

# AIMer



# HAETAETAE



# DSA 구현 목차 상세

DSA 알고리즘 개요 및 구현 관점 이해

암호 구현의 핵심연산

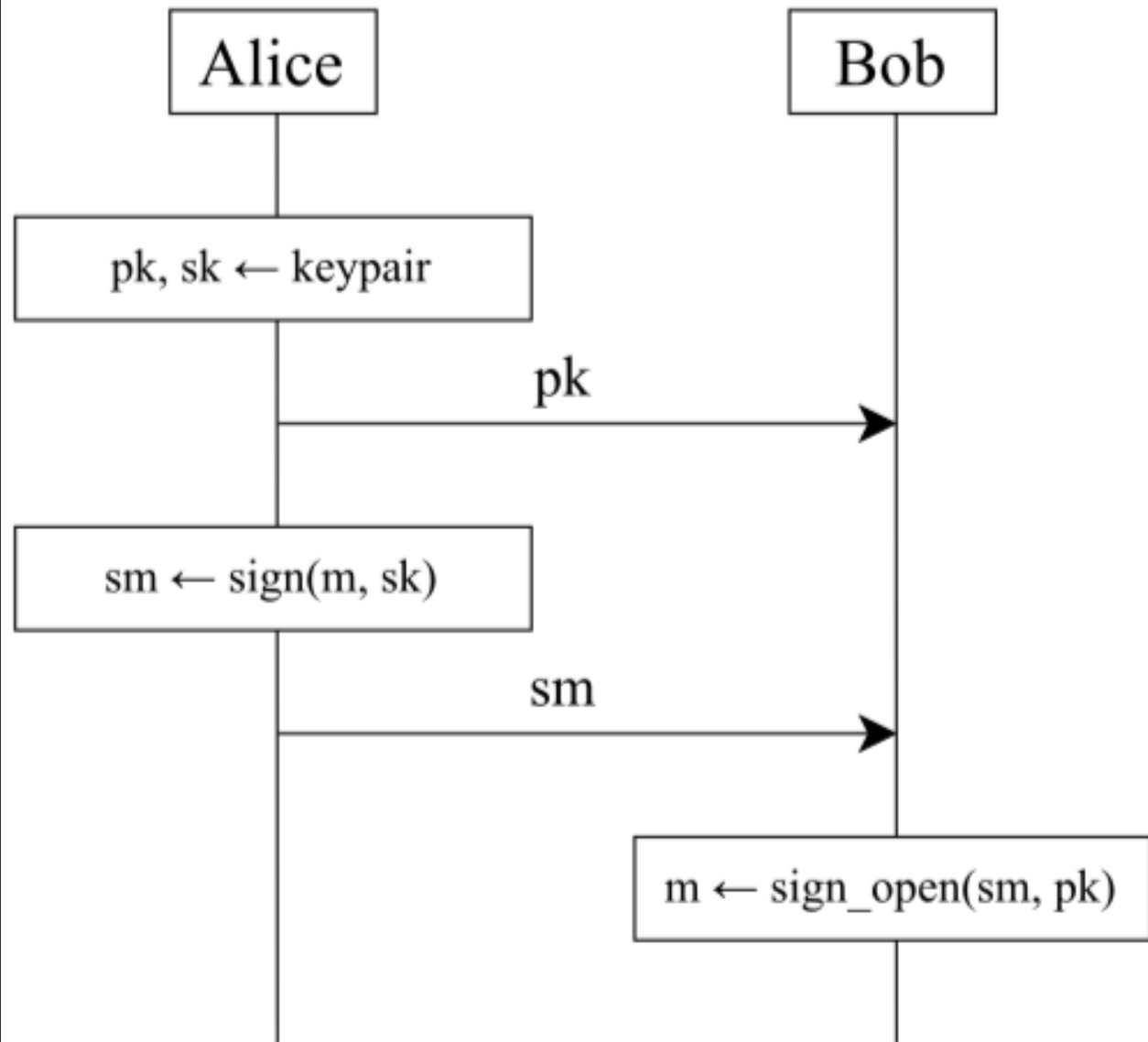
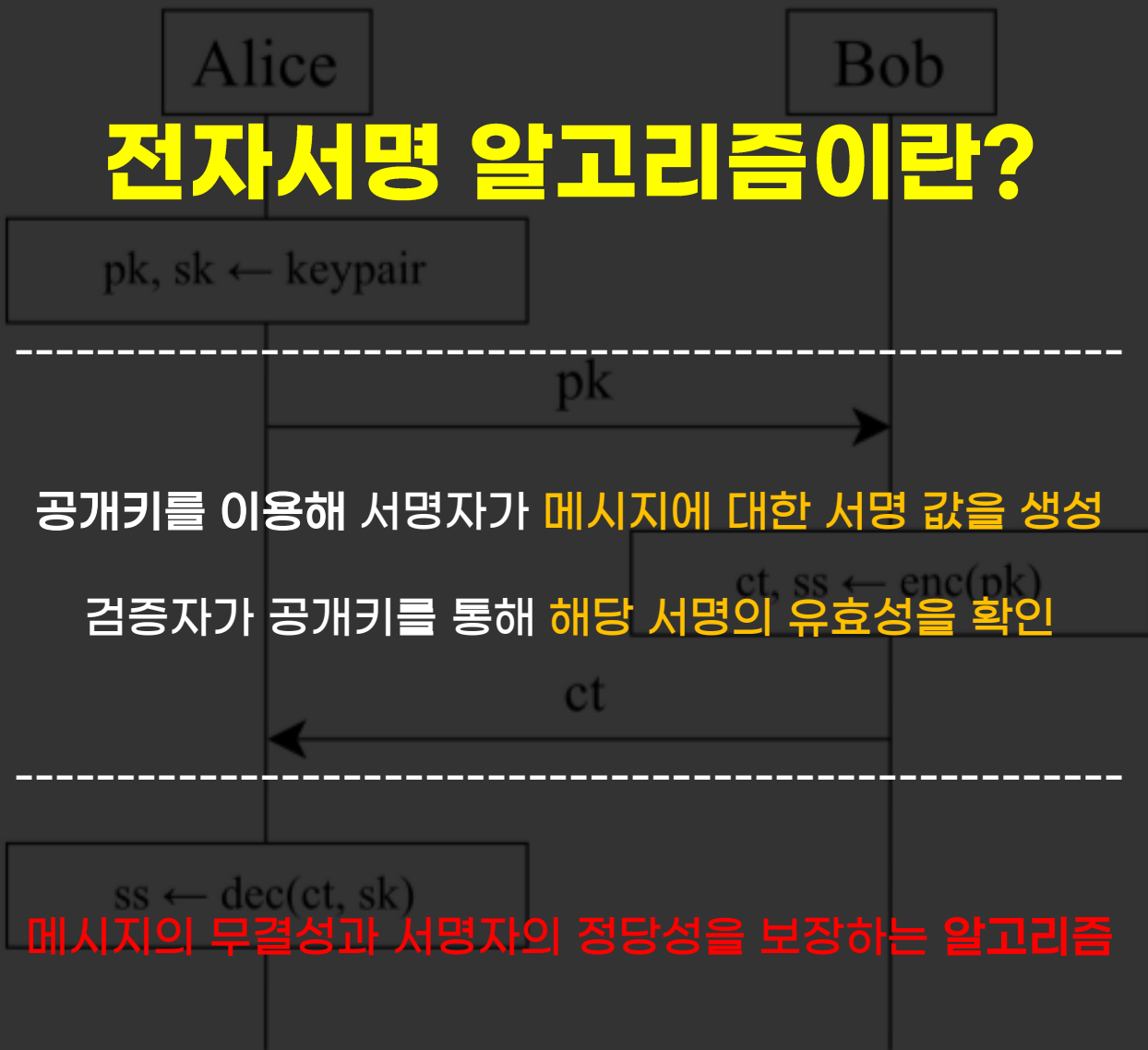
HAETAE 구조 분석

HAETAE: NTT 및 다항식 연산 최적 구현

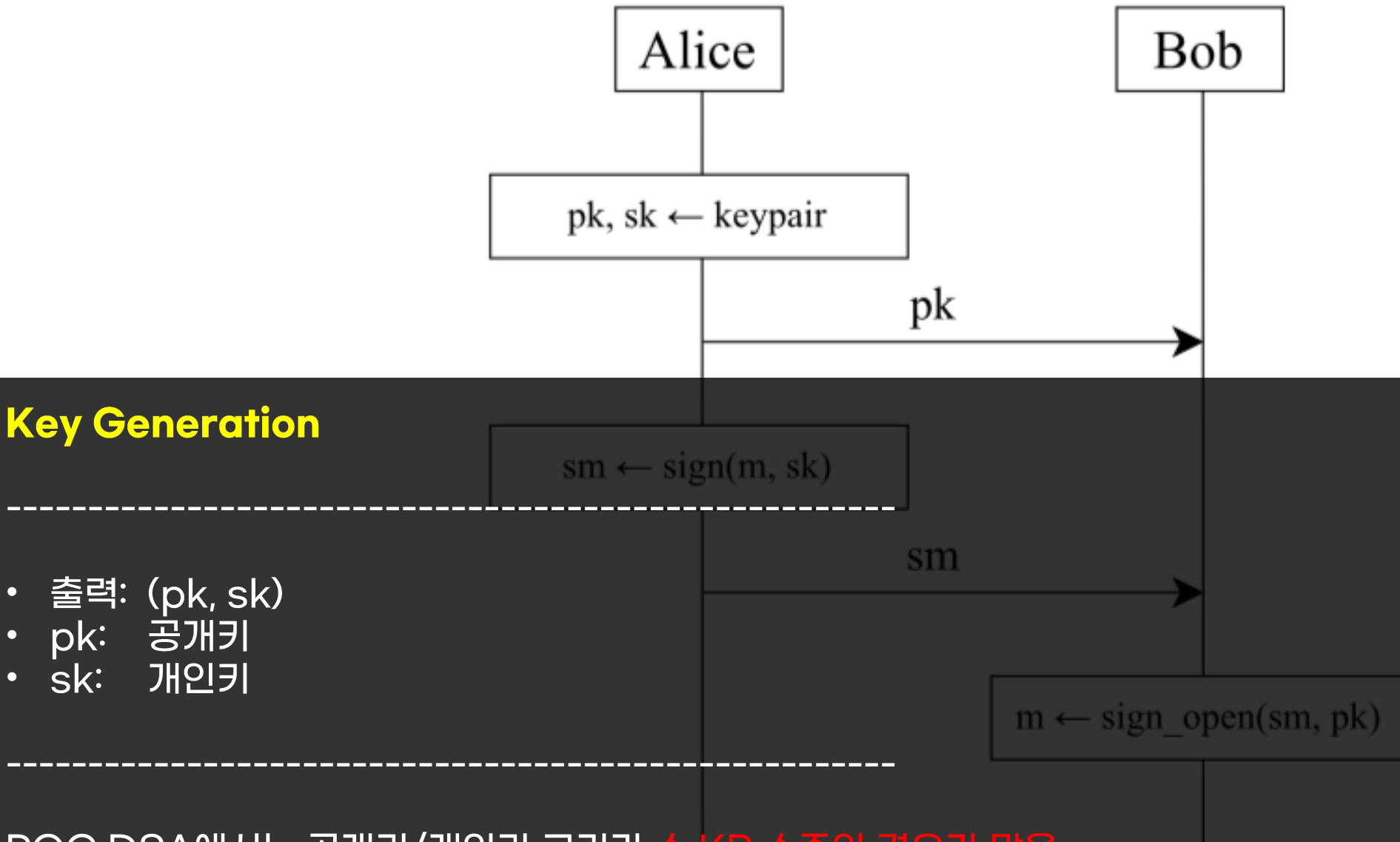
AIMer 및 AIM2 설계

Binary Field 연산과 Constant-Time 구현 기법

# 전자서명 알고리즘이란?



# 암호구현 관점에서의 DSA (상위 API 관점)



PQC DSA에서는 공개키/개인키 크기가 수 KB 수준인 경우가 많음

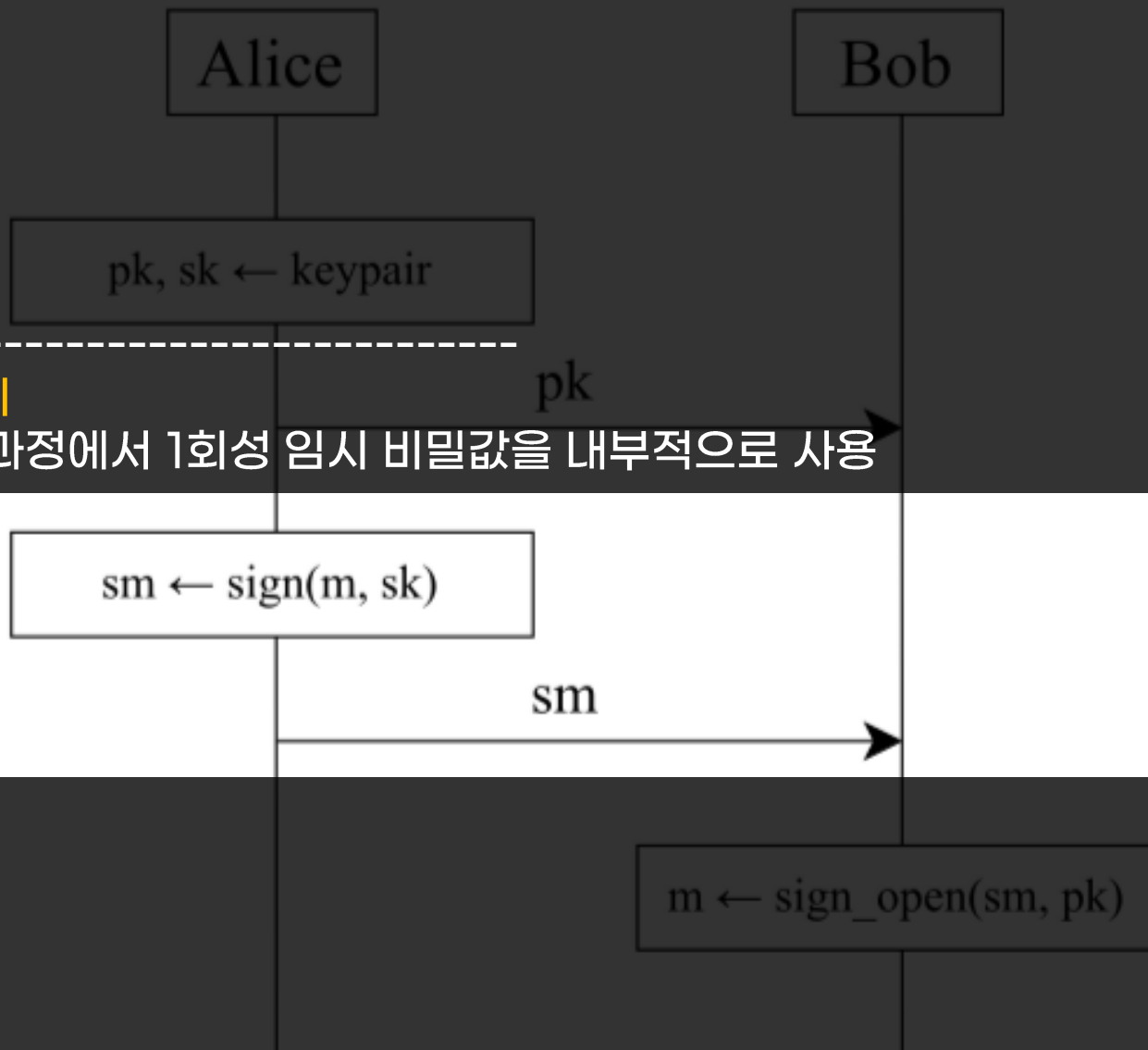
## 암호구현 관점에서의 DSA (상위 API 관점)

- 입력:  $sk, msg$
- 출력:  $sig$

- $msg$ : 메시지
- $sig$ : 서명 값

서명은 장기 비밀키에 의해 정의

장기 비밀키 보호를 위해 서명 과정에서 1회성 임시 비밀값을 내부적으로 사용



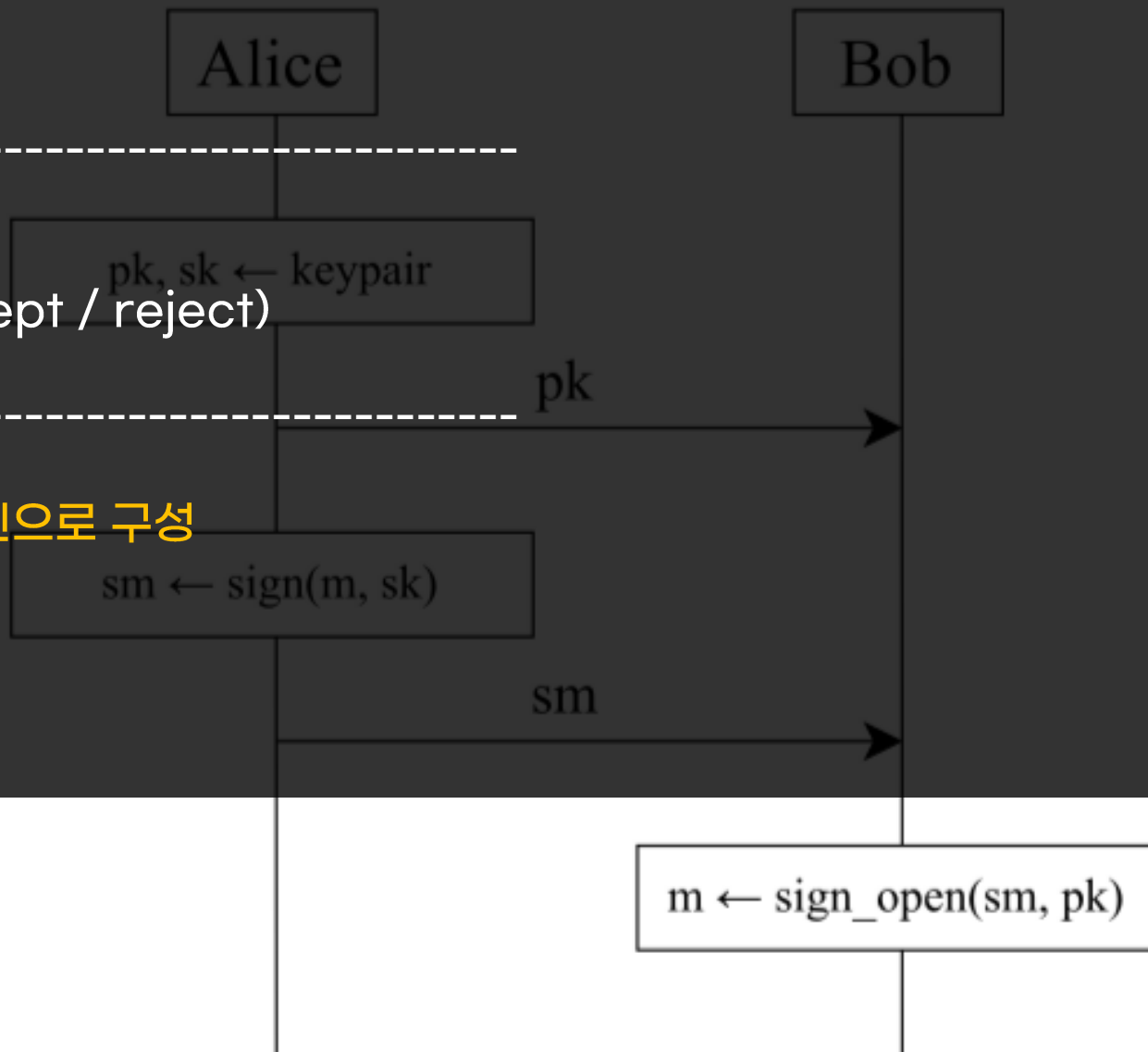


# 암호구현 관점에서의 DSA (상위 API 관점)

## Verification

- 입력:  $pk, msg, sig$
- 출력: 서명 유효 여부 (accept / reject)

검증 과정은 수학적 관계식 확인으로 구성

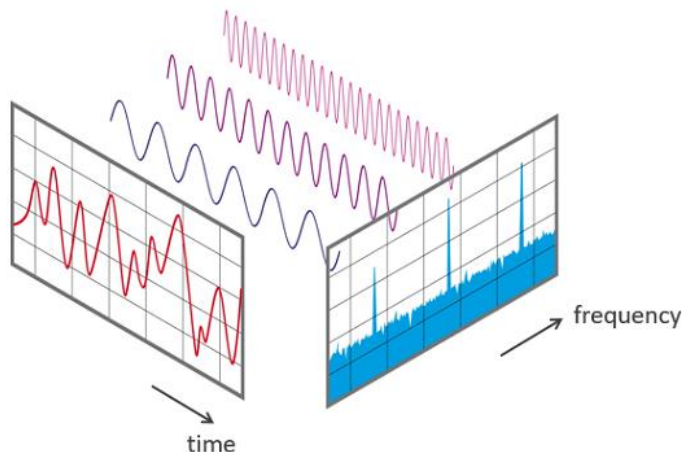


# 암호 최적화 구현 원리

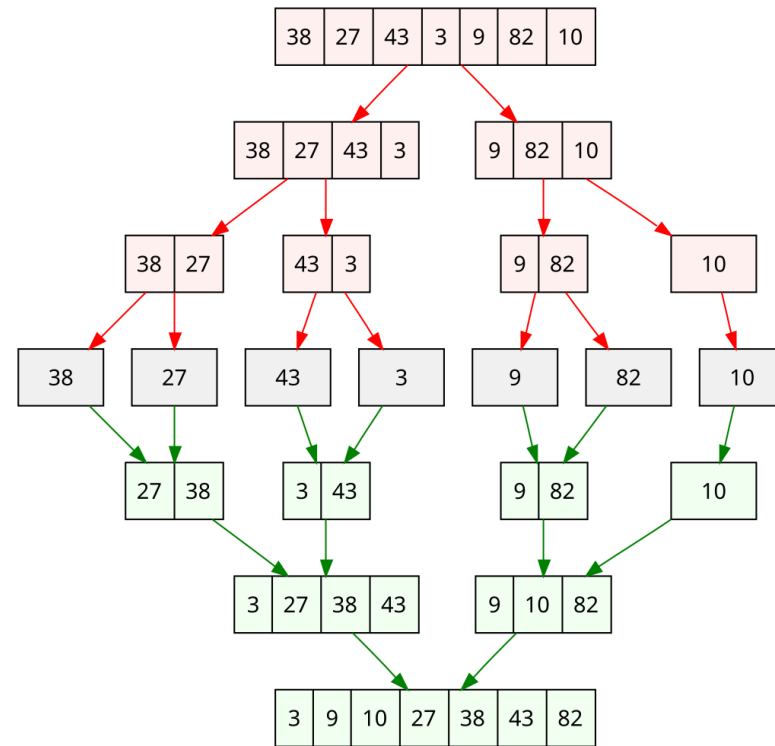
- 암호화 알고리즘을 타겟 플랫폼 상에서 **안전하면서도 가능한 빠르게 구현**

- **암호 구현에서 많이 사용되는 최적화 기술**

- Transform Techniques (변환기법)
- Divide & Conquer (분할 정복 알고리즘)



변환기법을 사용하는 FFT 알고리즘



분할 정복 알고리즘을 통한 Sorting 알고리즘

# 암호에서 가장 중요한 (비싼 혹은 복잡한) 연산은?

## RSA

암호화:  $c \equiv m^e \pmod{n}$

복호화:  $c^d \equiv (m^e)^d \equiv m \pmod{n}$

## ECC

암호화:  $C_1 = k \cdot G, C_2 = M + k \cdot Q$

복호화:  $M = C_2 - d \cdot C_1$

## PQC (격자기반)

암호화:  $c_1 = r^T \cdot A \pmod{q} (\in Z_q^n), c_2 = r^T \cdot b + \left\lfloor \frac{q}{2} \right\rfloor \mu \pmod{q} (\in Z_q)$

복호화:  $v = c_2 - c_1 \cdot s \pmod{q}, \mu = 0 \text{ if } v \approx 0, \mu = 1 \text{ if } v \approx \left\lfloor \frac{q}{2} \right\rfloor$



## 암호에서 가장 중요한 연산은?

여기서 사용하는 곱셈은 우리가 아는 일반적인 곱셈

## RSA

암호화:  $c \equiv m^e \pmod{n}$

복호화:  $c^d \equiv (m^e)^d \equiv m \pmod{n}$

## ECC

암호화:  $C_1 = k \cdot G, C_2 = M + k \cdot Q$

복호화:  $M = C_2 - d \cdot C_1$

## PQC (격자기반)

ECC와 스칼라 곱셈이고 PQC의 매트릭스 곱셈

암호화:  $c_1 = r^T \cdot A \pmod{q} (\in \mathbb{Z}_q^n), c_2 = r^T \cdot b + \left\lfloor \frac{q}{2} \right\rfloor \mu \pmod{q} (\in \mathbb{Z}_q)$

복호화:  $v = c_2 - c_1 \cdot s \pmod{q}, \mu = 0 \text{ if } v \approx 0, \mu = 1 \text{ if } v \approx \left\lfloor \frac{q}{2} \right\rfloor$

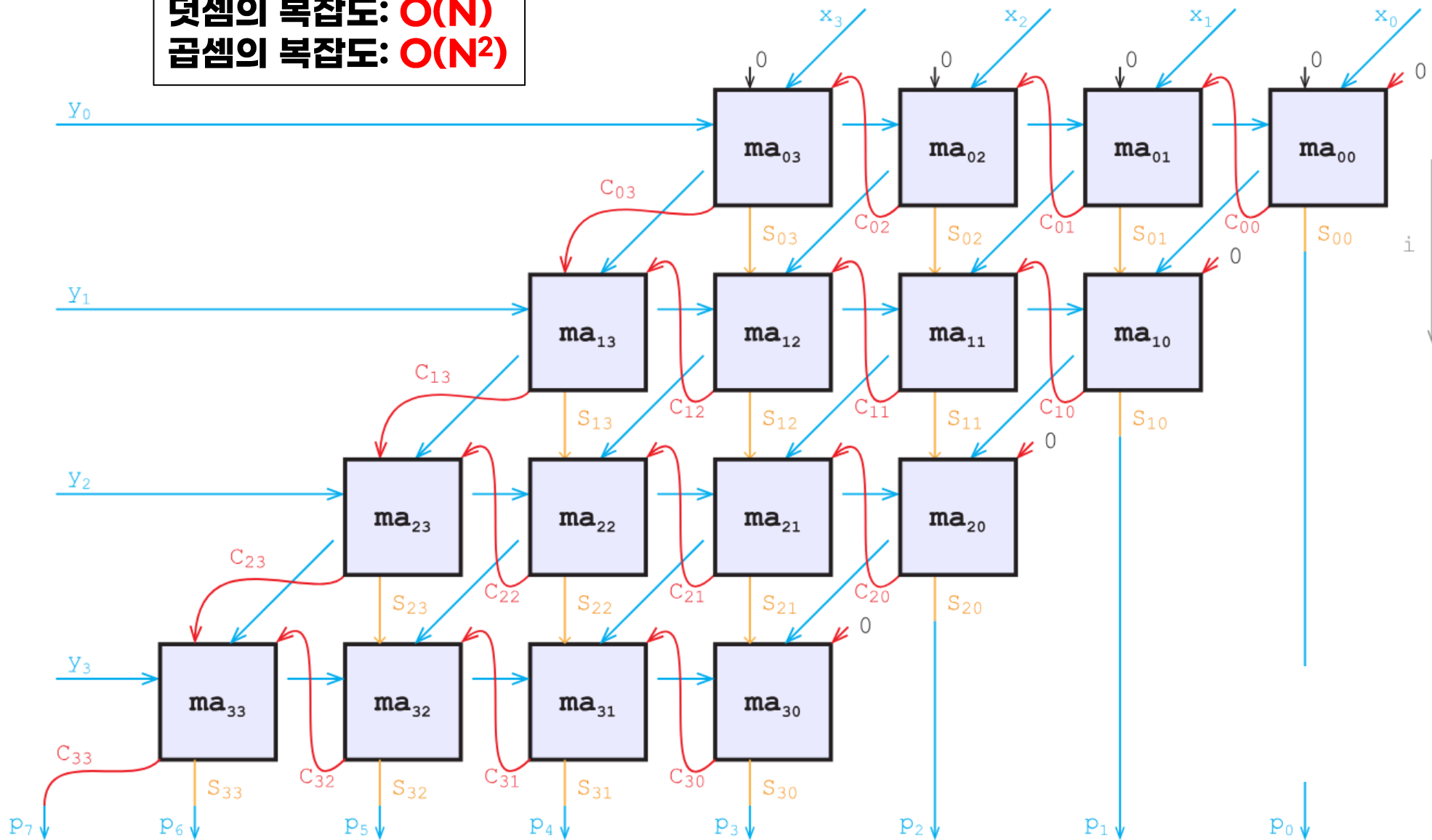
곱셈



복잡한 암호도 결국 사칙연산을 통해 구현됨

사칙연산 중에서 **가장 비싼 연산은 나눗셈**이지만 이는 대부분 알고리즘을 통해 **곱셈으로 대체** 가능  
따라서 곱셈 연산이 암호 구현에서 가장 중요 (비용이 많이 소요) → 최적화 효율성이 가장 높음

덧셈의 복잡도:  $O(N)$   
곱셈의 복잡도:  $O(N^2)$



# 곱셈 구현 기법 #1

- **Karatsuba Multiplication: 단위 곱셈 횟수를 줄여주는 알고리즘**

- $n$  워드  $A$ 와  $B$ 에 대한 곱셈 ( $A = A_H 2^{\frac{n}{2}} + A_L, B = B_H 2^{\frac{n}{2}} + B_L$ )

- 기본적인 방법은 **4번의 곱셈** 필요:  $O(n^2)$

- $A_H B_H 2^n + A_H B_L 2^{\frac{n}{2}} + A_L B_H 2^{\frac{n}{2}} + A_L B_L$

- Karatsuba의 경우 **3번의 곱셈**만 필요:  $O(n^{\log_2 3})$

- $A_H B_H 2^n + ((A_H + A_L)(B_H + B_L) - A_L B_L - A_H B_H) 2^{\frac{n}{2}} + A_L B_L$

# Karatsuba Multiplication 예시

$$x = 1234, y = 5678$$

## 1. 수분할 (기준: 2자리씩 나누기)

- Karatsuba Multiplication: 만위 곱셈 횟수를 줄여주는 알고리즘

$$x = 1234 \rightarrow a = 12, b = 34$$

$$y = 5678 \rightarrow c = 56, d = 78$$

- $n$  워드  $A$ 와  $B$ 에 대한 곱셈 ( $A = A_H 2^{\frac{n}{2}} + A_L, B = B_H 2^{\frac{n}{2}} + B_L$ )

## 2. 세가지 곱셈 수행

- 기본적인 방법은 4번의 곱셈 필요:  $O(n^2)$

$$ac = a \times c = 12 \times 56 = 672$$

$$bd = b \times d = 34 \times 78 = 2652$$

$$(a + b)(c + d) = (12 + 34) \times (56 + 78) = 46 \times 134 = 6164$$

- Karatsuba의 경우 3번의 곱셈만 필요:  $O(n^{\log_2 3})$

## 3. 중간 항 계산 (w/o 곱셈 연산)

$$ad + bc = (a + b)(c + d) - ac - bd = 6164 - 672 - 2652 = 2840$$

## 4. 자리수에 맞게 정렬하여 결과 합산

$$x \times y = ac \times 10^4 + (ad + bc) \times 10^2 + bd$$

$$= 672 \times 10^4 + 2840 \times 10^2 + 2652$$

$$= 6720000 + 284000 + 2652$$

$$= 7006652$$

# Reduction 구현 기법#1

- **Montgomery Reduction: 나눗셈 연산을 곱셈으로 대체하는 알고리즘**
  - Modulus  $m$ 은  $r$ 과 서로소 관계
  - Montgomery radix는 다음 공식에 따라 설정
    - $r^{n-1} < m < r^n$
- Reduction이 필요한 수  $c$  ( $0 \leq c < m^2$ )  
→ Montgomery reduction이 수행될 목표
- Montgomery reduction 공식
  - $\frac{c + (\mu \cdot c \bmod r^n) \cdot m}{r^n} \equiv c \cdot r^{-n} \pmod{m}$
  - $\mu = -m^{-1} \bmod r^n$  해당 값은 사전 계산 가능

# Reduction 구현 기법#1

## • Montgomery reduction 공식

$$u = \frac{c + (c \cdot \mu \bmod r^n) \cdot m}{r^n} \equiv c \cdot r^{-n} \pmod{m}$$

- $\mu = -m^{-1} \bmod r^n$  해당 값은 사전 계산 가능

- $t \equiv (c \cdot \mu) \bmod r^n$  ( $0 \leq t < r^n$ )

- 어떤 정수  $k$ 가 존재하여  $t = c \cdot \mu + k \cdot R$

$$u = c + t \cdot m$$

- $u = c + t \cdot m \equiv c + (c \cdot \mu) \cdot m \pmod{r^n}$

- 여기서  $\mu \equiv -m^{-1} \pmod{r^n}$  이므로  $\mu \cdot m \equiv -1 \pmod{r^n}$

- 따라서  $u \equiv c + c \cdot (m \cdot \mu) \equiv c + c(-1) \equiv 0 \pmod{r^n}$

- 곱셈 결과값의 하위  $r^n$  부분은 0으로 세팅되어 시프트 연산으로 reduction 수행



# Reduction 구현 기법#1

- **Montgomery Reduction: 나눗셈 연산을 곱셈으로 대체하는 알고리즘**

- Montgomery Reduction 공식

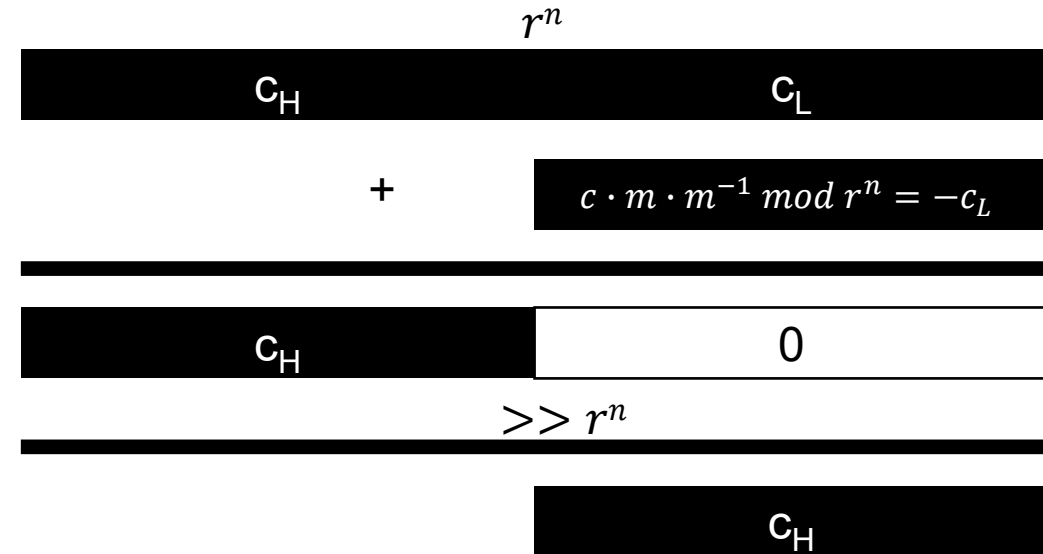
- $\frac{c + (\mu \cdot c \bmod r^n) \cdot m}{r^n} \equiv c \cdot r^{-n} \pmod{m}$
- $\mu = -m^{-1} \bmod r^n$  해당 값은 사전 계산 가능

- Montgomery Domain 상에서의 연산

- $\tilde{a} = a \cdot r^n \bmod m$
- $\tilde{b} = b \cdot r^n \bmod m$

- 두 변수의 곱에 대한 Montgomery Reduction

- $\tilde{a} \cdot \tilde{b} \equiv a \cdot b \cdot r^{2n} \pmod{m}$  에 대한 Montgomery Reduction
- $a \cdot b \cdot r^{2n} r^{-n} \equiv a \cdot b \cdot r^n \pmod{m}$



- **Montgomery Reduction**을 **Montgomery Multiplication**으로 확장
  - $a$ 와  $b$ 가 주어진 경우
  - 1단계:  $a$ 와  $b$ 를 Montgomery Domain으로 변환
    - $\tilde{a} = a \cdot r^n \bmod m$
    - $\tilde{b} = b \cdot r^n \bmod m$
  - 2단계: Montgomery Multiplication 수행
    - $\tilde{c} = \tilde{a} \cdot \tilde{b} \cdot r^{-n} \bmod m$
  - 3단계:  $\tilde{c}$ 를 일반 도메인 ( $c$ )으로 변환

# • Montgomery Reduction을 Montgomery Multiplication으로 확장

• 예시:  $a \cdot a \bmod m$  ( $a = 5, m = 11$ )

•  $r^n = 16$  ( $n = 4$ ),  $r2 = r^{2n} \bmod m = 3$ ,  $\mu = -m^{-1} \bmod r^n = 13$

• 1단계:  $MontMul(a, r2) = a \cdot r^n \bmod m = \tilde{a}$

$$= \frac{(a \cdot r2) + ((a \cdot r2) \cdot \mu \bmod r^n \cdot m)}{r^n} = \frac{(5 \cdot 3) + ((5 \cdot 3) \cdot 13 \bmod 16 \cdot 11)}{r^n}$$

$$= \frac{15 + (195 \bmod 16 \cdot 11)}{r^n} = \frac{15 + 33}{r^n} = 48 \gg 4 = 3$$

• 2단계:  $MontMul(\tilde{a}, \tilde{a}) = a \cdot a \cdot r^n \bmod m = \tilde{c}$

$$= \frac{(a \cdot r^n \cdot a \cdot r^n) + ((a \cdot r^n \cdot a \cdot r^n) \cdot \mu \bmod r^n \cdot m)}{r^n}$$

$$= \frac{(3 \cdot 3) + ((3 \cdot 3) \cdot 13 \bmod 16 \cdot 11)}{r^n} = \frac{(9) + (117 \bmod 16 \cdot 11)}{r^n}$$

$$= \frac{9 + 55}{r^n} = 64 \gg 4 = 4$$

$  \begin{aligned}  a \cdot a \bmod m &= 5 \cdot 5 \bmod 11 \\  &= 25 \bmod 11 \\  &= 3  \end{aligned}  $
---

• 3단계:  $MontMul(\tilde{c}, 1) = a \cdot a \bmod m$

$$= \frac{(a \cdot a \cdot r^n \cdot 1) + ((a \cdot a \cdot r^n \cdot 1) \cdot \mu \bmod r^n \cdot m)}{r^n} = \frac{4 + (4 \cdot 13 \bmod 16 \cdot 11)}{r^n}$$

$$= \frac{4 + 44}{r^n} = 48 \gg 4 = 3$$

## 곱셈 구현 기법 #2

- **NTT: Ring 상에서의 CRT 적용 기법**

- RSA와 같이 ML-DSA도 많은 곱셈 연산으로 구성 (다만 입력 데이터 형식은 다름)
  - RSA는 large number를 사용 (일반 곱셈 w/ big integer)
  - ML-DSA는 ring을 사용 (다항식 곱셈)
- Toy ring:  $R = Z_q[X]/(X^n + 1) = Z_{17}[X]/(X^4 + 1)$ 
  - Multiplication in rings:  $c_k = (a \cdot b)_k = \sum_{i+j=k} a_i b_j$ 
    - Toy ring 상에서는 16 번의 inner multiplication 필요 ( $a_i \cdot b_i$ )

# 곱셈 구현 기법 #2

## • NTT: Ring 상에서의 CRT 적용 기법

- Ring 상에 CRT를 적용하여 coefficient multiplication 횟수를 줄일 수 있음

- Ring 상에서의 modulus

- $N = X^4 + 1$

- $m_1 \cdot m_2 = X^4 + 1$

- $m_1 = (X^2 - 4)$

- $m_2 = (X^2 + 4)$

- $m_1 \cdot m_2 = (X^2 - 4)(X^2 + 4) = X^4 - 4X^2 + 4X^2 - 16 = X^4 - 16 \equiv X^4 + 1$

- $R = Z_{17}[X]/(X^4 + 1)$

- $R_1 = Z_{17}[X]/(X^2 - 4)$

- $R_2 = Z_{17}[X]/(X^2 + 4)$

- 예시)  $a = 2 + 7X^3$

- $a \bmod (X^2 - 4) \equiv 2 + 7X^3 \equiv 2 + 7X^3 - 7X(X^2 - 4) \equiv 2 + (7 \cdot 4)X \equiv 2 + 28X \equiv 2 + 11X$

- $a \bmod (X^2 + 4) \equiv 2 + 7X^3 \equiv 2 + 7X^3 - 7X(X^2 + 4) \equiv 2 - (7 \cdot 4)X \equiv 2 - 28X \equiv 2 + 6X$

## 곱셈 구현 기법 #2

- **NTT: Ring 상에서의 CRT 적용**

- **Recurring down (degree-0까지)**

- $R = \mathbb{Z}_{17}[X]/(X^4 + 1)$

- $R_1 = \mathbb{Z}_{17}[X]/(X^2 - 4)$

- $R_2 = \mathbb{Z}_{17}[X]/(X^2 + 4)$

- $R_1 = \mathbb{Z}_{17}[X]/(X^2 - 4) = \mathbb{Z}_{17}[X]/(X + \zeta)(X - \zeta)$

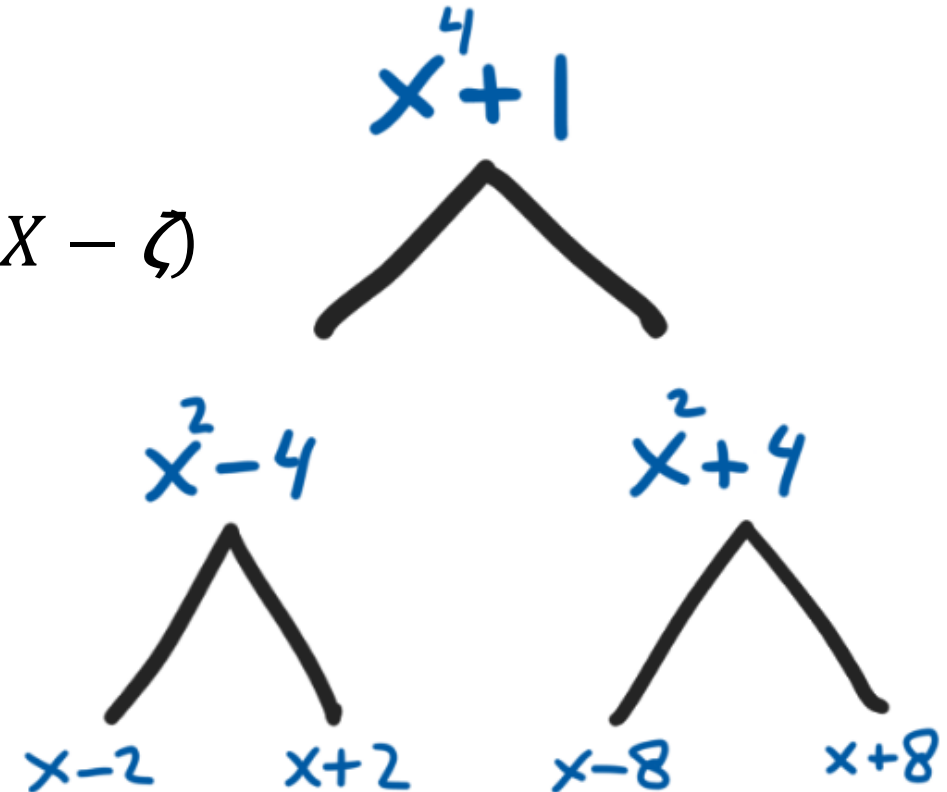
- $\zeta^2 = 4$

- $\zeta = \sqrt{4} = 2$

- $R_1 = \mathbb{Z}_{17}[X]/(X^2 - 4)$

- $R_{1,1} = \mathbb{Z}_{17}[X]/(X + 2)$

- $R_{1,2} = \mathbb{Z}_{17}[X]/(X - 2)$



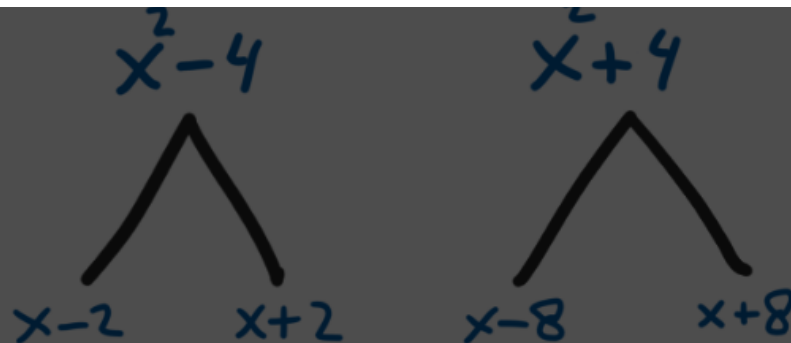


## 곱셈 구현 기법 #2

- NTT: Ring 상에서의 CRT 적용

두 polynomial  $a$ 와  $b$ 를 곱하여  $c$ 를 생성하기 위해선

- 1)  $a$ 와  $b$ 를  $N$  degree-0 polynomial로 변환 (NTT)
- 2)  $a_i, b_i \in R_i$  일 때 pointwise 곱셈을 통해  $c_i = a_i b_i$  연산
- 3) CRT를 사용하여  $c_i$ 를  $c$ 로 변환 (INTT)




## 곱셈 구현 기법 #2

- **ML-DSA/HAETAЕ의 ring에 대한 정의 (실제로는 negacyclic)**

- $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$  : NTT 적용을 위해  $(X^{128} - \alpha)(X^{128} + \alpha) = (X^{256} + 1)$

$$(X^{256} + 1) = (X^{128} - \alpha)(X^{128} + \alpha)$$
$$(X^{256} + 1) = X^{256} + (-\alpha + \alpha)X^{128} - \alpha^2$$


$$1 = -\alpha^2$$

$$\alpha^2 = -1$$

$$\alpha^4 = 1$$

$\alpha = \sqrt[4]{-1}$  (fourth primitive root of -1;

달리 말하면  $\alpha^1, \alpha^2, \alpha^3$ 은 1이 아니어야 함; 오직  $\alpha^4 = 1$ )

## 곱셈 구현 기법 #2

$$(X^{128} - \alpha) = (X^{64} - \beta)(X^{64} + \beta) = X^{128} + (-\beta + \beta)X^{64} - \beta^2$$

$$-\alpha = -\beta^2$$

$$\alpha = \beta^2$$

$$\beta = \sqrt{\alpha}$$

$$(X^{256} + 1) = (X^{128} - \alpha)(X^{128} + \alpha)$$


$$(X^{128} + \alpha) = (X^{64} - \gamma)(X^{64} + \gamma) = X^{128} + (-\gamma + \gamma)X^{64} - \gamma^2$$

$$\alpha = -\gamma^2$$

$$\gamma = \sqrt{-\alpha} = \sqrt{(-1) \cdot \alpha} = \sqrt{\alpha^2 \cdot \alpha} = (\sqrt{\alpha})^3$$

# 곱셈 구현 기법 #2

$$(X^{128} - \alpha) = (X^{64} - \beta)(X^{64} + \beta) = X^{128} + (-\beta + \beta)X^{64} - \beta^2$$



$$\begin{aligned} -\alpha &= -\beta^2 \\ \alpha &= \beta^2 \\ \beta &= \sqrt{\alpha} \end{aligned}$$

## 일반화

특정 polynomial의 값이  $x$ 일 때 one layer를 내려갈 경우 값은 다음과 같이 변화

$$y_1 = \sqrt{x} \text{ 그리고 } y_2 = (\sqrt{x})^3$$

이를 보다 일반화 하여 사용할 경우  $\zeta_k$  로 표현 가능

$\zeta_k$ 는  $k$ -th primitive root of 1을 나타내며 수학식으로는  $\zeta_k = \sqrt[k]{1}$ 로 나타냄

$$\gamma = \sqrt{-\alpha} = \sqrt{(-1) \cdot \alpha} = \sqrt{\alpha^2 \cdot \alpha} = (\sqrt{\alpha})^3$$

# 곱셈 구현 기법 #2

$$(X^{128} - \alpha) = (X^{64} - \beta)(X^{64} + \beta) = X^{128} + (-\beta + \beta)X^{64} - \beta^2$$

$$\beta = \sqrt{\alpha} = \sqrt{\zeta_4} = \zeta_8$$

$$\gamma = (\sqrt{\alpha})^3 = (\sqrt{\zeta_4})^3 = \zeta_8^3$$

$$(X^{256} + 1) = (X^{128} - \alpha)(X^{128} + \alpha)$$

$$= (X^{64} - \zeta_8)(X^{64} + \zeta_8)(X^{64} - \zeta_8^3)(X^{64} + \zeta_8^3)$$

$$= (X^{32} - \zeta_{16})(X^{32} + \zeta_{16})(X^{32} - \zeta_{16}^5)(X^{32} + \zeta_{16}^5)(X^{32} - \zeta_{16}^3)(X^{32} + \zeta_{16}^3)(X^{32} - \zeta_{16}^7)(X^{32} + \zeta_{16}^7)$$

$$(X^{128} + \alpha) = (X^{64} - \gamma)(X^{64} + \gamma) = X^{128} + (-\gamma + \gamma)X^{64} - \gamma^2$$

$$\alpha = -\gamma^2$$

$$\gamma = \sqrt{-\alpha} = \sqrt{(-1) \cdot \alpha} = \sqrt{\alpha^2 \cdot \alpha} = (\sqrt{\alpha})^3$$

## 곱셈 구현 기법 #2

- Polynomial에 대한 reduction을 반복적으로 수행하여 **degree-0 polynomial**로 만듦
- 512-th primitive root of 1 ( $\zeta_{512}$ )까지 reduction (negacyclic for ML-DSA/HAETA-E)
  - 여기서는 간단히  $\zeta_{512}$ 를  $\zeta$ 로 표기
- 따라서  $(X^{256} + 1)$ 에 대한 factorization 결과는 다음과 같음
  - Negacyclic 구조로써 원시근의 홀수항만 존재
  - $(X - \zeta)(X + \zeta)(X - \zeta^{129})(X + \zeta^{129}) \dots (X - \zeta^{127})(X + \zeta^{127})(X - \zeta^{255})(X + \zeta^{255})$



## 곱셈 구현 기법 #2

### • Forward Transform

- $a \in Z_q[X]/(X^{256} + 1)$ 
  - $a_L^{(1)} \in Z_q[X]/(X^{128} - \zeta^{128})$

$$\rightarrow X^{128} = \zeta^{128} \rightarrow a_i X^i = a_i \zeta^{128} X^{i-128}$$

$$\bullet a_L^{(1)} = (a_0 + \zeta^{128} a_{128}) + (a_1 + \zeta^{128} a_{129})X + (a_2 + \zeta^{128} a_{130})X^2 + \dots$$

- $a_R^{(1)} \in Z_q[X]/(X^{128} + \zeta^{128})$

$$\rightarrow X^{128} = -\zeta^{128} \rightarrow a_i X^i = -a_i \zeta^{128} X^{i-128}$$

$$\bullet a_R^{(1)} = (a_0 - \zeta^{128} a_{128}) + (a_1 - \zeta^{128} a_{129})X + (a_2 - \zeta^{128} a_{130})X^2 + \dots$$

- 두 식을 살펴보면 병렬로 적용가능한 부분 확인이 가능

- 즉  $\zeta^{128}$ 를 곱하는 부분이 공통분모이며 이를 더하고 빼는 루틴을 반복

- 이를 butterfly 연산이라고 함

- 해당 연산을 degree-0까지 수행 ( $\log_2 n$  layer 만큼)하고 base 곱셈을 수행

- Transform:  $O(n \log n)$

- Base 곱셈:  $O(n)$

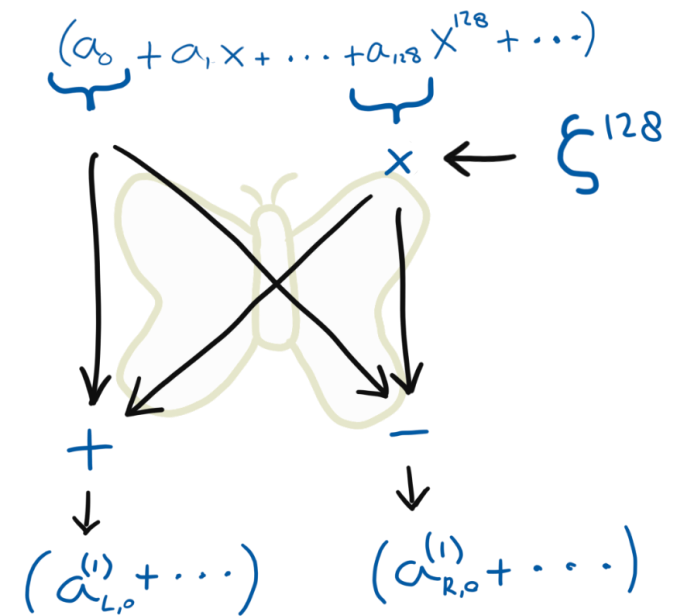
$$\rightarrow X^{128} = \zeta^{128} \rightarrow a_i X^i = a_i \zeta^{128} X^{i-128}$$

- $a_L^{(1)} = (a_0 + \zeta^{128} a_{128}) + (a_1 + \zeta^{128} a_{129})X + (a_2 + \zeta^{128} a_{130})X^2 + \dots$

- $a_R^{(1)} \in Z_q[X]/(X^{128} + \zeta^{128})$

$$\rightarrow X^{128} = -\zeta^{128} \rightarrow a_i X^i = -a_i \zeta^{128} X^{i-128}$$

- $a_R^{(1)} = (a_0 - \zeta^{128} a_{128}) + (a_1 - \zeta^{128} a_{129})X + (a_2 - \zeta^{128} a_{130})X^2 + \dots$



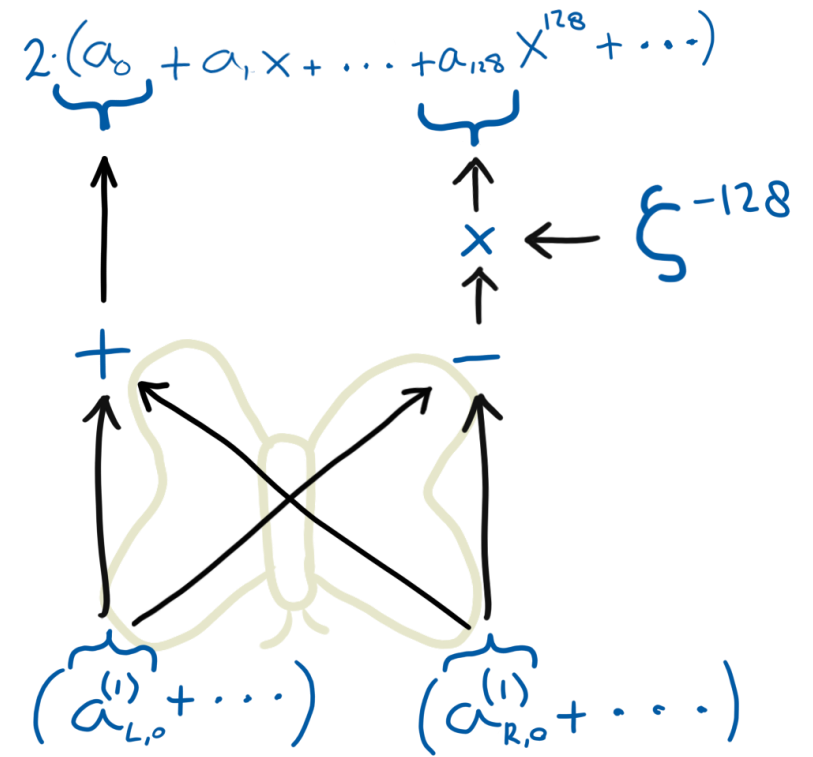
### • Inverse Transform

- $a_{L,0}^{(1)} = a_0 + \zeta^{128} a_{128}$        $a_{R,0}^{(1)} = a_0 - \zeta^{128} a_{128}$
- $a_{L,0}^{(1)} + a_{R,0}^{(1)} = a_0 + \zeta^{128} a_{128} + a_0 - \zeta^{128} a_{128} = 2a_0$
- $a_0 = 2^{-1} (a_{L,0}^{(1)} + a_{R,0}^{(1)})$
- $a_{L,0}^{(1)} - a_{R,0}^{(1)} = a_0 + \zeta^{128} a_{128} - a_0 + \zeta^{128} a_{128} = 2\zeta^{128} a_{128}$
- $a_{128} = 2^{-1} \zeta^{-128} (a_{L,0}^{(1)} - a_{R,0}^{(1)})$

## 곱셈 구현 기법 #2

### • Inverse Transform

- $a_{L,0}^{(1)} = a_0 + \zeta^{128} a_{128}$
- $a_{L,0}^{(1)} + a_{R,0}^{(1)} = a_0 + \zeta^{128} a_{128} + a_0 -$
- $a_0 = 2^{-1} (a_{L,0}^{(1)} + a_{R,0}^{(1)})$



- 각 layer에서 추가적으로  $2^{-1}$  곱셈 연산이 수행
  - $l$  개의 layer라면  $2^{-l}$  constant를 한번 곱해서 결과값 도출 가능
  - 따라서 다음과 같은 식 도출
    - $2a_0 = (a_{L,0}^{(1)} + a_{R,0}^{(1)})$
    - $2a_{128} = \zeta^{-128} (a_{L,0}^{(1)} - a_{R,0}^{(1)})$

## Forward Transformation (전방향 변환) $\pi$

## Reverse Transformation (역방향 변환)

Original Data  
(원본 데이터)

NTT Transform  
(NTT 변환)

State changed, still H2O

$\Sigma$

Montgomery Transform  
(몽고메리 변환)

Color changed,  
essence unchanged

$$A \cdot R^{-1} \bmod N$$

Both Applied  
(몽고메리 + NTT)

Looks completely different,  
but fundamentally the same water  
- Original data with both transformations

$$\text{NTT}(A \cdot R^{-1} \bmod N)$$

Montgomery<sup>-1</sup>  
(역 몽고메리)

- Removing color

$$A \bmod N$$

INTT (역 NTT)

- Melting ice back to liquid

$$\sum_{\text{H}_2\text{O}}^{-1}$$

## Forward Transformation (전방향 변환)

## Reverse Transformation (역방향 변환)

도메인 변환은 다양한 조합이 가능

동일한 도메인으로 맞추어 놓고

계산하지 않으면 엉뚱한 결과값 도출

$$a \xrightarrow{Mont} \tilde{a} \xrightarrow{NTT} \hat{\tilde{a}}$$

$$a \xleftarrow{IMont} \tilde{a} \xleftarrow{INTT} \hat{\tilde{a}}$$

$$a \xrightarrow{NTT} \hat{a} \xrightarrow{Mont} \tilde{\hat{a}}$$

$$a \xleftarrow{INTT} \hat{a} \xleftarrow{IMont} \tilde{\hat{a}}$$

### Both Applied (몽고메리 + NTT)

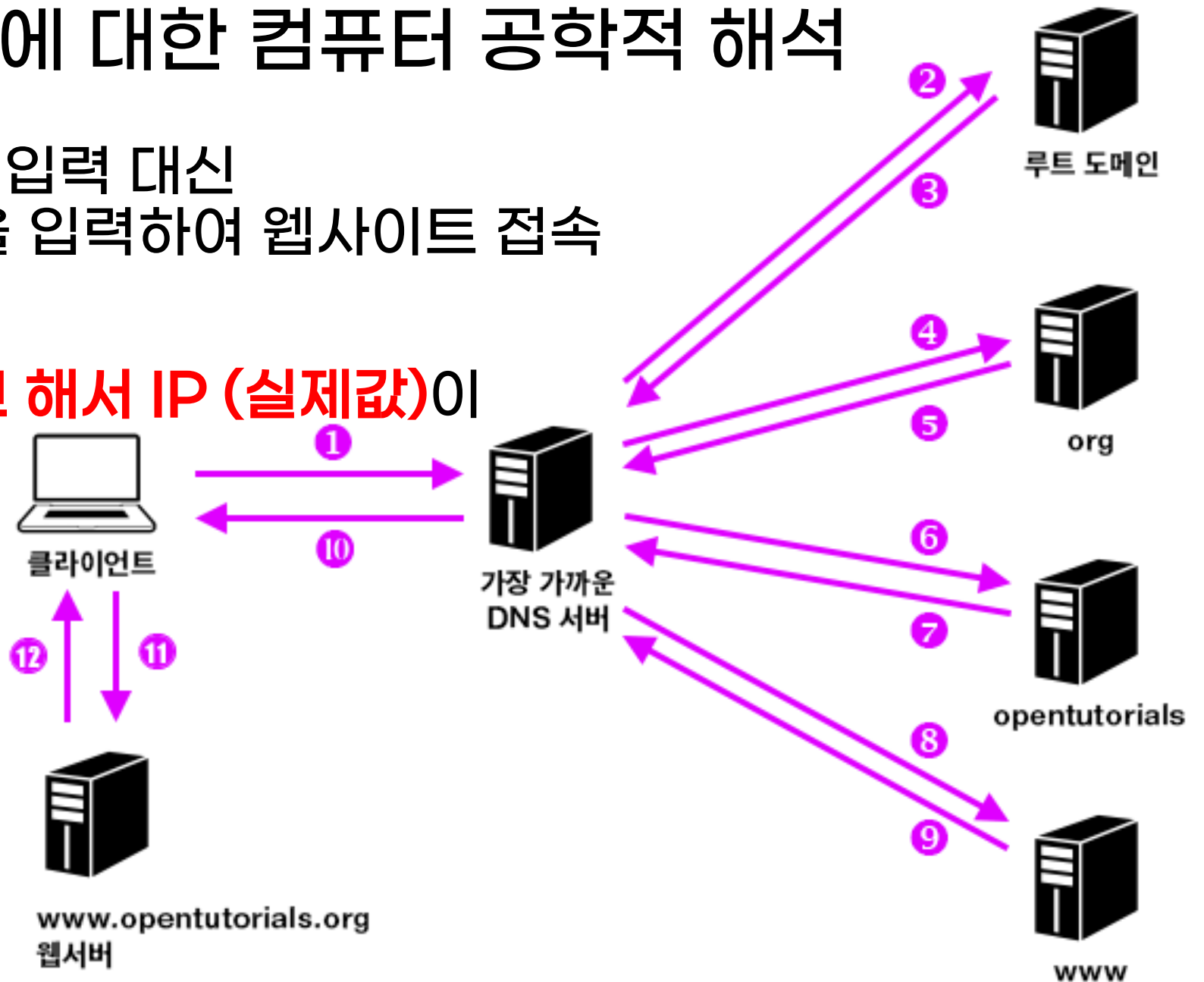
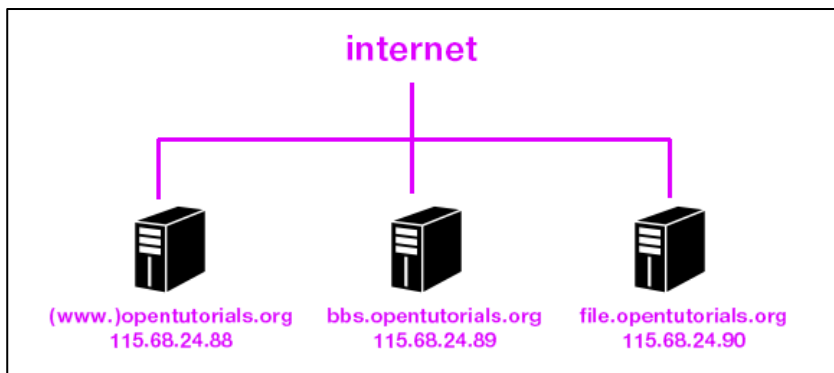
Looks completely different,  
but fundamentally the same water  
- Original data with both transformations

$$NTT(A \cdot R^{-1} \bmod N)$$



# 수학적 도메인 변경에 대한 컴퓨터 공학적 해석

- 인터넷 접속 시 **실제 IP** 입력 대신 **IP에 매핑되는 도메인**을 입력하여 웹사이트 접속
- 즉 **도메인이 달라진다고 해서 IP (실제값)**이 변하는 것은 아님



# HAETAE 기반 문제 및 설계 특징

- **Module-LWE**
  - 공개키 생성 및 난수성 보장 근거
- **Module-SIS**
  - 서명 위조 불가능성 보장
- **Ring-LWE 대비 구조 단순화 및 모듈 차원의 유연성 확보**
- **Fiat-Shamir with Abort 구조**
  - 서명 분포의 안전성 확보
- **Sparse / Small-norm 비밀값**
  - 메모리 사용량 절감
- **NTT 기반 고속 구현 가능**

# 본 강연에서 다루는 코드

- 강연 준비를 1월 초부터 한 관계로 **KpqClean ver2**로 작업을 수행
- [https://github.com/kpqc-cryptocraft/KpqClean\\_ver2](https://github.com/kpqc-cryptocraft/KpqClean_ver2)

The screenshot shows the GitHub repository page for `kpqc-cryptocraft/KpqClean_ver2`. The repository is public and has 1 branch and 2 tags. The main branch is selected. The repository contains several files and folders, including `common`, `crypto_kem`, `crypto_sign`, `.gitignore`, and `README.md`. The `README.md` file is selected, showing the title `KpqClean_ver2` and the description `Benchmark on Korean Post Quantum Cryptography version 2!`.

Repository: `kpqc-cryptocraft / KpqClean_ver2` (Public)

Navigation: `<> Code`, `Issues`, `Pull requests`, `Actions`, `Projects`, `Security`, `Insights`

Branches: `main` (1 Branch), `2 Tags`

Search: `Go to file`

Commits: `minjoo97` SMAUG-T5 AVX2: fix SECRETKEYBYTES definition in KEM (0896ebe · 5 months ago) 69 Commits

File/Folder	Description	Time
<code>common</code>	Update source for four standardized KpqC algorithms	8 months ago
<code>crypto_kem</code>	SMAUG-T5 AVX2: fix SECRETKEYBYTES definition in KEM	5 months ago
<code>crypto_sign</code>	Update SMAUG_T parameter & api and NTRU+ latest code	6 months ago
<code>.gitignore</code>	Create .gitignore	2 years ago
<code>README.md</code>	Update NTRU+ submission status and code update date	6 months ago

Selected File: `README`

## KpqClean\_ver2

Benchmark on Korean Post Quantum Cryptography version 2!

# HAETAE 코드

- HAETAE5를 기준으로 코드를 분석

- 핵심 연산자는 NTT, 다항식 연산, 샘플링 알고리즘, XOF256

clean/	
└─ api.h	# API 인터페이스 정의
└─ config.h	# 컴파일 설정
└─ params.h	# 파라미터 정의 (보안 레벨별)
└─ decompose.c/h	# HighBits/LowBits 분해
└─ encoding.c/h	# rANS 인코딩/디코딩
└─ fft.c/h	# FFT 관련 (미사용 가능)
└─ fixpoint.c/h	# 고정소수점 연산
└─ ntt.c/h	# NTT 변환
└─ packing.c/h	# 키/서명 패킹/언패킹
└─ poly.c/h	# 다항식 연산
└─ polyfix.c/h	# 고정소수점 다항식
└─ polymat.c/h	# 다항식 행렬 연산
└─ polyvec.c/h	# 다항식 벡터 연산
└─ reduce.c/h	# 모듈러 환원
└─ sampler.c/h	# 샘플링 알고리즘
└─ sign.c/h	# 서명 생성/검증
└─ symmetric-shake.c	# XOF256 (SHAKE256) 구현
└─ symmetric.h	# 대칭키 암호 헤더
└─ rans_byte.h	# rANS 인코딩 헤더
└─ Makefile	# 빌드 시스템

# HAETAE 코드

## • HAETAE5를 기준으로 코드를 분석

어제 질문에 대한 답변  
해시 연산자는 NTT, 다항식 연산, 샘플링 알고리즘, XOF256  
PQC 상에서  
NTT와 SHA-3의 전체 연산 부하 비중은?

NTT가 대략 20~30%

SHA-3가 대략 70~80%

### NIST PQC: Hashing Dominance

	scheme	impl	enc/sign	dec/open
	kyber512	speed	82%	73%
	dilithium2	speed	66%	81%
	falcon-512-tree	speed	1%	36%
	sphincs-shake256-128f-simple	clean	96%	96%
	sphincs-shake256-128s-simple	clean	96%	96%

- Results on Arm Cortex-M4 from pqm4 (<https://github.com/mupq/pqm4>)
- Kyber, Dilithium, and SPHINCS+ are dominated by hashing
- For 64-bit platforms slightly less for Kyber/Dilithium, but still dominating

⇒ **Faster hashing will be great for PQC!**

12 December 2022

Matthias J. Kannwischer

Institute of Information Science, Academia Sinica



2/20

```
— reduce.c/h      # 모듈러 환원
— sampler.c/h      # 샘플링 알고리즘
— sign.c/h         # 서명 생성/검증
— symmetric-shake.c # XOF256 (SHAKE256) 구현
— symmetric.h      # 대칭키 암호 헤더
— cans_byte.h      # cans 인코딩 헤더
— Makefile         # 빌드 시스템
```

# HAETAE 서명 1 (입력/비밀키 언팩 및 메시지 해시 바인딩)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen, const uint8_t *m,
                          size_t mlen, const uint8_t *sk) {
```

```
// Unpack secret key
```

```
unpack_sk(A: A1, s0: &s1, s1: &s2, key, sk);
```

- A1: 공개키로부터 복원가능한 행렬의 일부
- s1, s2: 비밀 벡터; key: 서명용 시드

```
xof256_absorbe_twice(state: &state, in1: sk, in1len: HAETAE_CRYPTOPUBLICKEYBYTES, in2: m, in2len: mlen);
```

```
xof256_squeeze(mu, CRHBYTES, &state);
```

```
xof256_absorbe_twice(state: &state, in1: key, in1len: SEEDBYTES, in2: mu, in2len: CRHBYTES);
```

```
xof256_squeeze(seedbuf, CRHBYTES, &state);
```

```
polyvecm_ntt(x: &s1);
```

```
polyveck_ntt(x: &s2);
```

- (Line 1~2)  $\mu = H(pk \parallel m)$  형태로 메시지에 대한 digest 생성
- (Line 3~4)  $seedbuf = H(key \parallel \mu)$ 로 서명 샘플링 시드 생성
- (Line 5~6) 곱셈 효율을 위해 s1, s2를 NTT 도메인에 올려둠

# HAETAE 서명 2 (Hyperball에서 샘플링)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen, const uint8_t *m,  
                           size_t mlen, const uint8_t *sk) {
```

```
/*----- 1. Sample y1 and y2 from hyperball -----*/  
counter = polyfixvec1k_sample_hyperball(y1: &y1, y2: &y2, b: &b, seed: seedbuf, nonce: counter);
```

- y1, y2: 임시 벡터 (서명 nonce 역할)이며 분포는 **hyperball 샘플링**

# HAETAЕ 서명 3 (챌린지 계산)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen, const uint8_t *m,
                          size_t mlen, const uint8_t *sk) {
```

```
/*----- 2. Compute a challenge c -----*/
// Round y1 and y2
polyfixvec1_round(a: &z1rnd, b: &y1);
polyfixveck_round(a: &z2rnd, b: &y2);

// A * round(y) mod q = A1 * round(y1) + 2 * round(y2) mod q
z1rnd0 = z1rnd.vec[0];
polyvec1_ntt(x: &z1rnd);
polymatkl_pointwise_montgomery(t: &Ay, mat: A1, v: &z1rnd);
polyveck_invntt_tomont(x: &Ay);
polyveck_double(b: &z2rnd);
polyveck_add(w: &Ay, u: &Ay, v: &z2rnd);

// recover A * round(y) mod 2q
polyveck_poly_fromcrt(w: &Ay, u: &Ay, v: &z1rnd0);
polyveck_freeze2q(v: &Ay);

// HighBits of (A * round(y) mod 2q)
polyveck_highbits_hint(w: &highbits, v: &Ay);

// LSB(round(y_0) * j)
poly_lsb(a0: &lsb, a: &z1rnd0);

// Pack HighBits of A * round(y) mod 2q and LSB of round(y0)
polyveck_pack_highbits(buf, v: &highbits);
poly_pack_lsb(buf: buf + POLYVECK_HIGHBITS_PACKEDBYTES, a: &lsb);

// c = challenge(highbits, lsb, mu)
poly_challenge(c: &c, highbits_lsb: buf, mu);
```

- 메시지에 바인딩된  $\mu$ 와  
임시값  $y$ 에서 나온 요약정보 (highbits, lsb)로  
챌린지 다항식  $c$ 를 결정



# HAETAЕ 서명 4 (응답 계산)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen, const uint8_t *m,
                          size_t mlen, const uint8_t *sk) {
```

```
/*----- 3. Compute  $z = y + (-1)^b c * s$  -----*/
//  $cs = c * s = c * (s_1 || s_2)$ 
cs1.vec[0] = c;
chat = c;
poly_ntt(a: &chat);

for (i = 1; i < L; ++i) {
    poly_pointwise_montgomery(c: &cs1.vec[i], a: &chat, b: &s1.vec[i - 1]);
    poly_invntt_tomont(a: &cs1.vec[i]);
}

polyveck_poly_pointwise_montgomery(w: &cs2, u: &s2, v: &chat);
polyveck_invntt_tomont(x: &cs2);

//  $z = y + (-1)^b cs = z_1 + z_2$ 
polyveck_cneg(v: &cs1, b: b & 1);
polyveck_cneg(v: &cs2, b: b & 1);
polyfixveck_add(w: &z1, u: &y1, v: &cs1);
polyfixveck_add(w: &z2, u: &y2, v: &cs2);

// reject if  $\text{norm}(z) \geq B$ 
reject1 = ((uint64_t)B1SQ * LN * LN - polyfixveck_sqnorm2(a: &z1, b: &z2)) >> 63;
reject1 &= 1;

polyfixveck_double(b: &z1tmp, a: &z1);
polyfixveck_double(b: &z2tmp, a: &z2);

polyfixfixveck_sub(w: &z1tmp, u: &z1tmp, v: &y1);
polyfixfixveck_sub(w: &z2tmp, u: &z2tmp, v: &y2);

// reject if  $\text{norm}(2z - y) < B$  and  $b' = 0$ 
reject2 =
    (polyfixveck_sqnorm2(a: &z1tmp, b: &z2tmp) - (uint64_t)B0SQ * LN * LN) >> 63;
reject2 &= 1;
reject2 &= (b & 0x2) >> 1;

if (reject1 | reject2) {
    goto reject;
}
```

- NTT 상에서  $c*s1, c*s2$  수행
- 이후 부호 선택 후 더하기

- 두 개의 거절 판정을 수행
  - $\|z\|$ 가 너무 크면 reject
  - $\|2z - y\| < B$  이면서 특정 플래그 (b)가 0이면 reject

# HAETAЕ 서명 5 (힌트 생성 및 서명 패킹)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen, const uint8_t *m,
                          size_t mlen, const uint8_t *sk) {
```

```
/*----- 4. Make a hint -----
// Round z1 and z2
polyfixvec1_round(a: &z1rnd, b: &z1);
polyfixveck_round(a: &z2rnd, b: &z2);

// recover A1 * round(z1) - qcj mod 2q
polyveck_double(b: &z2rnd);
polyveck_sub(w: &htmp, u: &Ay, v: &z2rnd);
polyveck_freeze2q(v: &htmp);

// HighBits of (A * round(z) - qcj mod 2q) and (A1 * round(z1) -
polyveck_highbits_hint(w: &htmp, v: &htmp);
polyveck_sub(w: &h, u: &highbits, v: &htmp);
polyveck_caddDQ2ALPHA(h: &h);

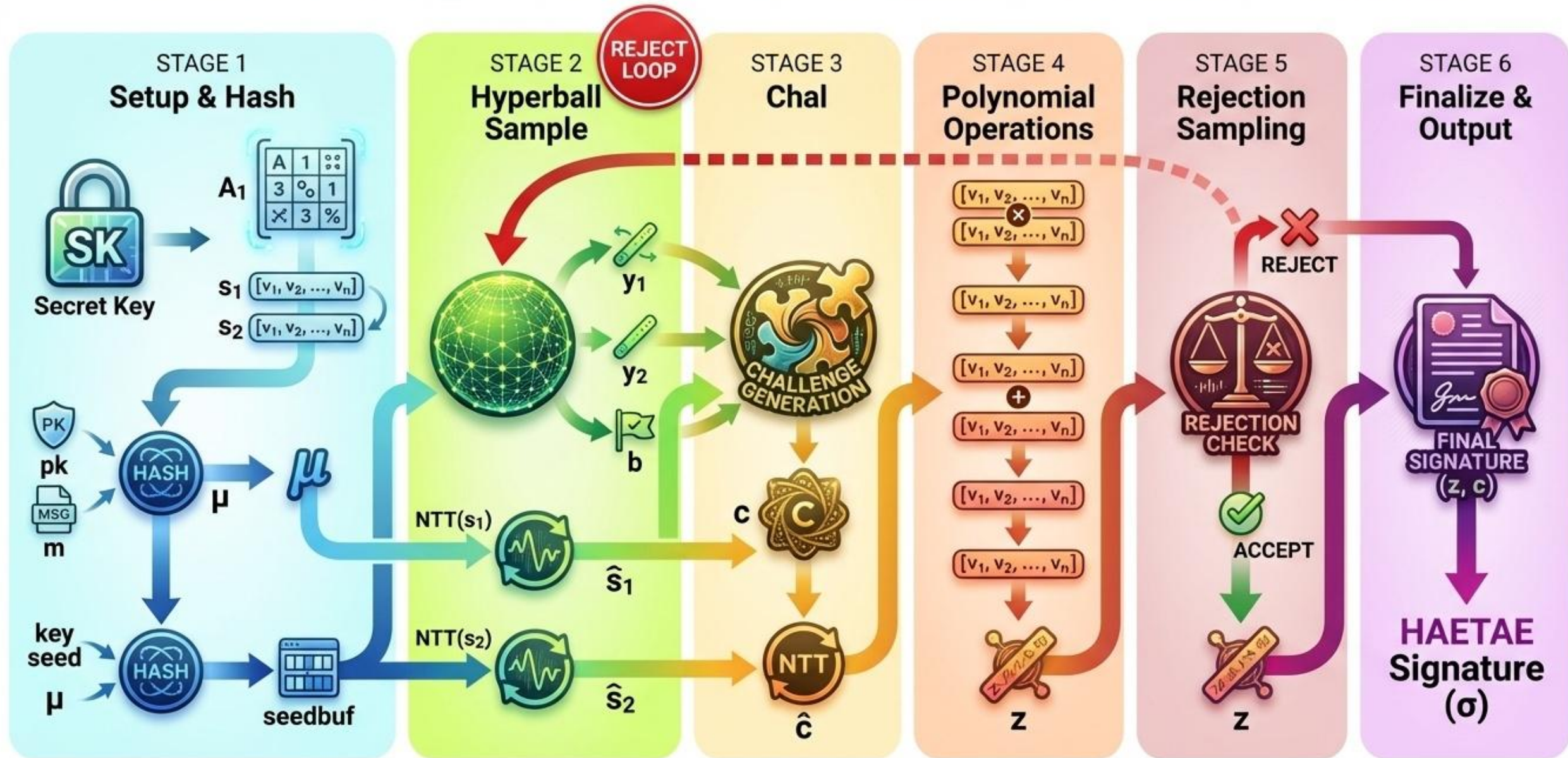
/*----- Decompose(z1) and Pack signature -----
polyvec1_lowbits(v2: &lb_z1, v: &z1rnd); // TODO do this in one f
polyvec1_highbits(v2: &hb_z1, v: &z1rnd);

if (pack_sig(sig, c: &c, lowbits_z1: &lb_z1, highbits_z1: &hb_z1,
             h: &h)) { // reject if signature is too big
    goto reject;
}
*siglen = HAETAЕ_CRYPTO_BYTES;

return 0;
```

- (Line 6~8)  $(A * \text{round}(z) - qcj)$  쪽 highbits를 계산해 앞에서 얻었던  $\text{highbits}(A * \text{round}(y))$ 와의 차이를 힌트로 만듦
- (Line 9~10)  $z1$ 를 high/ low로 분해하여 시그니처 크기를 줄이고 표현을 제한
- (Line 11~13) 최종 서명 패킹

# HAETAE Signature Generation - Detailed Flow



# HAETAE NTT 구현

- **Cyclic convolution (일반적인 구조)**

- 환이  $Z_q[x]/(x^{256} - 1)$  인 경우 256차 원시근 ( $\omega$ )만 있으면 됨
- $\omega^{256} = 1$

- **Negacyclic convolution (e.g., HAETAE, ML-DSA)**

- 환이  $Z_q[x]/(x^{256} + 1)$  인 경우
  - $x^{256} + 1 = 0$
  - 즉  $x^{256} = -1$
  - 해당 조건을 만족하는 평가점이 필요

# HAETAE NTT 구현

- **Negacyclic convolution (HAETAE, ML-DSA)**

- 환이  $Z_q[x]/(x^{256} + 1)$  인 경우

- $x^{256} + 1 = 0$
- 즉  $x^{256} = -1$
- 해당 조건을 만족하는 평가점이 필요

- 양변을 제공하면

- $x^{256} = -1$
- $\psi^{512} = 1$  는 512차 원시근
- $\omega = \psi^2$  라고 두면  $\omega^{256} = 1$  이 되어 256차 원시근이 됨

# HAETAE NTT 구현

- **Negacyclic convolution (HAETAE, ML-DSA)**
  - Negacyclic NTT는 다음을 사용
    - $a(x) \rightarrow a(\psi^{2i+1})$
    - 즉 평가점을  $\psi, \psi^3, \psi^5, \dots, \psi^{511}$  홀수 지수만 사용
    - 해당 점들은  $x^{256} = -1$ 을 만족



## 구조

## 필요한 원시근

$$(x^n - 1)$$

$N$ 차 원시근

$$(x^n + 1)$$

$2N$ 차 원시근

$$\bullet x^{256} = -1$$

짝수 지수

$$(\psi^{2k})^{256} = \psi^{512k} = 1$$

이것은  $x^{256} = 1$  cyclic convolution용 평가점

홀수 지수

$$(\psi^{2k+1})^{256} = \psi^{512k+256} = \psi^{256} = -1$$

이것은  $x^{256} = -1$  을 만족

```

1 q = 64513
2 psi = 426
3 R = pow(2, 32, q) # 14321
4
5 def bitrev8(x):
6     b = format(x, "08b")[::-1]
7     return int(b, 2)
8
9 zetas = [0]*256
10 for i in range(1, 256):
11     e = bitrev8(i)
12     z = (pow(psi, e, q) * R) % q
13     if z > q//2:
14         z -= q
15     zetas[i] = z
16
17 print(zetas[:20])
18 print(zetas[-10:])
19

```

$$\psi^{256} = -1$$

$$\psi^{512} = 1$$



```

[0, 26964, -16505, 22229, 30746, 20243, 19064, -31218, 9395, -30985, 22859,
-8851, 32144, 13744, 21408, 17599, -16039, -22946, 6241, -19553]
[25492, -12831, 7947, 17463, -12979, 29003, 31612, 26554, 8241, -20175]

```

```

static const int32_t zetas[N] = {
[0]=0, [1]=26964, [2]=-16505, [3]=22229, [4]=30746, [5]=20243, [6]=19064, [7]=-31218, [8]=9395,
[9]=-30985, [10]=22859, [11]=-8851, [12]=32144, [13]=13744, [14]=21408, [15]=17599, [16]=-16039, [17]=-22946,
[18]=6241, [19]=-19553, [20]=10681, [21]=22935, [22]=22431, [23]=-29104, [24]=28147, [25]=-27527, [26]=-29133,
[27]=-20035, [28]=20143, [29]=-11361, [30]=30820, [31]=25252, [32]=-22562, [33]=-6789, [34]=-10049, [35]=9383,
[36]=16304, [37]=-12296, [38]=16446, [39]=18239, [40]=-1296, [41]=-19725, [42]=-32076, [43]=11782, [44]=-17941,
[45]=29643, [46]=-8577, [47]=7893, [48]=-21464, [49]=-19646, [50]=-15130, [51]=-2391, [52]=30608, [53]=-23970,
[54]=-16608, [55]=19616, [56]=-7941, [57]=26533, [58]=-19129, [59]=27690, [60]=7597, [61]=-11459, [62]=10615,
[63]=-9430, [64]=11591, [65]=7814, [66]=12697, [67]=32114, [68]=-3761, [69]=-9604, [70]=19813, [71]=20353,
[72]=17456, [73]=-16267, [74]=-19555, [75]=598, [76]=-29942, [77]=4538, [78]=835, [79]=15546, [80]=3970,
[81]=-27685, [82]=1488, [83]=8311, [84]=-12442, [85]=31352, [86]=-17631, [87]=1806, [88]=-5342, [89]=9790,
[90]=29068, [91]=16507, [92]=-29051, [93]=22131, [94]=6759, [95]=15510, [96]=-14941, [97]=28710, [98]=1160,
[99]=-31327, [100]=24985, [101]=11261, [102]=-10623, [103]=-27727, [104]=21502, [105]=18731, [106]=-16186, [107]=-4127,
[108]=-18832, [109]=12050, [110]=-14501, [111]=7929, [112]=29563, [113]=-31064, [114]=5913, [115]=5322, [116]=-16405,
[117]=2844, [118]=29439, [119]=5876, [120]=-9522, [121]=-18586, [122]=-9874, [123]=23844, [124]=30362, [125]=-21442,
[126]=9560, [127]=17671, [128]=-27989, [129]=3350, [130]=787, [131]=-13857, [132]=1657, [133]=-21224, [134]=-7374,
[135]=-9190, [136]=2464, [137]=25555, [138]=-3529, [139]=-28772, [140]=16588, [141]=-15739, [142]=23475, [143]=13666,
[144]=5764, [145]=30980, [146]=13633, [147]=-7401, [148]=-30317, [149]=28847, [150]=7682, [151]=-11808, [152]=-8796,
[153]=14864, [154]=-24162, [155]=-19194, [156]=689, [157]=-1311, [158]=-31332, [159]=-16319, [160]=1025, [161]=10971,
[162]=-23016, [163]=-2648, [164]=-21900, [165]=-12543, [166]=-25921, [167]=28254, [168]=28521, [169]=-16160, [170]=12380,
[171]=-12882, [172]=-30332, [173]=-16630, [174]=23439, [175]=7742, [176]=17182, [177]=17494, [178]=5920, [179]=13642,
[180]=7382, [181]=-18166, [182]=21422, [183]=-30274, [184]=-28190, [185]=13283, [186]=-20316, [187]=-9939, [188]=10672,
[189]=21454, [190]=6080, [191]=-17374, [192]=-29735, [193]=-25912, [194]=-10170, [195]=3808, [196]=10639, [197]=-26985,
[198]=-10865, [199]=25636, [200]=17261, [201]=-26851, [202]=-8253, [203]=-3304, [204]=18282, [205]=-2202, [206]=-31368,
[207]=-22243, [208]=13882, [209]=12069, [210]=-11242, [211]=-7729, [212]=-10226, [213]=1761, [214]=-27298, [215]=-4800,
[216]=-17737, [217]=-22805, [218]=-3528, [219]=65, [220]=10770, [221]=8908, [222]=-23751, [223]=26934, [224]=21921,
[225]=-27010, [226]=-21944, [227]=8889, [228]=-1035, [229]=23224, [230]=-9488, [231]=-5823, [232]=-994, [233]=-20206,
[234]=7655, [235]=-16251, [236]=-22820, [237]=-27740, [238]=15822, [239]=23078, [240]=13803, [241]=-8099, [242]=2931,
[243]=9217, [244]=-21126, [245]=-14203, [246]=25492, [247]=-12831, [248]=7947, [249]=17463, [250]=-12979, [251]=29003,
[252]=31612, [253]=26554, [254]=8241, [255]=-20175}; // q = 64513

```



1 128	1	36 36	2	71 226	3	106 86	4	141 177	5	176 13	6	211 203
2 64		37 164		72 18		107 214		142 113		177 141		212 43
3 192		38 100		73 146		108 54		143 241		178 77		213 171
4 32	3	39 228	4	74 82	5	109 182	6	144 9	7	179 205	8	214 107
5 160		40 20		75 210		110 118		145 137		180 45		215 235
6 96		41 148		76 50		111 246		146 73		181 173		216 27
7 224	4	42 84	5	77 178	6	112 14	7	147 201	8	182 109	9	217 155
8 16		43 212		78 114		113 142		148 41		183 237		218 91
9 144		44 52		79 242		114 78		149 169		184 29		219 219
10 80	5	45 180	6	80 10	7	115 206	8	150 105	9	185 157	10	220 59
11 208		46 116		81 138		116 46		151 233		186 93		221 187
12 48		47 244		82 74		117 174		152 25		187 221		222 123
13 176	6	48 12	7	83 202	8	118 110	9	153 153	10	188 61	11	223 251
14 112		49 140		84 42		119 238		154 89		189 189		224 7
15 240		50 76		85 170		120 30		155 217		190 125		225 135
16 8	7	51 204	8	86 106	9	121 158	10	156 57	11	191 253	12	226 71
17 136		52 44		87 234		122 94		157 185		192 3		227 199
18 72		53 172		88 26		123 222		158 121		193 131		228 39
19 200	8	54 108	9	89 154	10	124 62	11	159 249	12	194 67	13	229 167
20 40		55 236		90 90		125 190		160 5		195 195		230 103
21 168		56 28		91 218		126 126		161 133		196 35		231 231
22 104	9	57 156	10	92 58	11	127 254	12	162 69	13	197 163	14	232 23
23 232		58 92		93 186		128 1		163 197		198 99		233 151
24 24		59 220		94 122		129 129		164 37		199 227		234 87
25 152	10	60 60	11	95 250	12	130 65	13	165 165	14	200 19	15	235 215
26 88		61 188		96 6		131 193		166 101		201 147		236 55
27 216		62 124		97 134		132 33		167 229		202 83		237 183
28 56	11	63 252	12	98 70	13	133 161	14	168 21	15	203 211	16	238 119
29 184		64 2		99 198		134 97		169 149		204 51		239 247
30 120		65 130		100 38		135 225		170 85		205 179		240 15
31 248	12	66 66	13	101 166	14	136 17	15	171 213	16	206 115	17	241 143
32 4		67 194		102 102		137 145		172 53		207 243		242 79
33 132		68 34		103 230		138 81		173 181		208 11		243 207
34 68	13	69 162	14	104 22	15	139 209	16	174 117	17	209 139	18	244 47
35 196		70 98		105 150		140 49		175 245		210 75		245 175

$i$ 에 매핑되는 bitrev된  $e$  값

bitrev는 bit의 위치를 역순서로 만들어줌

main.py

+

```

1 q = 64513
2 psi = 426
3 R = pow(2, 32, q) # 14321
4
5 def bitrev8(x):
6     b = format(x, "08b")[::-1]
7     return int(b, 2)
8
9 zetas = [0]*256
10 for i in range(1, 256):
11     e = bitrev8(i)
12     z = (pow(psi, e, q) * R) % q # Montgomery form twiddle
13     if z > q//2:
14         z -= q # signed rep
15     zetas[i] = z
16
17 print(zetas[:20])
18 print(zetas[-10:])
19

```

bitrev8(5) 실행 과정:  
 5 (십진수)  
 ↓  
 00000101 (8비트 이진수)  
 ↓ (비트 순서 반전)  
 10100000 (역순 이진수)  
 ↓  
 160 (십진수 결과)

$$\alpha = \psi^m$$

$$m = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

$$m = b_7 \cdot 128 + b_6 \cdot 64 + \dots + b_1 \cdot 2 + b_0 \cdot 1$$

Stage	len	회전	결정하는 비트	버터플라이 개수
1	128	$\psi^{128}$	b7	1
2	64	$\psi^{64}$	b6	2
3	32	$\psi^{32}$	b5	4
4	16	$\psi^{16}$	b4	8
5	8	$\psi^8$	b3	16
6	4	$\psi^4$	b2	32
7	2	$\psi^2$	b1	64
8	1	$\psi^1$	b0	128

들어줌

m twiddle

$$\alpha = \psi^m$$

$$m = b_7b_6b_5b_4b_3b_2b_1b_0$$

$$m = b_7 \cdot 128 + b_6 \cdot 64 + \cdots + b_1 \cdot 2 + b_0 \cdot 1$$

$$m = 2t + b_0$$

$$(\psi^m)^{256} = \psi^{256m} = \psi^{256(2t+b_0)} = \psi^{512t+256b_0}$$

$$\psi^{512t} = (\psi^{512})^t = 1^t = 1$$

따라서  $(\psi^m)^{256} = \psi^{512t+256b_0} = 1 \cdot \psi^{256b_0}$

만약 $b_0 = 0$ 인 경우 $\psi^{256 \cdot 0} = \psi^0 = 1$	$(\psi^m)^{256} = 1$
만약 $b_0 = 1$ 인 경우 $\psi^{256 \cdot 1} = \psi^{256} = -1$	$(\psi^m)^{256} = -1$

1 128	1	36 36		71 226		106 86		141 177		176 13		211 203		246 111
2 64		37 164		72 18		107 214		142 113		177 141		212 43		247 239
3 192		38 100		73 146		108 54		143 241		178 77		213 171		248 31
4 32	2	39 228		74 82		109 182		144 9		179 205		214 107		249 159
5 160		40 20		75 210		110 118		145 137		180 45		215 235		250 95
6 96		41 148		76 50		111 246		146 73		181 173		216 27		251 223
7 224	3	42 84		77 178		112 14		147 201		182 109		217 155		252 63
8 16		43 212		78 114		113 142		148 41		183 237		218 91		253 191
9 144		44 52		79 242		114 78		149 169		184 29		219 219		254 127
10 80	4	45 180		80 10		115 206		150 105		185 157		220 59		255 255
11 208		46 116		81 138		116 46		151 233		186 93		221 187		
12 48		47 244		82 74		117 174		152 25		187 221		222 123		
13 176	5	48 12		83 202		118 110		153 153		188 61		223 251		
14 112		49 140		84 42		119 238		154 89		189 189		224 7		
15 240		50 76		85 170		120 30		155 217		190 125		225 135		
16 8	6	51 204		86 106		121 158		156 57		191 253		226 71		
17 136		52 44		87 234		122 94		157 185		192 3		227 199		
18 72		53 172		88 26		123 222		158 121		193 131		228 39		
19 200	7	54 108		89 154		124 62		159 249		194 67		229 167		
20 40		55 236		90 90		125 190		160 5		195 195		230 103		
21 168		56 28		91 218		126 126		161 133		196 35		231 231		
22 104	8	57 156		92 58		127 254		162 69		197 163		232 23		
23 232		58 92		93 186		128 1		163 197		198 99		233 151		
24 24		59 220		94 122		129 129		164 37		199 227		234 87		
25 152	9	60 60		95 250		130 65		165 165		200 19		235 215		
26 88		61 188		96 6		131 193		166 101		201 147		236 55		
27 216		62 124		97 134		132 33		167 229		202 83		237 183		
28 56	10	63 252		98 70		133 161		168 21		203 211		238 119		
29 184		64 2		99 198		134 97		169 149		204 51		239 247		
30 120		65 130		100 38		135 225		170 85		205 179		240 15		
31 248	11	66 66		101 166		136 17		171 213		206 115		241 143		
32 4		67 194		102 102		137 145		172 53		207 243		242 79		
33 132		68 34		103 230		138 81		173 181		208 11		243 207		
34 68	12	69 162		104 22		139 209		174 117		209 139		244 47		
35 196		70 98		105 150		140 49		175 245		210 75		245 175		

8단계에서는 가장 하위  
비트를 odd로 설정 (회전)하여  
negacyclic을 완성

$$(\psi^1)^{256} = \psi^{256} = -1$$

$$(\psi^3)^{256} = (\psi)^{768} = (\psi^{256})^3 = (-1)^3 = -1$$

$$U + \psi^1 V, U - \psi^1 V$$

$$U + \psi^1 V, U - \psi^1 V$$

$$-\psi \leftrightarrow \psi^{256+1} = \psi^{257}$$

# HAETAETAE NTT

- **버터플라이 연산**

- 64-bit 곱셈 1회
- Montgomery reduction 1회
- 32-bit 덧셈 / 뺄셈
- 분기 없음
- 배열 접근 2회

```
void ntt(int32_t a[N]) {
    unsigned int len, start, j, k;
    int32_t zeta, t;

    k = 0;
    for (len = 128; len > 0; len >>= 1) {
        for (start = 0; start < N; start = j + len) {
            zeta = zetas[++k];
            for (j = start; j < start + len; ++j) {
                t = montgomery_reduce(a: (int64_t)zeta * a[j + len]);
                a[j + len] = a[j] - t;
                a[j] = a[j] + t;
            }
        }
    }
}
```

# HAETAETAE iNTT

- NTT와 정반대로 수행

- $a = \frac{1}{N} \cdot \text{invNTT}(\hat{a})$

$$\text{NTT}^{-1}(\text{NTT}(a)) = N \cdot a$$

- 실제 구현에서는

- $f = \text{mont}^2 / N$
- 이미 Montgomery 도메인에 있음
- 결과도 Montgomery 도메인 유지

```
void invntt_tomont(int32_t a[N]) {
    unsigned int start, len, j, k;
    int32_t t, zeta;
    const int32_t f = -29720; // mont^2/256

    k = 256;
    for (len = 1; len < N; len <= 1) {
        for (start = 0; start < N; start = j + len) {
            zeta = -zetas[--k];
            for (j = start; j < start + len; ++j) {
                t = a[j];
                a[j] = t + a[j + len];
                a[j + len] = t - a[j + len];
                a[j + len] = montgomery_reduce(a: (int64_t)zeta * a[j + len]);
            }
        }
    }

    for (j = 0; j < N; ++j) {
        a[j] = montgomery_reduce(a: (int64_t)f * a[j]);
    }
}
```

$\text{mont reduce}(x \cdot R) = x \bmod Q$

```
poly_ntt(a: &chat);

for (i = 1; i < L; ++i) {
    poly_pointwise_montgomery(c: &cs1.vec[i], a: &chat, b: &s1.vec[i - 1]);
    poly_invntt_tomont(a: &cs1.vec[i]);
}
```

# HAETAE iNTT

- **NTT 행렬**  $F_{j,i} = \omega^{ij}$
- **inverse NTT 행렬**  $F_{i,j}^{-1} = \omega^{-ij}$

- $F^{-1} \cdot F = N \cdot I$

- $\sum_{j=0}^{N-1} \omega^{j(\ell-i)} = \begin{cases} N & \ell = i \\ 0 & \ell \neq i \end{cases}$

$\ell - i = 0$  인 경우

$$\omega^{j(\ell-i)} = \omega^{j(0)} = \omega^0 = 1$$

$$\sum_{j=0}^{N-1} \omega^{j(\ell-i)} = \sum_{j=0}^{N-1} 1 = N$$

$$F^{-1} \cdot F = N$$

# HAETAE iNTT

$$\bullet \sum_{j=0}^{N-1} \omega^{j(\ell-i)} = \begin{cases} N & \ell = i \\ 0 & \ell \neq i \end{cases}$$

$\ell - i \neq 0$  인 경우  $\omega^{\ell-i} \neq 1$

$$\sum_{j=0}^{N-1} \omega^{j(\ell-i)} = 0$$

$$m = \ell - i$$

$$\sum_{j=0}^{N-1} \omega^{jm} = 1 + \omega^m + \omega^{2m} + \dots + \omega^{(N-1)m}$$

공비가  $r = \omega^m$  인 유한 기하급수

$$\sum_{j=0}^{N-1} r^j = \frac{r^N - 1}{r - 1}$$

$$\sum_{j=0}^{N-1} \omega^{jm} = \frac{(\omega^m)^N - 1}{\omega^m - 1}$$



# HAETAETAE iNTT

$\ell - i \neq 0$  인 경우  $\omega^{\ell-i} \neq 1$

•  $\sum_{j=0}^{N-1} \omega^{j(\ell-i)} =$

$$F^{-1} \cdot F = \begin{pmatrix} N & 0 & \cdots & 0 \\ 0 & N & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & N \end{pmatrix} = NI$$

## 분자계산

$$(\omega^m)^N = \omega^{mN} = (\omega^N)^m = 1^m = 1$$
$$(\omega^m)^N - 1 = 0$$

## 분모가 0이 아님을 확인

$$\omega^m - 1 \neq 0$$

$$\omega^m = 1 \leftrightarrow m \equiv 0 \pmod{N}$$

지금은  $m \neq 0 \pmod{N}$ 이므로 분모는 0이 아님

$$m = \ell - i$$

결론  $\sum_{j=0}^{N-1} \omega^{j(\ell-i)} = \frac{0}{\omega^m - 1} = 0$

$$+ \omega^m + \omega^{2m} + \cdots + \omega^{(N-1)m}$$

인 유한 기하급수

$$\sum_{j=0}^{N-1} r^j = \frac{r^N - 1}{r - 1}$$

$$\omega^{jm} = \frac{(\omega^m)^N - 1}{\omega^m - 1}$$

# HAETAE Montgomery Reduction

$$\bullet \frac{c + (c \cdot \mu \bmod r^n) \cdot m}{r^n}$$

```
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;  
}
```

$$Q' = Q^{-1} \pmod{R}$$

사용한 역원의 부호가 상이하여  
발생한 차이점

$$\bullet \frac{c + (c \cdot \mu \bmod r^n) \cdot m}{r^n}$$

$$Q' = -Q^{-1} \pmod{R}$$

# AIMer 기반 문제

- **MPC-in-the-Head + BN++ 프로토콜**

- 다자 계산 시뮬레이션 기반 서명 생성
- $\tau$  반복을 통한 soundness error 지수적 감소
- 서명 검증 시 일부 party view만 공개

- **AIM2 대칭 프리미티브**

- Mersenne S-box + Affine Layer + Feed-forward 구조
- XOF (SHAKE128/256) 기반 파라미터, 행렬 생성
- 구조적 단순성으로 구현 안전성 확보

- **유연한 파라미터 설계**

- $(N, \tau)$  조절을 통한 서명 크기-성능 트레이드오프
- Fast ( $f$ ) / Small ( $s$ ) 파라미터 세트 제공
- IoT 환경 요구사항에 따른 선택 가능

# AIMer 특징

- **SPHINCS+ 대비**
  - 더 작은 서명 크기
  - 빠른 키 생성 및 서명
- **장기 IoT 배포용 대칭기반 양자내성 서명으로 유망**
  - 업데이트가 어려운 환경에서 보수적 보안 모델 제공

```
AIMer/
└─ clean/                                # 참조 구현 디렉토리
    └─ field.c                            # 유한체 연산 구현
    └─ field.h                            # 유한체 연산 헤더
    └─ aim2.c                             # AIM2 암호화 함수 구현
    └─ aim2.h                             # AIM2 암호화 함수 헤더
    └─ hash.c                             # 해시 함수 래퍼 구현
    └─ hash.h                             # 해시 함수 래퍼 헤더
    └─ tree.c                             # 시드 트리 구현
    └─ tree.h                             # 시드 트리 헤더
    └─ sign.c                             # 서명 생성/검증 구현
    └─ sign.h                             # 서명 생성/검증 헤더
    └─ params.h                           # 파라미터 정의
    └─ api.h                             # API 인터페이스
    └─ crypto_declassify.h                # 보안 관련 매크로
    └─ Makefile                           # 빌드 시스템
```

# AlMer Sign MPC-in-the-Head

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen,  
    const uint8_t *m, size_t mlen,  
    const uint8_t *sk)
```

- Phase 1 데이터를 만들기  
 위해 **대형 로컬 배열들을 잡음**

- run\_phase\_1을 시행

```
////////////////////////////////////  
// Phase 1: Committing to the seeds and the execution views of parties. //  
////////////////////////////////////
```

```
// nodes for seed trees  
uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE];
```

```
// commitments for seeds  
uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE];
```

```
// multiplication check inputs  
mult_chk_t mult_chk[AIMER_T][AIMER_N];
```

```
// multiplication check outputs  
GF alpha_v_shares[AIMER_T][2][AIMER_N];
```

```
// commitments for phase 1  
run_phase_1(sign, commits, nodes, mult_chk, alpha_v_shares, sk, m, mlen);
```

# AIMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,  
uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE],  
uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE],  
mult_chk_t mult_chk[AIMER_T][AIMER_N],  
GF alpha_v_shares[AIMER_T][2][AIMER_N],  
const uint8_t *sk, const uint8_t *m, size_t mlen)
```

- (Line 1~3) **sk에서 비밀 pt와 공개 파라미터를 꺼냄**
- (Line 4~11)  
 $\mu = H((IV||ct)||m)$  을 계산

```
GF pt_GF = {0,}, ct_GF = {0,};  
GF_from_bytes(out: pt_GF, in: sk);  
GF_from_bytes(out: ct_GF, in: sk + AIM2_NUM_BYTES_FIELD + AIM2_IV_SIZE);
```

```
// message pre-hashing  
hash_instance ctx;  
hash_init_prefix(ctx: &ctx, prefix: HASH_PREFIX_0);  
hash_update(ctx: &ctx, data: sk + AIM2_NUM_BYTES_FIELD,  
            data_len: AIM2_IV_SIZE + AIM2_NUM_BYTES_FIELD);  
hash_update(ctx: &ctx, data: m, data_len: mlen);  
hash_final(ctx: &ctx);  
  
uint8_t mu[AIMER_COMMIT_SIZE];  
hash_squeeze(ctx: &ctx, buffer: mu, buffer_len: AIMER_COMMIT_SIZE);  
hash_ctx_release(ctx: &ctx);
```

# AlMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,  
uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE],  
uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE],  
mult_chk_t mult_chk[AIMER_T][AIMER_N],  
GF alpha_v_shares[AIMER_T][2][AIMER_N],  
const uint8_t *sk, const uint8_t *m, size_t mlen)
```

- (Line 1~2) 비밀 pt에 대해 앞쪽 L개의 S-box 출력값을 계산해 둬
  - 해당 값이 MPC에서 공유되어야 할 값
- (Line 3~5) IV로 부터 matrix\_A와 vector\_b를 재생성
- (Line 6~7) Signature에 사용될 randomness 추출

```
// compute first L sboxes' outputs  
GF sbox_outputs[AIMER_L];  
aim2_sbox_outputs(sbox_outputs, pt: pt_GF);
```

```
// derive the binary matrix and the vector from the initial vector  
GF matrix_A[AIMER_L][AIM2_NUM_BITS_FIELD];  
GF vector_b = {0,};  
generate_matrix_LU(matrix_A, vector_b, iv: sk + AIM2_NUM_BYTES_FIELD)  
  
// generate per-signature randomness  
uint8_t random[SECURITY_BYTES];  
randombytes(random, SECURITY_BYTES);
```



# AlMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,  
                uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE],  
                uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE],  
                mult_chk_t mult_chk[AIMER_T][AIMER_N],  
                GF alpha_v_shares[AIMER_T][2][AIMER_N],  
                const uint8_t *sk, const uint8_t *m, size_t mlen)
```

*// generate execution views and commitments*

```
commit_and_expand_tape(tape: &tape, commit: commits[rep][party], ctx_precom: &ctx_precom,  
                      seed: nodes[rep][party + AIMER_N - 1], rep, party);  
hash_update(ctx: &ctx, data: commits[rep][party], data_len: AIMER_COMMIT_SIZE);
```

- 각 party는 leaf seed로부터 **commitment** 1개 (commits[rep][party])
- **실행 테이프 (tape)** 1개를 얻음

# AlMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,
```

개념	정의	생성 방식	포함 내용	공개 여부
seed	각 가상 party의 난수 출발점	랜덤 생성	난수 출발점	일부 공개
tape	seed로부터 생성된 난수 스트림에서 파생된 내부 랜덤 상태	PRG(seed)	랜덤 share	직접 공개 X
view	한 party가 MPC 실행 중에 가지는 전체 내부 상태	tape + MPC 계산	전체 내부 상태	일부 재구성
commitment	각 party의 seed에 대한 해시값	H(seed)	seed 고정값	항상 포함

```
hash_update(ctx, &ctx_data, commits[rep][party], data, len, ATMER_COMMIT_SIZE);
```

## MPC-in-the-Head (MithH)

비밀 계산을 실제로 여러 참여자가 수행하는 대신,  
서명자가 머릿속에서 (로컬에서) 여러 가상 party로 나누어 MPC를  
시뮬레이션하는 기법

모든 party의 실행 기록 (view)을 먼저 커밋한 뒤  
무작위로 일부만 공개하여 계산이 올바르게 수행되었음을 확률적으로 증명

# AIMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,
                uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE],
                uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE],
                mult_chk_t mult_chk[AIMER_T][AIMER_N],
                GF alpha_v_shares[AIMER_T][2][AIMER_N],
                const uint8_t *sk, const uint8_t *m, size_t mlen)

// compute offsets
GF_add(c: delta.pt_share, a: delta.pt_share, b: tape.pt_share);
GF_add(c: delta.t_shares[0], a: delta.t_shares[0], b: tape.t_shares[0]);
GF_add(c: delta.t_shares[1], a: delta.t_shares[1], b: tape.t_shares[1]);
GF_add(c: delta.t_shares[2], a: delta.t_shares[2], b: tape.t_shares[2]);
GF_add(c: delta.a_share, a: delta.a_share, b: tape.a_share);
GF_add(c: delta.c_share, a: delta.c_share, b: tape.c_share);
GF_set0(a: mult_chk[rep][party].x_shares[AIMER_L]);
```

- delta에 모든 party **tape** share를 누적합으로 모음

# AIMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,  
                uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE],  
                uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE],  
                mult_chk_t mult_chk[AIMER_T][AIMER_N],  
                GF alpha_v_shares[AIMER_T][2][AIMER_N],  
                const uint8_t *sk, const uint8_t *m, size_t mlen)
```

```
// adjust the last share and prepare the proof and h_1
```

```
if (party == AIMER_N - 1)
```

```
{
```

```
    GF_add(c: delta.pt_share, a: delta.pt_share, b: pt_GF);
```

```
    GF_add(c: delta.t_shares[0], a: delta.t_shares[0], b: sbx_outputs[0]);
```

```
    GF_add(c: delta.t_shares[1], a: delta.t_shares[1], b: sbx_outputs[1]);
```

```
    GF_add(c: delta.t_shares[2], a: delta.t_shares[2], b: sbx_outputs[2]);
```

```
    GF_mul_add_s(c: delta.c_share, a: pt_GF, b: delta.a_share);
```

```
    GF_to_bytes(out: sign->proofs[rep].delta_pt_bytes, in: delta.pt_share);
```

```
    GF_to_bytes(out: sign->proofs[rep].delta_ts_bytes[0], in: delta.t_shares[0]);
```

```
    GF_to_bytes(out: sign->proofs[rep].delta_ts_bytes[1], in: delta.t_shares[1]);
```

```
    GF_to_bytes(out: sign->proofs[rep].delta_ts_bytes[2], in: delta.t_shares[2]);
```

```
    GF_to_bytes(out: sign->proofs[rep].delta_c_bytes, in: delta.c_share);
```

```
    GF_add(c: tape.pt_share, a: delta.pt_share, b: tape.pt_share);
```

```
    GF_add(c: tape.t_shares[0], a: delta.t_shares[0], b: tape.t_shares[0]);
```

```
    GF_add(c: tape.t_shares[1], a: delta.t_shares[1], b: tape.t_shares[1]);
```

```
    GF_add(c: tape.t_shares[2], a: delta.t_shares[2], b: tape.t_shares[2]);
```

```
    GF_add(c: tape.c_share, a: delta.c_share, b: tape.c_share);
```

```
    GF_copy(out: mult_chk[rep][party].x_shares[AIMER_L], in: vector_b);
```

```
}
```

- 마지막 party  
(party==AIMER\_N-1)에서만  
합이 실제 값이 되도록 보정

# AIMer Sign (run\_phase\_1)

```
void run_phase_1(signature_t *sign,  
                uint8_t commits[AIMER_T][AIMER_N][AIMER_COMMIT_SIZE],  
                uint8_t nodes[AIMER_T][2 * AIMER_N - 1][AIMER_SEED_SIZE],  
                mult_chk_t mult_chk[AIMER_T][AIMER_N],  
                GF alpha_v_shares[AIMER_T][2][AIMER_N],  
                const uint8_t *sk, const uint8_t *m, size_t mlen)
```

```
    crypto_declassify(ct_GF, sizeof(ct_GF));  
    aim2_mpc(mult_chk: &mult_chk[rep][party],  
            matrix_A: (const GF (*)[AIM2_NUM_BITS_FIELD])matrix_A, ct_GF);  
}  
  
// NOTE: depend on the order of values in proof_t  
hash_update(ctx: &ctx, data: sign->proofs[rep].delta_pt_bytes,  
            data_len: AIM2_NUM_BYTES_FIELD * (AIMER_L + 2));  
}  
hash_ctx_release(ctx: &ctx_precom);  
  
// commit to salt, (all commitments of parties' seeds,  
// delta_pt, delta_t, delta_c) for all repetitions  
hash_final(ctx: &ctx);  
hash_squeeze(ctx: &ctx, buffer: sign->h_1, buffer_len: AIMER_COMMIT_SIZE);  
hash_ctx_release(ctx: &ctx);  
}
```

- (Line 2)  
AIM2의 관계식을 MPC share 형태로 만족하도록 시뮬레이션
- (Line 5~7)  
mu, salt, 모든 commits, 모든 repetition의 delta 값들을 통해 h\_1 생성

# AIMer Sign (run\_phase\_2\_3)

```
void run_phase_2_and_3(signature_t *sign,  
GF alpha_v_shares[AIMER_T][2][AIMER_N],  
const mult_chk_t mult_chk[AIMER_T][AIMER_N])
```

```
GF epsilons[AIMER_L + 1];  
  
hash_instance ctx_e;  
hash_init(ctx: &ctx_e);  
hash_update(ctx: &ctx_e, data: sign->h_1, data_len: AIMER_COMMIT_SIZE);  
hash_final(ctx: &ctx_e);  
  
hash_instance ctx;  
hash_init_prefix(ctx: &ctx, prefix: HASH_PREFIX_2);  
hash_update(ctx: &ctx, data: sign->h_1, data_len: AIMER_COMMIT_SIZE);  
hash_update(ctx: &ctx, data: sign->salt, data_len: AIMER_SALT_SIZE);  
  
GF alpha = {0,};  
for (size_t rep = 0; rep < AIMER_T; rep++)  
{  
    GF_set0(a: alpha);  
    hash_squeeze(ctx: &ctx_e, buffer: (uint8_t *)epsilons, buffer_len: sizeof(epsilons));  
  
    crypto_declassify(epsilons, sizeof(epsilons));  
}
```

- Epsilons 챌린지 생성
  - ctx\_e=H(sign->h\_1)에서 squeeze해서 매 repetition마다 eplisions[AIMER\_L+1]을 뽑음

# AIMer Sign (run\_phase\_2\_3)

```
void run_phase_2_and_3(signature_t *sign,
                      GF alpha_v_shares[AIMER_T][2][AIMER_N],
                      const mult_chk_t mult_chk[AIMER_T][AIMER_N])
```

```
for (size_t party = 0; party < AIMER_N; party++)
{
    // alpha_share = a_share + sum x_share[i] * eps[i]
    // v_share = c_share - pt_share * alpha + sum z_share[i] * eps[i]
    GF_mul_add(c: alpha_v_shares[rep][0][party],
              a: mult_chk[rep][party].x_shares[0], b: epsilons[0]);
    GF_mul_add(c: alpha_v_shares[rep][1][party],
              a: mult_chk[rep][party].z_shares[0], b: epsilons[0]);
    GF_mul_add(c: alpha_v_shares[rep][0][party],
              a: mult_chk[rep][party].x_shares[1], b: epsilons[1]);
    GF_mul_add(c: alpha_v_shares[rep][1][party],
              a: mult_chk[rep][party].z_shares[1], b: epsilons[1]);
    GF_mul_add(c: alpha_v_shares[rep][0][party],
              a: mult_chk[rep][party].x_shares[2], b: epsilons[2]);
    GF_mul_add(c: alpha_v_shares[rep][1][party],
              a: mult_chk[rep][party].z_shares[2], b: epsilons[2]);
    GF_mul_add(c: alpha_v_shares[rep][0][party],
              a: mult_chk[rep][party].x_shares[3], b: epsilons[3]);
    GF_mul_add(c: alpha_v_shares[rep][1][party],
              a: mult_chk[rep][party].z_shares[3], b: epsilons[3]);

    GF_add(c: alpha, a: alpha, b: alpha_v_shares[rep][0][party]);
}
```

```
// alpha is opened, so we can finish calculating v_share
crypto_declassify(alpha, sizeof(alpha));
for (size_t party = 0; party < AIMER_N; party++)
{
    GF_mul_add(c: alpha_v_shares[rep][1][party],
              a: mult_chk[rep][party].pt_share, b: alpha);
}
```

- alpha\_v\_shares 배열을 누적 방식으로 업데이트
- alpha는 모든 party의 alpha\_share를 더해  
오픈되는 값으로 만듦
- alpha가 오픈되면 각 party는 v\_share에  
 $pt\_share * alpha$ 를 더해 최종식을 맞춤



# AImer Sign (run\_phase\_2\_3)

```
void run_phase_2_and_3(signature_t *sign,  
                        GF alpha_v_shares[AIMER_T][2][AIMER_N],  
                        const mult_chk_t mult_chk[AIMER_T][AIMER_N])
```

```
    hash_update(ctx: &ctx, data: (const uint8_t *)alpha_v_shares[rep],  
               data_len: AIM2_NUM_BYTES_FIELD * 2 * AIMER_N);  
}  
hash_final(ctx: &ctx);  
hash_squeeze(ctx: &ctx, buffer: sign->h_2, buffer_len: AIMER_COMMIT_SIZE);  
hash_ctx_release(ctx: &ctx);  
hash_ctx_release(ctx: &ctx_e);
```

- 각 repetition의 (alpha\_share, v\_share) 전부를 해시하여 sign->h\_2를 만듦



# AIMer Sign (phase 4)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen,  
    const uint8_t *m, size_t mlen,  
    const uint8_t *sk) {  
    #define AIMER_L                AIM2_NUM_INPUT_SBOX  
    #define AIMER_T                65 // number of parallel repetitions (Tau)  
    #define AIMER_N                16 // number of MPC parties (N)  
    #define AIMER_LOGN            4 // log_2(N)
```

```
////////////////////////////////////  
// Phase 4: Challenging views of the MPC protocols. //  
////////////////////////////////////
```

```
hash_init(ctx: &ctx);  
hash_update(ctx: &ctx, data: sign->h_2, data_len: AIMER_COMMIT_SIZE);  
hash_final(ctx: &ctx);
```

```
uint8_t indices[AIMER_T]; // AIMER_N <= 256  
hash_squeeze(ctx: &ctx, buffer: indices, buffer_len: AIMER_T);  
hash_ctx_release(ctx: &ctx);  
for (size_t rep = 0; rep < AIMER_T; rep++)  
{  
    indices[rep] &= (1 << AIMER_LOGN) - 1;  
}
```

- repetition마다 party의 view는 숨기고 나머지는 공개

# AIMer Sign (phase 5)

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen,  
    const uint8_t *m, size_t mlen,  
    const uint8_t *sk)
```

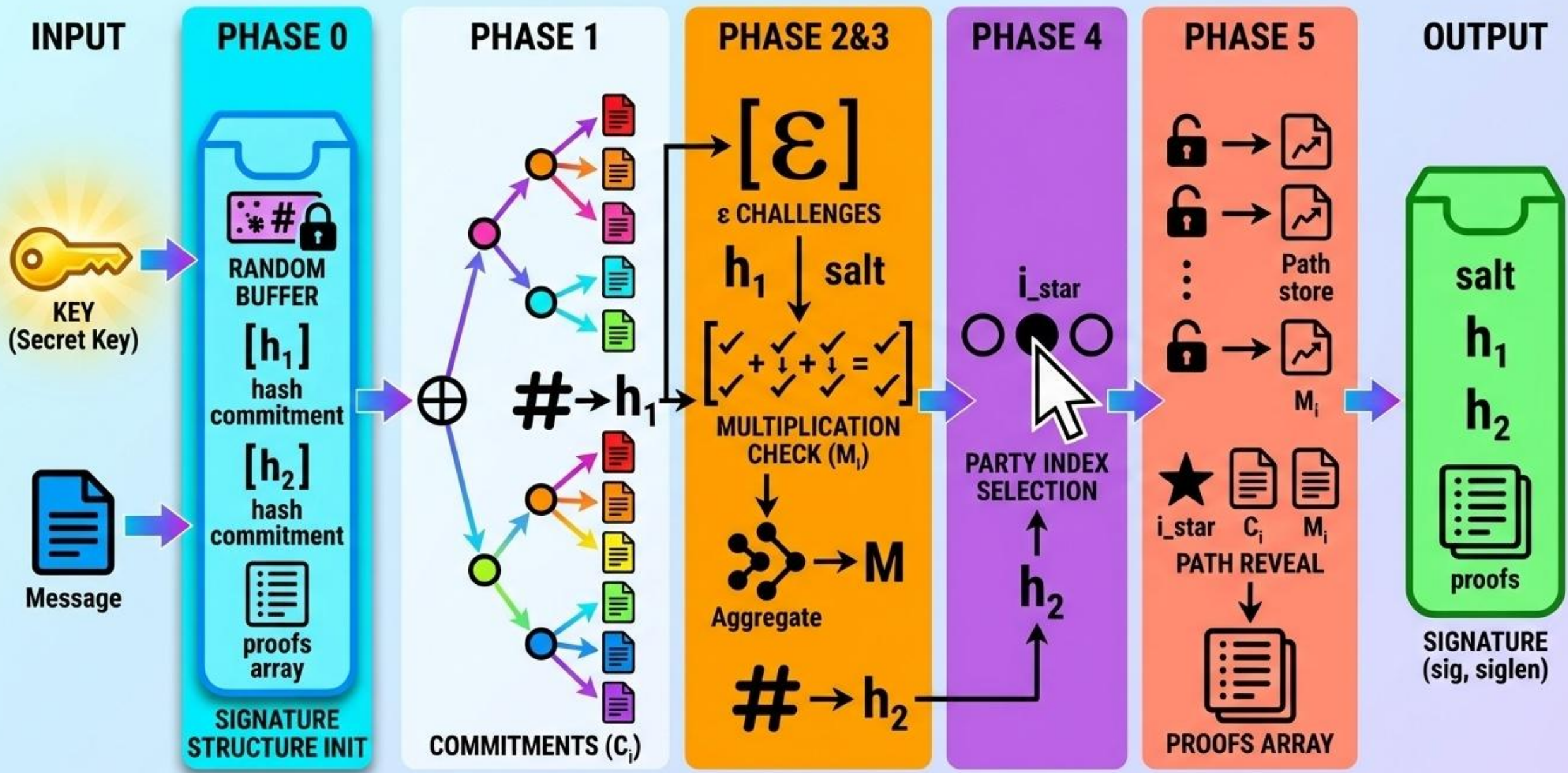
```
/////////////////////////////////////////  
// Phase 5: Opening the views of the MPC protocols. //  
/////////////////////////////////////////
```

```
crypto_declassify(indices, sizeof(indices));  
for (size_t rep = 0; rep < AIMER_T; rep++)  
{  
    size_t i_bar = indices[rep];  
    reveal_all_but(reveal_path: sign->proofs[rep].reveal_path,  
        nodes: (const uint8_t (*)(AIMER_SEED_SIZE))nodes[rep], cover_index: i_bar);  
    memcpy(dest: sign->proofs[rep].missing_commitment, src: commits[rep][i_bar],  
        n: AIMER_COMMIT_SIZE);  
    GF_to_bytes(out: sign->proofs[rep].missing_alpha_share_bytes,  
        in: alpha_v_shares[rep][0][i_bar]);  
}  
*siglen = CRYPTO_BYTES;
```

- Seed tree에서 i\_bar만 제외하고 나머지 leaf seed들을 재현 가능하도록 경로를 넣음

Verifier가 i\_bar party에 대해서 seed가 없으니 대신 다음 2개를 서명에 첨부

- missing\_commitment = commits[rep][i\_bar]
- missing\_alpha\_share\_bytes = alpha\_v\_shares[rep][0][i\_bar] (alpha\_share만 저장)



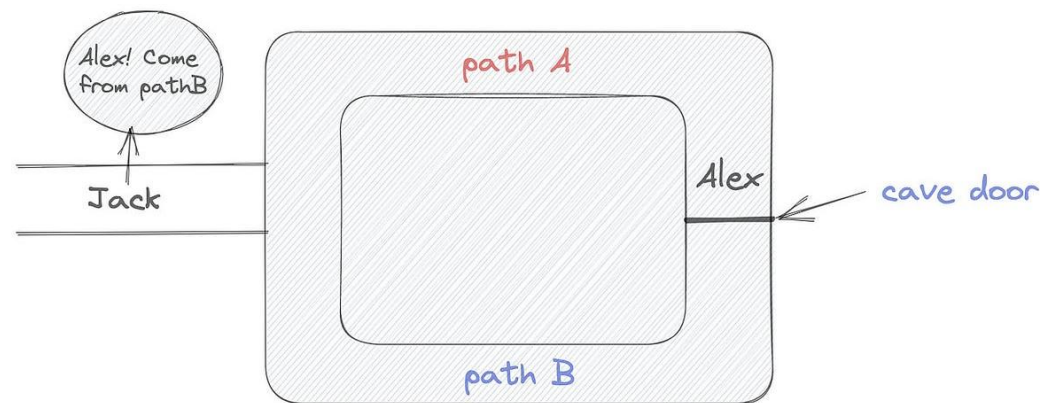
# 영지식 증명

- **AIMer 알고리즘에는 영지식 증명 개념이 포함되어 있음**

- 비밀은 공개하지 않으면서  
비밀을 알고 있다는 사실만 증명하는 방법

- 가장 유명한 예시는 Ali Baba 동굴 비유

- 동굴에는 두 갈래 길 A, B가 있음
- 안쪽에는 비밀번호를 알아야 열 수 있는 문이 있음
- 증명자 (Prover)는 비밀번호를 알고 있음
- 검증자 (Verifier)는 모름
- 검증자가 “A로 나와!” 또는 “B로 나와!”라고 무작위로 요구
  - 증명자는 문을 열고 원하는 방향으로 나올 수 있음
- 이를 여러 번 반복하면
  - 검증자는 증명자가 “비밀번호를 알고 있구나 ” 라는 확신을 얻음
  - 검증자는 비밀번호 자체는 절대 알 수 없음



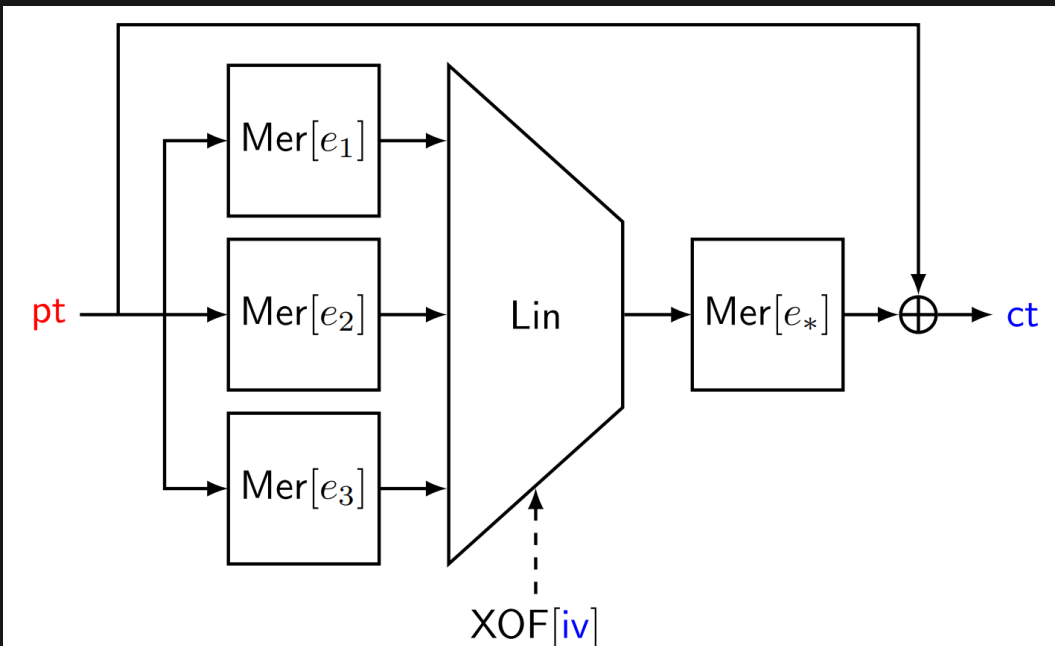
# AIM2 설계 철학 (구현 관점)

- **역 Mersenne 256-bit S-box 구현** ( $2^{256} - 1$ )
  - 해당 필드 상에서의 multiplication과 squaring
- **S-box 기반 비선형성 + Affine mixing**
- **MPC-in-the-head / ZK-friendly 구조**
  - 곱셈 수 최소화
  - 분기 없는 계산
- **비트 연산 중심 설계**



# AIMer의 핵심 연산자 AIM2

- [Stage 0] 입력로드 + 랜덤 선형층 생성
  - iv 를 해시 입력으로 넣고 hash\_squeeze()로 스트림을 뽑아
    - S-box별(3개)로 L, U 삼각행렬 생성
    - 그리고 벡터 b 생성
  - L/U는 삼각구조 + 대각선 1로 항상 가역 구조



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF_matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF_matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF_vector_b = {0,};

    GF_state[AIM2_NUM_INPUT_SBOX];
    GF_pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# AIMer의 핵심 연산자 AIM2

행렬  $A$ 가 가역이라는 것은  $\exists A^{-1}$  s.t.  $AA^{-1} = I$  과 동치이고 선형대수적으로는 아래와 동일함

• [Stage 0] 입력로드 + 랜덤 선형층 생성

$$\det(A) \neq 0$$

•  $iv$  를 해시 입력으로 넣고  $\text{hash\_squeeze}()$ 로 스트림을 뽑아

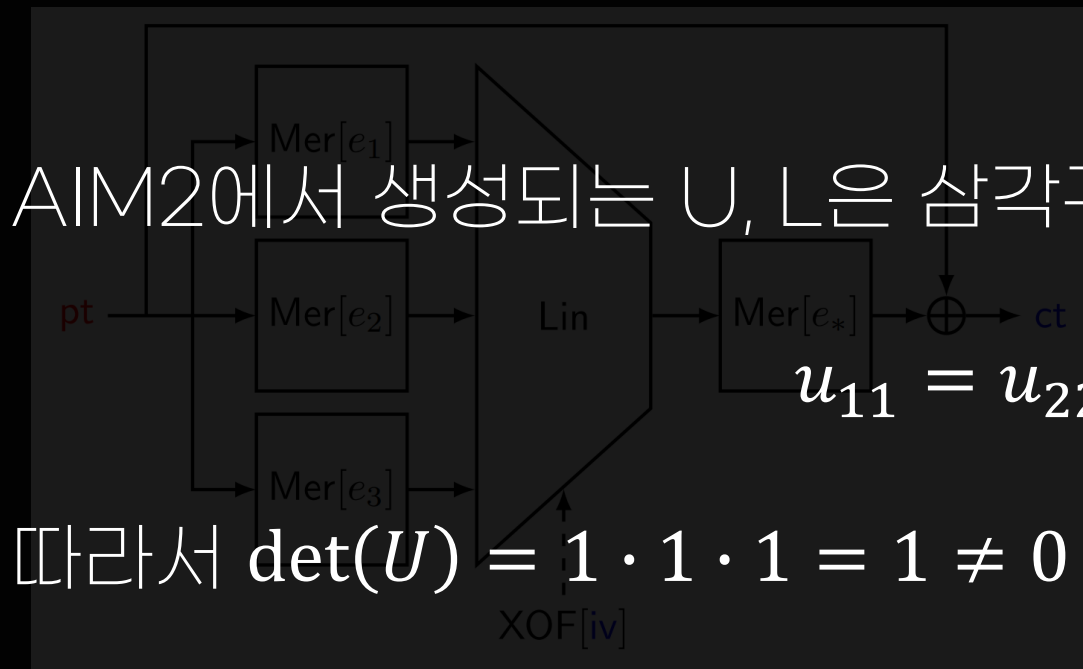
• S-box별(3개)  $L, U$  삼각행렬 생성

• 그리고 벡터  $b$  생성

•  $L/U$ 는 삼각구조 + 대각선 1로 항상 가역 구조

$$U = \begin{bmatrix} u_{11} & * & * \\ 0 & u_{22} & * \\ 0 & 0 & u_{33} \end{bmatrix}$$

$$L = \begin{bmatrix} l_{11} & 0 & 0 \\ * & l_{22} & 0 \\ * & * & l_{33} \end{bmatrix}$$



AIM2에서 생성되는  $U, L$ 은 삼각구조, 대각선 원소를 강제로 1로 설정

$$u_{11} = u_{22} = u_{33} = 1$$

따라서  $\det(U) = 1 \cdot 1 \cdot 1 = 1 \neq 0$

```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF_vector_b = {0,};

    GF_state[AIM2_NUM_INPUT_SBOX];
    GF_pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // Linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // Linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

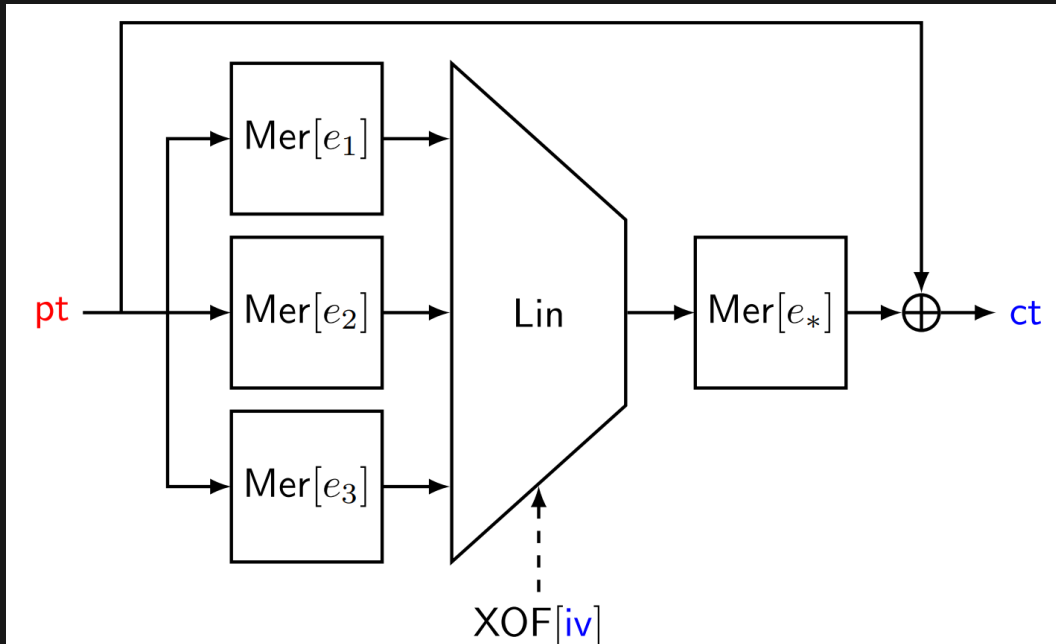
    // Linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# AIMer의 핵심 연산자 AIM2

- [Stage 1] 상수 추가

- aim2\_constants[3]은 고정 상수



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF vector_b = {0,};

    GF state[AIM2_NUM_INPUT_SBOX];
    GF pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```



# AIMer의 핵심 연산자 AIM2

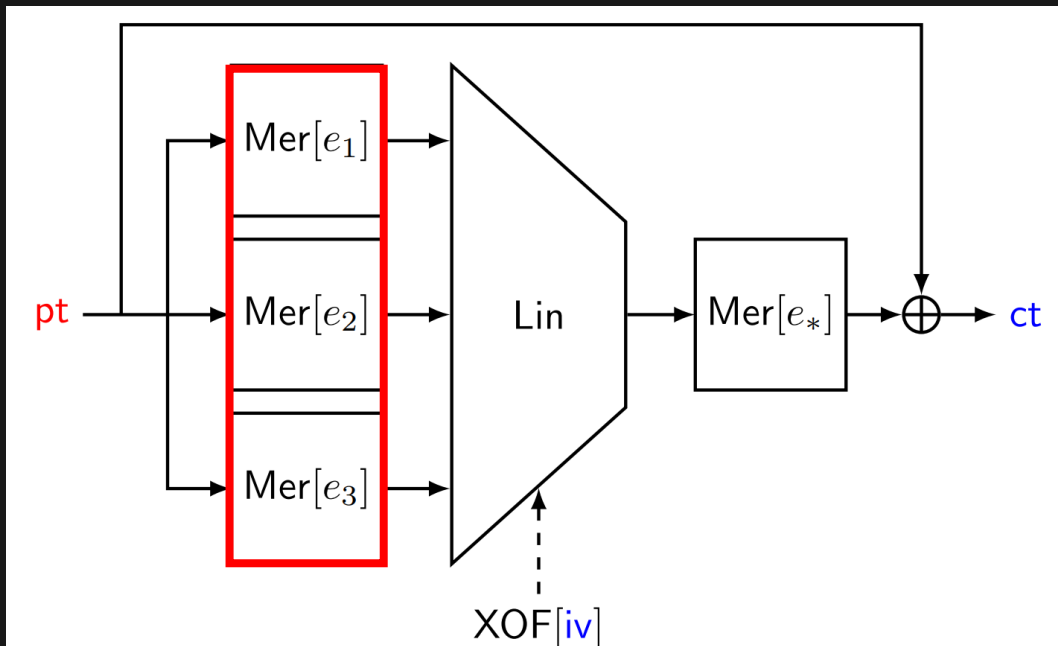
## • [Stage 2] 역 S-box 3개

- Mersenne 형태 지수 ( $2^e - 1$ )를 쓰는 S-box를 두고
- 그 역함수는 지수의 모듈러 역원

$$(2^e - 1)^{-1} \bmod (2^{256} - 1)$$

- 로 거듭 제공한 것

- 즉 각 state는  $x \rightarrow x^{(2^e-1)^{-1} \bmod (2^{256}-1)}$



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF_matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF_matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF_vector_b = {0,};

    GF_state[AIM2_NUM_INPUT_SBOX];
    GF_pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

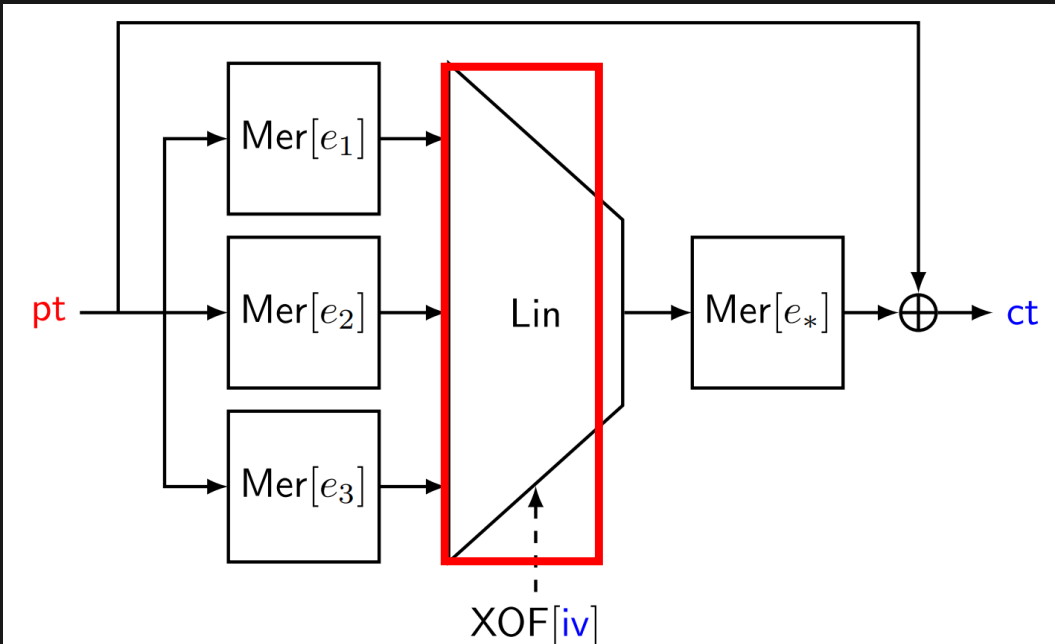
    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# AIMer의 핵심 연산자 AIM2

## • [Stage 3] Affine 변환 (U→L 순서)

- Matrix\_U[i], matrix\_L[i]는 각각 상삼각/하삼각 구조
- 각 S-box 출력에 대해 서로 다른 랜덤 선형 변환을 적용



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF vector_b = {0,};

    GF state[AIM2_NUM_INPUT_SBOX];
    GF pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

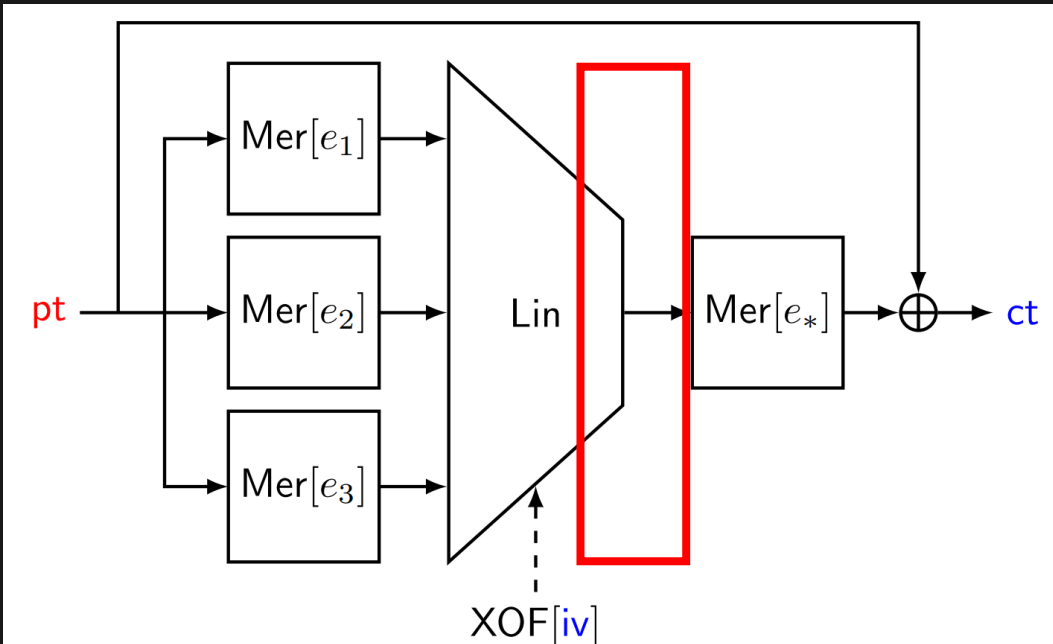
    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# AIMer의 핵심 연산자 AIM2

- [Stage 4] 선형결합
  - 3개 state를 하나로 섞음



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF vector_b = {0,};

    GF state[AIM2_NUM_INPUT_SBOX];
    GF pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

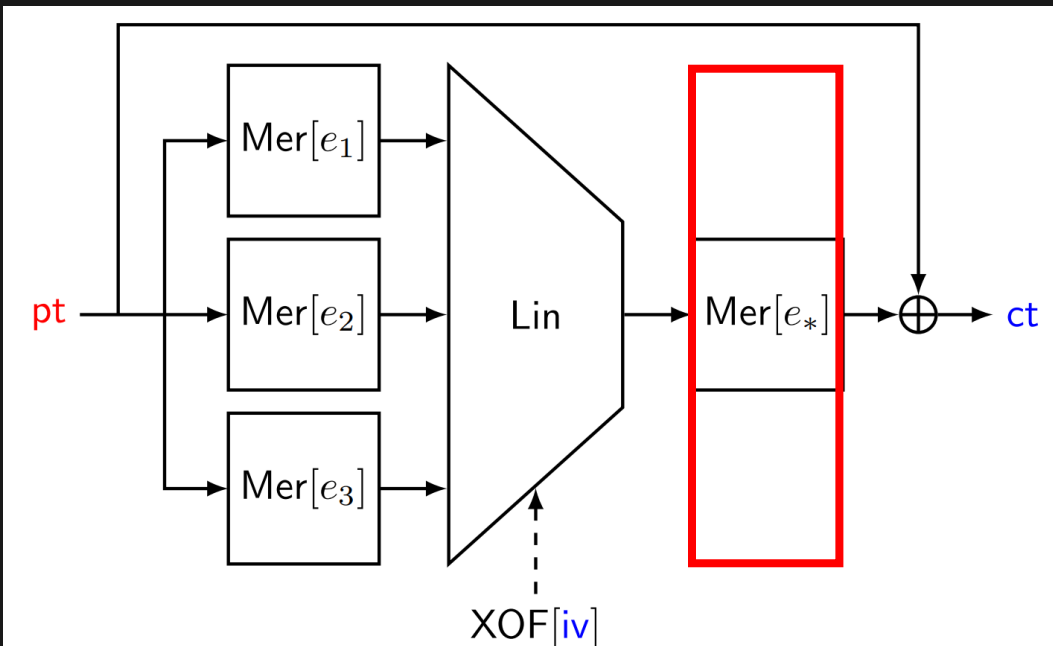
    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# AIMer의 핵심 연산자 AIM2

## • [Stage 5] Mersenne S-box

- $x \rightarrow x^{2^3-1} = x^7$



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF vector_b = {0,};

    GF state[AIM2_NUM_INPUT_SBOX];
    GF pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

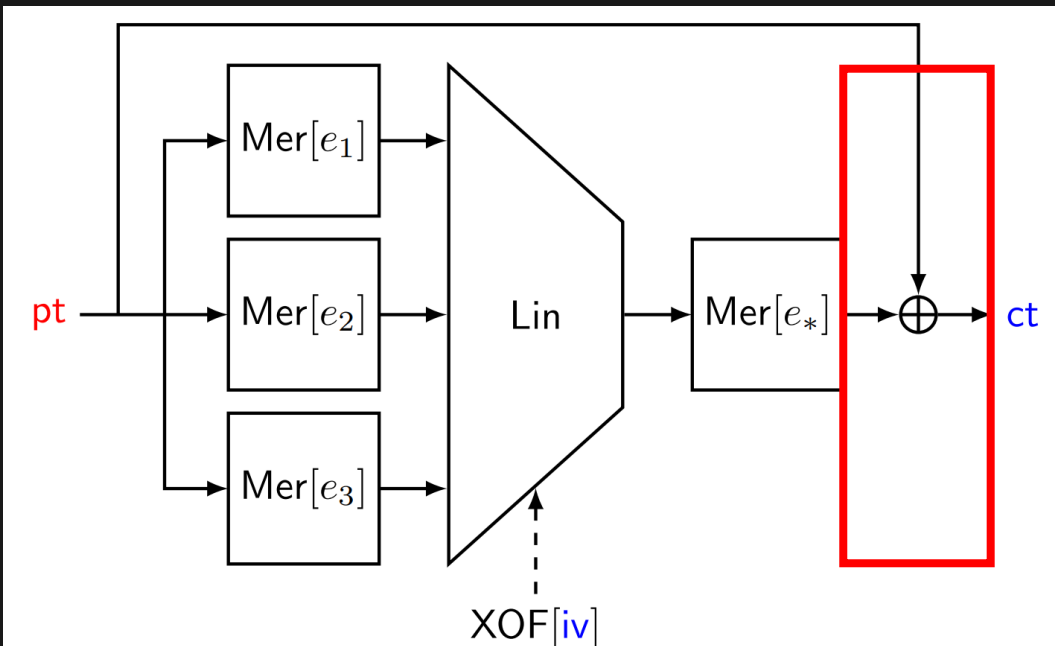
    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# AIMer의 핵심 연산자 AIM2

- [Stage 6] Feed-forward
  - 평문을 다시 XOR



```
void aim2(uint8_t ct[AIM2_NUM_BYTES_FIELD],
          const uint8_t pt[AIM2_NUM_BYTES_FIELD],
          const uint8_t iv[AIM2_IV_SIZE])
{
    GF matrix_L[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF matrix_U[AIM2_NUM_INPUT_SBOX][AIM2_NUM_BITS_FIELD];
    GF vector_b = {0,};

    GF state[AIM2_NUM_INPUT_SBOX];
    GF pt_GF = {0,}, ct_GF = {0,};
    GF_from_bytes(out: pt_GF, in: pt);

    // generate random matrix
    generate_matrices_L_and_U(matrix_L, matrix_U, vector_b, iv);

    // linear component: constant addition
    GF_add(c: state[0], a: pt_GF, b: aim2_constants[0]);
    GF_add(c: state[1], a: pt_GF, b: aim2_constants[1]);
    GF_add(c: state[2], a: pt_GF, b: aim2_constants[2]);

    // non-linear component: inverse Mersenne S-box
    GF_exp_invmer_e_1(out: state[0], in: state[0]);
    GF_exp_invmer_e_2(out: state[1], in: state[1]);
    GF_exp_invmer_e_3(out: state[2], in: state[2]);

    // linear component: affine layer
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_U[0]);
    GF_transposed_matmul(c: state[0], a: state[0], b: (const GF *)matrix_L[0]);

    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_U[1]);
    GF_transposed_matmul(c: state[1], a: state[1], b: (const GF *)matrix_L[1]);

    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_U[2]);
    GF_transposed_matmul(c: state[2], a: state[2], b: (const GF *)matrix_L[2]);

    GF_add(c: state[0], a: state[0], b: state[1]);
    GF_add(c: state[2], a: state[2], b: vector_b);
    GF_add(c: state[0], a: state[0], b: state[2]);

    // non-linear component: Mersenne S-box
    GF_exp_mer_e_star(out: state[0], in: state[0]);

    // linear component: feed-forward
    GF_add(c: ct_GF, a: state[0], b: pt_GF);

    GF_to_bytes(out: ct, in: ct_GF);
}
```

# Binary field

- **필드 정의**

- $F_{2^n} = F_2[x]/\langle f(x) \rangle$
- $F_2 = \{0,1\}$
- $f(x)$ : 차수  $n$ 의 기약 다항식 (irreducible polynomial)
- AIM2에서는  $n = 256$

- **원소 표현**

- $a \in F_{2^n} \leftrightarrow a(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}, a_i \in \{0,1\}$
- 즉  $n$ 비트 벡터 = 하나의 필드 원소

# Binary Field Addition

- **두 원소  $a(x), b(x) \in F_2^n$ 에 대해**

- $a(x) + b(x) = \sum_{i=0}^{n-1} (a_i \oplus b_i) x^i$

- 계수 연산은  $F_2$ 에서 수행
  - $\oplus$ 는 XOR 연산
  - Carry 없음
  - 데이터 의존 분기 없음

- **성질**

- $a + a = 0, a + 0 = a$

# Binary Field Multiplication

- **수학적 정의 (다항식 곱)**

- $a(x) \cdot b(x) = (\sum_i a_i x^i)(\sum_j b_j x^j) \bmod f(x)$
- 일반 다항식 곱 (degree  $\leq 2n - 2$ )
- 결과를 기약 다항식  $f(x)$ 로 나눈 나머지

- **성질**

- 교환법칙, 결합법칙 성립
- 덧셈보다 비쌘

- **구현관점**

- Shift + XOR 기반
- Reduction 단계 필요



# Binary Field Squaring

- **수학적 정의**

- $a(x)^2 = (\sum_i a_i x^i)^2 = \sum_i a_i x^{2i}$  (characteristic 2)
- 교차항이 모두 사라짐
  - $(a + b)^2 = a^2 + b^2$
- 연산 이후  $a(x)^2 \bmod f(x)$

- **구현 관점**

- 단순히 기존 비트열 사이에 0을 추가하는 연산
- Pre-computed reduction 가능
- 곱셈보다 훨씬 저렴

# Binary Field Squaring

- 수학적 정의

- $a(x)^2 = (\sum_i a_i x^i)^2 = \sum_i a_i x^{2i}$  (characteristic 2)

- 교차항이 모두 사라짐

- $(a + b)^2 = a^2 + b^2$

연산	수학적 정의	구현 특징
덧셈	비트별 XOR	매우 저렴
곱셈	다항식 곱 + mod f(x)	상대적으로 비쌘
제곱	계수 위치 이동	매우 저렴

- 구현 관점

- 단순히 기존 비트열 사이에 0을 추가하는 연산
  - precomputed reduction 가능
  - 곱셈보다 훨씬 저렴

# 사전테이블 기반 Binary Field Multiplication

## • 64-비트 곱셈 연산

$$c(x) = a(x) \cdot b(x) = \sum_{k=0}^{126} c_k x^k$$

## • 사전테이블 (table[16])이 의미하는 것

- 핵심 아이디어는  $b$ 를 4비트씩 끊어서 곱셈 수행
- $t \in [0,15]$ 에 대해 다음을 미리 계산해 둬
  - $table[t] = (a_{low61}) \cdot t$
  - $a$ 의 하위 61비트만 테이블에 넣음  
→ 64-비트 내에서 안전하게 다루기 위함 (오버플로 제어)
- GF(2) 선형성 때문에 테이블 구성은 복잡하지 않음
  - $a \cdot (2) = a \cdot x$
  - $a \cdot (4) = a \cdot x^2$
  - $a \cdot (8) = a \cdot x^3$

```
uint64_t table[16];
uint64_t temp, mask, high, low;
uint64_t top3 = a >> 61;

table[0] = 0;
table[1] = a & 0xffffffffffffffffULL;
table[2] = table[1] << 1;
table[4] = table[2] << 1;
table[8] = table[4] << 1;

table[3] = table[1] ^ table[2];

table[5] = table[1] ^ table[4];
table[6] = table[2] ^ table[4];
table[7] = table[1] ^ table[6];

table[9] = table[1] ^ table[8];
table[10] = table[2] ^ table[8];
table[11] = table[3] ^ table[8];
table[12] = table[4] ^ table[8];
table[13] = table[5] ^ table[8];
table[14] = table[6] ^ table[8];
table[15] = table[7] ^ table[8];
```

# 사전테이블 기반 Binary Field Multiplication

- **$b$ 를 4비트씩 나누어서 연산**

- $b = \sum_{k=0}^{15} b_k \cdot 2^{4k} \quad (b_k \in [0, 15])$
- $a \cdot b = \bigoplus_{k=0}^{15} (a \cdot b_k) \ll (4k)$

- **$\text{high} \hat{=} \text{temp} \gg (64\text{-shift})$ 의 이유**

- $\text{temp} \ll \text{shift}$ 는 64비트 경계를 넘는 상위 비트가 생김
- C언어에서 64비트 변수에  $\ll$ 를 하면 넘친 비트는 버려짐
- 넘친 비트를 따로  $\text{high}$ 에 XOR로 넘겨 넣어야 함

```
temp = table[(b >> shift) & 0xf];  
low ^= temp << shift;  
high ^= temp >> (64 - shift);
```

```
low = table[b & 0xf];  
temp = table[(b >> 4) & 0xf];  
low ^= temp << 4;  
high = temp >> 60;  
temp = table[(b >> 8) & 0xf];  
low ^= temp << 8;  
high ^= temp >> 56;  
temp = table[(b >> 12) & 0xf];  
low ^= temp << 12;  
high ^= temp >> 52;  
temp = table[(b >> 16) & 0xf];  
low ^= temp << 16;  
high ^= temp >> 48;  
temp = table[(b >> 20) & 0xf];  
low ^= temp << 20;  
high ^= temp >> 44;  
temp = table[(b >> 24) & 0xf];  
low ^= temp << 24;  
high ^= temp >> 40;
```

# 사전테이블 기반 Binary Field Multiplication

- 상위 3비트에 대한 보정

- top3은 상위 3비트를 뽑아 둔것

```
static void poly64_mul(uint64_t *c1, uint64_t *c0, uint64_t a, uint64_t b)
{
    uint64_t table[16];
    uint64_t temp, mask, high, low;
    uint64_t top3 = a >> 61;
```

- 만약 a의 61번째 비트가 1이면 곱셈에서 다음 항이 추가되어야 함

- $(x^{61}) \cdot b(x) = b(x) \ll 61$
- 마찬가지로 62, 63번째 비트도 각각  $b \ll 62$ ,  $b \ll 63$ 을 XOR로 더해야 함

```
mask = -(int64_t)(top3 & 0x1);
low ^= mask & (b << 61);
high ^= mask & (b >> 3);
mask = -(int64_t)((top3 >> 1) & 0x1);
low ^= mask & (b << 62);
high ^= mask & (b >> 2);
mask = -(int64_t)((top3 >> 2) & 0x1);
low ^= mask & (b << 63);
high ^= mask & (b >> 1);
```

테이블 기반 구현은 성능이  
좋지만 constant timing은 아님

# 비트슬라이싱이란?

## • Timing Attack 분석

- Timing이 비밀정보라면 Timing을 모르게 만들면 안전성 확보
- 따라서 시간 정보에서 비밀정보 제거 혹은 시간을 항상 일정하게 만듦



Timing Blinding  
- shuffling, dummy



Constant Timing

## • 비트슬라이싱의 장점

- 분기 없음 → constant time
- 메모리 접근없음 → 캐시 안전
- XOR/shift/mask만 사용 → 이식성 우수
- SIMD없이도 SWAR (SIMD Within A Register) 병렬성 확보

1 0 1 1 0 1 0 1 1 1

	n1	n2	n3	n4
s1	0	1	0	0
s2	1	1	0	1
s3	0	0	1	1
s4	0	1	0	1

# 비트슬라이싱을 위한 비트교환 커널

```
s = ((a >> k) ^ b) & mask;  
a ^= s << k;  
b ^= s;
```

- **a와 b 사이에서 k 만큼 떨어진 비트를 mask가 지정한 위치에 한해 정확히 서로 교환 (swap)**
  - XOR만으로 swap 수행
  - 임시 변수 하나 (s)만 사용
  - 완전히 가역적 (reversible)

# 비트슬라이싱을 위한 비트교환 커널

```
s = ((a >> k) ^ b) & mask;  
a ^= s << k;  
b ^= s;
```

## • 1비트 교환 예제

- $k=1$
- $\text{mask} = 0b01010101$  (짝수 비트 위치만 선택)
- 8비트 예제
- $\text{mask}$  가 1인 위치  $i$ 에 대해  $a[i] \leftrightarrow b[i-1]$  교환

### 변경 전

$a = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$   
 $b = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

### 변경 후

$a = b_6 a_6 b_4 a_4 b_2 a_2 b_0 a_0$   
 $b = b_7 a_7 b_5 a_5 b_3 a_3 b_1 b_1$



# 비트슬라이싱을 위한 비트교환 커널

```
s = ((a >> k) ^ b) & mask;  
a ^= s << k;  
b ^= s;
```

- **$a \gg k$**

- $a$ 의 비트가  $k$ 만큼 오른쪽으로 이동
- 비교 대상 (비트) 정렬

- **$(a \gg k) \wedge b$**

- 두 비트가 다르면 1, 같으면 0
- 교환이 발생하는 지점

$s$ 의  $i$ 번째 비트가 1일 경우 의미

$a$ 의  $i+k$  번째 비트와  $b$ 의  $i$ 번째 비트가 다르므로 교환 수행

- **$\&mask$**

- 교환을 허용한 위치만 남김

# 비트슬라이싱을 위한 비트교환 커널

```
s = ((a >> k) ^ b) & mask;
```

```
a ^= s << k;
```

```
b ^= s;
```

- **s의 i번째 비트가 1인 위치**

- a의 i+k번째 비트에 대해 반전 수행 (XOR)
- b의 i번째 비트에 대해 반전 수행 (XOR)

- **교환의 이유**

- 두 비트가 서로 달랐기 때문에 둘 다 뒤집으면 서로 값이 교환되는 효과

a	b	s	a'	b'
0	0	0	0	0
1	1	0	1	1
0	1	1	1	0
1	0	1	0	1

# 비트슬라이싱을 위한 비트교환 커널

- **mask에 따른 비트교환 단위**

k	mask 의미	교환 단위
1	0x5555...	1비트
2	0x3333...	2비트
4	0x0f0f...	4비트
8	0x00ff...	8비트
16	0x0000ffff...	16비트
32	0x00000000ffffffff	32비트

# SWAR 기반 비트전치 + 컨볼루션 곱셈기 구현

- **비트 전치 단계**

- 64비트 안에 섞여 있는 비트들을 분리하여 각 레지스터가 특정 비트 위치 집합만 갖도록 재배열

- **즉 bit-slicing / bit-matrix transpose를 수행**

- **해당 단계는 SWAR 병렬성을 만들기 위한 준비 단계**

```
x1 = x0 ^ (x0 >> 32); // will ignore outside C5
x2 = ((x0 ^ (x0 >> 16)) & C4) | ((x1 ^ (x1 << 16)) & (C4 << 16));
x3 = ((x0 ^ (x0 >> 8)) & C3) | ((x1 ^ (x1 << 8)) & (C3 << 8));
x4 = x2 ^ (x2 >> 8); // will ignore outside C3
x5 = ((x0 ^ (x0 >> 4)) & C2) | ((x1 ^ (x1 << 4)) & (C2 << 4));
x6 = ((x2 ^ (x2 >> 4)) & C2) | ((x3 ^ (x3 << 4)) & (C2 << 4));
x7 = x4 ^ (x4 >> 4); // will ignore outside C2
s = ((x0 >> 2) ^ x2) & C1;
x0 ^= s << 2;
x2 ^= s;
s = ((x1 >> 2) ^ x3) & C1;
x1 ^= s << 2;
x3 ^= s;
s = ((x4 >> 2) ^ x6) & C1;
x4 ^= s << 2;
x6 ^= s;
s = ((x5 >> 2) ^ x7) & C1;
x5 ^= s << 2;
x7 ^= s;
s = ((x0 >> 1) ^ x1) & C0;
x0 ^= s << 1;
x1 ^= s;
s = ((x2 >> 1) ^ x3) & C0;
x2 ^= s << 1;
x3 ^= s;
s = ((x4 >> 1) ^ x5) & C0;
x4 ^= s << 1;
x5 ^= s;
s = ((x6 >> 1) ^ x7) & C0;
x6 ^= s << 1;
x7 ^= s;
```

# SWAR 기반 비트전치 + 컨볼루션 곱셈기 구현

## • 다항식 컨볼루션 연산 수행

- $z_k = \bigoplus_{i+j=k} x_i y_j$

```
f0 = x0 & y0;
f1 = (x0 & y1) ^ (x1 & y0);
f2 = (x0 & y2) ^ (x1 & y1) ^ (x2 & y0);
f3 = (x0 & y3) ^ (x1 & y2) ^ (x2 & y1) ^ (x3 & y0);
g0 = (x1 & y3) ^ (x2 & y2) ^ (x3 & y1);
g1 = (x2 & y3) ^ (x3 & y2);
g2 = x3 & y3;

f4 = x4 & y4;
f5 = (x4 & y5) ^ (x5 & y4);
f6 = (x4 & y6) ^ (x5 & y5) ^ (x6 & y4);
f7 = (x4 & y7) ^ (x5 & y6) ^ (x6 & y5) ^ (x7 & y4);
g4 = (x5 & y7) ^ (x6 & y6) ^ (x7 & y5);
g5 = (x6 & y7) ^ (x7 & y6);
g6 = x7 & y7;
```

덧셈은 XOR, 곱은 AND  
Carry는 없음

# SWAR 기반 비트전치 + 컨볼루션 곱셈기 구현

- 부분결과 재정렬

```
s = ((f0 >> 1) ^ f1) & C0;
f0 ^= s << 1;
f1 ^= s;
s = ((f2 >> 1) ^ f3) & C0;
f2 ^= s << 1;
f3 ^= s;
s = ((f0 >> 2) ^ f2) & C1;
f0 ^= s << 2;
f2 ^= s;
s = ((f1 >> 2) ^ f3) & C1;
f1 ^= s << 2;
f3 ^= s;
s = ((g0 >> 1) ^ g1) & C0;
g0 ^= s << 1;
g1 ^= s;
s = (g2 >> 1) & C0;
g2 ^= s << 1;
g3 = s;
s = ((g0 >> 2) ^ g2) & C1;
g0 ^= s << 2;
g2 ^= s;
s = ((g1 >> 2) ^ g3) & C1;
g1 ^= s << 2;
g3 ^= s;
```

- 계층적 병합

- 4→8→16→  
32→64-비트

```
t = f0 ^ g0;
f0 ^= ((f5 ^ t) & C2) << 4;
g0 ^= (g5 ^ (t >> 4)) & C2;
t = f1 ^ g1;
f1 ^= (f5 ^ (t << 4)) & (C2 << 4);
g1 ^= ((g5 ^ t) >> 4) & C2;
t = f2 ^ g2;
f2 ^= ((f6 ^ t) & C2) << 4;
g2 ^= (g6 ^ (t >> 4)) & C2;
t = f3 ^ g3;
f3 ^= (f6 ^ (t << 4)) & (C2 << 4);
g3 ^= ((g6 ^ t) >> 4) & C2;
t = f4 ^ g4;
f4 ^= ((f7 ^ t) & C2) << 4;
g4 ^= (g7 ^ (t >> 4)) & C2;
```

# Karatsuba 기반 곱셈기 최적화

- **표현 방식: 256-비트를 64-비트 기반 다항식으로 봄**

- $X = x^{64}$ 라고 둠

- $A(X) = a_0 + a_1X + a_2X^2 + a_3X^3$ ,  $B(X) = b_0 + b_1X + b_2X^2 + b_3X^3$

- 여기서  $a_i, b_i$  자체가 GF(2) 에서의 64비트 다항식

- Poly64\_mul\_s은 64x64 다항식 곱을 생성

- School-book 방식의 경우 16번의 다항식 곱 필요  
→ Karatsuba 방식은 2단 중첩 구조를 통해 9번만 호출

# Karatsuba 기반 곱셈기 최적화

- **대각항 (4번)에 대한 곱셈 수행**

- $a_0b_0, a_1b_1, a_2b_2, a_3b_3$

```
poly64_mul_s(z1: &t[0], z0: &temp[0], x0: a[0], y0: b[0]);  
poly64_mul_s(z1: &t[2], z0: &t[1], x0: a[1], y0: b[1]);  
t[0] ^= t[1];  
  
poly64_mul_s(z1: &t[3], z0: &t[1], x0: a[2], y0: b[2]);  
t[1] ^= t[2];  
  
poly64_mul_s(z1: &temp[7], z0: &t[2], x0: a[3], y0: b[3]);  
t[2] ^= t[3];
```



# Karatsuba 기반 곱셈기 최적화

- 내부 카라츠바 연산 수행

- $(a_0 + a_1X)(b_0 + b_1X) = a_0b_0 + ((a_0 + a_1)(b_0 + b_1) + a_0b_0 + a_1b_1)X + a_1b_1X^2$

```
poly64_mul_s(z1: &t[1], z0: &t[0], x0: (a[0] ^ a[1]), y0: (b[0] ^ b[1]));  
temp[1] ^= t[0];  
temp[2] ^= t[1];  
  
poly64_mul_s(z1: &t[1], z0: &t[0], x0: (a[2] ^ a[3]), y0: (b[2] ^ b[3]));  
temp[3] ^= t[0];  
temp[6] ^= t[1];
```

# Reduction

- $f(x) = x^{256} + x^{10} + x^5 + x^2 + 1$

- $x^{256} \equiv x^{10} + x^5 + x^2 + 1 \pmod{f(x)}$

- **Reduction은 상위 256비트를 하위로 내려서 XOR 연산**

- 10/5/2 만큼 오른쪽 shift
- 혹은 그대로

```
c[3] = temp[3] ^ temp[7];
c[3] ^= (temp[7] << 10) | (temp[6] >> 54);
c[3] ^= (temp[7] << 5) | (temp[6] >> 59);
c[3] ^= (temp[7] << 2) | (temp[6] >> 62);

c[2] = temp[2] ^ temp[6];
c[2] ^= (temp[6] << 10) | (temp[5] >> 54);
c[2] ^= (temp[6] << 5) | (temp[5] >> 59);
c[2] ^= (temp[6] << 2) | (temp[5] >> 62);

c[1] = temp[1] ^ temp[5];
c[1] ^= (temp[5] << 10) | (t[0] >> 54);
c[1] ^= (temp[5] << 5) | (t[0] >> 59);
c[1] ^= (temp[5] << 2) | (t[0] >> 62);

c[0] = temp[0] ^ t[0];
c[0] ^= (t[0] << 10);
c[0] ^= (t[0] << 5);
c[0] ^= (t[0] << 2);
```

## 결론

- 전자서명 알고리즘의 성능과 안전성은 결국 곱셈기법에 의해 좌우됨
- 격자기반 HEATAE와 대칭기반 AIMer는 서로 다른 수학적 기반을 가짐
  - 각각 NTT 및 Binary Field 연산 최적화가 핵심 경쟁력으로 작용
- Constant-time 구현, 분기 제거, 메모리 접근 최소화와 같은 구현 보안 기술은 알고리즘 자체만큼 중요
- 향후 PQC 전환 시대에는 이론적 안전성뿐 아니라 플랫폼 친화적이고 안전한 구현 기술 확보가 필수 과제