

(Session III)

코드기반 및 기타 알고리즘 최적 구현 기술

서화정

Sampler

Classic McEliece / BIKE (+constant & 메모리 최적화)

SPHINCS (+AVX 연산자를 통한 가속화)

구현 기술 세미나를 들어가며...

- 암호 구현의 중점 사항

- 복잡도를 줄이자 (어떠한 방법을 동원하더라도?!)
- 그렇다면 복잡도가 줄면 정말로 구현에도 무조건 좋을까?
- 많은 경우에는 좋음; 하지만 모든 실제 상황에 좋다고 할 수는 없음

- 구현 기술 세미나의 중점 사항

- 컴퓨터가 가진 연산 능력과 저장공간의 trade-off 고려
- 저장공간 (LUT)을 써서 사전연산을 하는 건 좋지만 임베디드 환경에선...
- 컴퓨터 연산자와 레지스터 그리고 파이프라이닝을 함께 고려하는 구현

UPDATES

2022

NIST PQC는 격자와 코드 그리고 해시의 삼자 구조

PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates

July 05, 2022

	Category	Public-Key Encryption/KEMs	Digital Signatures
격자 해시	Algorithm to be Standardized	CRYSTALS-Kyber	CRYSTALS-Dilithium Falcon, SPHINCS+
코드 아이소	Fourth Round	BIKE, Classic McEliece, HQC, SIKE	-

대박난 크리스탈 집안

암호 알고리즘 같은 (?)
복잡한 설계

설마 되겠어?
키가 작으면 취약할 건데...

악플보다 무서운 무플 (HQC)

마내의 화려한 등장 그리고 조용한 퇴장

외로운(?) 해시기반

구관이 명관이다!

샘플러

- 격자 상에서의 가우시안 샘플링은 격자기반 암호의 핵심 기술

- 가우시안 샘플링: 가우시안 분포 상의 랜덤한 값을 출력하는 과정

- 대표적인 가우시안 샘플링

- Klein (SODA 2000) & Gentry, Peikert and Vaikuntanathan (STOC 2008)

- Randomized version of Babai's nearest-plane algorithm

- Slow ($O(n^2)$)** but **high-quality sampler**

- Peikert (CRYPTO 2010)

- Randomized version of Babai's rounding algorithm

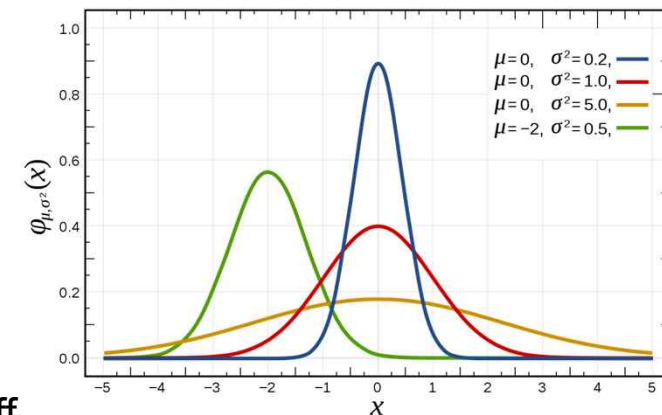
- Fast ($O(n)$)** but **low-quality sampler**

- Ducas, Prest (ePrint 2015)

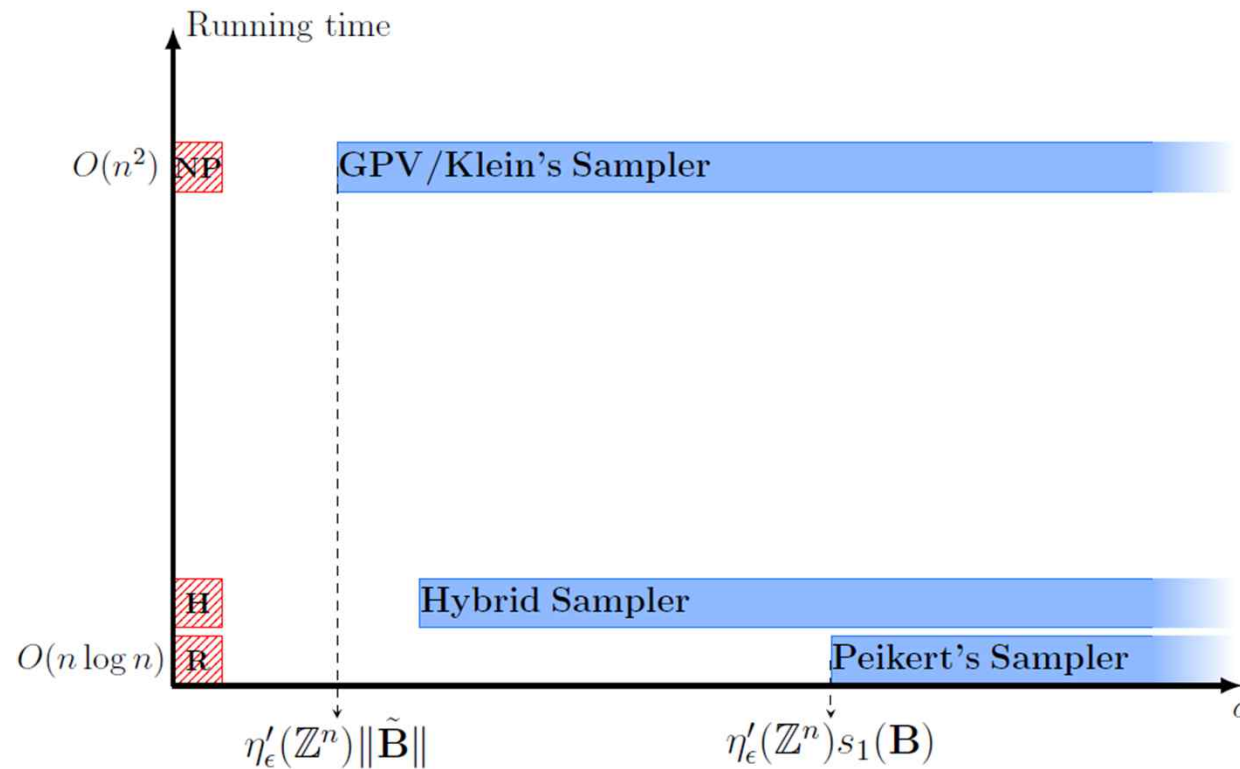
- Hybrid sampler – High level (Klein) + low level (Peikert) → Trade-off

- Ducas, Prest (ACM ISSAC 2016)

- Fast Fourier sampler – FFT를 통해 matrix에 대한 직교화 (**FALCON sampling에 적용**)



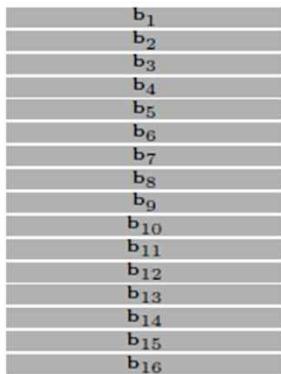
샘플러 적용



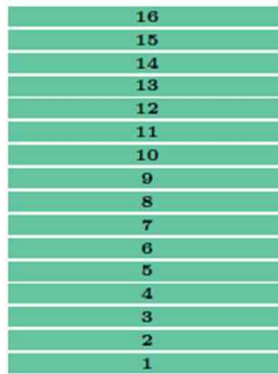
- NTRU prime ring ($R = \mathbb{Z}[x]/(x^p - x - 1)$) 상에서의 성능 비교 분석
- NP (Nearest Plane), H (Hybrid), R (Rounding)
- X축 (표준편차 분포), Y축 (동작 시간)

샘플러 적용

- Klein (192-bit Security Level)은 basis 벡터단위로 샘플링
- Peikert (120-bit Security Level) 는 전체를 한번에 샘플링
- Hybrid (160-bit Security Level) 는 일정수의 벡터를 한번에 샘플링



Original basis



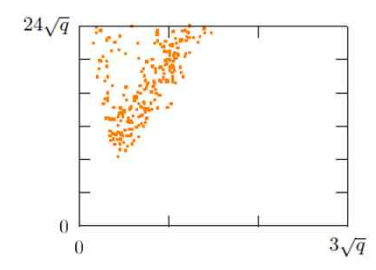
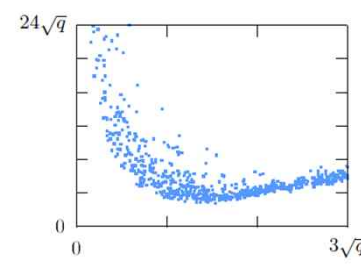
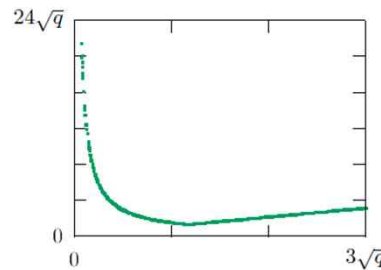
Klein



Hybrid



Peikert



Klein's Sampling in Source Level

Algorithm 1 Ring_Klein($\mathcal{R}, \mathbf{B}, \tilde{\mathbf{B}}, \sigma, \mathbf{c}$)

Require: Basis $\mathbf{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_n\} \in \mathcal{R}^{n \times m}$, its GSO $\tilde{\mathbf{B}} = \{\tilde{\mathbf{b}}_1, \dots, \tilde{\mathbf{b}}_n\} \in \mathbb{K}^{n \times m}$, $\sigma \in \mathbb{K}^+$, target $\mathbf{c} \in \mathbb{K}^m$

Ensure: \mathbf{v} sampled in a distribution close to $\mathcal{D}_{\text{Span}_{\mathcal{R}}(\mathbf{B}), \sigma, \mathbf{c}}$

```

1:  $\mathbf{c}_n \leftarrow \mathbf{c} \in \mathbb{K}^m$ 
2:  $\mathbf{v}_n \leftarrow \mathbf{0} \in \mathbb{R}^m$ 
3: for  $i \leftarrow n, \dots, 1$  do
4:    $d_i \leftarrow \langle \mathbf{c}_i, \tilde{\mathbf{b}}_i \rangle_{\mathbb{K}} / \|\tilde{\mathbf{b}}_i\|_{\mathbb{K}}^2 \in \mathbb{K}$ 
5:    $\Sigma_i \leftarrow \sigma^2 / \|\tilde{\mathbf{b}}_i\|_{\mathbb{K}}^2 \in \mathbb{K}$ 
6:    $z_i \leftarrow \text{SampleR}(\mathcal{R}, \Sigma_i, d_i) \in \mathcal{R}$ 
7:    $\mathbf{c}_{i-1} \leftarrow \mathbf{c}_i - z_i \tilde{\mathbf{b}}_i \in \mathbb{K}^m$ 
8:    $\mathbf{v}_{i-1} \leftarrow \mathbf{v}_i + z_i \mathbf{b}_i \in \mathbb{R}^m$ 
9: end for
10: return  $\mathbf{v}_0$ 

```

Pros: 빠른 구현 / **Cons:** 라이브러리 의존도

NTL: A Library for doing Number Theory

NTL is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields.

By default, NTL is **thread safe**.

NTL is distributed under **LGPLv2.1+** (i.e., LGPL version 2.1 or later) [\[more details\]](#)

If you are interested in contributing to the development of NTL, please contact me (see contact info below). I have a few projects in mind, and would be happy to discuss other ideas as well.

```

vec_RR Samplers::Klein(const mat_ZZ& B, const mat_RR& BTilde, RR sigma, long precision, int tailcut, const vec_RR center) {
    cout << "\n[*] Klein's sampler status: "; // Class RR은 Arbitrary-precision floating point numbers을 지칭
    RR::SetPrecision(precision); // floating point의 precision을 설정

    double sizeof_RR = pow(2.0, sizeof(RR));
    RR aux_B, c_prime, innerp, innerpl, sigma_prime, Z;
    vec_RR c, v;
    int cols, i, j, rows;

    cols = BTilde.NumCols();
    rows = BTilde.NumRows();

    c.SetLength(cols);
    v.SetLength(cols);

    clear(v); // 초기화 및 센터값 설정
    c = center;

```

샘플링 연산 수행 프로세스

- 샘플을 취할 목표에 대한 파라미터 세팅
- 해당 파라미터 상에서 샘플링 수행

잠시 외부 라이브러리 관련하여 이야기하면...

- NIST 공모전에서도 초기에 특정 플랫폼 상에서 최적화 수행
 - Cortex-M4
- 해당 플랫폼 상에는 당연히 NTL, OpenSSL이 올라가지 않음
- 따라서 pure한 구현 결과인 PQClean이 제안되게 됨

PQClean

[See the build status for each component here](#)

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

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

What PQClean is **not** aiming for is

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

As a first main target, we are collecting C implementations that fulfill the requirements listed below. We also accept optimised implementations, but still requiring the source code.



Spoiler Alert: ICISC'22에 pqm4, pqClean 제작자 초청강연 예정

Klein's Sampling in Source Level

Algorithm 1 Ring_Klein($\mathcal{R}, \mathbf{B}, \tilde{\mathbf{B}}, \sigma, \mathbf{c}$)

Require: Basis $\mathbf{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_n\} \in \mathcal{R}^{n \times m}$, its GSO $\tilde{\mathbf{B}} = \{\tilde{\mathbf{b}}_1, \dots, \tilde{\mathbf{b}}_n\} \in \bar{\mathbb{K}}^{n \times m}$, $\sigma \in \bar{\mathbb{K}}^+$, target $\mathbf{c} \in \bar{\mathbb{K}}^m$

Ensure: \mathbf{v} sampled in a distribution close to $\mathcal{D}_{\text{Span}_{\mathcal{R}}(\mathbf{B}), \sigma, \mathbf{c}}$

```

1:  $\mathbf{c}_n \leftarrow \mathbf{c}$   $\in \bar{\mathbb{K}}^m$ 
2:  $\mathbf{v}_n \leftarrow \mathbf{0}$   $\in \mathcal{R}^m$ 
3: for  $i \leftarrow n, \dots, 1$  do
4:    $d_i \leftarrow \langle \mathbf{c}_i, \tilde{\mathbf{b}}_i \rangle_{\bar{\mathbb{K}}} / \|\tilde{\mathbf{b}}_i\|_{\bar{\mathbb{K}}}^2$   $\in \bar{\mathbb{K}}$ 
5:    $\Sigma_i \leftarrow \sigma^2 / \|\tilde{\mathbf{b}}_i\|_{\bar{\mathbb{K}}}^2$   $\in \bar{\mathbb{K}}$ 
6:    $z_i \leftarrow \text{SampleR}(\mathcal{R}, \Sigma_i, d_i)$   $\in \mathcal{R}$ 
7:    $\mathbf{c}_{i-1} \leftarrow \mathbf{c}_i - z_i \mathbf{b}_i$   $\in \bar{\mathbb{K}}^m$ 
8:    $\mathbf{v}_{i-1} \leftarrow \mathbf{v}_i + z_i \mathbf{b}_i$   $\in \mathcal{R}^m$ 
9: end for
10: return  $\mathbf{v}_0$ 

```

```

for(i = rows-1; i >= 0; i--) {

```

```

    NTL::InnerProduct(innerp1, c, BTilde[i]);
    NTL::InnerProduct(innerp, BTilde[i], BTilde[i]);
    div(c_prime, innerp1, innerp);
    div(sigma_prime, sigma, sqrt(innerp));

```

```

    if(NTL::floor(sigma_prime) == 0)
        Z = c_prime;

```

```

    else {

```

```

        if(sigma_prime > sizeof_RR)

```

```

            NTL::mul(sigma_prime, to_RR(2), sigma); // to_RR: 반올림 연산

```

```

            this->BuildProbabilityMatrix(precision, tailcut, sigma_prime, c_prime);

```

```

            Z = to_RR(this->KnuthYao(tailcut, sigma_prime, c_prime));

```

```

    } //end-else

```

```

    for(j = 0; j < B.NumCols(); j++) {

```

```

        aux_B = to_RR(B[i][j]);

```

```

        v[j] = v[j] + aux_B*Z;

```

```

        c[j] = c[j] - aux_B*Z;

```

```

    } //end-for

```

```

} //end-for

```

전체 샘플링 과정 중에서
가장 많은 연산과
저장공간이 필요한 부분은
샘플링을 실제 수행하는 Step 6

Knuth-Yao Sampling

- Discrete Gaussian distribution으로부터 sampling을 위해 Rejection (Kyber, Dilithium) 혹은 Inversion sampling이 사용됨
- 높은 precision을 위해서는 긴 난수값이 필요한데 이를 임베디드 디바이스에서 효율적으로 sampling을 수행하기 위해 **Knuth-Yao 기법 사용**

- 연산 수행은 두단계로 구성

- Probability matrix 작성
- 작성된 matrix 기반 random walk

Algorithm 1: Knuth-Yao column scanning Sampling

```
input : Probability matrix P
output: Sample value s
1 d ← 0; // Distance between the visited and the rightmost internal node
2 Hit ← 0; // 1 when sampling process hits a terminal node
3 col ← 0; // column number of probability matrix
4 while Hit=0 do
5   r ← RandomBit(); // 무작위 값을 통해 특정한 샘플링 값 추출
6   d ← 2 * d + r; // 노드가 2개씩 늘어나기 때문 → 바이너리 트리
7   for row=MAXROW down to 0 do
8     d ← d - P[row][col]; // 무작위값과 확률분포를 빼줌
9     if d=-1 then
10      s ← row; // 해당 샘플링 값의 위치를 리턴 후 종료
11      Hit ← 1;
12      ExitForLoop();
13   col ← col + 1;
14 return s
```

Knuth-Yao Sampling

- Random-walk model (w/ random bits)

- 샘플 공간 $S = \{0, 1, 2\}$

- $p_0 = 0.01110$
- $p_1 = 0.01101$
- $p_2 = 0.00101$

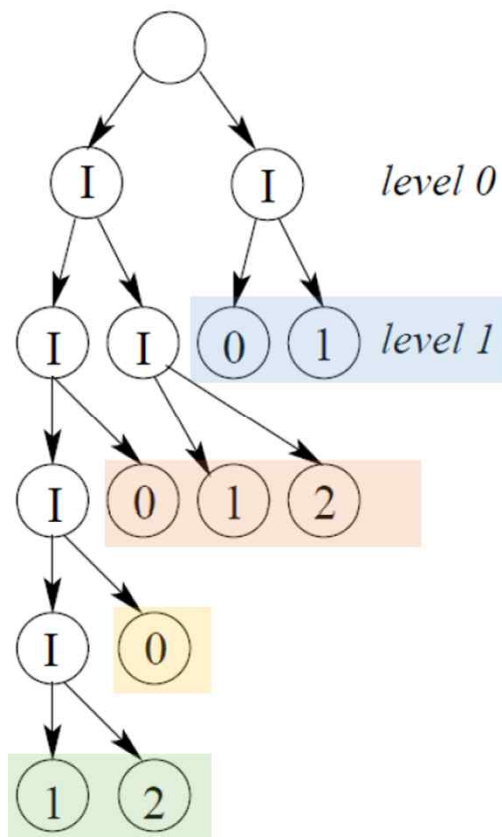
- Probability matrix

$$P_{mat} = \begin{matrix} & \begin{matrix} \text{column 0} \\ \downarrow \end{matrix} \\ \begin{matrix} \text{row 0} \rightarrow \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \end{matrix}$$

Knuth-Yao Sampling

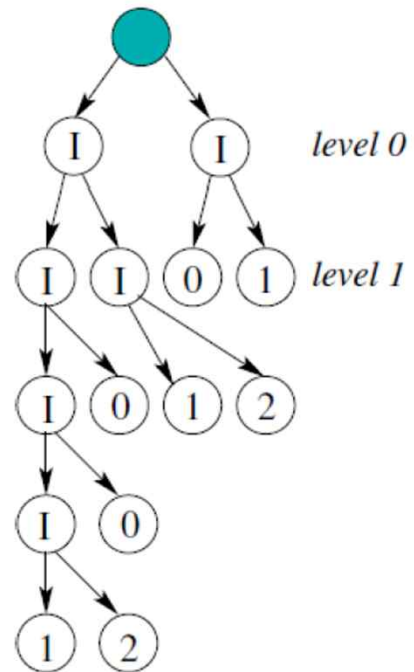
- **Discrete Distribution Generation (DDG)**

- Probability matrix (P_{mat})에 대응되는 바이너리 트리

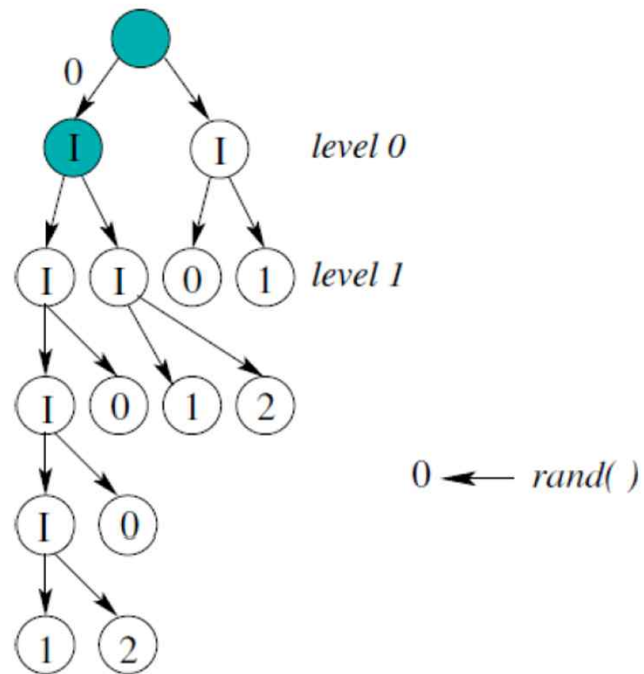


$$P_{mat} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

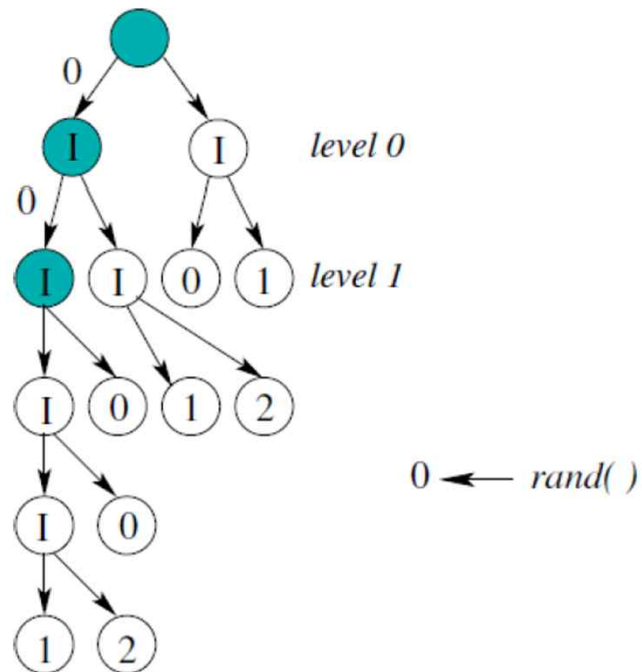
Knuth-Yao Sampling: Random Walk



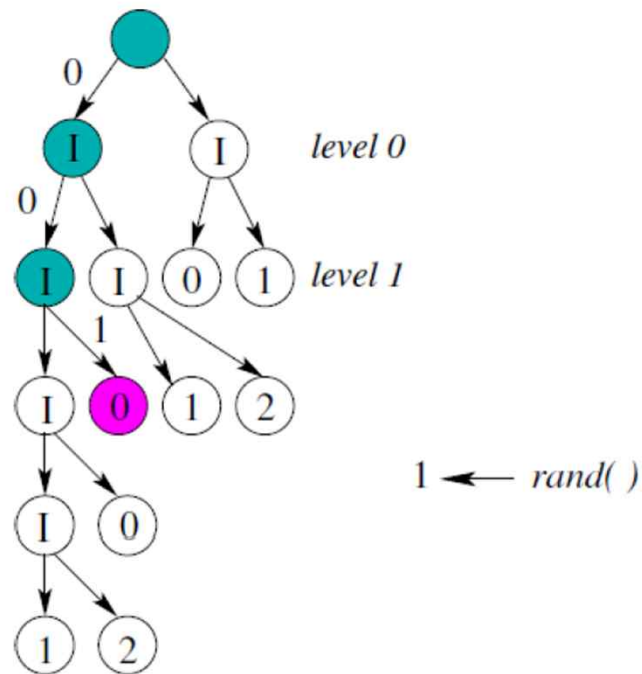
Knuth-Yao Sampling: Random Walk



Knuth-Yao Sampling: Random Walk



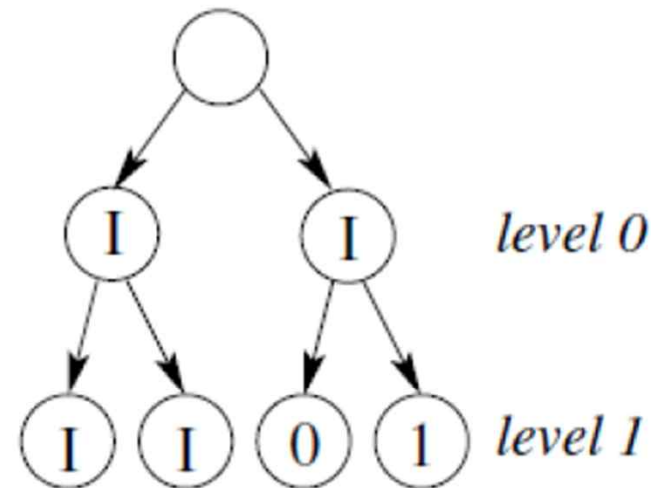
Knuth-Yao Sampling: Random Walk



Knuth-Yao Sampling: Implementation


- Probability matrix를 이용하여 어떤 level도 DDG tree 구성 가능

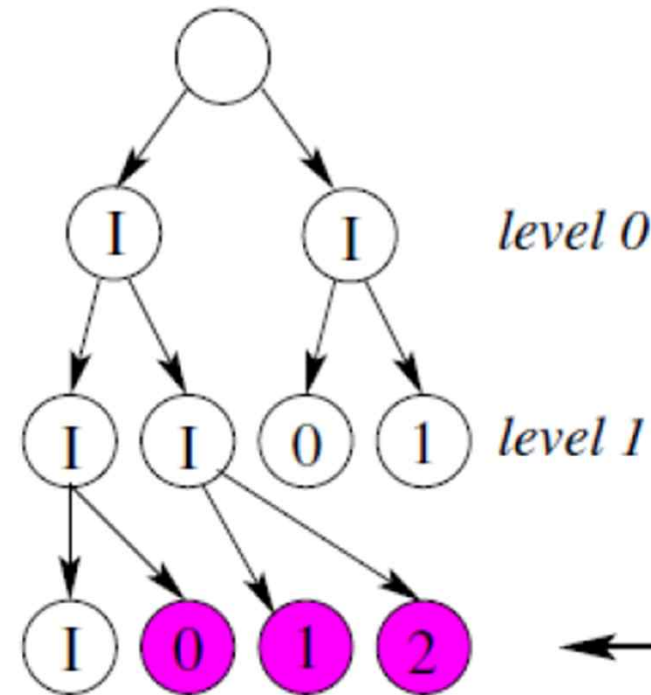
$$P_{mat} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$



Knuth-Yao Sampling: Implementation

- Probability matrix를 이용하여 어떤 level도 DDG tree 구성 가능

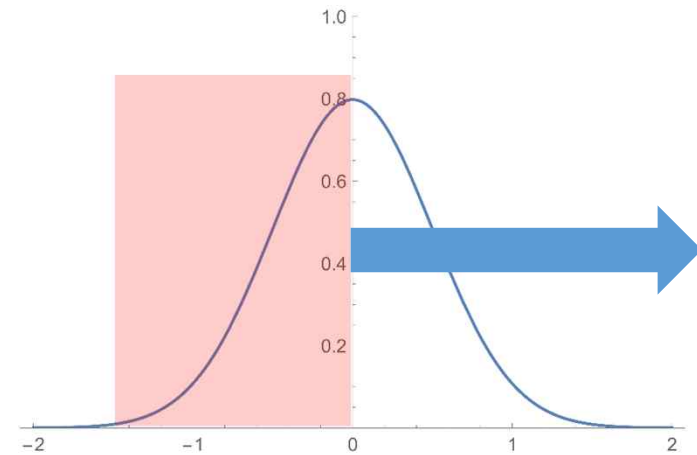
$$P_{mat} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$




Probability Matrix: 107-bit precision, 39-tail-bound

[illegible]

Probability Matrix 생성 상세



분포에 해당하는 값들을 0과 1로 매핑

매핑된 값들은 워드크기로 패킹
(소프트웨어 구현의 경우)

워드 크기

The image features a complex, high-contrast background composed of a dense, repeating pattern of binary digits (0s and 1s). This pattern is arranged in a grid-like fashion, with some areas appearing more concentrated than others. A prominent, thick red diagonal line cuts across the entire image, starting from the upper-left quadrant and extending towards the lower-right corner. In the bottom-left corner, the Korean text '본 맵' (Bong Map) is displayed in a bold, black, sans-serif font. The overall aesthetic is digital and abstract, suggesting a theme related to data, mapping, or digital art.

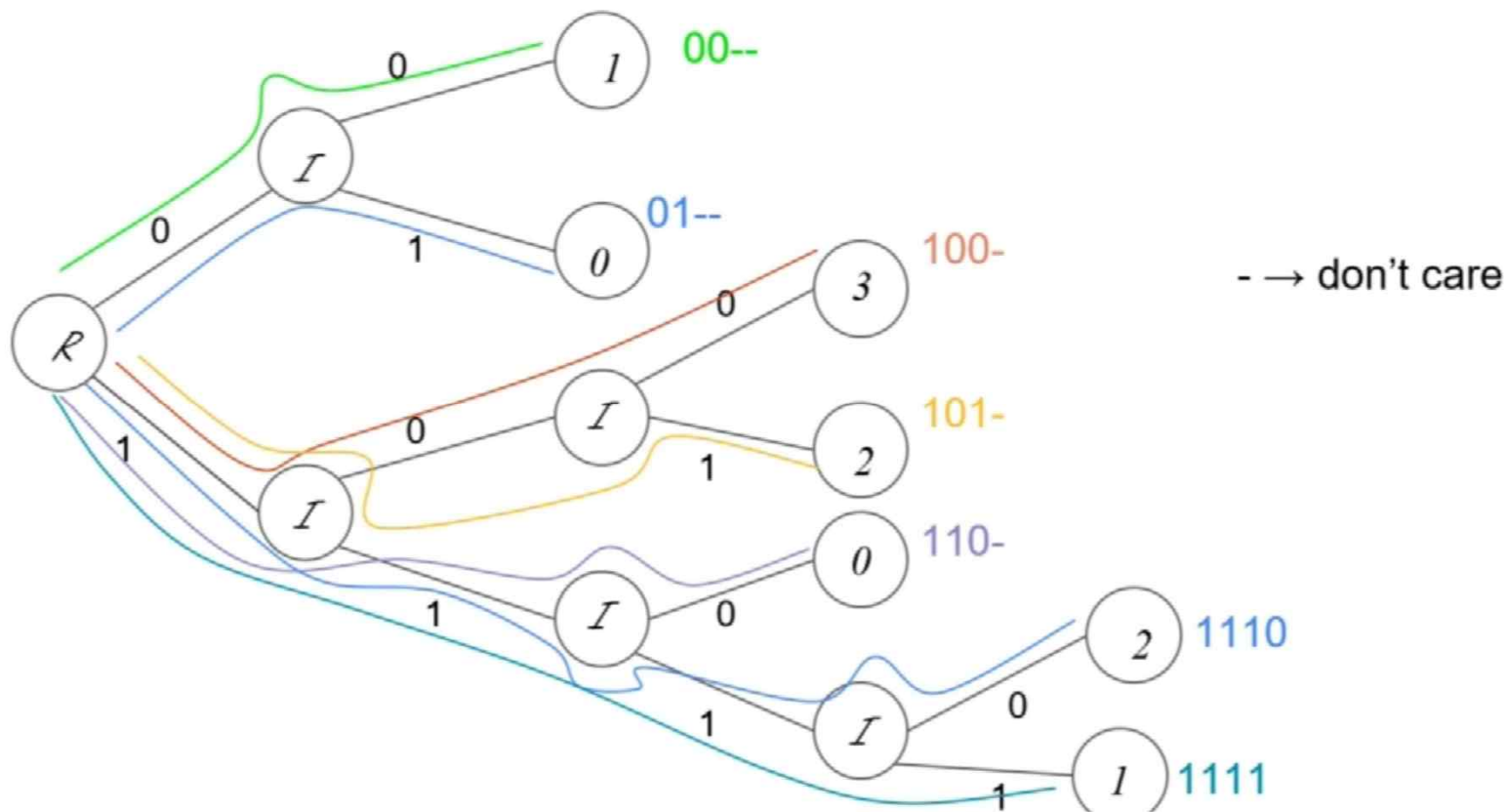
Probability Matrix: Column-wise Optimization

- **Probability matrix는 ROM에 column 축을 기반으로 저장**
 - Column의 하위 zero 값은 저장하지 않음
 - Column의 길이 또한 저장: 길이가 대부분 1씩 연속적으로 차이가 남
 - 하나의 비트를 통해 column-length 결정 (1: 증가, 0: 유지)

```
000111111111010111000101110101
0011111001101110110011011001101
001101001000110011101100011010
001010010010001110000011001110
000111010011001101100110100000
000100101100101100100011010010
000010101111011110010010001110
000001011100110110001001011000
000000101100100010110010101101
000000010011011000000110100010
000000000111101001000111111011
000000000010101110111011001001
00000000000111000101110001100
000000000000010000101011010101
000000000000000100011100100010
00000000000000001000100110001
000000000000000001111000100
00000000000000000010111111
```

Knuth-Yao Sampling (non-constant timing)

- 샘플링 루트에 따라 연산 속도가 달라지는 non-constant timing



Constant Timing

- **Constant Timing이란?**

- 연산 수행에 소모되는 시간은 언제나 일정

- 만약 일정하지 않다면 (variable timing)?

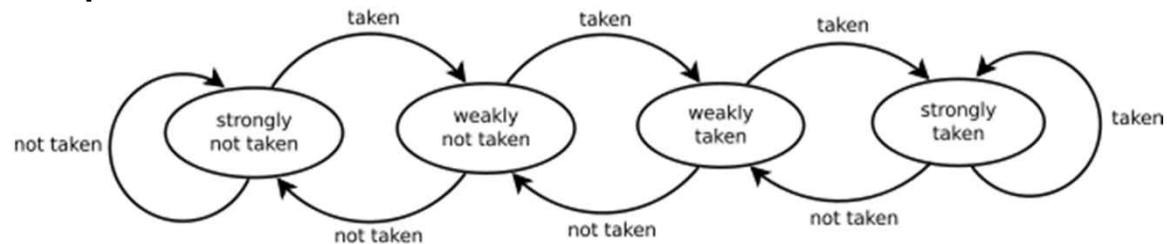
- 예상과는 다른 연산이 수행될 확률이 높음!

- 그러면 동일한 코드가 왜 variable timing일까?

- 기계도 트릭을 쓴다!!!
- 캐시, branch prediction, speculative execution...

- 이득: 시간

- 손해: 보안



Timing Attack in Real World (내 음식은 언제 오나?)



쿠팡이츠 "라이더 안전 위해 1:1 배송 시스템 구축"

기존 배달 시스템과 차별화...1명이 인근지역 3~4건 주문 합배송 안해
라이더 안전 개선 노력...여러 음식점 들리며 배달하지 않아 과속 등 위험요인 제거
고객 만족도 Up...배달 시간 줄이고 동선 및 도착 시간 안내하며 특화된 서비스 지원



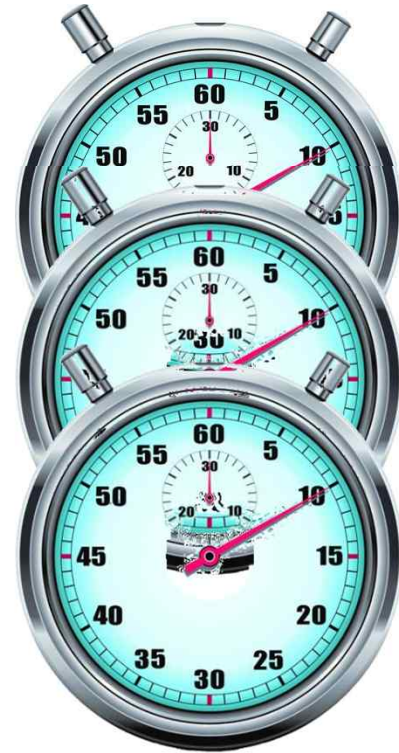
Timing Attack 방어?

- **Timing Attack 원인?**

- 만약 Timing == 비밀정보이라면!!!
- 해결방안 → 시간 정보에서 비밀정보 제거



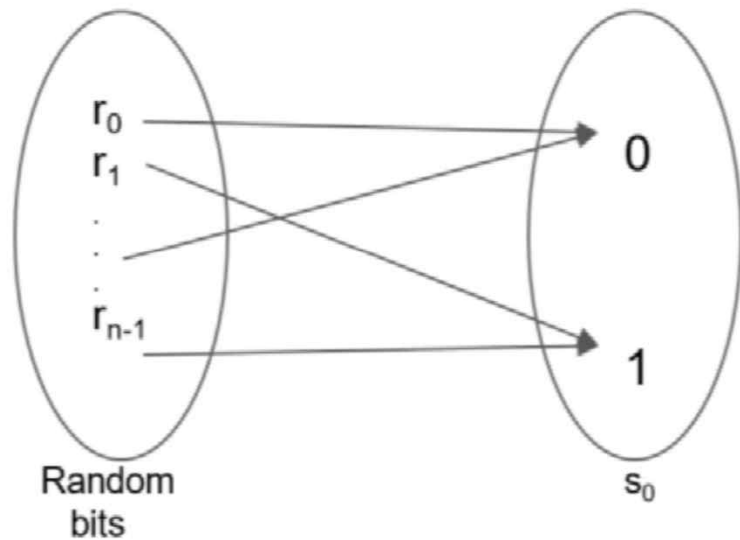
Timing Blinding
- Shuffling, dummy



Constant Timing

Knuth-Yao Sampling (constant timing)

- 난수값에 따라 트리를 따라가는 방법 대신 입력값을 Boolean 연산자로 엮어서 결과값 도출
→ 즉 constant routine (LUT)으로 값을 결정



$$s_0 = f^0(r_0, r_1, r_1, \dots, r_{n-1})$$

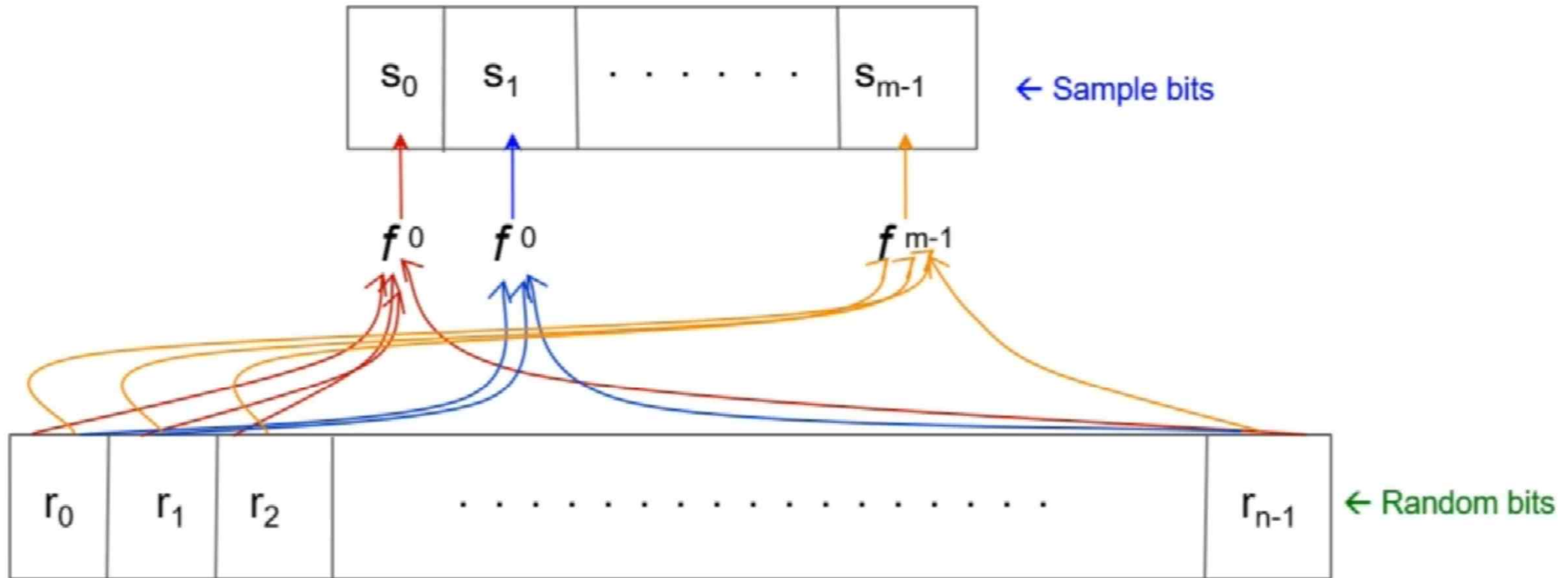
$$s_1 = f^1(r_0, r_1, r_1, \dots, r_{n-1})$$

$$s_{m-1} = f^{m-1}(r_0, r_1, r_1, \dots, r_{n-1})$$

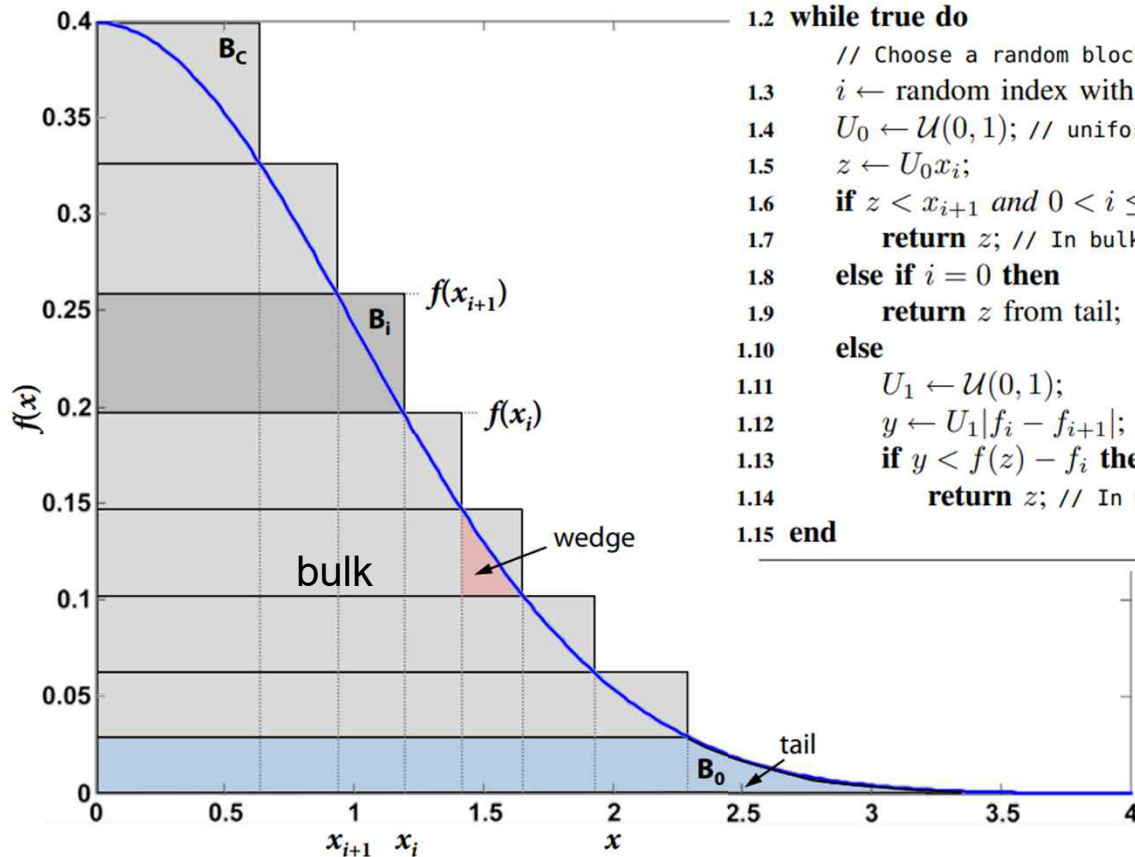
$n = \text{depth of tree}$
 $m = \max(\log_2(S))$

Knuth-Yao Sampling (constant timing)

- 모든 샘플비트 (s)에 해당하는 Boolean 식 생성



이외에도...

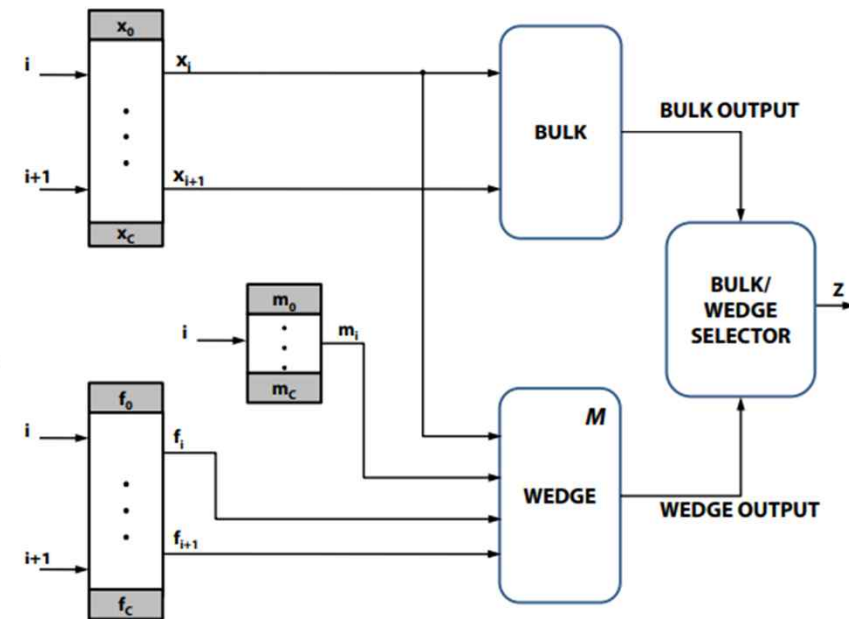


Algorithm 1: Ziggurat Algorithm

input : Number of blocks C
output: Gaussian random number z

```

1.1 Generate points  $f_i = f(x_i)$   $i = 0, \dots, C$ ;
1.2 while true do
    // Choose a random block
1.3  $i \leftarrow$  random index with probability  $1/C$ ;
1.4  $U_0 \leftarrow \mathcal{U}(0, 1)$ ; // uniform random number
1.5  $z \leftarrow U_0 x_i$ ;
1.6 if  $z < x_{i+1}$  and  $0 < i \leq C$  then
1.7     return  $z$ ; // In bulk
1.8 else if  $i = 0$  then
1.9     return  $z$  from tail;
1.10 else
1.11      $U_1 \leftarrow \mathcal{U}(0, 1)$ ;
1.12      $y \leftarrow U_1 |f_i - f_{i+1}|$ ;
1.13     if  $y < f(z) - f_i$  then
1.14         return  $z$ ; // In wedge
1.15 end
    
```



- Ziggurat 알고리즘도 Knuth-Yao와 함께 사용되지만 최근에는 Knuth-Yao에 대한 구현이 보다 많이 연구됨
- 해당 기법은 rejection sampling과 유사하지만 약간은 정밀하게 분포를 traverse하는 것 같음

Peikert & Ducas-Prest Sampler

Algorithm 1 Ring_Klein($\mathcal{R}, \mathbf{B}, \tilde{\mathbf{B}}, \sigma, \mathbf{c}$)

Require: Basis $\mathbf{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_n\} \in \mathcal{R}^{n \times m}$, its GSO $\tilde{\mathbf{B}} = \{\tilde{\mathbf{b}}_1, \dots, \tilde{\mathbf{b}}_n\} \in \bar{\mathbb{K}}^{n \times m}$, $\sigma \in \bar{\mathbb{K}}^+$, target $\mathbf{c} \in \bar{\mathbb{K}}^m$

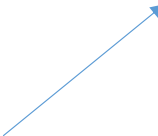
Ensure: \mathbf{v} sampled in a distribution close to $\mathcal{D}_{\text{Span}_{\mathcal{R}}(\mathbf{B}), \sigma, \mathbf{c}}$

```

1:  $\mathbf{c}_n \leftarrow \mathbf{c}$   $\in \bar{\mathbb{K}}^m$ 
2:  $\mathbf{v}_n \leftarrow \mathbf{0}$   $\in \mathcal{R}^m$ 
3: for  $i \leftarrow n, \dots, 1$  do
4:    $d_i \leftarrow \langle \mathbf{c}_i, \tilde{\mathbf{b}}_i \rangle_{\bar{\mathbb{K}}} / \|\tilde{\mathbf{b}}_i\|_{\bar{\mathbb{K}}}^2$   $\in \bar{\mathbb{K}}$ 
5:    $\Sigma_i \leftarrow \sigma^2 / \|\tilde{\mathbf{b}}_i\|_{\bar{\mathbb{K}}}^2$   $\in \bar{\mathbb{K}}$ 
6:    $z_i \leftarrow \text{SampleR}(\mathcal{R}, \Sigma_i, d_i)$   $\in \mathcal{R}$ 
7:    $\mathbf{c}_{i-1} \leftarrow \mathbf{c}_i - z_i \mathbf{b}_i$   $\in \bar{\mathbb{K}}^m$ 
8:    $\mathbf{v}_{i-1} \leftarrow \mathbf{v}_i + z_i \mathbf{b}_i$   $\in \mathcal{R}^m$ 
9: end for
10: return  $\mathbf{v}_0$ 

```

Ring_Klein의 Step6에서 Ring_Peikert 호출



Algorithm 2 Ring_Peikert(\mathcal{R}, Σ, c)

Require: A variance $\Sigma \in \bar{\mathbb{K}}^+$, a target $c \in \bar{\mathbb{K}}$, a precomputed value $b \in \bar{\mathbb{K}}$ such that $\Sigma_e(b\bar{b} + \eta^2) = \Sigma$

Ensure: z sampled according to $D_{\mathcal{R}, \sigma, c}$

```

1:  $p \leftarrow b \cdot D_{\bar{\mathbb{K}}, \sqrt{\Sigma_e}}$   $\in \bar{\mathbb{K}}$ 
2:  $z \leftarrow D_{\mathcal{R}, \eta\sqrt{\Sigma_e}, c+p}$   $\in \mathcal{R}$ 
3: Return  $z$ 

```

Sampler on CRYSTAL (PQC Standard)

- Kyber (CBD, Rejection Sampling)
- Dilithium (Rejection Sampling)

Post-quantum key exchange – a new hope*

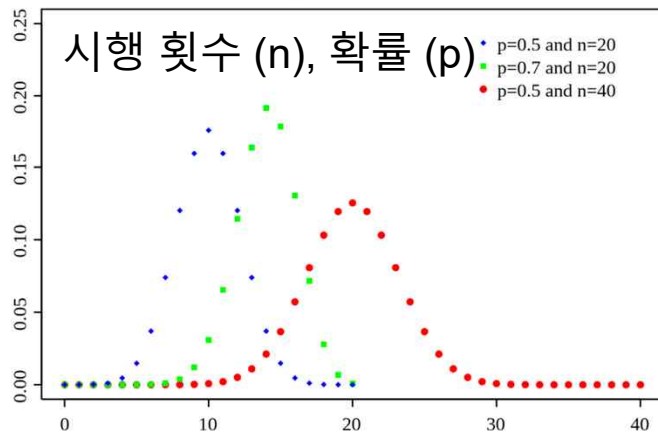
Erdem Alkim
Department of Mathematics, Ege University, Turkey

Léo Ducas
Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands

Thomas Pöppelmann
Infineon Technologies AG, Munich, Germany

Peter Schwabe
Digital Security Group, Radboud University, The Netherlands

Binomial noise. Theoretic treatments of LWE-based encryption typically consider LWE with Gaussian noise, either rounded Gaussian [93] or discrete Gaussian [28]. As a result, many early implementations also sampled noise from a discrete Gaussian distribution, which turns out to be either fairly inefficient (see, for example, [25]) or vulnerable to timing attacks (see, for example, [29, 88, 44]). The performance of the best known attacks against LWE-based encryption does not depend on the exact distribution of noise, but rather on the standard deviation (and potentially the entropy). This motivates the use of noise distributions that we can easily, efficiently, and securely sample from. One example is the centered binomial distribution used in [10]. Another example is the use of “learning-with-rounding” (LWR), which adds deterministic uniform noise by dropping bits as in KYBER’s Compress_q function. In the design of KYBER we decided to use centered binomial noise and thus rely on LWE instead of LWR as the underlying problem.

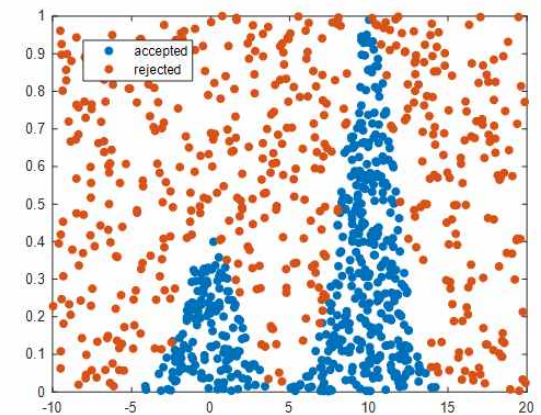
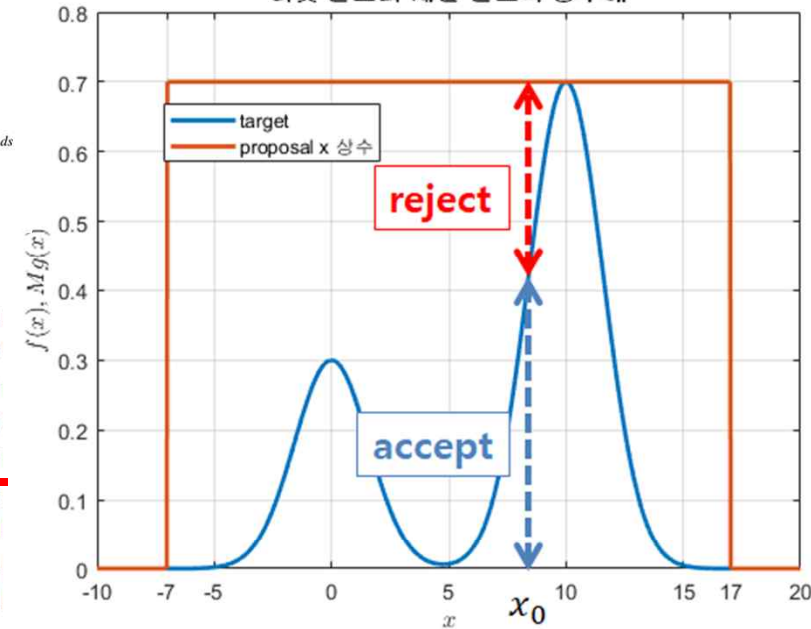


Kyber 팀의 판단

- 가우시안 샘플링은 비효율적 + 취약
- 보안성은 distribution이 아니라 standard deviation에 있음

Rejection sampling

타겟 분포와 제안 분포의 상수배



Sampler on CRYSTAL (PQC Standard)

- Kyber (CBD, Rejection Sampling)
- Dilithium (Rejection Sampling)
- **두 sampler 구현 모두 간단한 사칙 연산으로 수행 가능**
 - 가우시안 샘플링에 비해 간단하며 Kyber 구현은 constant timing
 - Dilithium 구현은 분기문에 대한 고려가 필요해 보임 (AVX 명령어 사용)

```
static void cbd2(poly *r, const uint8_t buf[2*KYBER_N/4])
{
    unsigned int i,j;
    uint32_t t,d;
    int16_t a,b;

    for(i=0;i<KYBER_N/8;i++) {
        t = load32_littleendian(buf+4*i);
        d = t & 0x55555555;
        d += (t>>1) & 0x55555555;

        for(j=0;j<8;j++) {
            a = (d >> (4*j+0)) & 0x3;
            b = (d >> (4*j+2)) & 0x3;
            r->coeffs[8*i+j] = a - b;
        }
    }
}
```

```
int poly_reject_cs(int64_t in[][N])
{
    int tmp1;
    int tmp2;

    int result = 1;

    for (int i = 0; i < _M; i++)
    {
        for (int j = 0; j < _N; j++)
        {
            if(((Bound1 - in[i][j]) & (in[i][j] - (Q - Bound1))) >> 63) return 0;
        }
    }

    return 1;
}
```

Sampler	Fast	Short output s	NTRU-friendly
Klein [Kle00]	No	Yes	Yes
Peikert [Pei10]	Yes	No	Yes
Micciancio-Peikert [MP12]	Yes	Yes	No
Ducas-Prest [DP16]	Yes	Yes	Yes

FALCON: Fast Fourier Sampling

Classic McEliece (1 / 2)

- 가장 오래된 코드 기반 암호 기반 alternate candidate: Classic McEliece
 - 기본적인 구조는 1978년 McEliece 암호시스템에 기반
 - Germany Federal Office for Information Security에서는 **Classic McEliece**와 **FrodoKEM**을 long-term security로 추천
- 하지만 공개키의 크기가 **256KB~1.3MB** 까지 소요
- 따라서 **192KB RAM**을 가진 **M4** 상에서는 구현이 불가능
 - 해당 구현에서 **1MB ROM**에 공개키 저장
 - 공개키를 줄이기 위해 Streaming 방식으로도 구현 가능

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

Classic McEliece (2 / 2)

- McEliece의 variant인 Niederreiter에 기반함
- 각 비밀키는 n -길이의 바이너리 Goppa code로 표현
 - $\Gamma_2(g, \alpha_1, \dots, \alpha_n) = \{c \in F_2^n \mid \sum_i \frac{c_i}{x - \alpha_i} \equiv 0 \pmod{g}\}$
 - $g \in F_{2^m}[x]$: degree- t monic irreducible polynomial
 - $\alpha_1, \dots, \alpha_n$: F_{2^m} 상의 원소, $g(\alpha_i) \neq 0$ for all i
 - $\Gamma_2(g, \alpha_1, \dots, \alpha_n)$ 의 dimension은 $k = n - mt$, t 개의 에러 수정
 - 공개키: 코드의 systematic parity-check matrix 표현
- Encapsulation의 동작 방식
 - Ciphertext: error vector의 syndrome 표현
 - Session key: error vector의 해시값
 - Secret 키를 통해 error vector를 디코딩 \rightarrow session 키 도출

Classic McEliece (공개키 최적화, 1 / 2)

• 공개키 생성: 코드에 대한 parity-check matrix 생성

- $\hat{H} = (M|T) \in F_2^{(n-K) \times n}$, $M \in F_2^{(n-k) \times (n-k)}$, $T \in F_2^{(n-k) \times k}$
- 알고리즘을 통해 \hat{H} 을 systematic form으로 치환
 - $(I_{n-k}|\hat{T}) = M^{-1}\hat{H}$, M 은 invertible (29%의 확률)

• 즉 $\hat{H} = (M|T) \rightarrow (I|M^{-1}T)$

- LUP decomposition을 on-the-fly로 수행, T 는 필요시 생성

3rn round 구현

$$M \longrightarrow \begin{array}{|c|} \hline U \\ \hline L^{-1} \\ \hline \end{array} \begin{array}{|c|} \hline P \\ \hline \end{array} \quad pk_i \leftarrow (U^{-1}(L^{-1}(PT_i)))$$

최신 구현

$$M \longrightarrow \begin{array}{|c|} \hline U \\ \hline L \\ \hline \end{array} \begin{array}{|c|} \hline P \\ \hline \end{array} \quad \begin{array}{l} 1. U^{-1}, L^{-1}, \\ 2. M^{-1} \leftarrow U^{-1}L^{-1}P \\ 3. pk_i \leftarrow M^{-1}T_i \end{array}$$

Classic McEliece (공개키 최적화, 2 / 2)

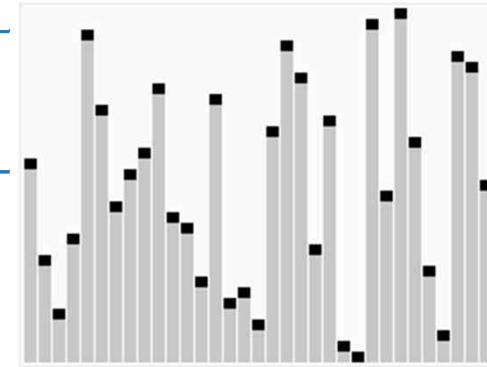
- P 값을 T_i 에 **sorting network**를 통해 적용 (**constant timing**)

$$\begin{aligned} & (0, r_0), (1, r_1), (2, r_2), \dots, (n-k-1, r_{n-k-1}) \rightarrow \\ & (r'_0, 0), (r'_1, 1), (r'_2, 2), \dots, (r'_{n-k-1}, n-k-1) \rightarrow \\ & (r'_0, A_0), (r'_1, A_1), (r'_2, A_2), \dots, (r'_{n-k-1}, A_{n-k-1}) \rightarrow \\ & (0, A_{r'_0}), (1, A_{r'_1}), (2, A_{r'_2}), \dots, (n-k-1, A_{r'_{n-k-1}}) \end{aligned}$$

- L^{-1} 과 U^{-1} 을 **inverse matrix 없이 계산 수행**

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_0 & 1 & 0 \\ l_1 & l_2 & 1 \end{pmatrix}, \quad L^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & l_2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ l_0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Classic McEliece (Encapsulation)



Quick sort

• 에러 생성

- 에러 생성 후 해당 값을 에러의 인덱스로 설정
- 동일한 인덱스가 있는지 확인하기 위해 정렬 후 인접한 에러 비교
- 기존에는 정렬 순서를 모두 숨기는 구현
- 하지만 에러 중 t 개의 에러의 위치만 안전하게 보호하면 됨 (정렬은 랜덤한 순서만 유출)
- 따라서 quick sort를 통해 정렬함으로써 성능 향상 도출

parameter set	quicksort	sorting network
mceliece348864*	74 688	111 577
mceliece460896*	172 711	259 354

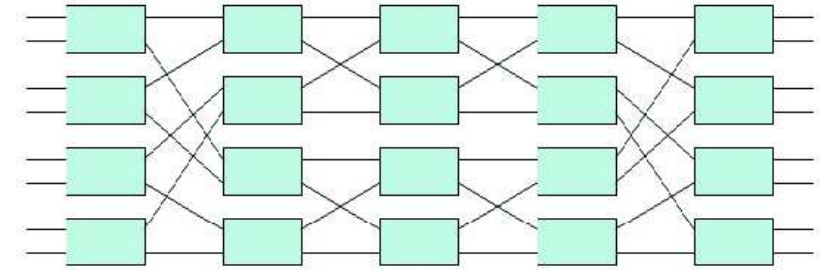
• T_e 연산

- 비트슬라이싱 기법으로 블록 단위로 묶어서 곱셈 연산 수행

Classic McEliece (Decapsulation)

• 연산 구성

- Berlekamp-Massey: field 상에서 minimal polynomial을 가지는 recurrent sequence 도출
- FFT: root finding
- Transposed FFT: syndrome computation
- Benes Network: permutation



• M4 상의 최적화

- 비트슬라이싱과 더불어 Radix-16 구현 수행 (UMULL 및 sparse form 사용)

$$(a_0 \cdot 2^0 + a_1 \cdot 2^4 + a_2 \cdot 2^8 + \dots + a_7 \cdot 2^{28}) \cdot (b_0 \cdot 2^0 + b_1 \cdot 2^4 + b_2 \cdot 2^8 + \dots + b_7 \cdot 2^{28}) \\ = a_0 \cdot b_0 \cdot 2^0 + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 2^4 + (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2) \cdot 2^8 + \dots + (a_7 \cdot b_7) \cdot 2^{56}$$

Classic McEliece on M4

scheme (implementation)	level	key generation	encapsulation	decapsulation
frodokem640aes (m4)	1	48 348 105	47 130 922	46 594 383
kyber512 (m4)	1	463 343	566 744	525 141
kyber768 (m4)	3	763 979	923 856	862 176
lightsaber (m4f)	1	361 687	513 581	498 590
saber (m4f)	3	654 407	862 856	835 122
ntruhs2048509 (m4f)	1	79 658 656	564 411	537 473
ntruhs2048677 (m4f)	3	143 734 184	821 524	815 516
sikep434 (m4)	1	48 264 129	78 911 465	84 276 911
sikep610 (m4)	3	119 480 622	219 632 058	221 029 700

- 유사하게 높은 보안 강도를 가지는 FrodoKEM 대비 **encapsulation과 decapsulation이 각각 79배 그리고 17배 빠름**

- 문제는 키크기

	key gen. (impl. a)	key gen. (impl. b)	encap.	decap.
mceliece348864f	115 588 + 61 440	114 584	1412	18 492
mceliece348864	115 588 + 61 440	114 624		
mceliece460896*			1996	34 956
mceliece6688128*				35 708
mceliece8192128*				36 200

BIKE on Intel Haswell & ARM Cortex-M4

- **BIKE**

- NIST PQC fourth round alternate candidate
- Code-based KEM based on “moderate-density parity-check” (MDPC) code
- Level-1 (1.5KB), level-3 (3KB), level-5 (5KB)
- 연산 속도가 상대적으로 느림 (저장공간과 trade-off)

instance	encap	decap	level	platform
bikel1	195000	1280000	1	Ice Lake
mceliece348864	43832	134184	1	Haswell

BIKE on Intel Haswell & ARM Cortex-M4

- **Haswell cycles:**

	key gen.	encap.	decap.	imple.
bike11	833968	131276	2636108	avx2
	688372	124892	1900908	our code
bike13	2515016	313976	9040604	avx2
	1857256	283932	6010004	our code

- **M4 cycles:**

	key gen.	encap.	decap.	imple.
bike11	65414337	4824059	114592442	portable
	24935033	3253379	49911673	our code
bike13	212999628	15041356	374777003	portable
	59820502	8376212	139234176	our code

BIKE on Intel Haswell & ARM Cortex-M4

KeyGen: $() \mapsto (h_0, h_1, \sigma), h$ Output: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M}, h \in \mathcal{R}$ 1: $h_0, h_1 \xleftarrow{\$} \mathcal{H}_w$ 2: $h = h_1 \cdot h_0^{-1}$ 3: $\sigma \xleftarrow{\$} \mathcal{M}$	Encaps: $h \mapsto K, c$ Input: $h \in \mathcal{R}$ Output: $K \in \mathcal{K}, c \in \mathcal{R} \times \mathcal{M}$ 1: $m \xleftarrow{\$} \mathcal{M}$ 2: $e_0, e_1 \leftarrow \mathbf{H}(m)$ 3: $c = (e_0 + e_1 \cdot h, m \oplus \mathbf{L}(e_0, e_1))$ 4: $K \leftarrow \mathbf{K}(m, c)$
Decaps: $(h_0, h_1, \sigma), c \mapsto K$ Input: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M}, c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$ Output: $K \in \mathcal{K}$ 1: $e' \leftarrow \text{decoder}(c_0 h_0, h_0, h_1) \quad \triangleright e' \in \mathcal{R}^2 \cup \{\perp\}$ 2: $m' = c_1 \oplus \mathbf{L}(e')$ \triangleright with the convention $\perp = (0, 0)$ 3: if $e' = \mathbf{H}(m')$ then $K \leftarrow \mathbf{K}(m', c)$, else $K \leftarrow \mathbf{K}(\sigma, c)$	

- $R = F_2[x] / (x^r - 1)$: **additions, multiplications, inversions**
 - Itoh-Tsuji 연산은 곱셈에서 bottleneck이 발생
- e_1, h_0, h_1 : **low-weight operands**
- $c_0 h_0 = e_0 h_0 + e_1 h_1$ 는 **matrix-vector 곱셈 연산으로 볼 수 있음**

BIKE on Intel Haswell & ARM Cortex-M4

- **Optimizations**

- **Multiplications in R_z (circular shift on f)**

- Constant-time shifts w/ SEL instruction (M4)
 - Constant-time shifts w/ matrix transposition (HW)
 - Adding $y^i f$ w/ Boyer-Peralta algorithm (HW and M4)

- **Multiplications in R (polynomial multiplication w/ modulo $x^r - 1$)**

- Usage of Bernstein's 5-way recursive algorithm (HW)
 - Usage of Frobenius additive FFT (M4)

BIKE on Intel Haswell & ARM Cortex-M4

- **Logical shifts w/ SEL instruction (M4)**
 - 변수 f' 를 s -비트 만큼 shift하는 연산 ($0 \leq s < r$)
 - $s = s^{(1)} + s^{(0)}$, $s^{(0)} = s \bmod 32$
 - $s^{(1)}$ 에 대한 shift 연산은 Barrel shifter를 사용
- Conditional shift (by BIKE team)
 - $w[i] = (w[i] \& mask) \mid (w[i + dx] \& mask)$;
- **SEL 명령어를 통해 두개의 워드 중에서 선택**
- $s^{(0)} < 32$ -비트 인 경우에는 logical instruction 혹은 multiplication-and-accumulate instruction (UMLAL) 사용

SEL

Select bytes from each operand according to the state of the APSR GE flags.

Syntax

`SEL { cond } { Rd }, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the first operand.

Rm

is the register holding the second operand.

Operation

The `SEL` instruction selects bytes from `Rn` or `Rm` according to the APSR GE flags:

- If GE[0] is set, `Rd[7:0]` come from `Rn[7:0]`, otherwise from `Rm[7:0]`.
- If GE[1] is set, `Rd[15:8]` come from `Rn[15:8]`, otherwise from `Rm[15:8]`.
- If GE[2] is set, `Rd[23:16]` come from `Rn[23:16]`, otherwise from `Rm[23:16]`.
- If GE[3] is set, `Rd[31:24]` come from `Rn[31:24]`, otherwise from `Rm[31:24]`.

BIKE on Intel Haswell & ARM Cortex-M4

- **Logical shifts w/ matrix transposition (HW)**

- 다음과 같이 정의: $s = s^{(1)} + s^{(0)}$, where $s^{(0)} = s \bmod b$.

- f' 를 matrix 형식으로 표현

$$M^{(0)} = \begin{pmatrix} f'_0 & f'_1 & \dots & f'_{b-1} \\ f'_b & f'_{b+1} & \dots & f'_{2b-1} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}.$$

- Shifting f' by $s^{(0)}$ -비트 \approx shifting the rows

- Shifting f' by $s^{(1)}$ -비트 = shifting the columns

- 따라서 column-major로 f' 저장

- 각 s -비트에 대한 shift는 column단위로 $s^{(1)}$ -비트 shift후 matrix transpose 이후에 f' 를 $s^{(0)}$ -비트만큼 shift 수행

- Matrix transposition은 빠르게 수행가능 → partition the matrix into 4 pieces; recursively swap upper-right and bottom left

- Haswell 구현에서는 $b=256$ 으로 설정

- M4 상에서는 동작하지 않음 (아마도 레지스터 문제일듯)

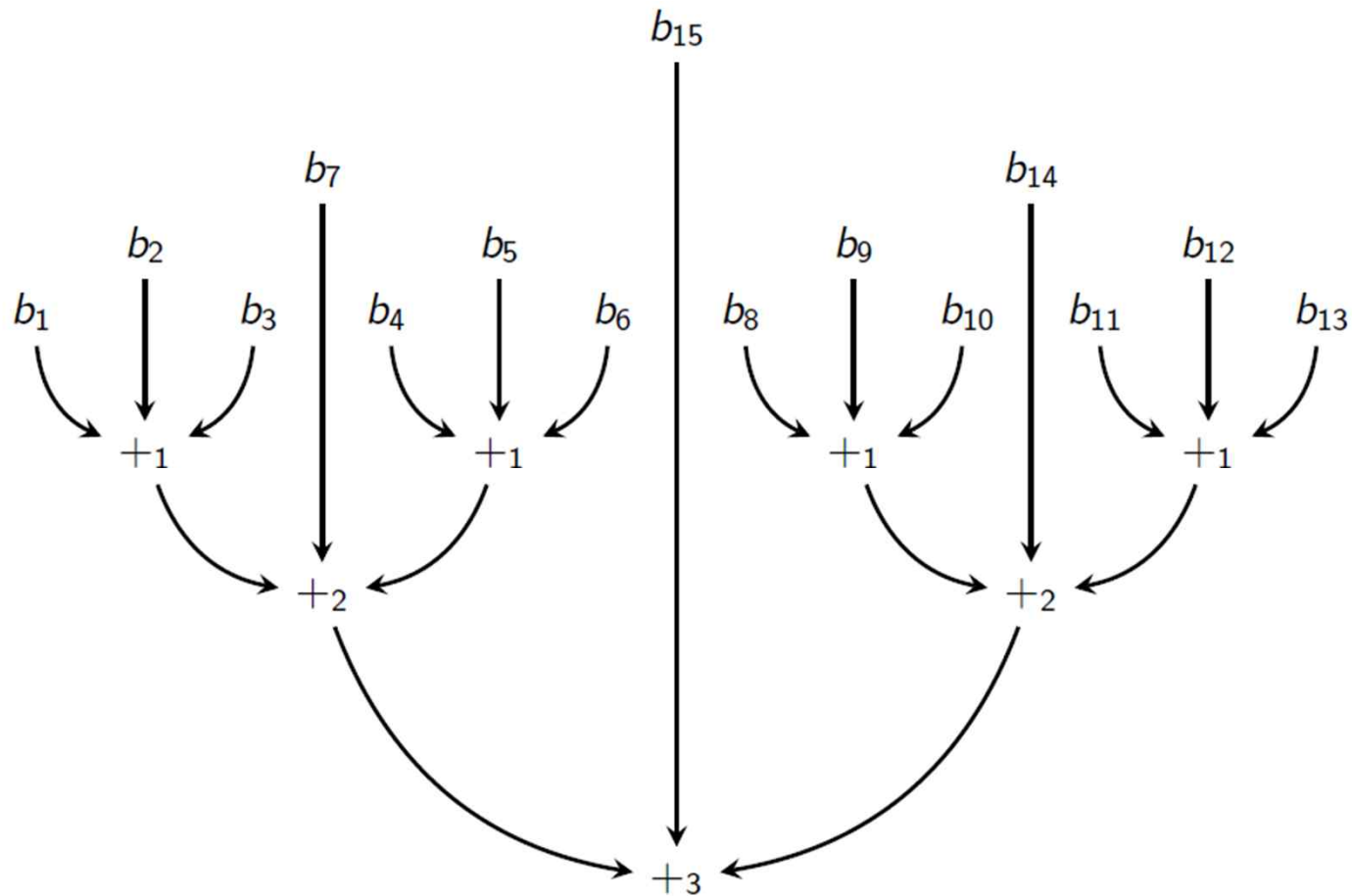
BIKE on Intel Haswell & ARM Cortex-M4

• Boyar-Parelda algorithm

- 필요한 연산: $\sum_{i \in l} f \cdot y^i \in R_z = Z[y]/(y^r - 1)$.
- Length가 $|l|$ 인 r 벡터 상에서 1의 숫자를 카운팅
- 이전 연구에서는 half adder 사용
- 본 논문에서는 full adder 사용

	$ l $	naive-1	naive-2	BP
bikel1	71	923	676	326
bikel3	103	1339	1092	484
bikel5	137	2055	1553	655

BIKE on Intel Haswell & ARM Cortex-M4



BIKE on Intel Haswell & ARM Cortex-M4

- 기존 BIKE team의 구현 (Karatsuba; $O(n^{bg} z^3)$)
 - $F = F_0 + F_1x^n$ 그리고 $G = G_0 + G_1x^n$
 - $FG = F_0G_0 + F_1G_1x^{2n} + ((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1)x^n$
- 제안하는 구현 기법 (Bernstein's 5-way Recursive Algorithm $O(n^{bg} z^5)$)
 - $F = F_0 + F_1z + F_2z^2$ 그리고 $G = G_0 + G_1z + G_2z^2$; $z = x^n$
 - $H = FG$

$$H(0) = F_0 \cdot G_0,$$

$$H(1) = (F_0 + F_1 + F_2) \cdot (G_0 + G_1 + G_2),$$

$$H(x) = (F_0 + F_1x + F_2x^2) \cdot (G_0 + G_1x + G_2x^2),$$

$$H(x+1) = ((F_0 + F_1 + F_2) + F_1x + F_2x^2) \cdot ((G_0 + G_1 + G_2) + G_1x + G_2x^2),$$

$$H(\infty) = F_2 \cdot G_2.$$

$$H = U + H(\infty)(z^4 + z) + \frac{U + V + H(\infty)(x^4 + x)}{x^2 + x}(z^2 + z)$$

$$U = H(0) + (H(0) + H(1))z \text{ and } V = H(x) + (H(x) + H(x+1))(x+z) .$$

BIKE on Intel Haswell & ARM Cortex-M4

- Consider F_2 를 F_{2^m} 으로 간주하고 additive FFT를 수행

$$\begin{aligned}\mathbb{F}_{2^2} &= \mathbb{F}_2[x_1]/(x_1^2 + x_1 + 1), \\ \mathbb{F}_{2^4} &= \mathbb{F}_{2^2}[x_2]/(x_2^2 + x_2 + x_1), \\ \mathbb{F}_{2^8} &= \mathbb{F}_{2^4}[x_4]/(x_4^2 + x_4 + x_2x_1), \\ \mathbb{F}_{2^{16}} &= \mathbb{F}_{2^8}[x_8]/(x_8^2 + x_8 + x_4x_2x_1), \\ \mathbb{F}_{2^{32}} &= \mathbb{F}_{2^{16}}[x_{16}]/(x_{16}^2 + x_{16} + x_8x_4x_2x_1).\end{aligned}$$

- AFFT (Evaluation) \rightarrow Pointwise multiplication \rightarrow IFFT

Algorithm 2 FAFFT-based Polynomial Multiplication

Parameters: n : maximum length of output polynomial

```

1: procedure FAFFT( $f(x) \in \mathbb{F}_2[x]_{<n}, \Sigma$ )
2:    $(f_0, f_1, \dots, f_{n-1}) \in \mathbb{F}_2^n \leftarrow \text{RadixConversions}(f(x))$ 
3:    $(f'_0, \dots, f'_{n'-1}) \in \mathbb{F}_{2^{n'}}^n \leftarrow \text{Encode}((f_0, f_1, \dots, f_{n-1}), \Sigma)$ 
4:    $(\hat{f}_0, \dots, \hat{f}_{n'-1}) \in \mathbb{F}_{2^{n'}}^{n'} \leftarrow \text{Butterflies}((f'_0, \dots, f'_{n'-1}), \Sigma)$ 
5:   return  $(\hat{f}_0, \dots, \hat{f}_{n'-1})$ 
6: end procedure

7: procedure FAFFT $^{-1}((\hat{f}_0, \dots, \hat{f}_{n'-1}) \in \mathbb{F}_{2^{n'}}^{n'}, \Sigma)$ 
8:    $(f'_0, \dots, f'_{n'-1}) \in \mathbb{F}_{2^{n'}}^{n'} \leftarrow \text{Butterflies}^{-1}((\hat{f}_0, \dots, \hat{f}_{n'-1}), \Sigma)$ 
9:    $(f_0, \dots, f_{n-1}) \in \mathbb{F}_2^n \leftarrow \text{Encode}^{-1}((f'_0, f'_1, \dots, f'_{n'-1}), \Sigma)$ 
10:   $f(x) \in \mathbb{F}_2[x]_{<n} \leftarrow \text{RadixConversions}^{-1}(f_0, f_1, \dots, f_{n-1})$ 
11:  return  $f(x)$ 
12: end procedure

13: procedure BITPOLYMUL( $a(x) \in \mathbb{F}_2[x]_{<\frac{n}{2}}, b(x) \in \mathbb{F}_2[x]_{<\frac{n}{2}}$ )
14:    $(\hat{a}_0, \dots, \hat{a}_{n'-1}) \in \mathbb{F}_{2^{n'}}^{n'} \leftarrow \text{FAFFT}(a(x), \Sigma)$ 
15:    $(\hat{b}_0, \dots, \hat{b}_{n'-1}) \in \mathbb{F}_{2^{n'}}^{n'} \leftarrow \text{FAFFT}(b(x), \Sigma)$ 
16:    $(\hat{c}_0, \dots, \hat{c}_{n'-1}) \in \mathbb{F}_{2^{n'}}^{n'} \leftarrow (\hat{a}_0 \cdot \hat{b}_0, \dots, \hat{a}_{n'-1} \cdot \hat{b}_{n'-1})$   $\triangleright$  pointwise multiplication
17:    $c(x) \in \mathbb{F}_2[x]_{<n} \leftarrow \text{FAFFT}^{-1}((\hat{c}_0, \dots, \hat{c}_{n'-1}), \Sigma)$ 
18:   return  $c(x)$ 
19: end procedure

```

Evaluation의 경우 Frobenius additive FFT를 만족하는 조건을 찾아 evaluation points의 수를 최적화

Pointwise multiplication의 경우
Recursive Karatsuba 곱셈 연산 수행

SPHINCS (<https://sphincs.cr.yp.to/>)

- NIST PQC 표준화 확정
- Hash-based Signature Scheme
- Stateless
- 128-bit post-quantum security
- **Size**
 - Public Key: 1KB
 - Secret Key: 1KB
 - Signature: 41KB



SPHINCS: practical stateless hash-based signatures

[Introduction](#)

[Papers](#)

[Software](#)

SPHINCS-256 is a high-security post-quantum stateless hash-based signature scheme that signs hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU. Signatures are 41 KB, public keys are 1 KB, and private keys are 1 KB. SPHINCS-256 is designed to provide long-term 2^{128} security even against attackers equipped with quantum computers. Unlike most hash-based signature schemes, SPHINCS-256 is stateless, allowing it to be a drop-in replacement for current signature schemes.

Special note to law-enforcement agents: The word "state" is a technical term in cryptography. Typical hash-based signature schemes need to record information, called "state", after every signature. Google's Adam Langley refers to this as a "[huge foot-cannon](#)" from a security perspective. By saying "eliminate the state" we are advocating a security improvement, namely adopting signature schemes that do not need to record information after every signature. We are not talking about eliminating other types of states. We love most states, especially yours! Also, "hash" is another technical term and has nothing to do with cannabis.

Contributors (alphabetical order)

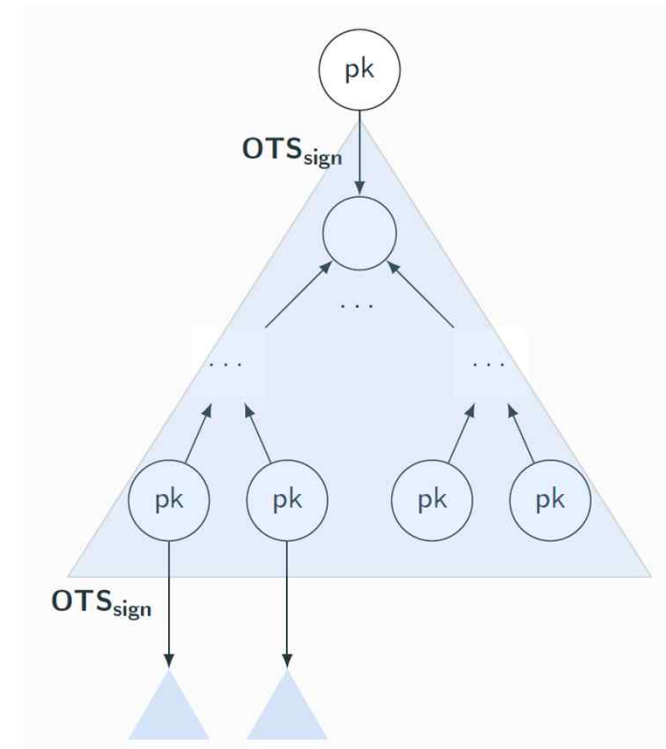
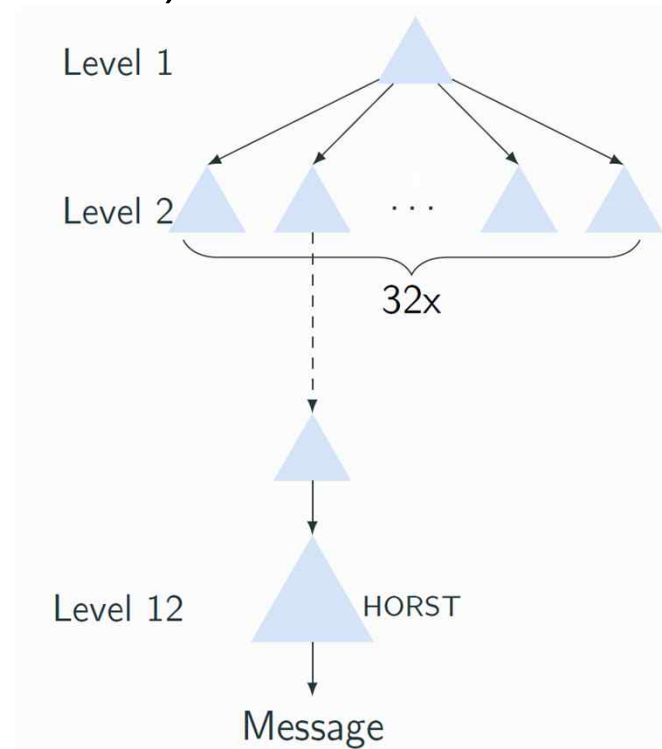
- [Daniel J. Bernstein](#), University of Illinois at Chicago, USA
- [Daira Hopwood](#), Jacaranda Software, UK
- [Andreas Hülsing](#), Technische Universiteit Eindhoven, Netherlands
- [Tanja Lange](#), Technische Universiteit Eindhoven, Netherlands
- [Ruben Niederhagen](#), Technische Universiteit Eindhoven, Netherlands
- Louiza Papachristodoulou, Radboud Universiteit Nijmegen, Netherlands
- Michael Schneider, Deutsche Bank, Germany
- [Peter Schwabe](#), Radboud Universiteit Nijmegen, Netherlands
- Zooko Wilcox-O'Hearn, Least Authority, USA

Version: This is version 2017.12.05 of the Introduction web page.

SPHINCS

- 주 연산자

- One-time Signature (WOTS)
- Few-time Signature (HORST)
- Merkle-Tree



SPHINCS - WOTS

서명

$$\sigma = (\sigma_3, \sigma_2, \sigma_1, \sigma_0) = (f(x_3), x_2, f(x_1), f^3(x_0)) = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \in \{0, 1\}^{(3,4)}$$

1 0 1 3

$x \mapsto x + 1 \bmod 8$, 임의의 해시함수

$d = 01 || 00$ 메시지
1 0

$$c = (4 - 1) + (4 - 0) = 7$$

$c = 01 || 11$. 메시지에서 계산된 값
1 3

검증

$$(f^2(\sigma_3), f^3(\sigma_2), f^2(\sigma_1), \sigma_0) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in \{0, 1\}^{(3,4)}$$

2 3 2 0

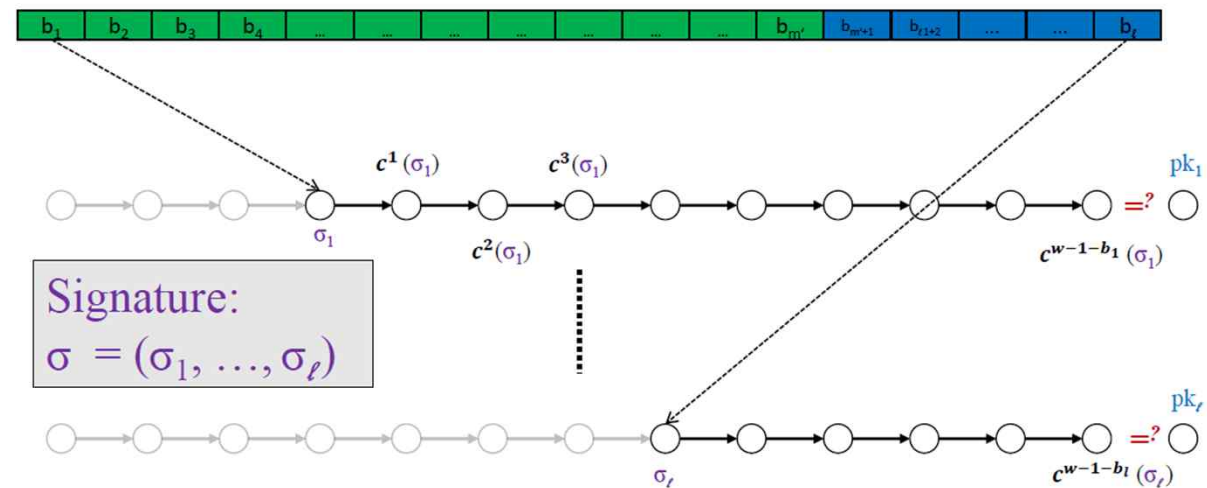
키 길이가 줄이지만 연산횟수는 증가

비밀키

$$X = (x_3, x_2, x_1, x_0) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \in \{0, 1\}^{(3,4)}$$

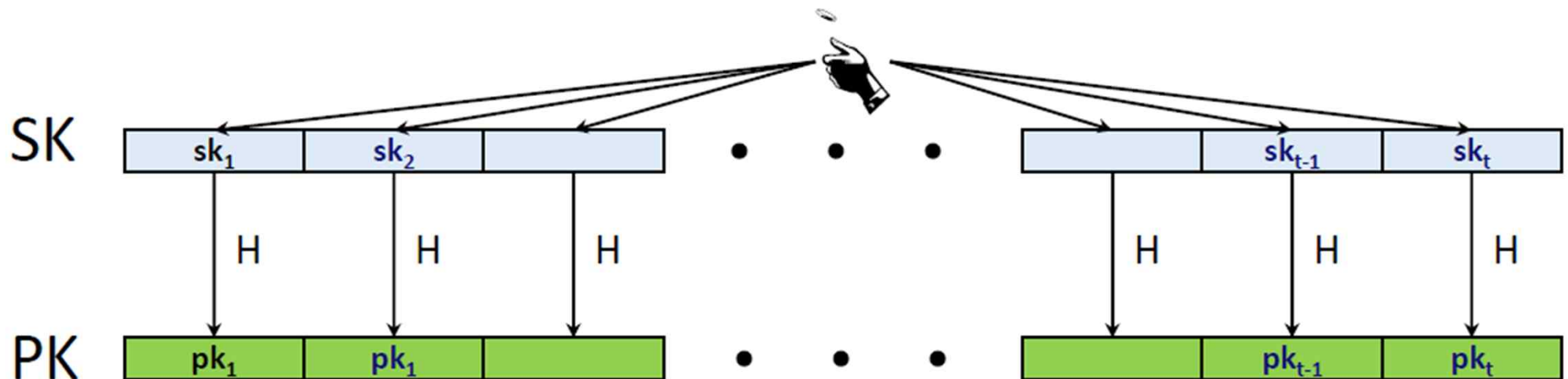
공개키

$$Y = (y_3, y_2, y_1, y_0) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in \{0, 1\}^{(3,4)}.$$



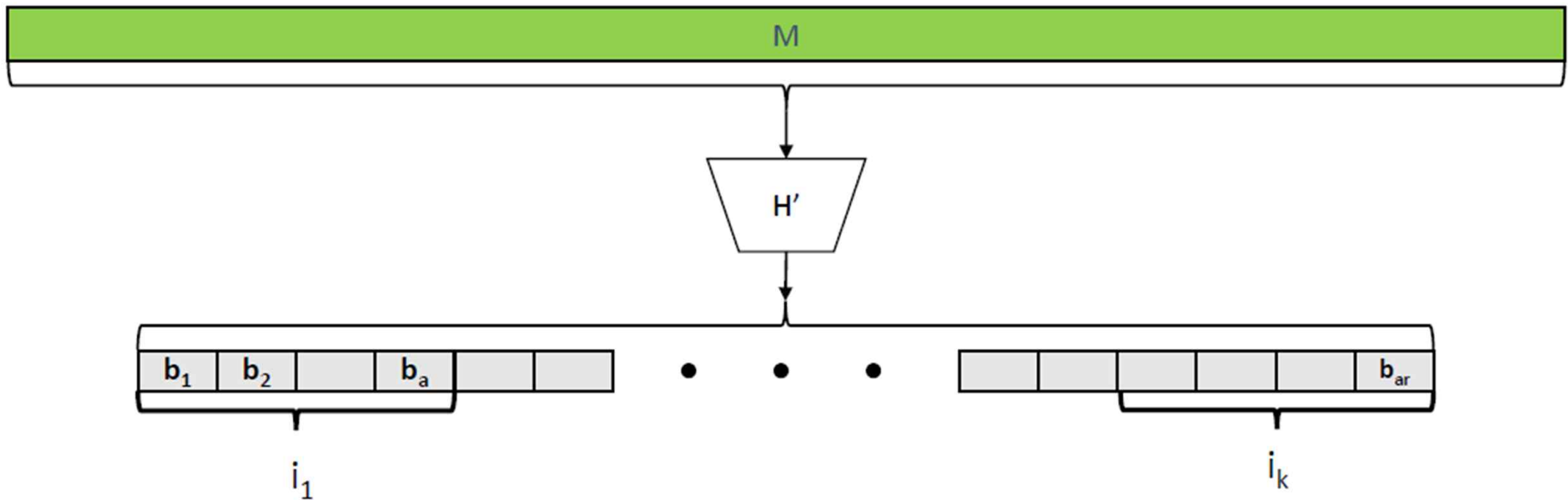
SPHINCS – HORS (Key Generation)

- Master secret key로부터 $t = 2^a$ 개의 비밀키 sk 를 생성
- 비밀키에 n -비트 해시 함수 H 를 적용하여 대응하는 pk 를 생성 ($n=256$)



SPHINCS – HORS (Mapping function)

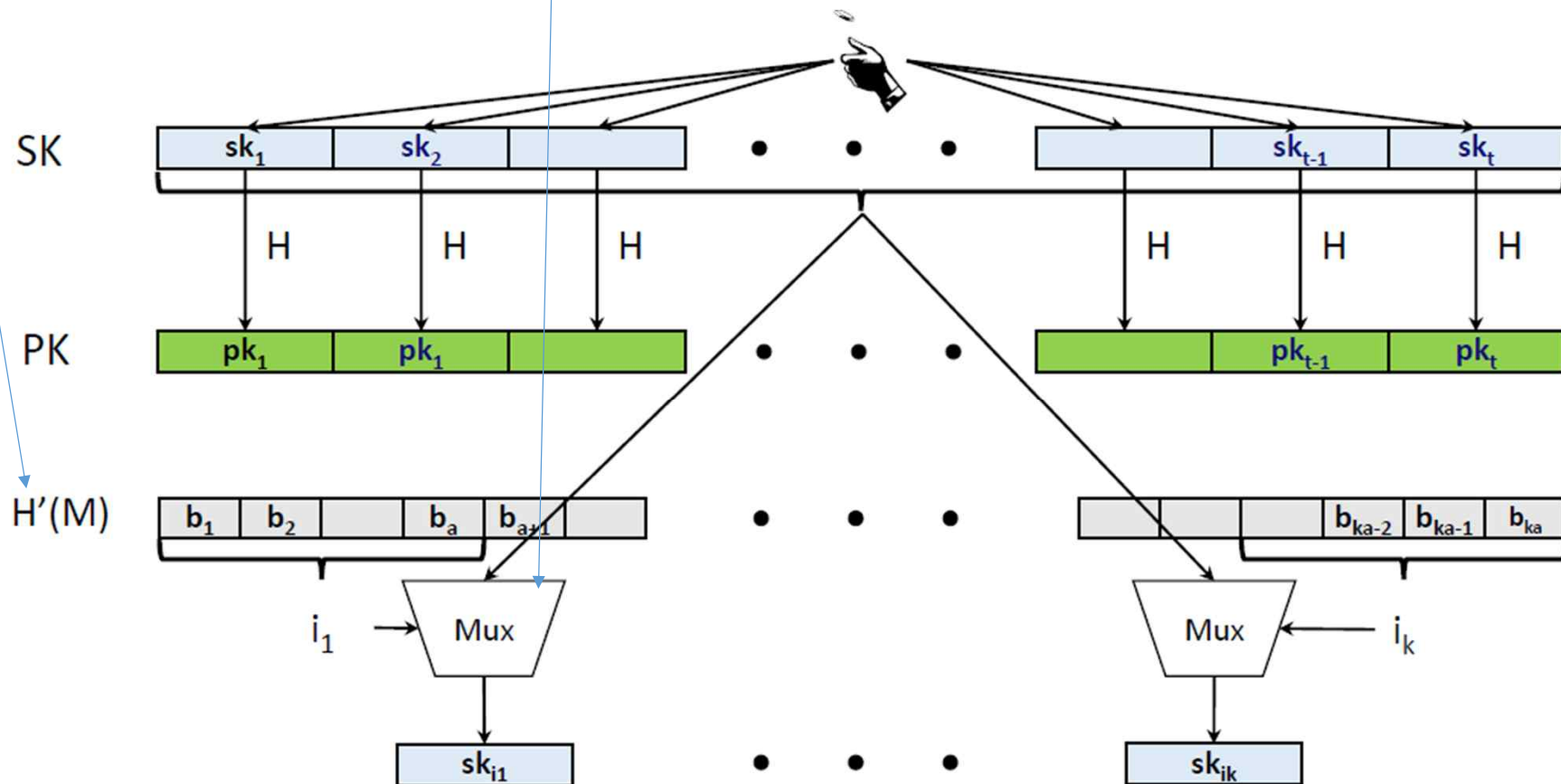
- Message의 길이는 m 이며 다음 특성을 가짐 ($m=ka$; 예시: $a=16, k=32$)
- H' 는 512-비트 출력값 생성



a -비트 크기를 가지는 블록

SPHINCS – HORS (Signing)

- $H'(M)$ 을 통해 얻어진 출력값을 MUX의 입력으로 넣어서 비밀키 결정
- 각 서명은 t 개의 비밀키 가운데 k 개를 공개하는 것 (동일한 서명이 나올 확률이 일정 확률 이하)



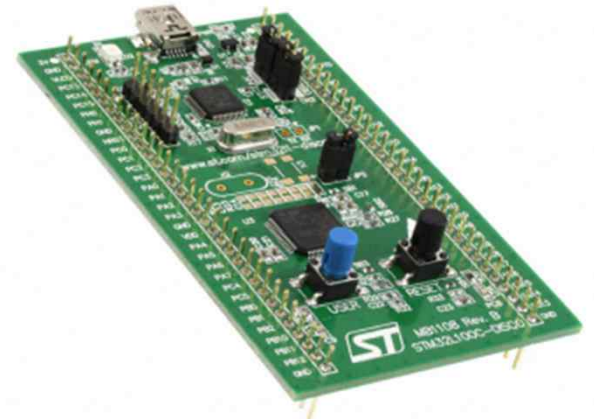
SPHINCS – 메모리 최적구현

- **HORST (Few Time Signatures)**

- 16-layer Merkle tree ($2^{16} = 65,536$)
- Goal: 32 authentication paths, root node
- Path는 (deterministically 선택된) 무작위 leaf에서 시작
- **Tree 구성에는 약 2MB RAM이 사용**

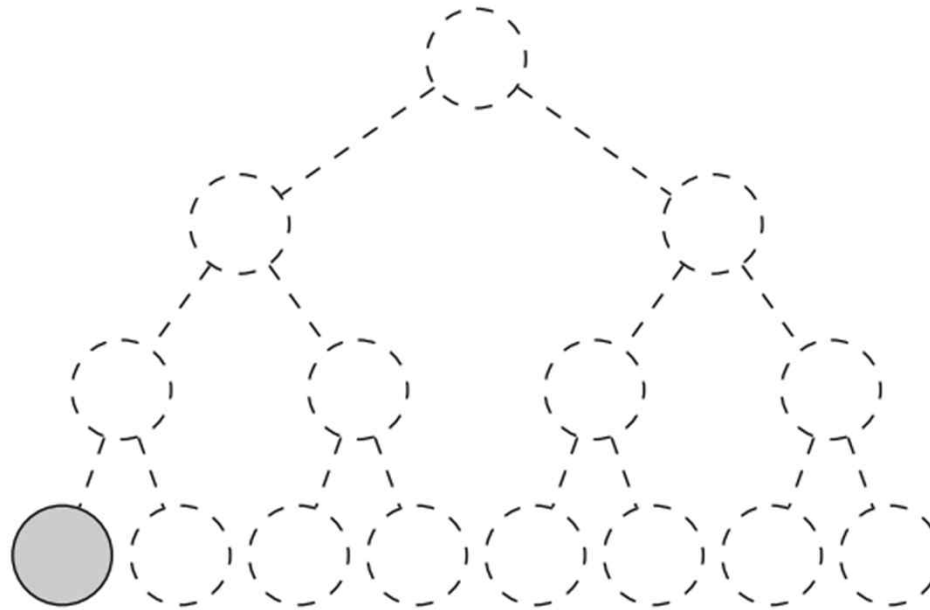
- **구현 플랫폼**

- STM32L100C 개발용 보드
- Cortex-M3, ARMv7-M
- 32MHz, 32-비트 구조
- 256KB Flash, **16KB RAM**



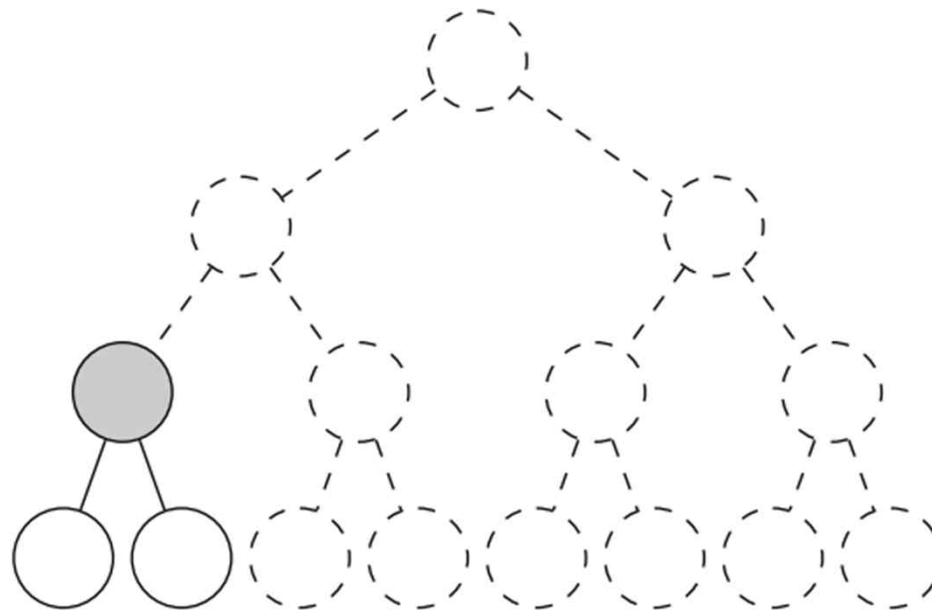
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treeshash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16$ nodes (아래 예시 $bg(8) = 3$)



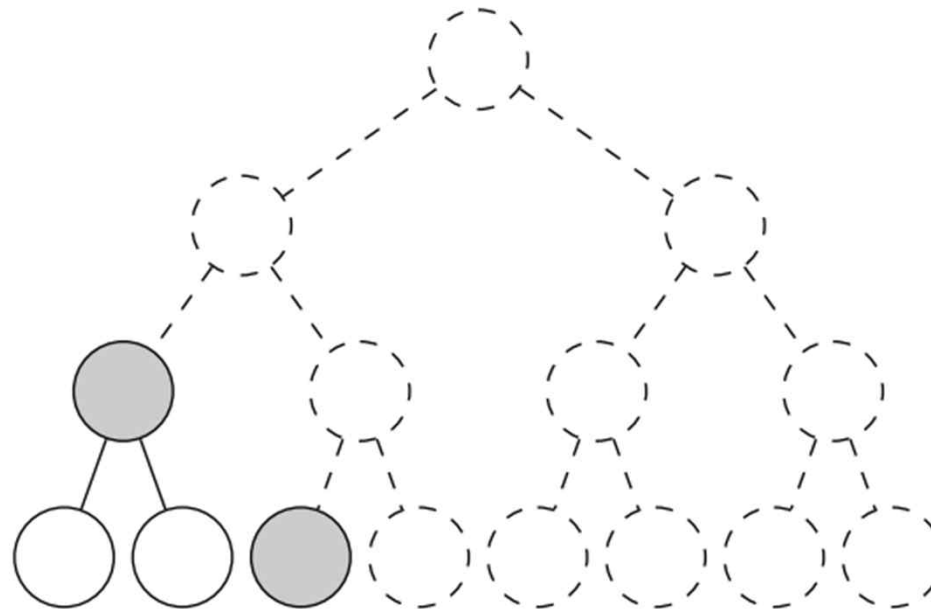
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treehash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16 \text{ nodes}$ (아래 예시 $bg(8) = 3$)



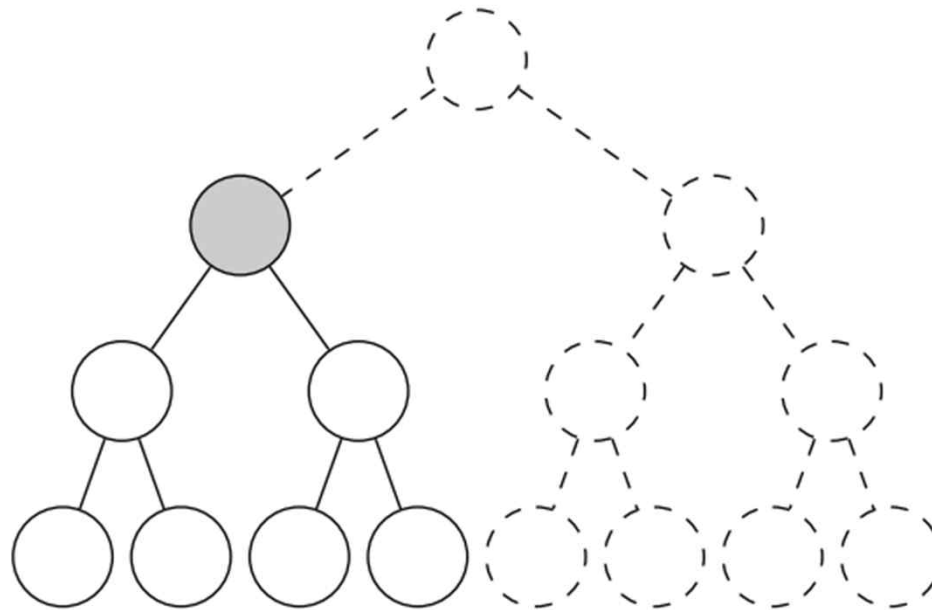
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treehash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16 \text{ nodes}$ (아래 예시 $bg(8) = 3$)



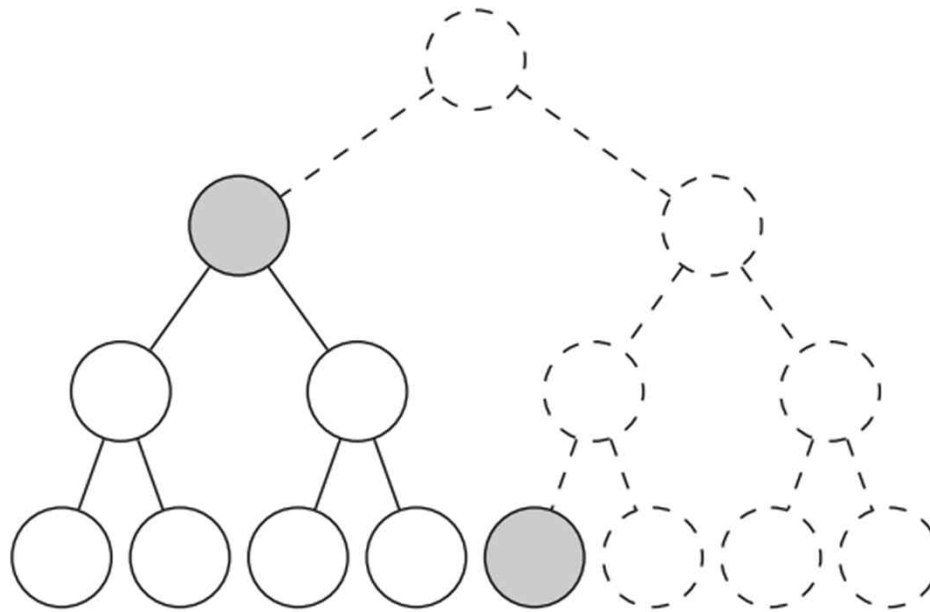
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treeshash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16$ nodes (아래 예시 $bg(8) = 3$)



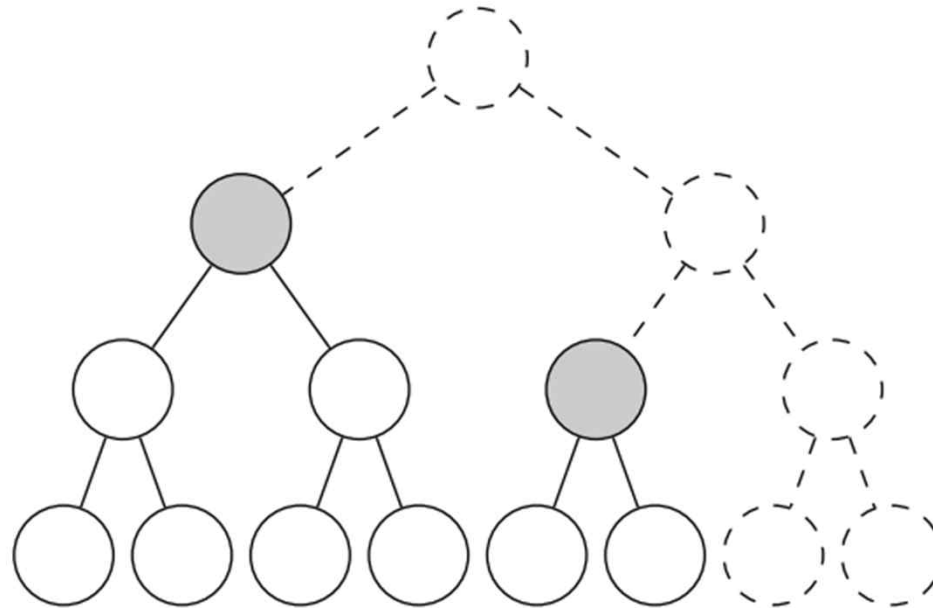
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treeshash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16$ nodes (아래 예시 $bg(8) = 3$)



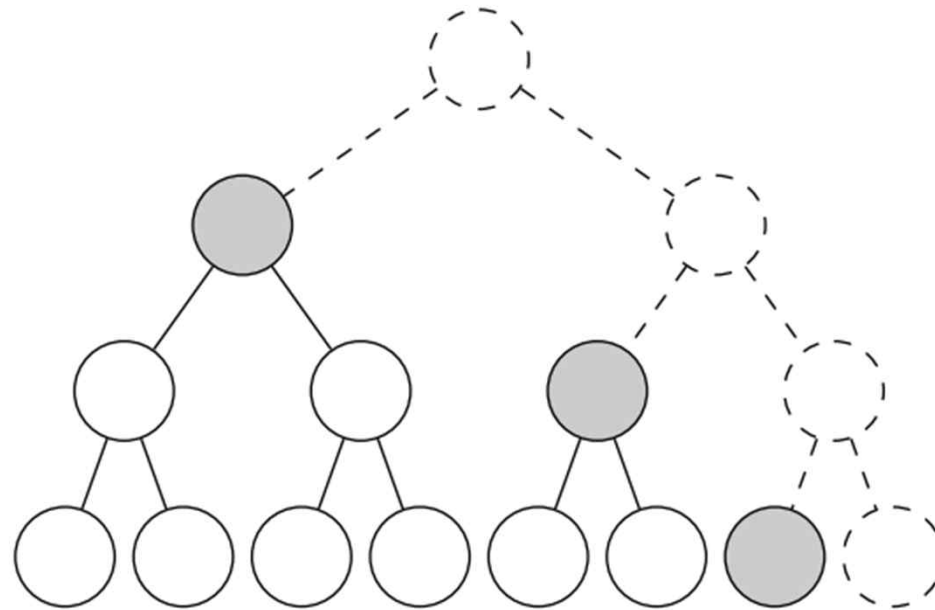
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treeshash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16$ nodes (아래 예시 $bg(8) = 3$)



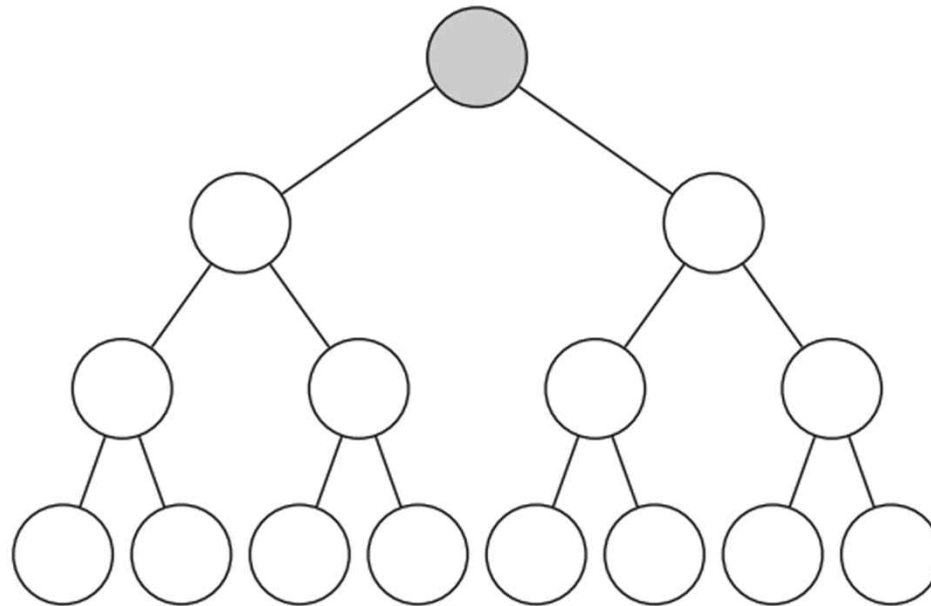
SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treehash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16 \text{ nodes}$ (아래 예시 $bg(8) = 3$)



SPHINCS – 메모리 최적구현

- **HORST tree는 매우 큼 (2MB)**
- **Treeshash: 오로지 관련된 노드들만을 저장**
 - 스택상에는 $bg(n) = 16$ nodes (아래 예시 $bg(8) = 3$)



SPHINCS – 메모리 최적구현

- **Goal: 32개의 HORST의 authentication path 구성**

- **구현 시 고려 사항**

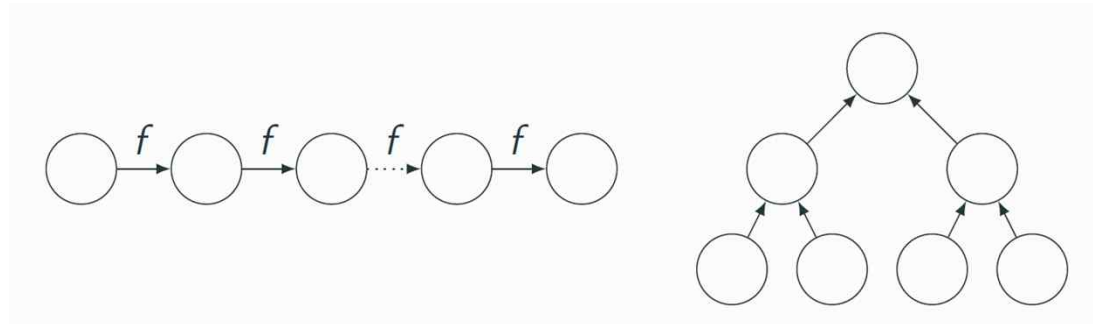
- 연관된 노드 확인 (131,071개 중에서 320개)
- 연관된 라운드 확인 (65,536개 중에서 320개)
- 연관된 라운드 중에서 연관된 노드 확인 (bitmask)
- 라운드 인덱스를 통해 마스크를 정렬
 - 포인터를 통해 유지

SPHINCS – 메모리 최적구현

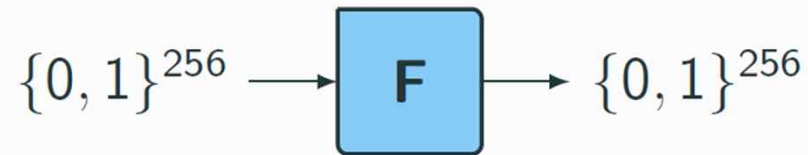
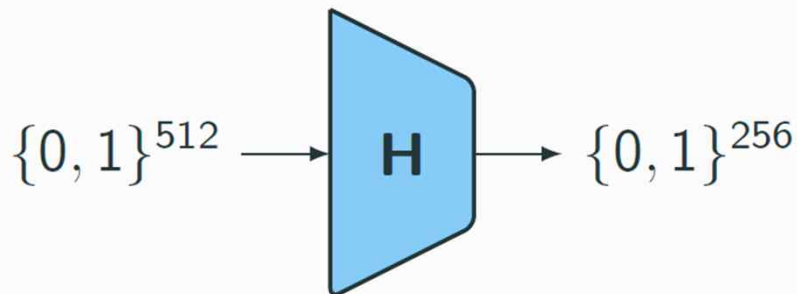
- **작은 RAM크기로 인해 서명을 저장할 수 있음**
 - 결과값을 streaming out 형식으로 출력
- **HORST는 정렬되지 않은 형식으로 결과값을 출력**
 - HOST 상에서 재정렬 및 탐색 수행
 - Tags (832 바이트; 64 + 640 + 128)
- **연산 성능**
 - 키생성: 8,857,708,189 cycles (276.80 초)
 - 서명: 19,441,021 cycles (0.61 초)
 - 검증: 4,961,447 cycles (0.16 초)

SPHINCS

- 수행되는 연산
 - 많은 해시 연산



- 하나의 서명 생성
 - $\approx 450,000$ 번의 F 연산
 - $\approx 90,000$ 번의 H 연산



SPHINCS

- SPHINCS에서 사용되는 해시 함수

- 표준

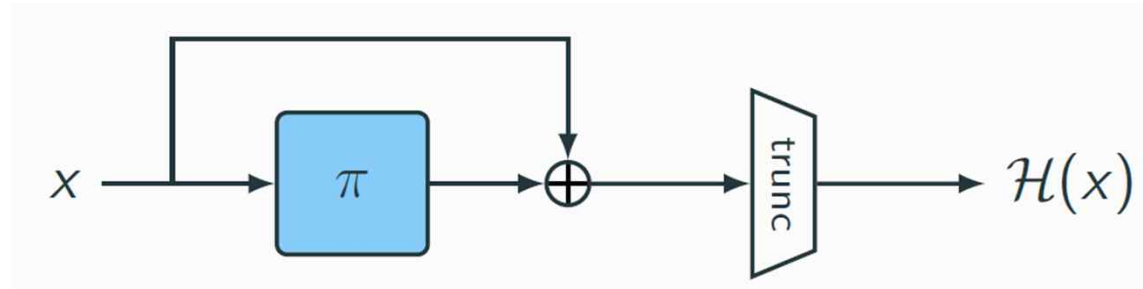
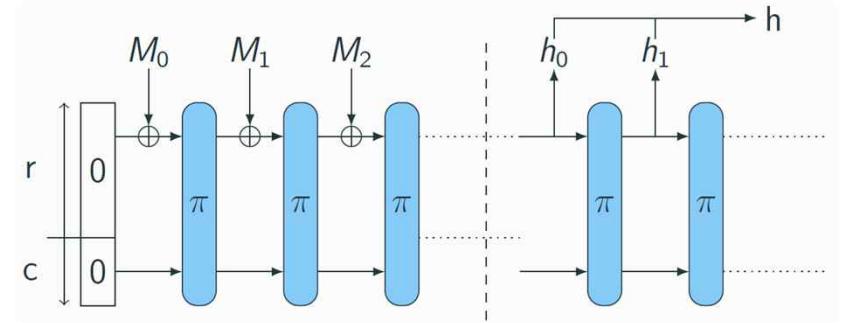
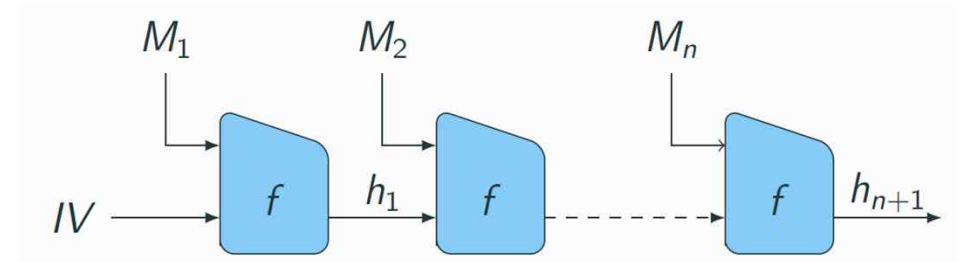
- SHA256: 512-비트 메시지 블록
 - SHA-3: 1088-비트 메시지 블록

- Keccak

- ChaCha12: SPHINCS 논문에서 제안 (SW 상에서 고속구현)

- Haraka: AES 기반 permutation (SPN 구조)

- Simpira: AES 기반 permutation (Feistel 구조)



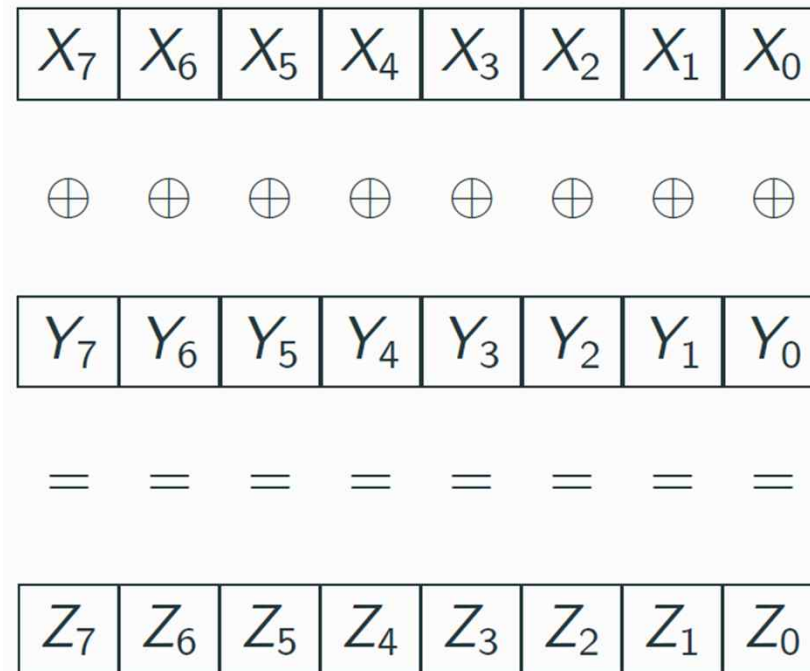
SPHINCS 고속 구현

- **SPHINCS는 저성능 장비에서는 권장되지 않음**
 - 서명크기가 RAM을 넘어섬
 - 서명연산에 소요되는 시간이 큼 (검증 시간은 상대적으로 짧음)
- **따라서 고성능 디바이스에서의 운용에 보다 적합함 (고성능 고려사항)**
 - Vectorization (AVX2, NEON, AVX-512)
 - Hardware Support (AES, SHA-2, SHA-3)
 - Pipelining

SPHINCS 고속 구현

• Vector Instructions

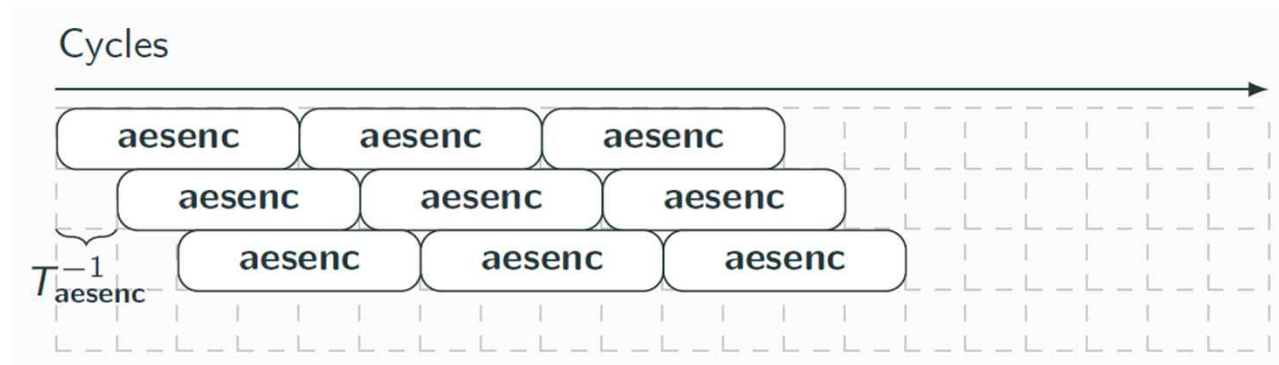
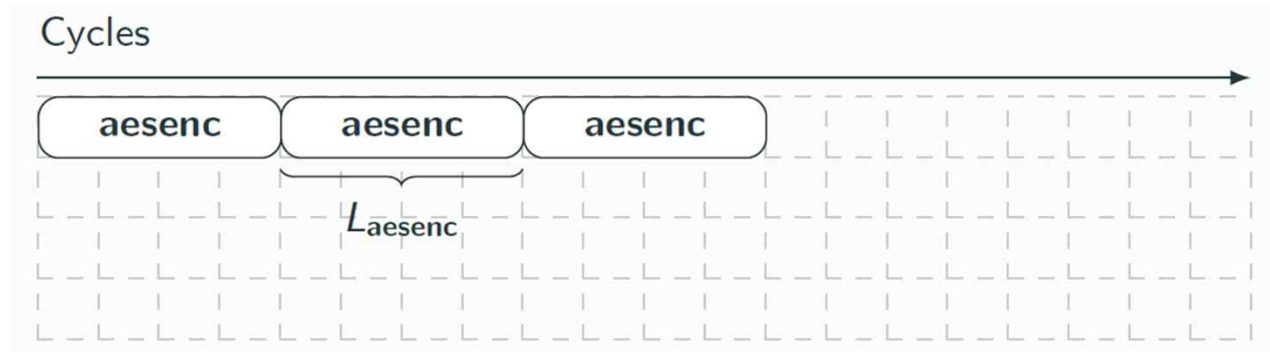
- 동일한 연산을 벡터내의 모든 인자들에 대해 수행
- 입력은 상호간에 독립적



SPHINCS 고속 구현

• Pipelining

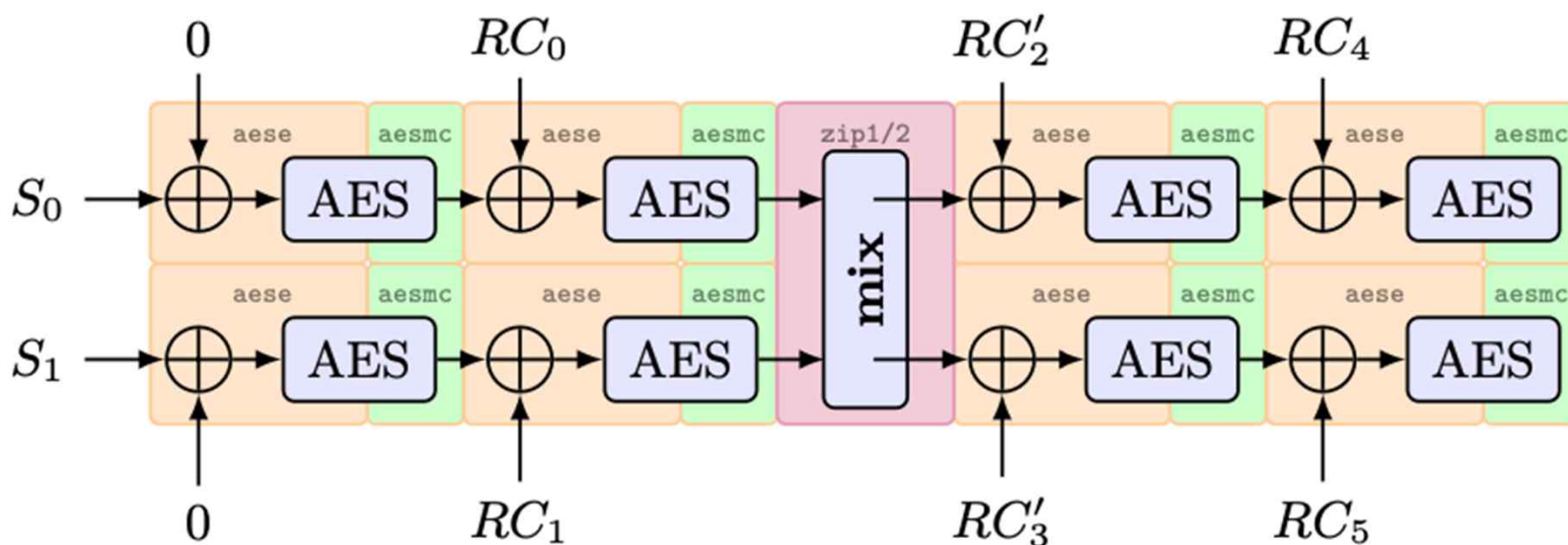
- Latency
- Inverse Throughput



Platform	Instruction	Latency	inv. Throughput
Skylake	vectorized XOR	1	0.33
Ryzen	vectorized XOR	1	0.5
Cortex A57	vectorized XOR	3	2

SPHINCS 고속 구현– Haraka

- AES module과 zip1/2 명령어 (mixing)를 이용하여 구현 가능



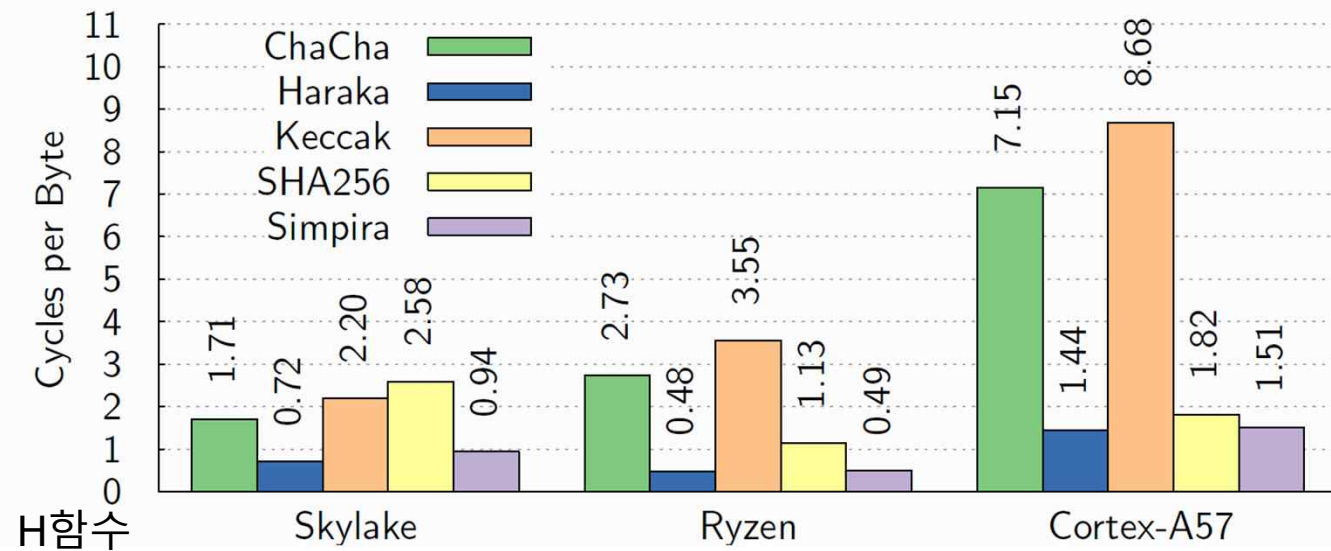
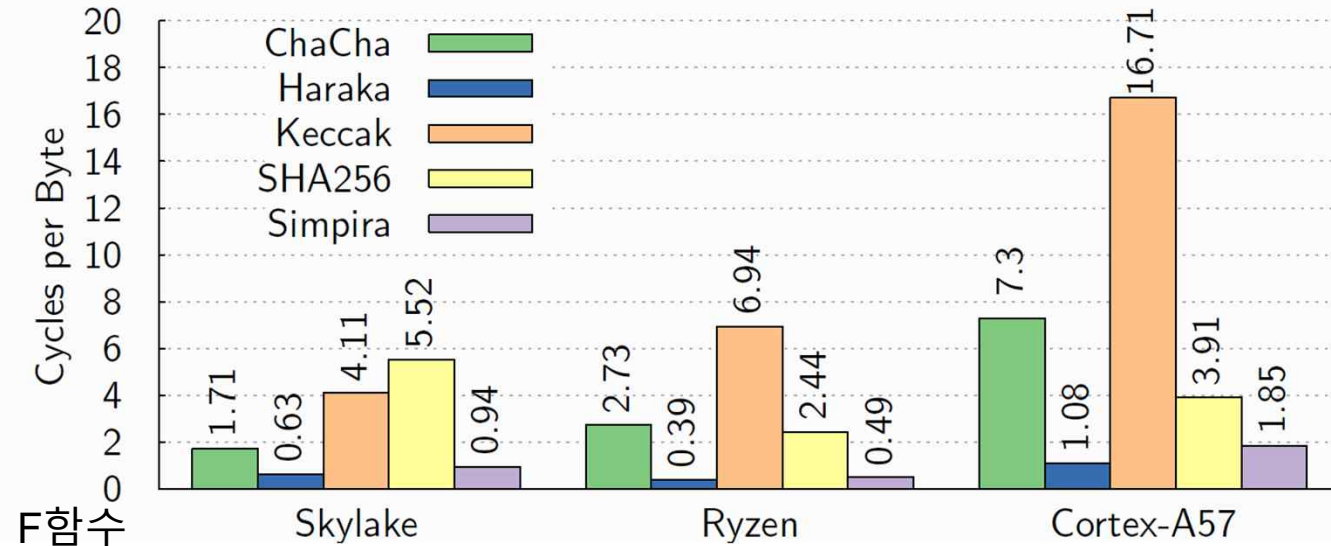
$\text{aesenc} = \text{AddKey} \circ \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}$

$\text{aesmc} \circ \text{aese} = \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes} \circ \text{AddKey}$

SPHINCS 고속 구현

• 해시함수 최적화

- SHA-2
 - Vectorize, Hardware Support
- SHA-3
 - Vectorize, Hardware Support
- ChaCha12
 - Vectorize
- Haraka
 - AES + Permute
- Simpira
 - AES



컴퓨터가 가진 특수 능력 (AVX 상세)?

• 컴퓨터의 특수 능력 (?)은 매우 더디게 발전 중

프로세서 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
설치된 RAM 32.0GB(31.6GB 사용 가능)

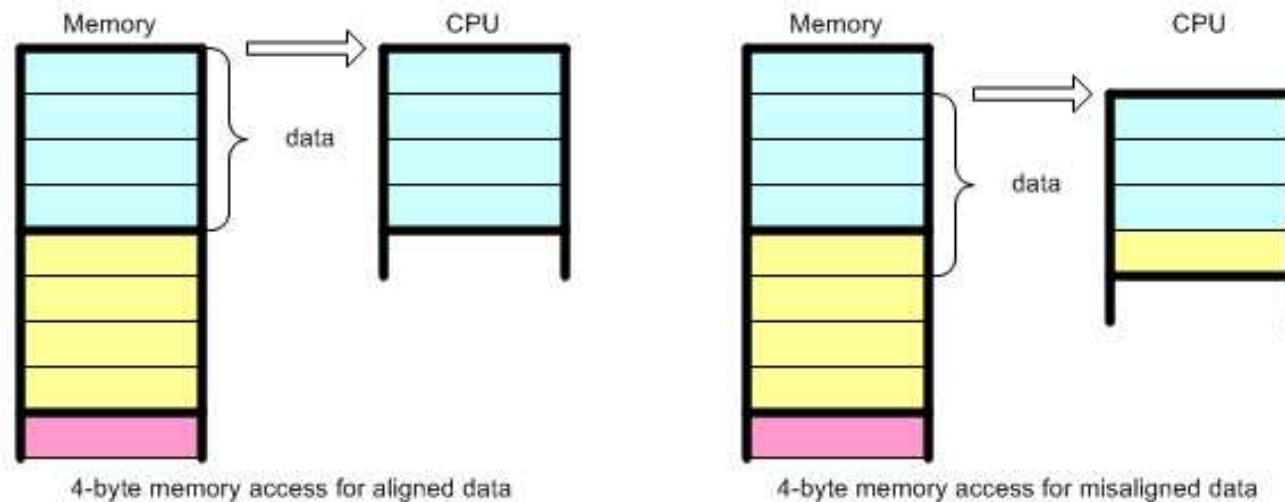
Instruction Set ?	64-bit
Instruction Set Extensions ?	Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2, Intel® AVX-512
Idle States ?	Yes
Thermal Monitoring Technologies ?	Yes
Intel® Volume Management Device (VMD) ?	Yes
Security & Reliability	
Intel® Control-Flow Enforcement Technology ?	Yes
Intel® Total Memory Encryption ?	No
Intel® AES New Instructions ?	Yes
Intel® Software Guard Extensions (Intel® SGX) ?	No
Intel® OS Guard	Yes
Intel® Trusted Execution Technology ‡ ?	No
Intel® Boot Guard ?	Yes
Mode-based Execute Control (MBEC) ?	Yes



데이터 정렬

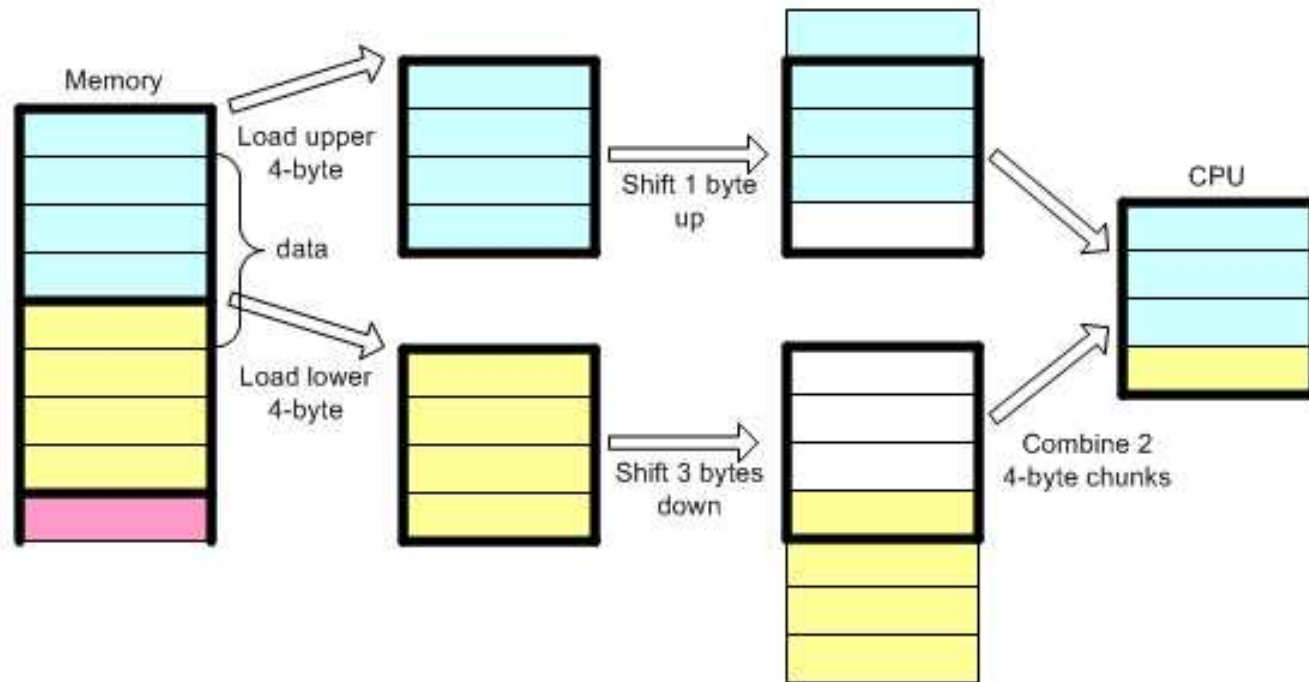
- **데이터는 2개의 특성을 가짐**
 - 실제 값
 - 저장되는 공간
- **데이터 주소는 2의 배수 형식으로 정렬됨**
 - 예시) 0x12FEEC (1244908)의 경우 4-바이트로 정렬
 - 컴퓨터는 높은 성능을 위해 1바이트가 아닌 16, 32바이트 형식으로 접근

데이터 정렬



- 위의 예시는 4-바이트 메모리 접근 시 정렬된 경우와 그렇지 않은 경우를 나타내고 있음
- 정렬되지 않은 메모리 접근은 추가적인 연산으로 인한 시간 소모 발생

데이터 정렬



- 정렬되지 않은 데이터에 대한 접근 예시
 - 두번의 메모리 접근 및 추가적인 연산 필요

데이터 정렬

- 사용하고자 하는 명령어셋에 최적화된 데이터 정렬 중요
- 실제 암호 구현 시 연산 딜레이 보다 메모리 접근이 성능에 큰 영향을 미침

데이터 형식	정렬 (바이트)
char	1
short	2
int	4
float	4
double	8
SSE	16
AVX256	32
AVX512	64

```
int x __attribute__((aligned (16))) = 0;
```

데이터 정렬 – case study: KISA LSH

```
/*
 * LSH: step constants
 */
// 32바이트 단위로 정렬 → AVX256이 32바이트 단위로 정보를 가지고 오기 때문
static const LSH_ALIGNED_32 lsh_u32 g_StepConstants[CONST_WORD_LEN * NUM_STEPS] = {
```

```
0x917caf90, 0x6c1b10a2, 0x6f352943, 0xc7778243, 0x2ceb7472, 0x29e96ff2, 0x8a9ba428, 0x2eeb2642,
0x0e2c4021, 0x872bb30e, 0xa45e6cb2, 0x46f9c612, 0x185fe69e, 0x1359621b, 0x263fcb2, 0x1a116870,
0x3a6c612f, 0xb2dec195, 0x02cb1f56, 0x40bfd858, 0x784684b6, 0x6cbb7d2e, 0x660c7ed8, 0x2b79d88a,
0xa6cd9069, 0x91a05747, 0xcdea7558, 0x00983098, 0xbecb3b2e, 0x2838ab9a, 0x728b573e, 0x55262b5,
0x745dfa0f, 0x31f79ed8, 0xb85fce25, 0x98c8c898, 0x8a0669ec, 0x60e445c2, 0xfde295b0,
0xd2580983, 0x29967709, 0x182df3dd, 0x61916130, 0x90705676, 0x452a0822, 0xe07846ad,
0x2a618d55, 0xc00d8032, 0x4621d0f5, 0xf2f29191, 0x00c6cd06, 0x6f322a67, 0x58bef48d,
0x8beee27f, 0xcd8db2f2, 0x67f2c63b, 0xe5842383, 0xc793d306, 0xa15c91d6, 0x17b381e5,
0x7ad1620a, 0x5b40a5bf, 0x5ab901a2, 0x69a7a768, 0x5b66d9cd, 0xfdee6877, 0xcb3566fc,
0x4c336c84, 0x9be6651a, 0x13baa3fc, 0x114f0fd1, 0xc240a728, 0xec56e074, 0x009c63c7,
0x7f9ff0d0, 0x824b7fb5, 0xce5ea00f, 0x605ee0e2, 0x02e7cfea, 0x43375560, 0x9d002ac7,
0x1f90c14f, 0xcdcb3537, 0x2cfeafdd, 0xb3fc342, 0xeab7b9ec, 0x7a8cb5a3, 0x9d2af264,
0xb052106e, 0x99006d04, 0x2bae8d09, 0xff030601, 0xa271a6d6, 0x0742591d, 0xc81d5701,
0x02627f1e, 0x996d719d, 0xda3b9634, 0x02090800, 0x14187d78, 0x499b7624, 0xe57458c9,
0x64e19d20, 0x06df0f36, 0x15d1cb0e, 0x0b110802, 0x2c95f58c, 0xe5119a6d, 0x59cd22ae,
0x467ebd84, 0xe5ee453c, 0xe79cd923, 0x1c190a0d, 0xc28b81b8, 0xf6ac0852, 0x26efd107,
0xc53c41ca, 0xd4338221, 0x8475fd0a, 0x35231729, 0x4e0d3a7a, 0xa2b45b48, 0x16c0d82d,
0x017e0c8f, 0x07b5a3f5, 0xfa73078e, 0x583a405e, 0x5b47b4c8, 0x570fa3ea, 0xd7990543,
0x7f8a9b90, 0xbd5998fc, 0x6d7a9688, 0x927a9eb6, 0xa2fc7d23, 0x66b38e41, 0x709e491a,
0x0a262c0f, 0x16f295b9, 0xe8111ef5, 0x0d195548, 0x9f79a0c5, 0x1a41cfa7, 0x0ee7638a,
0x30523b19, 0x09884ecf, 0xf93014dd, 0x266e9d55, 0x191a6664, 0x5c1176c1, 0xf64aed98,
0x828d5449, 0x91d71dd8, 0x2944f2d6, 0x950bf27b, 0x3380ca7d, 0x6d88381d, 0x4138868e,
0x0fe19dcb, 0x68f4f669, 0x6e37c8ff, 0xa0fe6e10, 0xb44b47b0, 0xf5c0558a, 0x79bf14cf,
0xf17f68da, 0x5deb5fd1, 0xa600c86d, 0x9f6c7eb0, 0xf92f864, 0xb615e07f, 0x38d3e448,
0x70e843cb, 0x494b312e, 0xa6c93613, 0x0beb2f4f, 0x928b5d63, 0xcbf66035, 0x0cb82c80,
0x592c0f3b, 0x947c5f77, 0x6fff49b9, 0xf71a7e5a, 0x1de8c0f5, 0xc2569600, 0xc4e4ac8c,
```

```
};
```

```
/*
 * LSH : permutation information
 */
static const LSH_ALIGNED_32 lsh_u32 g_StepConstants[CONST_WORD_LEN * NUM_STEPS] = {
```

```
0x03020100, 0x06050407, 0x09080b0a,
static const LSH_ALIGNED_32 lsh_u32 g_StepConstants[CONST_WORD_LEN * NUM_STEPS] = {
0x0f0e0d0c, 0x0b0a0908, 0x03020100,
```

```
};
```

```
#define LOAD(x) _mm256_loadu_si256((__m256i*)x)
#define STORE(x,y) _mm256_storeu_si256((__m256i*)x, y)
```

정보를 불러오는 연산의 경우 현재는 align을 강요하진 않음
아마도 align 되지 않은 데이터 처리 용이성을 위해서

Synopsis

```
_mm256i _mm256_loadu_si256 (__m256i const * mem_addr)
#include <immintrin.h>
Instruction: vmovdqu ymm, m256
CPUID Flags: AVX
```

Description

Load 256-bits of integer data from memory into `dst.mem_addr` does not need to be aligned on any particular boundary.

Operation

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	7	0.333333333
Icelake Intel Core	7	0.5
Icelake Xeon	7	0.56
Skylake	7	0.5

vmovdqa32

Synopsis

```
_mm256i _mm256_load_epi32 (void const* mem_addr)
#include <immintrin.h>
Instruction: vmovdqa32 ymm, m256
CPUID Flags: AVX512F + AVX512VL
```

Description

Load 256-bits (composed of 8 packed 32-bit integers) from memory into `dst.mem_addr` must be aligned on a 32-byte boundary or a general-protection exception may be generated.

```

)/* ----- *
* LSH: functions
* ----- */

#define LOAD(x) _mm256_loadu_si256((__m256i*)x)
#define STORE(x,y) _mm256_storeu_si256((__m256i*)x, y)
#define XOR(x,y) _mm256_xor_si256(x,y)
#define OR(x,y) _mm256_or_si256(x,y)
#define AND(x,y) _mm256_and_si256(x,y)
#define SHUFFLE8(x,y) _mm256_shuffle_epi8(x,y)

#define ADD(x,y) _mm256_add_epi32(x,y)
#define SHIFT_L(x,r) _mm256_slli_epi32(x,r)
#define SHIFT_R(x,r) _mm256_srli_epi32(x,r)

__m256i _mm256_shuffle_epi32 (__m256i a, const int imm8)

```

Synopsis

```

__m256i _mm256_shuffle_epi32 (__m256i a, const int imm8)
#include <immintrin.h>
Instruction: vpslufd ymm, ymm, imm8
CPUID Flags: AVX2

```

Description

Shuffle 32-bit integers in **a** within 128-bit lanes using the control in **imm8**, and store the results in **dst**.

Operation

```

DEFINE SELECT4(src, control) {
    CASE(control[1:0]) OF
    0:    tmp[31:0] := src[31:0]
    1:    tmp[31:0] := src[63:32]
    2:    tmp[31:0] := src[95:64]
    3:    tmp[31:0] := src[127:96]
    ESAC
    RETURN tmp[31:0]
}

dst[31:0] := SELECT4(a[127:0], imm8[1:0])
dst[63:32] := SELECT4(a[127:0], imm8[3:2])
dst[95:64] := SELECT4(a[127:0], imm8[5:4])
dst[127:96] := SELECT4(a[127:0], imm8[7:6])
dst[159:128] := SELECT4(a[255:128], imm8[1:0])
dst[191:160] := SELECT4(a[255:128], imm8[3:2])
dst[223:192] := SELECT4(a[255:128], imm8[5:4])
dst[255:224] := SELECT4(a[255:128], imm8[7:6])
dst[MAX:256] := 0

```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	1	0.5
Icelake Intel Core	1	0.5
Icelake Xeon	1	0.5
Skylake	1	1

그 외에 사용된 명령어 셋들은 대부분 벡터 레지스터 상에서 스칼라 값 정렬
그리고 고려사항: packing & unpacking

```

static INLINE void word_perm(__m256i* cv_l, __m256i* cv_r){
    __m256i temp;
    temp = _mm256_shuffle_epi32(*cv_l, 0xd2);
    *cv_r = _mm256_shuffle_epi32(*cv_r, 0x6c);
    *cv_l = _mm256_permute2x128_si256(temp, *cv_r, 0x31);
    *cv_r = _mm256_permute2x128_si256(temp, *cv_r, 0x20);
};

__m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, const int imm8)

```

Synopsis

```

__m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, const int imm8)
#include <immintrin.h>
Instruction: vperm2i128 ymm, ymm, ymm, imm8
CPUID Flags: AVX2

```

Description

Shuffle 128-bits (composed of integer data) selected by **imm8** from **a** and **b**, and store the results in **dst**.

Operation

```

DEFINE SELECT4(src1, src2, control) {
    CASE(control[1:0]) OF
    0:    tmp[127:0] := src1[127:0]
    1:    tmp[127:0] := src1[255:128]
    2:    tmp[127:0] := src2[127:0]
    3:    tmp[127:0] := src2[255:128]
    ESAC
    IF control[3]
        tmp[127:0] := 0
    FI
    RETURN tmp[127:0]
}

dst[127:0] := SELECT4(a[255:0], b[255:0], imm8[3:0])
dst[255:128] := SELECT4(a[255:0], b[255:0], imm8[7:4])
dst[MAX:256] := 0

```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	3	1
Icelake Intel Core	3	1
Icelake Xeon	3	1
Skylake	3	1

더 자세한 사항은 Intel Intrinsic에서...

Instruction Set

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVMML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math Functions
- ☐ General Support
- ☐ Load
- ☐ Logical

Q Search Intel Intrinsics

<code>void _mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)</code>	<code>vp2intersectd</code>
<code>void _mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)</code>	<code>vp2intersectd</code>
<code>void _mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16* k2)</code>	<code>vp2intersectd</code>
<code>void _mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)</code>	<code>vp2intersectq</code>
<code>void _mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)</code>	<code>vp2intersectq</code>
<code>void _mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2)</code>	<code>vp2intersectq</code>
<code>__m512i _mm512_4dpwssd_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)</code>	<code>vp4dpwssd</code>
<code>__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)</code>	<code>vp4dpwssd</code>
<code>__m512i _mm512_maskz_4dpwssd_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)</code>	<code>vp4dpwssds</code>
<code>__m512i _mm512_4dpwssds_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)</code>	<code>vp4dpwssds</code>
<code>__m512i _mm512_mask_4dpwssds_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)</code>	<code>vp4dpwssds</code>
<code>__m512i _mm512_maskz_4dpwssds_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)</code>	<code>vp4dpwssds</code>
<code>__m512 _mm512_4fmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)</code>	<code>v4fmaddps</code>
<code>__m512 _mm512_mask_4fmadd_ps (__m512 src, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)</code>	<code>v4fmaddps</code>
<code>__m512 _mm512_maskz_4fmadd_ps (__mmask16 k, __m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)</code>	<code>v4fmaddps</code>
<code>__m128 _mm_4fmadd_ss (__m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)</code>	<code>v4fmaddss</code>
<code>__m128 _mm_mask_4fmadd_ss (__m128 src, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)</code>	<code>v4fmaddss</code>
<code>__m128 _mm_maskz_4fmadd_ss (__mmask8 k, __m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)</code>	<code>v4fmaddss</code>
<code>__m512 _mm512_4fnmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)</code>	<code>v4fnmaddps</code>
<code>__m512 _mm512_mask_4fnmadd_ps (__m512 src, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)</code>	<code>v4fnmaddps</code>
<code>__m512 _mm512_maskz_4fnmadd_ps (__mmask16 k, __m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)</code>	<code>v4fnmaddps</code>
<code>__m128 _mm_4fnmadd_ss (__m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)</code>	<code>v4fnmaddss</code>
<code>__m128 _mm_mask_4fnmadd_ss (__m128 src, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)</code>	<code>v4fnmaddss</code>
<code>__m128 _mm_maskz_4fnmadd_ss (__mmask8 k, __m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)</code>	<code>v4fnmaddss</code>

새로운 구조: AVX512

- 2013년 7월에 공개된 SIMD 구조
- 512-비트 vector register를 사용하며 신규 명령어 셋 제공

기존 rotation은 두 번의 shift를 통해 수행: `_mm256_srli_epi32(cur, 9), _mm256_slli_epi32(cur, 23)`
AVX512에서는 rotation 전용 명령어셋 제공

```
__m512i _mm512_rol_epi32 (__m512i a, const int imm8)
```

Synopsis

```
__m512i _mm512_rol_epi32 (__m512i a, const int imm8)
#include <immintrin.h>
Instruction: vprold zmm, zmm, imm8
CPUID Flags: AVX512F
```

Description

Rotate the bits in each packed 32-bit integer in `a` to the left by the number of bits specified in `imm8`, and store the results in `dst`.

Operation

```
DEFINE LEFT_ROTATE_DWORDS(src, count_src) {
    count := count_src % 32
    RETURN (src << count) OR (src >> (32 - count))
}
FOR j := 0 to 15
    i := j*32
    dst[i+31:i] := LEFT_ROTATE_DWORDS(a[i+31:i], imm8[7:0])
ENDFOR
dst[MAX:512] := 0
```

511	256	255	128	127	0
ZMM0	YMM0	XMM0			
ZMM1	YMM1	XMM1			
ZMM2	YMM2	XMM2			
ZMM3	YMM3	XMM3			
ZMM4	YMM4	XMM4			
ZMM5	YMM5	XMM5			
ZMM6	YMM6	XMM6			
ZMM7	YMM7	XMM7			
ZMM8	YMM8	XMM8			
ZMM9	YMM9	XMM9			
ZMM10	YMM10	XMM10			
ZMM11	YMM11	XMM11			
ZMM12	YMM12	XMM12			
ZMM13	YMM13	XMM13			
ZMM14	YMM14	XMM14			
ZMM15	YMM15	XMM15			
ZMM16	YMM16	XMM16			
ZMM17	YMM17	XMM17			
ZMM18	YMM18	XMM18			
ZMM19	YMM19	XMM19			
ZMM20	YMM20	XMM20			
ZMM21	YMM21	XMM21			
ZMM22	YMM22	XMM22			
ZMM23	YMM23	XMM23			
ZMM24	YMM24	XMM24			
ZMM25	YMM25	XMM25			
ZMM26	YMM26	XMM26			
ZMM27	YMM27	XMM27			
ZMM28	YMM28	XMM28			
ZMM29	YMM29	XMM29			
ZMM30	YMM30	XMM30			
ZMM31	YMM31	XMM31			

새로운 명령어 – AES512

- 하나의 명령어 셋을 통해 4개의 평문에 대한 AES 라운드 연산 가능

```
__m512i _mm512_aesenc_epil28 (__m512i a, __m512i RoundKey)
```

Synopsis

```
__m512i _mm512_aesenc_epil28 (__m512i a, __m512i RoundKey)
#include <immintrin.h>
Instruction: vaesenc zmm, zmm
CPUID Flags: AVX512F + VAES
```

Description

Perform one round of an AES encryption flow on data (state) in `a` using the round key in `RoundKey`, and store the results in `dst`.

Operation

```
FOR j := 0 to 3
    i := j*128
    a[i+127:i] := ShiftRows(a[i+127:i])
    a[i+127:i] := SubBytes(a[i+127:i])
    a[i+127:i] := MixColumns(a[i+127:i])
    dst[i+127:i] := a[i+127:i] XOR RoundKey[i+127:i]
ENDFOR
dst[MAX:512] := 0
```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Icelake Intel Core	-	1
Icelake Xeon	3	1

맺음말

- 암호 최적 구현은 크게 두가지 접근 방법을 취해야 함
 - 암호 알고리즘 자체 복잡도를 낮춤
 - 암호 알고리즘이 컴퓨터 구조에 알맞게 구현

→ 단 두 방법론은 독립적이며 최악의 경우 서로 상충 (상호간 양보 필요)
- 일반적으로는 최적화된 암호 알고리즘을 현재 컴퓨터 구조에 적용하는 방향성을 가짐
- 다만 다양한 컴퓨터 구조로 인해 모두를 만족하는 암호구현은 어려움
 - 예시) 하드웨어 기반 암호는 비트단위 구현을 취함 (SW에 비효율적)
 - 하지만 암호엔지니어들은 어떻게든 이를 극복하는 시도를 하고 있음
- 따라서 처음부터 맞추어 가기 보다는 지속적인 상호간 피드백을 통해 점진적으로 암호구현 최적화를 해나가는 것이 올바른 방향으로 보임

Q & A