Layered ROLLO-256	13,452	13,192	260
SMAUG-128	10,632	10,608	24
SMAUG-192	17,688	17,664	24
SMAUG-256	36,272	36,248	24
TiGER-128	10,992	10,968	24
TiGER-192	19,280	19,256	24
TiGER-256	19,936	19,912	24
NTRUplus-576	16,656	16,632	24
NTRUplus-768	21,776	21,752	24
NTRUplus-864	24,336	24,312	24
NTRUplus-1152	31,984	31,960	24
IPCC-f1	733,200	733,160	40
IPCC-f3	789,200	789,160	40
IPCC-f4	803,240	803,200	40
PALOMA-128	16,641,656	16,641,600	56
PALOMA-192	16,641,656	16,641,600	56
PALOMA-256	16,641,656	16,641,600	56

메모리 사용량 측정 (Digital Signature)

메모리 사용량이 PQC에서 가장 중요한 factor로 보이며 이는 최적화 코드에서 가장 많이 개선되는 부분

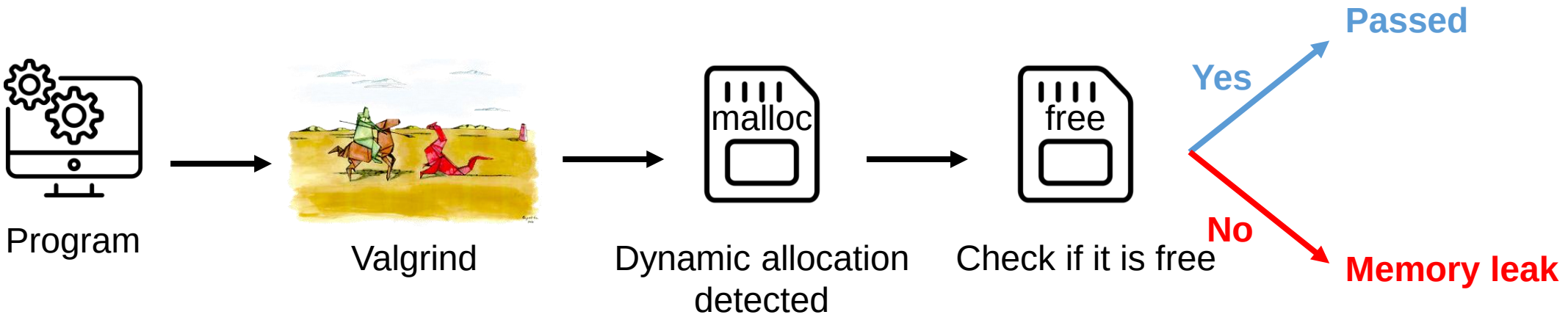
메모리 사용량이 작은 순서: Peregrine, AIMer, GCKSign, HAETAE, SOLMAE, NCCSign, MQSign, Enhanced pqsigRM

Algorithm	Stack+Heap (bytes)	Stack (bytes)	Heap (bytes)	Algorithm	Stack+Heap (bytes)	Stack (bytes)	Heap (bytes)
AIMer-l1-param1	9,418	9,176	242	HAETAE-II	88,600	88,560	40
AIMer-l1-param2	9,912	9,320	592	HAETAE-III	134,936	134,896	40
AIMer-l1-param3	9,470	9,160	310	HAETAE-V	170,424	170,384	40
AIMer-l1-param4	10,004	9,328	676	MQSign-72_46	1,245,952	1,245,896	56
AIMer-l3-param1	17,584	16,664	920	MQSign-112_72	4,674,624	4,674,568	56
AIMer-l3-param2	17,072	16,488	584	MQSign-148_96	10,906,344	10,906,320	24
AIMer-l3-param3	17,072	16,488	584	NCCSign-II (con.)	222,384	222,344	40
AIMer-l3-param4	16,856	16,488	368	NCCSign-III (con.)	294,848	294,808	40
AIMer-l5-param1	31,016	28,632	2384	NCCSign-V (con.)	373,024	372,984	40
AIMer-l5-param2	29,038	28,648	390	NCCSign-II (ori.)	187,200	187,160	40
AIMer-l5-param3	29,554	28,632	922	NCCSign-III (ori.)	262,416	262,376	40
AIMer-l5-param4	29,508	28,808	700	NCCSign-V (ori.)	349,392	349,352	40
GCKSign-II	44,840	44,800	40	Peregrine-512	3,172	3,008	164
GCKSign-III	45,464	45,424	40	Peregrine-1024	3,162	3,008	154
GCKSign-V	69,272	69,232	40	SOLMAE-512	126,896	126,840	56
Enhanced pqsigRM-612	1,619,112	1,618,816	296	SOLMAE-1024	252,800	252,744	56
Enhanced pqsigRM-613	4,288,922	4,288,616	306				

메모리 누수 확인

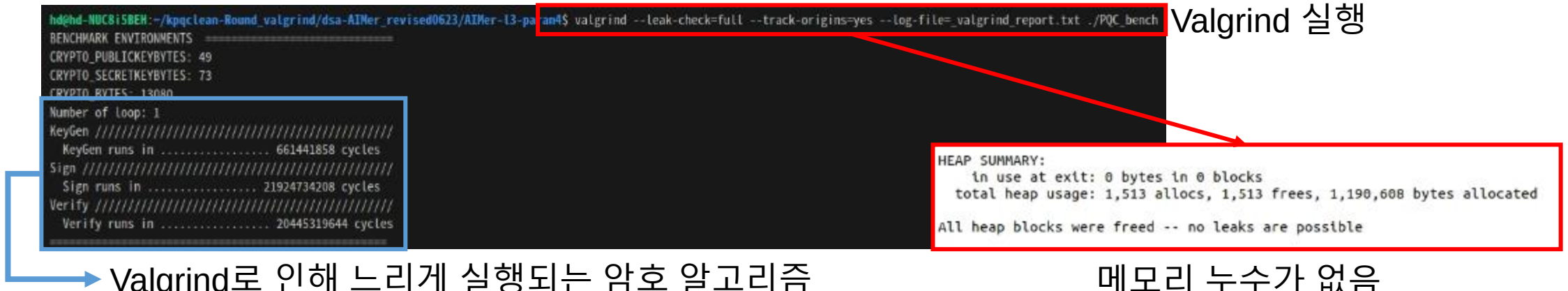
- **Valgrind를 통해 메모리 누수(Memory leak)를 확인할 수 있음**
  - 메모리 누수는 동적할당 후 **사용이 끝난 메모리를 반환하지 않을 때** 발생
  - 사용하지 않는 메모리가 반환되지 않기에 가용 자원이 줄어듦

- **메모리 누수 검사 원리**
  1. Valgrind가 동적할당 된 변수를 확인
  2. 해당 변수가 free 되었는지 점검
  3. **Free되지 않았다면, 메모리 누수가 발생한 것으로 판단**



메모리 누수 확인

- **Valgrind 명령어를 통해 검사 가능**
  - valgrind --leak-chek-=full --track-origins=yes --log-file=(로그파일이름).txt ./(프로그램이름)
  - 상수 시간(Constant time) 구현까지 확인 가능
- **적정한 측정 시간을 위해 반복 횟수를 1로 지정**
  - Valgrind가 실행될 경우, **프로그램 실행 속도가 매우 느려짐**
  - 적정한 측정 시간을 위해서라도 반복 횟수는 1회로 사용
- **로그 파일에 기록된 정보를 토대로 메모리 누수를 확인 가능**
  - 누수가 발생했다면 얼마만큼 (byte) 누수 되었는지 기록됨



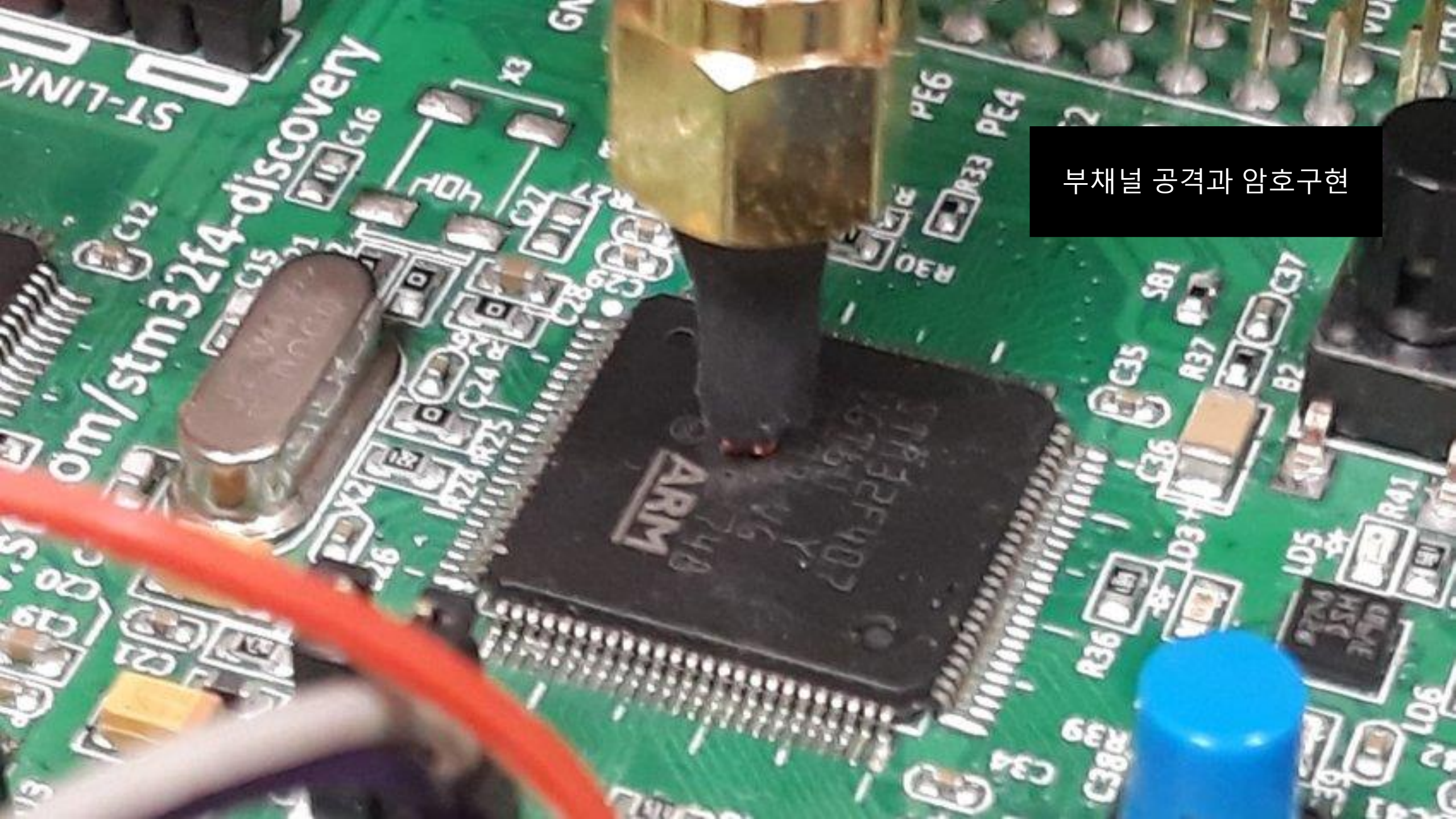
메모리 누수 확인

- **3개의 알고리즘에서 메모리 누수가 발생됨을 확인 (측정불가 알고리즘 2종)**
  - REDOG: Python으로 제공되는 코드로, Valgrind는 C언어를 대상으로 하기에 실행 불가
  - FIBS: Valgrind 적용 시 연산이 종료되지 않음

A 암호	
612	613
definitely lost: 96 bytes in 4 blocks indirectly lost: 947,354 bytes in 4 blocks possibly lost: 0 bytes in 0 blocks still reachable: 16,384 bytes in 1 blocks suppressed: 0 bytes in 0 blocks	definitely lost: 1,000 bytes in 35 blocks indirectly lost: 2,064 bytes in 3 blocks possibly lost: 0 bytes in 0 blocks still reachable: 32,768 bytes in 1 blocks suppressed: 0 bytes in 0 blocks

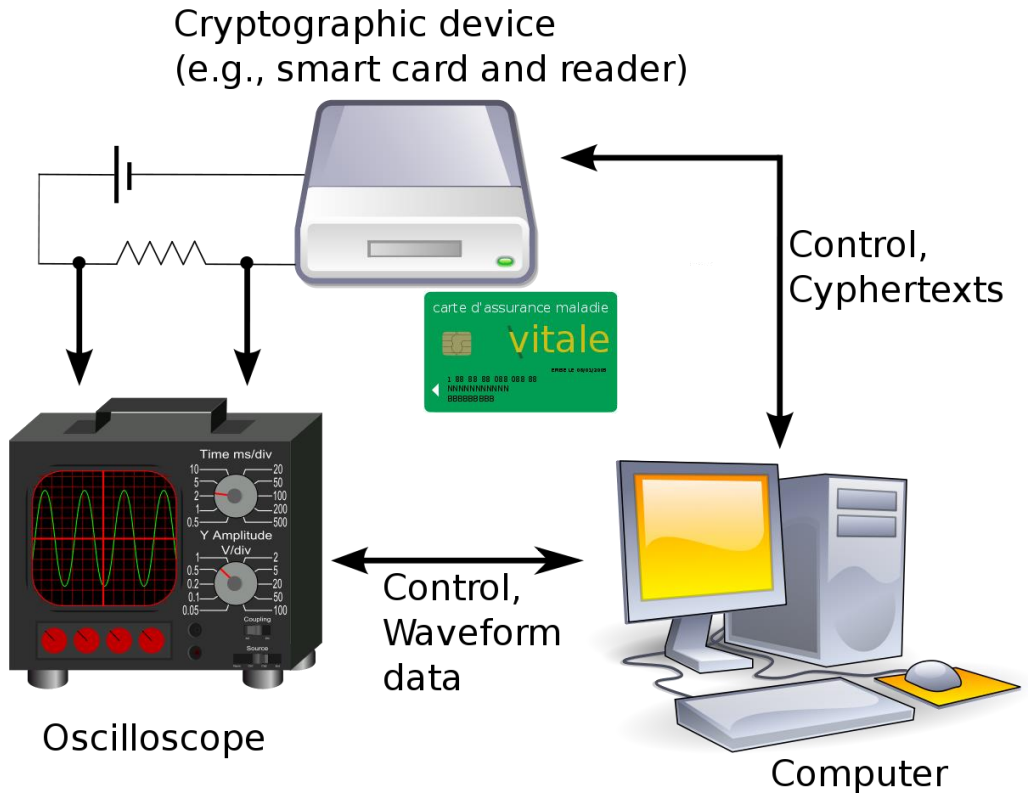
B 암호		
128	192	256
definitely lost: 112 bytes in 7 blocks indirectly lost: 5,108 bytes in 7 blocks possibly lost: 0 bytes in 0 blocks still reachable: 0 bytes in 0 blocks suppressed: 0 bytes in 0 blocks	definitely lost: 112 bytes in 7 blocks indirectly lost: 5,940 bytes in 7 blocks possibly lost: 0 bytes in 0 blocks still reachable: 0 bytes in 0 blocks suppressed: 0 bytes in 0 blocks	definitely lost: 112 bytes in 7 blocks indirectly lost: 8,308 bytes in 7 blocks possibly lost: 0 bytes in 0 blocks still reachable: 0 bytes in 0 blocks suppressed: 0 bytes in 0 blocks

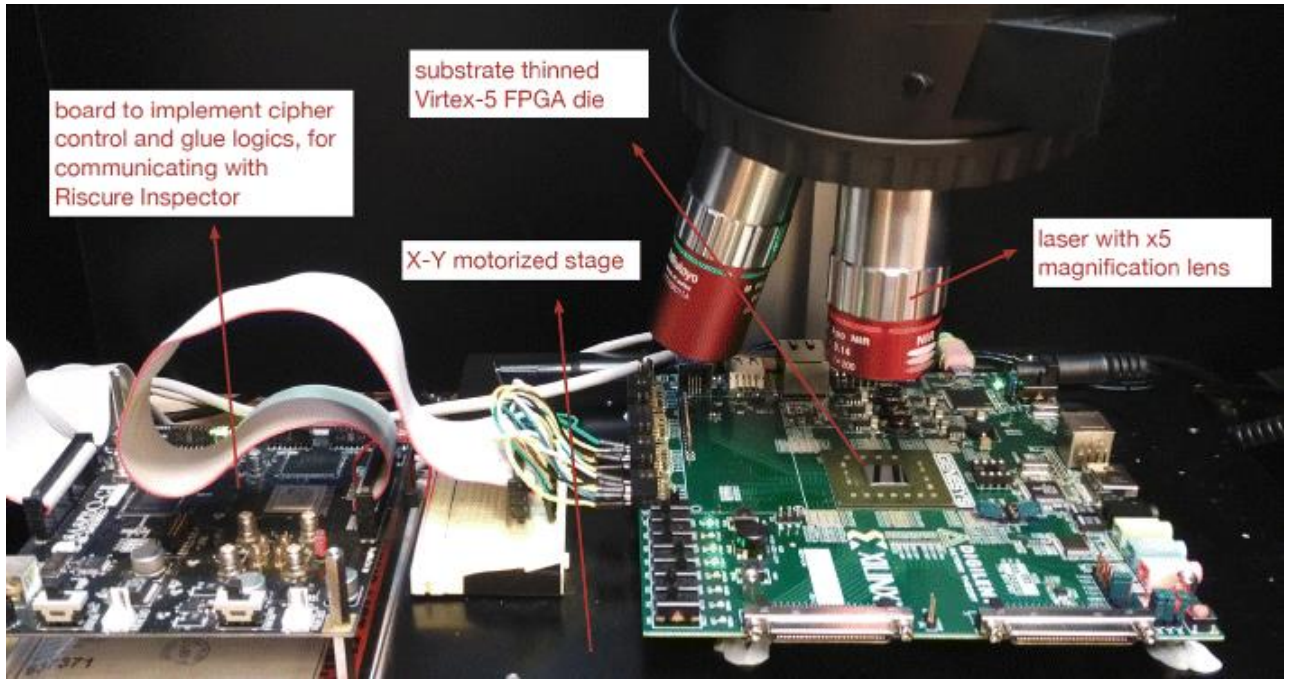
C 암호	
512	1024
definitely lost: 109,820 bytes in 6 blocks indirectly lost: 0 bytes in 0 blocks possibly lost: 0 bytes in 0 blocks still reachable: 0 bytes in 0 blocks suppressed: 0 bytes in 0 blocks	definitely lost: 220,934 bytes in 6 blocks indirectly lost: 0 bytes in 0 blocks possibly lost: 0 bytes in 0 blocks still reachable: 0 bytes in 0 blocks suppressed: 0 bytes in 0 blocks


부채널 공격과 암호구현

부채널 공격

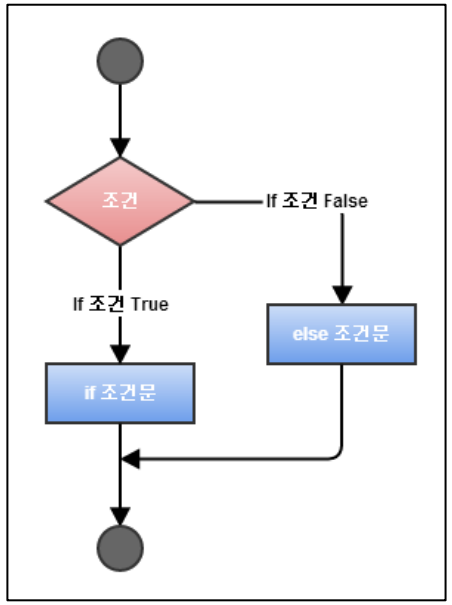
- **암호구현 시 부가적으로 발생하는 정보 (전력, 시간, 온도)를 이용한 암호분석 방법론**
- **부채널 공격은 다양한 스펙트럼을 가지며 공격에 대한 가정을 강화할 수록 공격 성공률이 높아지게 됨**
  - 특히 접근성이 높은 임베디드 장비 (사물인터넷)에 대한 공격이 이루어지고 있음
  - 물론 일반적인 고성능 컴퓨터 그리고 클라우드에서도 다양한 부채널 공격 가능성 존재

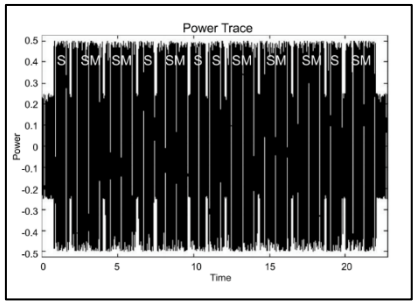




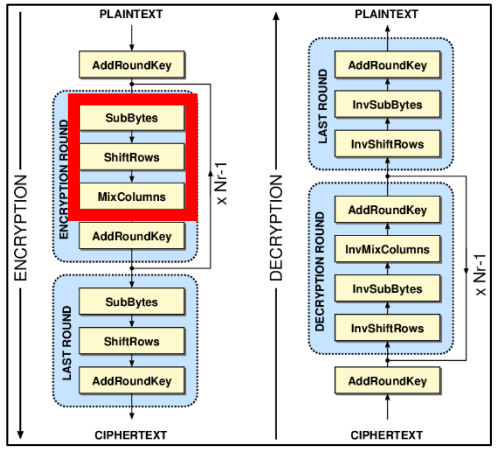
암호구현의 기본이 되는 부채널 방어

- **간단한 Timing Attack 예시**
  - 조건에 따라 상이한 연산 (조건문) → 상이한 연산 속도
  - RSA는 Simple Power Analysis와 Timing Attack에 취약
  - AES는 T-table 기반 최적화 시 캐시 공격 가능
  - Timing Attack을 방어하기 위해서는 → 비밀키에 의존적인 조건문 삭제 필요





S = *M*
for *i* **from** *1* **to** *n-1* **do**
S = *S*^2
if d_i = *1* **then**
S = *S* * *M*
return *S* (= M^d)



$$T_0[a]=\begin{bmatrix} S[a]\bullet 02 \\ S[a] \\ S[a] \\ S[a]\bullet 03 \end{bmatrix} \quad T_1[a]=\begin{bmatrix} S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \\ S[a] \end{bmatrix} \quad T_2[a]=\begin{bmatrix} S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \end{bmatrix} \quad T_3[a]=\begin{bmatrix} S[a] \\ S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \end{bmatrix}.$$

$$T_0[k[0]\oplus n[0]]$$

상수 시간 구현 확인

- **Valgrind는 특정 플래그 값이 분기문을 형성하는지 추적 가능**
  - 소스코드에 **플래그를 표시**해줘야 함
  - 해당 값이 if, for 같은 분기(반복)문을 형성할 경우, 로그를 생성
- **상수 시간 구현에 치명적인 부분을 발견할 수 있음**
  - 그러나 Valgrind는 **분기문의 의미를 해석하지는 않음**
  - Valgrind 로그 발생 ≠ 상수 시간 구현 깨짐
  - **따라서 로그가 발생한 위치를 직접 소스코드 상에서 확인할 필요가 있음**

Flag 지정 (일반적으로 비밀키)

```
cycles1 = cpucycles();
VALGRIND_MAKE_MEM_UNDEFINED(sk, CRYPTO_SECRETKEYBYTES);
crypto_sign(sm, &smlen, m, mlen, sk);
cycles2 = cpucycles();
fprintf(fp_sign, "%lld \n", cycles2-cycles1);
kcycles += cycles2-cycles1;
```

Valgrind 실행

```
/AIMer-l3-param4$ valgrind --leak-check=full --track-origins=yes --log-file=_valgrind_report.txt ./PQC_bench
```

로그 검출

```
Conditional jump or move depends on uninitialised value(s)
   at 0x10D9AA: aimer_instance_get (in /home/hd/kpqclean-Round_valgrind/dsa-AIMer_revised0623/AIMer-l3-param1/PQC_bench)
   by 0x1098AB: crypto_sign (in /home/hd/kpqclean-Round_valgrind/dsa-AIMer_revised0623/AIMer-l3-param1/PQC_bench)
   by 0x10964D: PQC_bench (in /home/hd/kpqclean-Round_valgrind/dsa-AIMer_revised0623/AIMer-l3-param1/PQC_bench)
```

실제 소스코드 상에서 확인

```
const aimer_instance_t *aimer_instance_get(aimer_parans_t paran)
{
  if (param <= PARAMETER_SET_INVALID || param >= PARAMETER_SET_MAX_INDEX)
  {
    return NULL;
  }

  return &instances[param];
}
```

```c
#include <valgrind/memcheck.h>

int compute(unsigned char secret[32]) {
    if (secret[0] == 0) {
        return 0;
    } else {
        return 1;
    }
}

int main(void) {
    unsigned char buf[32];
    for (int i = 0; i < 32; ++i)
        buf[i] = 0;
    VALGRIND_MAKE_MEM_UNDEFINED(buf, 32);
    compute(buf);
    return 0;
}
```

```
Conditional jump or move depends on uninitialised value(s)
        at 0x1093BC: compute (code.c:4)
        by 0x109464: main (code.c:16)
```

Valgrind 결과 해석

일반적인 구현 기법

Input: $g, k = (k_{t-1}, \ldots, k_0)_2$
Output: $y = g^k$ 기본적인 RSA 연산

$R_0 \leftarrow 1;\ R_1 \leftarrow g$
for $j = t-1$ downto 0 do
  $R_0 \leftarrow (R_0)^2$
  if $(k_j = 1)$ then $R_0 \leftarrow R_0 R_1$
return R_0 비트가 세팅된 경우에만 곱셈 수행

Montgomery Ladder 기법

Input: $g, k = (k_{t-1}, \ldots, k_0)_2$
Output: $y = g^k$

$R_0 \leftarrow 1;\ R_1 \leftarrow g$
for $j = t-1$ downto 0 do
  if $(k_j = 0)$ then
    $R_1 \leftarrow R_0 R_1;\ R_0 \leftarrow (R_0)^2$
  else [if $(k_j = 1)$]
    $R_0 \leftarrow R_0 R_1;\ R_1 \leftarrow (R_1)^2$
return R_0 언제나 곱셈 그리고 제곱연산이 수행됨

- **Valgrind의 False positive 출력**
  - Valgrind는 분기문의 내용을 분석하지는 못함
  - **로그에 걸린 값이 있더라도 개별적인 분석을 진행해야 함**
  - **현재 모든 경우에 대해서 분석했지만, 보다 더 정밀한 분석 진행 중**
    - 일부 알고리즘에 대해서 타이밍 공격의 가능성이 존재하지만 자세한 조사가 필요함
- **로그 분석 예시**
  - **플래그 값(비밀키 배열)**이 분기문에 걸려있지만, **구조상 문제가 없음 (뒷장 KpqC case study)**

```
const aimer_instance_t *aimer_instance_get(aimer_params_t param)
{
  if (param <= PARAMETER_SET_INVALID || param >= PARAMETER_SET_MAX_INDEX)
  {
    return NULL;
  }

  return &instances[param];
}
```

- if문에 값(비밀키의 일부)이 걸려서 분기가 형성
- 해당 값의 의미를 파악하기 위해 이동

```
typedef enum
{
  PARAMETER_SET_INVALID = 0,
  AIMER_L1_PARAM2 = 2,
  AIMER_L1_PARAM3 = 3,
  AIMER_L1_PARAM4 = 4,
  AIMER_L3_PARAM1 = 5,
  AIMER_L3_PARAM2 = 6,
  AIMER_L3_PARAM3 = 7,
  AIMER_L3_PARAM4 = 8,
  AIMER_L5_PARAM1 = 9,
  AIMER_L5_PARAM2 = 10,
  AIMER_L5_PARAM3 = 11,
  PARAMETER_SET_MAX_INDEX = 13
} aimer_params_t;
```

0: 파라미터 설정 오류 코드

13: 파라미터 최대 종류 수

1. if문에서 비밀키의 첫 번째 값 체크
2. 값이 0, 13(특정 코드)일 경우 연산 종료
3. 그 외일 경우(1-12) 정상적인 연산 실행
4. 이로 인해서 연산 시간 차이는 발생
   - Constant time이 깨짐
5. **그러나 정상 실행 여부 외에는 유출 정보가 없음**

AIMer의 상수 시간 구현 확인

- **aimer_instance_get**
  - **플래그 값(비밀키 배열)**이 분기문에 걸려있지만, **구조상 문제가 없음**
    - param은 비밀키의 앞 일부 데이터를 포함
  - **정상적인 실행 중에는 if문이 무시됨**
    - 파라미터가 정상 범위 외일 경우 NULL값 반환으로 if문 동작 종료
  - **따라서 비밀키 유출의 위험성은 없음**

if문에 값(비밀키의 일부)이 걸려서 분기가 형성

비밀키 일부를 파라미터로 받음

```
const aimer_instance_t* aimer_instance_get(aimer_params_t param)
{  if문에서 비밀키의 첫 번째 값 체크
   if (param <= PARAMETER_SET_INVALID || param >= PARAMETER_SET_MAX_INDEX)
   {
      return NULL;
   }

   return &instances[param];
}
```

값이 0,13(특정 코드)일 경우 연산 종료

값이 0,13외일 경우(1~12) 정상적인 연산 실행

이로 인해 연산 시간 차이가 발생(Constant time이 깨짐)

그러나 정상 실행 여부 외에는 유출 정보가 없음

```
typedef enum
{    0: 파라미터 설정 오류 코드
   PARAMETER_SET_INVALID    =  0,
   AIMER_L1_PARAM1          =  1,
   AIMER_L1_PARAM2          =  2,
   AIMER_L1_PARAM3          =  3,
   AIMER_L1_PARAM4          =  4,
   AIMER_L3_PARAM1          =  5,
   AIMER_L3_PARAM2          =  6,
   AIMER_L3_PARAM3          =  7,
   AIMER_L3_PARAM4          =  8,
   AIMER_L5_PARAM1          =  9,
   AIMER_L5_PARAM2          = 10,
   AIMER_L5_PARAM3          = 11,
   AIMER_L5_PARAM4          = 12,
   PARAMETER_SET_MAX_INDEX = 13
} aimer_params_t;13: 파라미터 최대 종류 수
```

Layered ROLLO의 상수 시간 구현 확인

```
#define SEEDEXPANDER_SEED_BYTES 40 /**< Seed size of the NIST seed expander */
```

```
//Secret key generation
for(size_t i = 0 ; i < SEEDEXPANDER_SEED_BYTES ; i++)
{
  skseed_st[i]=sk[i];
}

 for(size_t i = 0 ; i < SEEDEXPANDER_SEED_BYTES ; i++)
{
  sk_PI_seed_st[i]=sk[i+SEEDEXPANDER_SEED_BYTES];
}

for(size_t i = 0 ; i < SEEDEXPANDER_SEED_BYTES ; i++)
{
  sk_PO_seed_st[i]=sk[i+2*SEEDEXPANDER_SEED_BYTES];
}
```

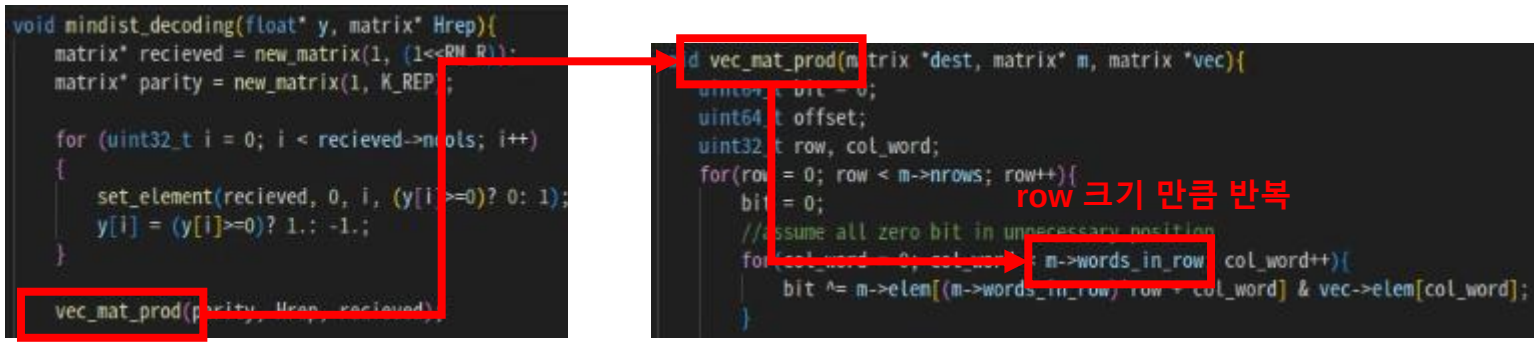
플래그 값(비밀키 배열)이 반복문(for)에서 SEEDEXPANDER_SEED_BYTES(40) 크기만큼 Copy 되지만, **구조상 문제가 없음**

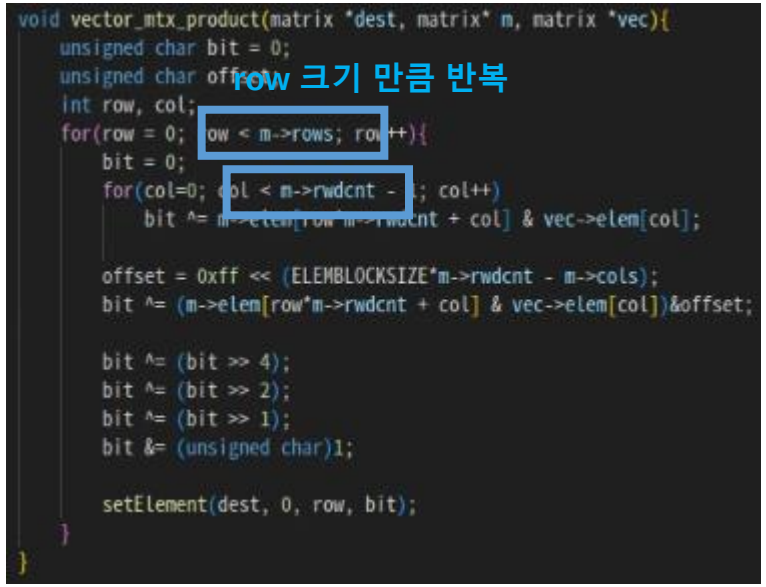
```
biix_secret_key_from_string(&skTmp1, skseed_st);
rbc_qre_set_arb_random(PI, BIIX_PARAM_PI_DEG, sk_PI_seed_st);
biix_decaps_multisym(ct, ctRes, sk_PO_seed_st);
```

복사된 비밀키 값이 다른 함수에서 사용되지만, 각 함수에서 이에 따른 **이상 (분기문)이 발견되지 않음**

Enhanced pqsigRM의 상수 시간 구현 확인

- **Mindest_decoding → vec_mat_prod**
- **Vector_mtx_product**
  - 모두 공통적으로 **행렬 크기 만큼 for문을 반복**시킴
  - 반복 횟수로 타이밍 정보가 달라질 수 있음
  - 그러나 **행렬 크기는 고정값**이므로 실제로는 **반복 횟수가 바뀌지 않음**
  - **비밀값에서 행렬 크기 정보를 사용하지만 비밀 정보 유출은 없음**





HAETAE의 상수 시간 구현 확인

- **rej_uniform**
  - rej_unfrom()에 연관된 값은 **해시값**
  - 유출이 되더라도 **해시값만 유출되어 비밀키 유출과 직접적 관련은 없음**

```
void poly_uniform(poly *a, const uint8_t seed[SEEDBYTES], uint16_t nonce) {
    unsigned int i, ctr, off;
    unsigned int buflen = POLY_UNIFORM_NBLOCKS * STREAM128_BLOCKBYTES;
    uint8_t buf[POLY_UNIFORM_NBLOCKS * STREAM128_BLOCKBYTES + 1];
    stream128_state state;

    stream128_init(&state, seed, nonce);
    stream128_squeezeblocks(buf, POLY_UNIFORM_NBLOCKS, &state);

    ctr = rej_uniform(a->coeffs, N, buf, buflen);

    while (ctr < N) {
        off = buflen % 2;
        for (i = 0; i < off; ++i)
            buf[i] = buf[buflen - off + i];

        stream128_squeezeblocks(buf + off, 1, &state);
        buflen = STREAM128_BLOCKBYTES + off;
        ctr += rej_uniform(a->coeffs + ctr, N - ctr, buf, buflen);
    }
}
```

```
unsigned int rej_uniform(int32_t *a, unsigned int len, const uint8_t *buf,
                         unsigned int buflen) {
    unsigned int ctr, pos;
    uint32_t t;

    ctr = pos = 0;
    while (ctr < len && pos + 2 <= buflen) {
        t = buf[pos++];
        t |= (uint32_t)buf[pos++] << 8;

        if (t < Q)
            a[ctr++] = t;
    }
    return ctr;
}
```

결론

- **KpqC 공모전**을 통해 국내에서 사용할 안전하고 효율적인 암호가 잘 선택되기 위해서는 지속적으로 **보안성과 효율성을 검토**해야 함

- **본 발표를 통해서 전반적인 암호의 효율성**에 대해 확인해 볼 수 있었으며 이를 기반으로 앞으로 더욱 고도화된 분석이 요구됨

- 아직까지 많은 **KpqC 암호 알고리즘들이 관심을 덜 받았다고 생각**되며 많은 후속 연구를 통해 이론적 기반이 탄탄해 질 수 있어야 할 것으로 보임

- **KpqC 공모전 전반부에 해당하는 지금은 보안성에 초점**이 더 맞추어 져야 할 것으로 보이며 **후반전에서는 실용적인 효율성이 중요**시 될 것으로 사료됨

Q & A