벤더 관점에서의 PQC 표준화

유튜브 주소 : https://youtu.be/5WFw_E3skjk

구현: NTT 최적화

포트폴리오

표준화

벤더 관점에서 본 PQC 표준화

벤더 관점에서는 구현을 우선시 하여 표준화 프로세스 진행

PQC 최적화를 통한 구현 비용 절감 중요

(실용적인 기술에 집중)

일반적인 표준화 과정 순서

벤더 관점의 표준화 과정 순서

Kyber vs Dilithium

항목	Kyber(ML-KEM)	Dilithium(ML-DSA)
기반 문제	격자 기반	
연산	NTT	
비트 크기	12-bit	23-bit
NTT 완성도	*Incomplete NTT	*Complete NTT
해시 함수	SHAKE 사용	
샘플링 방식	*이항 분포 샘플링	*균일 샘플링

***Incomplete NTT**: 일부 최적화 단계를 생략하여 **구현을 단순화**

* **Complete NTT**: 최적화 된 NTT로, **성능을 최대화**

* **이항 분포 샘플링:** 이항 분포를 사용하여 난수를 생성하는 방법**(각 값이 선택될 확률이 다름)**

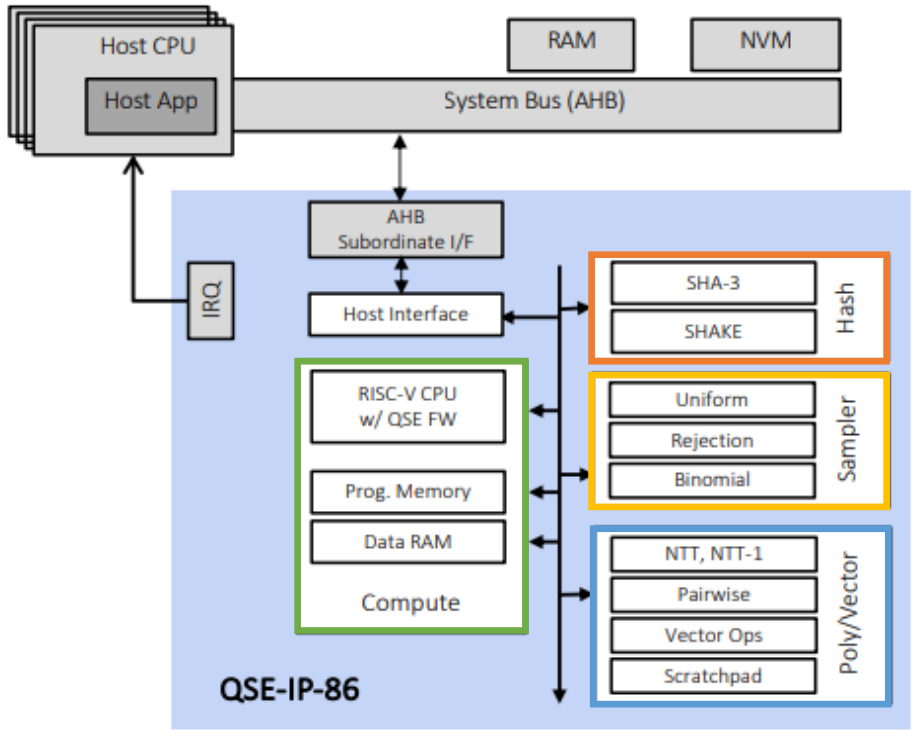
* **균일 샘플링:** 모든 값이 동일한 확률로 선택되는 방법**(각 값이 선택될 확률이 모두 동일)**

Rambus 양자 내성 엔진(Rambus QSE)

- **양자 컴퓨터 시대에 대비해 설계된 보안 엔진**
  - 연산을 효율적으로 수행하고, 하드웨어 가속을 통해 **알고리즘의 성능을 향상시키는 모듈**
  - FIPS 203, FIPS 204 표준 및 SHA-3, SHAKE-128, SHAKE-256 가속기 지원

<Rambus 알고리즘 동작 시 각 모듈의 역할>

- **Compute**: 다항식 및 벡터의 실제 연산 수행
  E.g. RISC-V CPU, Memory, Data RAM
- **Hash**: SHA-3 및 SHAKE 함수 등 해시 연산 수행
  E.g. SHA-3, SHAKE
- **Sampler**: 다양한 샘플링 방식의 연산 수행
  E.g. Binomial, Uniform, Rejection Sampling
- **Poly/Vector**: NTT, Karatsuba 곱셈 등의 연산 수행
  E.g. NTT, NTT-1, Pairwise, Vector Ops, Scratchpad



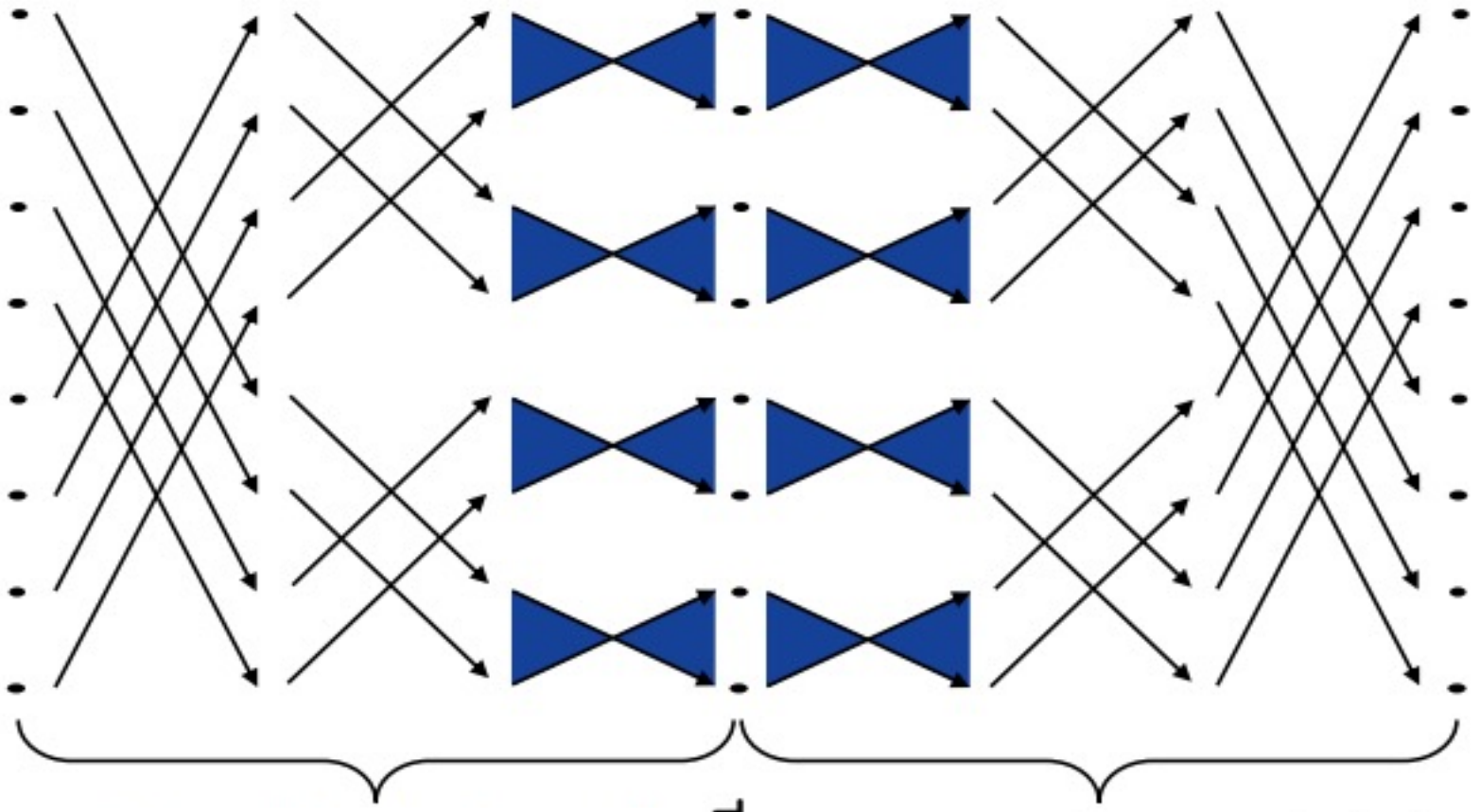
구현: NTT(Number-Theoretic Transform)

버터플라이 연산

NTT 기본 연산 단위

모듈러 연산 사용

NTT 기본 구조



Forward: Cooley-Tukey(CT) 알고리즘

- 입력 다항식(시간 영역)을 주파수 영역의 데이터로 변환

Inverse: Gentleman-Sande(GS) 알고리즘

- 주파수 영역으로 변환된 데이터를 시간 영역의 다항식으로 재 변환

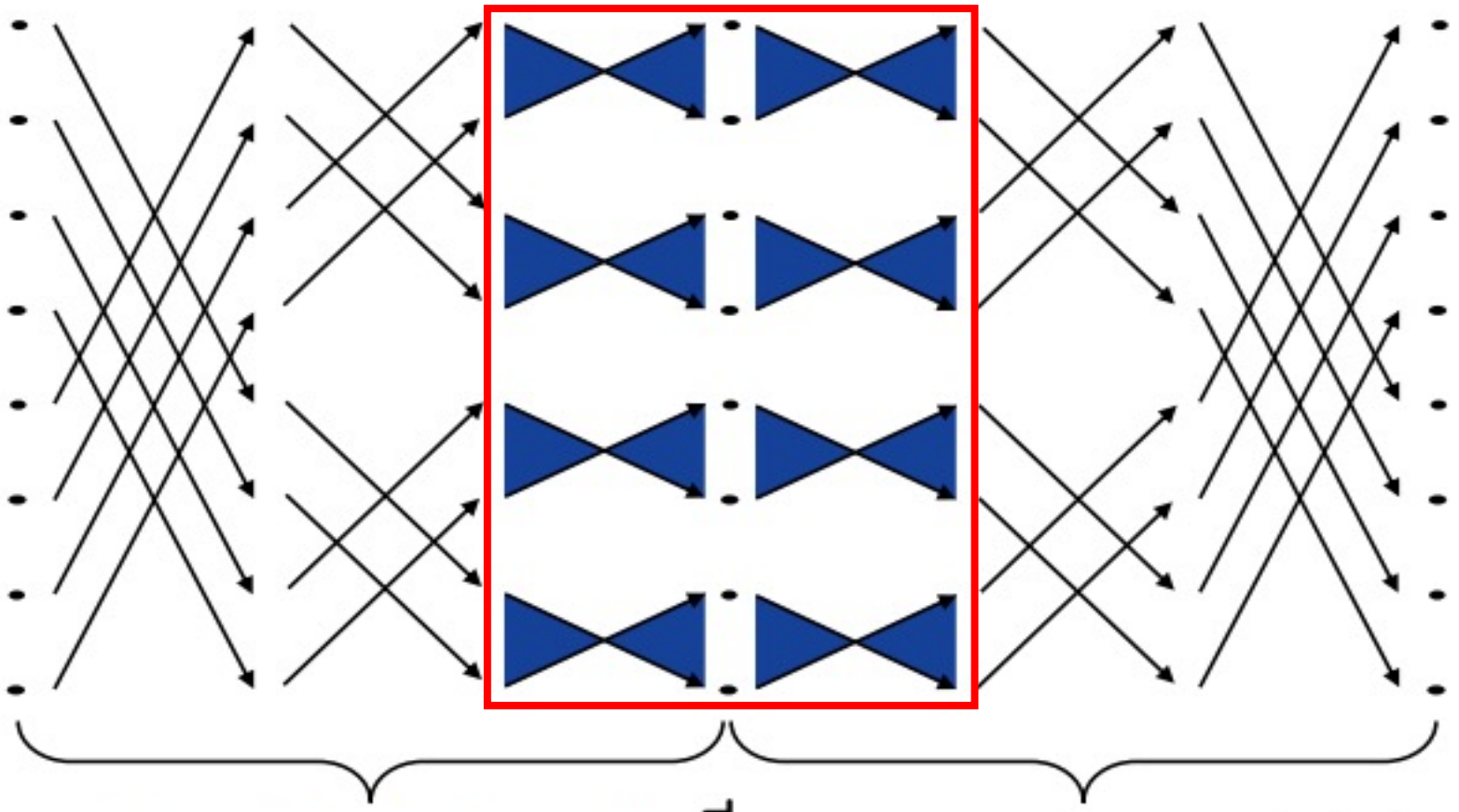
구현: 버터플라이 연산 재구성

버터플라이 연산

NTT 기본 연산 단위

모듈러 연산 사용

NTT 기본 구조



Forward: Cooley-Tukey(CT) 알고리즘

- 입력 다항식(시간 영역)을 주파수 영역의 데이터로 변환

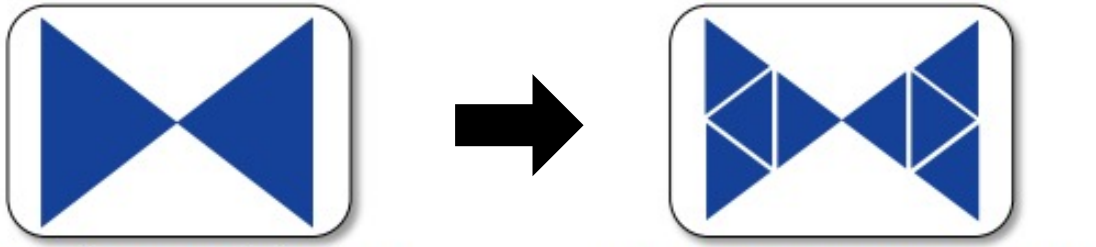
Inverse: Gentleman-Sande(GS) 알고리즘

- 주파수 영역으로 변환된 데이터를 시간 영역의 다항식으로 재 변환

구현: 버터플라이 연산 재구성

다양한 모듈러 연산을 지원하기 위해서는 설계 변경 필요

Kyber(12-bit)와 Dilithium(23-bit)을 **하나의 하드웨어 플랫폼에서 지원하기 위함**



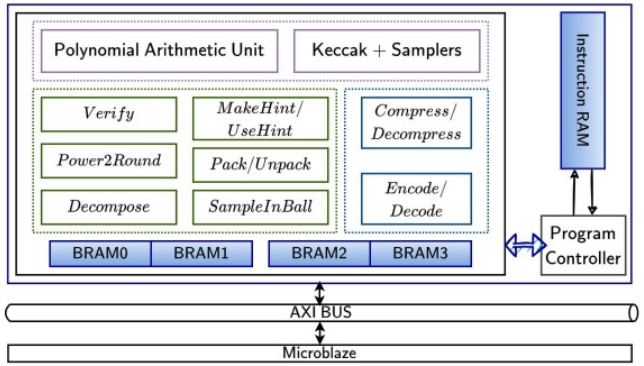
32-bit 모듈러 연산 **12-bit, 23-bit 모듈러 연산**

<설계 변경 시 비용 증가 항목>

- **설계 복잡성**
  - 32-bit는 2의 거듭제곱으로, 단순한 shift 연산으로 대체할 수 있는 경우가 많으나 12,23-bit는 대체하기 어려움
  - 즉, 맞춤형 설계가 필요하여 이에 따른 복잡성 증가
- **검증 비용**
  - 변경된 설계가 제대로 동작하는지에 대한 테스트 케이스 준비에 따른 비용 필요
- **개발 비용**
  - 변경된 설계를 기반으로 구현을 새로 해야 하므로, 개발 인력 투입에 따른 비용이 증가될 수 있음
- **칩 면적 비용**
  - 지원 bit가 늘어남에 따른 칩 면적이 늘어날 수 있어, 이에 따른 제조 비용이 증가될 수 있음
  - 설계가 복잡해짐에 따라 더 정교한 회로 배치가 필요할 수 있어, 이에 따른 제조 비용이 증가될 수 있음

구현: 버터플라이 연산 재구성

- **KaLi 시스템 상에서 버터플라이 연산 재구성[1]**
  - KaLi 시스템: Kyber와 Dilithium을 사용하는 암호화 아키텍처
  - **기존 버터플라이 연산기의 설계를 변경**
    - Dilithium과 Kyber 알고리즘을 모두 지원할 수 있게끔 연산 과정을 재구성
  - 지연 시간을 줄이기 위해 효율적인 메모리 관리 방법 제안
    - 메모리 재 정렬, 다중 메모리 뱅크 등



KaLi 시스템 구조

<기존 연산>

- 23-bit 버터플라이 연산기 1개 사용
  - Dilithium 연산 시 사용
- 12-bit 버터플라이 연산기 2개 사용
  - Kyber 연산 시 사용

<재구성한 연산>

- 23-bit 버터플라이 연산기 2개 사용
- Karatsuba 곱셈기 1개 사용

<연산 재구성 과정>

기존 23-bit 버터플라이 연산기 1개는 그대로 사용

12-bit 연산기 2개를 병렬로 사용하여 23-bit 연산기 1개로 만듦

- 23-bit 데이터를 상위 11-bit와 하위 12-bit로 나누어 각 부분을 별도로 연산
- 상위 12-bit 연산기의 나머지 1-bit는 패딩으로 처리하여 결과가 23-bit가 되도록 함

12-bit 연산기를 병렬로 연산하는 과정에서 Karatsuba 곱셈기 필요

- Karatsuba 알고리즘으로 23-bit 연산을 두 개의 12-bit 연산으로 나누어 처리 가능
- 두 개의 12-bit 연산기를 사용하는 것 보다 더 효율적으로 연산 가능

위와 같은 과정으로 23-bit 버터플라이 연산기 2개로 재구성

- Karatsuba 곱셈기 1개 포함

[1] Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, Sujoy Sinha Roy: KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium. IEEE Trans. Circuits Syst. I Regul. Pap. 70(2): 747-758 (2023)

구현: 버터플라이 연산 설계 최적화

재구성한 버터플라이 연산의 최적화

- **CT/GS 및 Karatsuba 알고리즘 최적화**
  - CT 및 GS 알고리즘 연산은 2N-bit 단위의 변환 연산을 수행
    - 즉, 2N-bit 단위의 덧셈, 뺄셈, 곱셈, 누적 곱셈 연산 수행
    - CT(Cooley-Tukey): 입력 다항식(시간 영역)을 주파수 영역의 데이터로 변환하는 연산 수행
    - GS(Gentleman-Sande): 주파수 영역으로 변환된 데이터를 다시 시간 영역의 다항식으로 재 변환하는 연산 수행
  - **CT/GS 알고리즘을 병렬로 수행하여, 4N-bit 단위의 CT/GS 버터플라이 연산 수행(최적화)**
    - 즉, 4N-bit 단위의 덧셈, 뺄셈, 곱셈, 누적 곱셈 연산 수행
  - Karatsuba 알고리즘은 2x2 형태의 곱셈기를 사용하여 최적화
    - Karatsuba 알고리즘: 큰 수의 곱셈을 효율적으로 수행하는 알고리즘
      - 시간 복잡도를 $O(N^2)$에서 $O(N^{log_2 3})$로 줄일 수 있음
    - 2x2형태의 곱셈기: 2개의 N-bit 곱셈기를 사용하여 더 큰 단위(2N-bit)의 곱셈을 수행

구현: 버터플라이 연산 설계 최적화

재구성한 버터플라이 연산의 최적화

- **CT/GS 및 Karatsuba 알고리즘 최적화**
  - CT 및 GS 알고리즘 연산은 2N-bit 단위의 변환 연산을 수행
    - 즉, 2N-bit 단위의 덧셈, 뺄셈, 곱셈, 누적 곱셈 연산 수행
    - CT(Cooley-Tukey): 입력 다항식(시간 영역)을 주파수 영역의 데이터로 변환하는 연산 수행

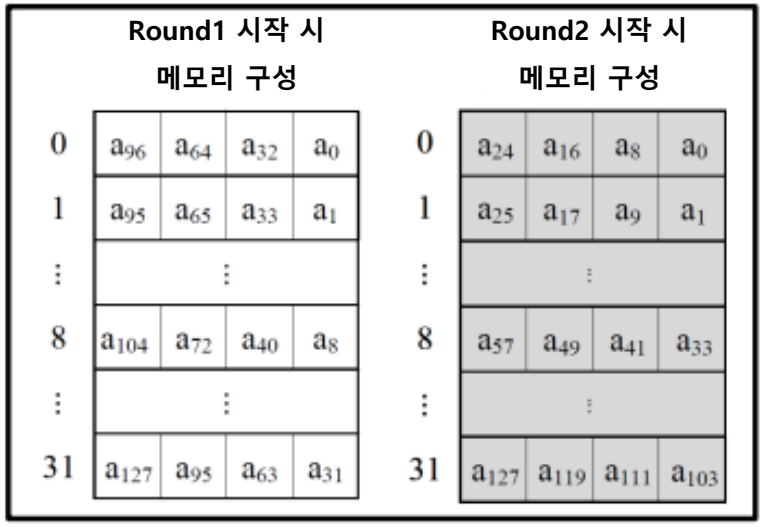
최적화 시, 1.하드웨어 자원을 효율적으로 사용하여 비용 절감 및 성능 향상
2.메모리 대역폭을 효율적으로 사용하여 데이터 전송 속도 향상

  - Karatsuba 알고리즘: 큰 수의 곱셈을 효율적으로 수행하는 알고리즘
    - 시간 복잡도를 $O(N^2)$에서 $O(N^{log_2 3})$로 줄일 수 있음
  - 2x2형태의 곱셈기: 2개의 N-bit 곱셈기를 사용하여 더 큰 단위(2N-bit)의 곱셈을 수행

구현: NTT 메모리 패턴 관리 방법

NTT 곱셈 가속기 상에서의 효율적인 메모리 패턴 관리 방법으로는 크게 아래 두 가지가 있음

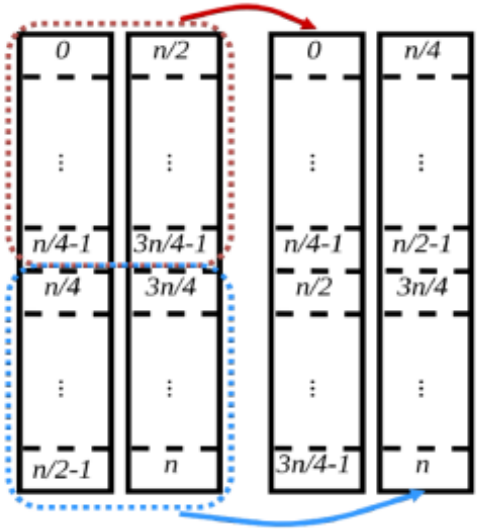
메모리 재 정렬[2]



다항식 계수 변환 및 데이터 순서 재 배열

- 메모리 재 정렬은 다항식 계수 재 정렬을 의미
- Round를 넘어갈 때 데이터 순서를 재 배열
  - Ex) Round1에서는 계수가 열 기준으로 정렬 되어 있음, Round2에서는 행 기준으로 재 정렬 됨

다중 메모리 뱅크[3]



데이터를 4개의 메모리 뱅크에 분산

- 각 메모리 뱅크는 다항식의 계수를 일정하게 분할하여 저장
  - 첫 번째 뱅크는 0부터 n/4-1까지, 두 번째 뱅크는 n/4부터 n/2-1까지 저장
- 여러 메모리 뱅크에 데이터가 분산 되어 병렬로 동시 접근 가능
  - 4개의 메모리 뱅크에서 동시에 데이터를 읽고 쓰는 등의 병렬처리 가능

[2] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, Mehran Mozaffari Kermani: High-Speed NTT-based Polynomial Multiplication Accelerator for Post-Quantum Cryptography. ARITH 2021: 94-101
[3] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, Erkay Savas: A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme. DATE 2021: 1020-1025

포트폴리오: 벤더 관점에서의 **선호 부분**

- **Kyber와 Dilithium의 공통점**
  - LWE(Learning With Errors) 사용
  - 고속 다항식 연산을 위해 NTT 사용
  - SHA-3의 XOF인 SHAKE 사용
    - XOF(Extendable OutPut Function): 확장 가능한 해시 함수

  → 공통된 부분으로 인해 효율성 상승 및 비용 절감 가능

- **NTT 친화적 소수**
  - NTT 친화적 소수는 모듈러 연산을 효율적으로 수행할 수 있도록 함
  - 효율적인 몽고메리 및 Barrett Reduction 연산 가능
    - 몽고메리 Reduction: 소수에 대한 모듈러 연산을 효율적으로 수행하는 알고리즘으로, 특히 곱셈 후의 모듈러 연산을 빠르게 수행 가능
    - Barrett Reduction: 나눗셈 연산을 곱셈과 시프트 연산으로 대체하여 모듈러 연산을 빠르게 수행하는 알고리즘

- **Matrix A를 저장할 필요 없음**
  - **Kyber와 Dilithium은 Matrix A를 실시간으로 생성하기에 저장할 필요 없음**
    - Matrix A: 다항식 계수를 포함하는 행렬로, SHAKE 함수를 사용하여 생성
  - 즉, SHAKE의 출력을 실시간으로 연산에 사용 가능
    - 이로 인해 메모리 사용량 감소 및 저장 공간 절약 가능

포트폴리오: 벤더 관점에서의 **비선호 부분**

- **산술 다양성: 서로 다른 크기의 모듈러 연산은 하드웨어 구현을 복잡하게 만듦**
  - E.g. Incomplete NTT, Complete NTT
- **다양한 샘플링 방식: 각 샘플링 방식마다 고유의 알고리즘과 구현 방식 필요**
  - 이로 인해 제품 간 일관성 유지가 어려움
- **FO 변환: 부채널 공격에 취약성을 제공할 수 있음**
  - FO(Fujisaki Okamoto) 변환: 원본 메시지에 무작위성을 추가하여 암호화 강도를 높임
  - 변환 시 Decapsulation 단계에서 재암호화 과정 포함
  - 암호문이 복호화되고, 다시 암호화하여 원래 암호문과 비교하여 유효성을 검사하므로 재암호화 단계에서 정보가 누출될 수 있음

포트폴리오: 벤더 관점에서의 **비선호 부분**

- **XOF의 잦은 호출: 모듈 간 분리 및 통합이 어려움**
  - XOF는 호출 마다 별도의 리소스가 필요하기에 모듈 관리가 어려움

*timing attack 테스트: 부채널 공격의 일종으로, 연산이 실행되는 시간을 분석하여 비밀 정보를 추출하는 공격

- **확률적 런타임: *timing attack 테스트가 복잡해짐**
  - 확률적 요소를 가지게 되면 동일한 연산이라도 매번 다른 시간이 소요되어, 타이밍 분석을 어렵게 함

- **부동 소수점 연산(FALCON)**: 부동 소수점 연산은 하드웨어 구현이 어려움
  - 부동 소수점 연산은 normalization(정규화), 반올림, 예외 처리 등의 추가적인 연산 단계 필요
  - 따라서 정수 연산에 비해 더 복잡한 연산기를 필요로 하여 하드웨어 구현이 어려움
    → 그 결과 더 많은 전력을 소모하며, 에너지 효율성 측면에서도 불리함

표준화: 벤더 관점에서의 중요 요소

- **선택된 알고리즘의 신뢰성**
  - **SIKE는 최종 알고리즘으로 선정되기 전 보안 취약점이 발견됨 → 표준화 과정이 효과적으로 작동**
    - 표준화 과정의 목적은 알고리즘의 안전성을 검증하고, 완전히 신뢰할 수 있는 알고리즘을 선정하는 것
  - 기존의 **ECC와 새로운 PQC 알고리즘을 혼합하여 사용**하는 것을 권장
    - ECC는 표준화 과정에서 검증되었기에, 새로운 PQC 알고리즘의 불안정성을 보완 가능

- **ML-KEM 및 ML-DSA**
  - ML-KEM 및 ML-DSA은 하드웨어 구현 시 용이함
    - 주로 간단한 다항식 연산을 기반으로 모듈러 연산을 수행하기 때문

- **SLH-DSA(Sphincs+)**
  - SLH-DSA는 ML-KEM 및 ML-DSA와 유사한 연산을 수행하기에 하드웨어에서 함께 구현하기 용이함
    - 3가지 알고리즘 모두 해시 함수와 모듈러 연산 사용
  - 해시 코어를 재사용하여 효율성 향상 가능
    - 해시 코어: 데이터를 hashing하는 하드웨어 모듈로, 동일한 하드웨어 자원을 활용하여 여러 알고리즘의 연산을 지원할 수 있음

표준화: 벤더 관점에서의 문제점 및 개선 사항

- **너무 많은 후보 알고리즘이 존재하여 학계의 하드웨어 관련 연구에 부담을 줌**
  - FALCON 알고리즘의 부동 소수점 연산에 대한 마스킹 대책이 미비함
    - 최근 FALCON의 부동소수점 곱셈 및 덧셈에 대한 마스킹 방법의 연구 사례가 발표되었음[5]
    - 연구가 계속 진행되고는 있으나, 추가적인 대책이 필요함
  - Fault attack에 대한 연구가 초기 단계에 머물러 있음

- **표준화 과정 막바지에 변경 사항이 생길 경우 부정적인 영향을 미침**
  - 제품 개발에 악영향을 미칠 수 있음
  - **변경 사항은 충분한 시간적 여유를 두고 신중히 검토되어야 함**

- **테스트 벡터가 일찍 제공되어야 함**
  - 알고리즘의 구현 검증 및 호환성 테스트를 위해 필요
  - 일찍 제공이 된다면, 개발자들이 더 나은 품질의 제품을 더 빨리 시장에 출시할 수 있음

[5] Keng-Yu Chen, Jiun-Peng Chen: Masking Floating-Point Number Multiplication and Addition of Falcon: First- and Higher-order Implementations and Evaluations. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2024(2), 276–303. https://doi.org/10.46586/tches.v2024.i2.276-303

표준화: 벤더 관점에서의 표준화 시 권장 사항

보안성이 보장되었을 경우 다음 사항을 권장

- **산술 다양성을 제한할 것**
  - 다양한 알고리즘을 지원하려면 각 알고리즘에 특화된 다양한 연산을 구현해야 함
  - **따라서 산술 다양성을 제한하여 복잡성을 낮추면, 보다 효율적으로 시스템을 설계 및 관리할 수 있음**
  - 또한, 모든 알고리즘에 적용되는 연산을 최적화 하는 것이 개발 및 유지 관리 면에서 효율적임
    - E.g. 구현 가능할 경우, 약간의 성능 저하가 있더라도 ML-DSA와 ML-KEM의 moduli를 재사용

- **메모리 복잡성을 ML-DSA, ML-KEM 수준으로 맞출 것**
  - ML-DSA, ML-KEM에서 사용되는 메모리 모델은 이미 검증되었음
  - 알고리즘이 사용하는 메모리 자원을 가능한 한 최소화하기 위함

- **FO 변환 사용을 지양할 것**
  - Decapsulation 단계에서 암호문이 복호화되고, 재암호화 된 후 원래 암호문과 비교하여 유효성을 검사
  - 따라서 재암호화 단계에서 부채널 공격을 통해 정보가 누출될 수 있음

Q & A