석사학위논문

격자 기반 암호 LizarMong의 최적화 구현

2020년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

권 용 빈

석사학위논문
지도교수 서화정

격자 기반 암호 LizarMong의 최적화 구현

Optimized implementation of LizarMong using AVX 2

2020년 6월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

권 용 빈

석 사 학 위 논 문
지도교수 서화정

격자 기반 암호 LizarMong의 최적화 구현

Optimized implementation of LizarMong
using AVX 2

위 논문을 공학 석사학위 논문으로 제출함

2020년 6월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

권 용 빈

권용빈의 공학 석사학위 논문을 인준함

2020년 6월 일

심사위원장 _____(인)

심 사 위 원 _____(인)

심 사 위 원 _____(인)

국 문 초 록

격자 기반 암호 LizarMong의 최적화 구현

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
권 용 빈

양자 컴퓨팅 환경의 발전이 기존의 공개키 암호 시스템을 위협하고 있다. P. Shor가 제안한 알고리즘은 현재 가장 널리 사용되고 있는 공개키 암호 시스템인 RSA 암호 시스템 및 타원곡선 시스템을 다항시간 내에 분석할 수 있음을 이론적으로 증명하였으며, Google 및 IBM은 양자 컴퓨팅의 상용화에 박차를 가하고 있다. 이에 따라 많은 기관들은 기존 폰 노이만 구조의 컴퓨터를 이용한 공격과 양자 컴퓨터를 이용한 공격 모두에 안전한 암호를 요구하고 있다. 이러한 요구에 맞춰 미국 국립표준연구소는 두 컴퓨터로부터 안전한 암호를 양자 내성 암호라고 정의하고 표준화를 위한 공모전을 진행하고 있다. 공모전 후보 알고리즘 중 과반을 차지하고 있는 격자 기반 암호는 훌륭한 성능과 오랜 연구에 따른 검증된 안전성으로 주목받고 있다. LizarMong은 후보 알고리즘 중 하나였던 Lizard에 기초를 둔 격자 기반 알고리즘으로 기존의 잘 연구된 Lizard 알고리즘에 최신 연구결과들을 집약한 알고리즘이다. LizarMong은 NIST 공모전 후보 알고리즘과 견주었을 때 탁월한 성능을 자랑하고 있다. 본 논문에서는 AVX2를 중심으로 LizarMong의 최적화 구현 기

법들을 제안한다. 이 기법은 성능을 15%에서 30% 향상시키며, 필요로 하는 난수의 수를 최소화한다. 이러한 연구 결과는 현재에도 타 후보 알고리즘 대비 탁월한 성능을 갖고 있는 LizarMong의 실용성을 더욱 높여줄 것으로 기대한다.

【주요어】 양자 내성 암호, 격자 기반 암호, LizarMong, AVX2, 최적화 구현, SIMD

목 차

표 목 차

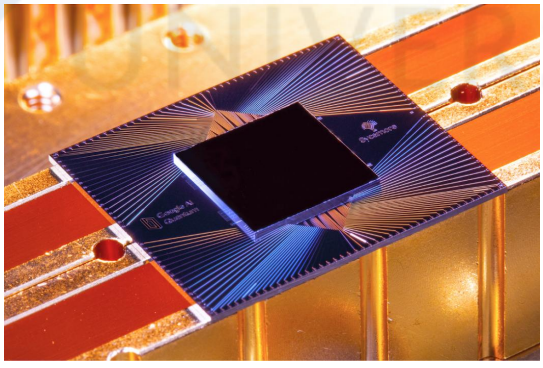
알고리즘 목 차

그 림 목 차

Ⅰ. 서론

1.1 연구 목적

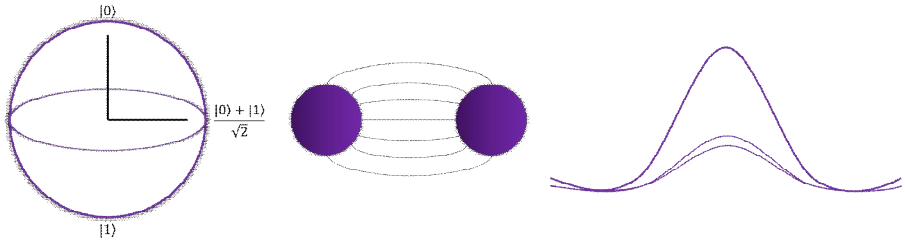
양자 컴퓨터에 대한 연구가 활발하게 진행되고 있는 가운데, 2019년 10월 구글(Google)이 과학 저널 네이처(Nature)에 양자 우월성(quantum supremacy) 달성을 증명한 논문을 발표하면서, 가까운 미래에 양자 컴퓨터 시대가 도달할 것임을 알렸다. 양자 우월성이란 양자 컴퓨터의 기술 수준을 지칭하는 말로, 양자 컴퓨터의 성능이 현존하는 슈퍼 컴퓨터의 성능을 뛰어넘는 수준을 말한다. 해당 논문에 따르면 구글에서는 난수성을 증명이라는 특정한 문제를 대상으로 양자 프로세서 시카모어(Sycamore)의 성능 실험을 했다. 그 결과 기존 슈퍼 컴퓨터를 이용하면 약 10,000년이 걸리는 작업을 양자 컴퓨터를 이용하여 약 200초에 해결함을 보였다. 다시 말해, 부분적인 양자 우월성에 도달했음을 보인 것이다.



[그림 1-1] 구글의 양자 프로세서 시카모어

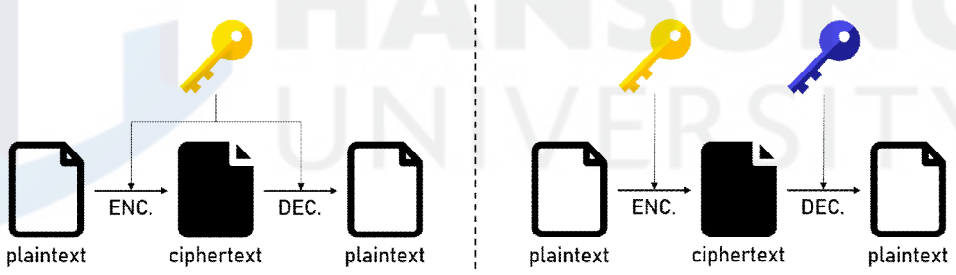
비트를 사용하는 기존의 컴퓨터와 다르게 양자 컴퓨터는 큐비트(qubit)라는 새로운 단위를 사용한다. 큐비트는 양자역학의 중첩(superposition), 얽힘(entanglement) 그리고 간섭(interference) 등의 성질을 가지고 있으며 이러한 새로운 성질을 이용한 알고리즘으로 기존의 컴퓨터가 해결하기 어

려웠던 문제들을 해결해 낼 수 있다.



[그림 1-2] 큐비트가 가지는 성질들 왼쪽부터 각각 중첩, 얽힘, 간섭

이러한 양자 컴퓨팅 환경이 보안 분야에 미치는 영향은 상당하다. 현대의 암호 시스템은 송수신자가 동일한 키를 사용하는 대칭키 시스템과 송수신자가 서로 다른 키를 사용하는 공개키 시스템으로 이루어져 있다.



[그림 1-3] 왼쪽: 대칭키 시스템, 오른쪽: 공개키 시스템

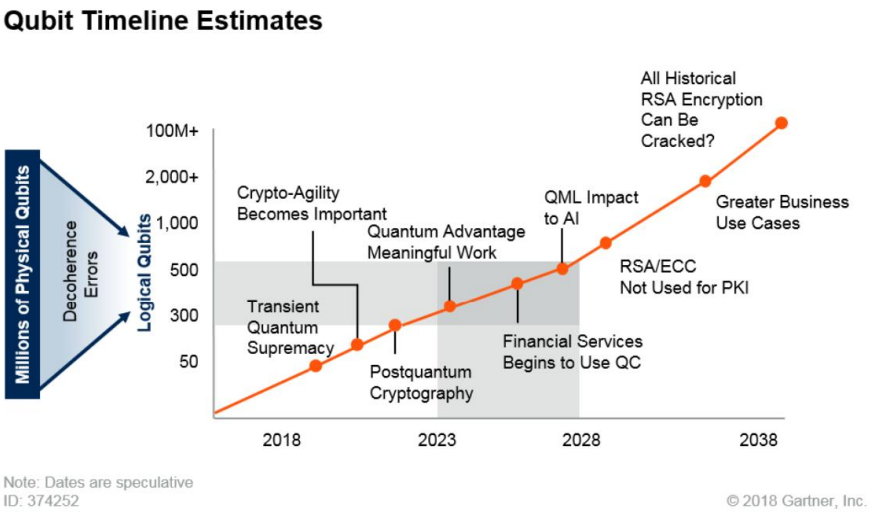
양자 컴퓨팅에서 실행 가능한 대표적인 공격 알고리즘인 그로버 알고리즘과 쇼어 알고리즘은 각각 대칭키 시스템, 공개키 시스템을 공격하는 알고리즘이다. 그로버 알고리즘은 탐색 알고리즘으로 모든 경우의 수를 대입해보는 전사 공격의 효율을 높이며 표준 대칭키 알고리즘인 AES에 적용될 수 있다. 쇼어 알고리즘은 현재 공개키 시스템의 표준으로 지정된 RSA 알고리즘이나 비트코인 등에서 사용하고 있는 ECC 알고리즘에 적용될 수 있다.

[표 1-1] 그로버, 쇼어 알고리즘의 공격 대상

Algorithms	Target System	Target Algorithms	Counter measure
Grover	Symmetric-Key	AES	o
Shor	Public-Key	RSA, ECC	x

그로버 알고리즘을 이용한 전사 공격의 경우 대응 기법이 알려져 있다. 양자 컴퓨팅 환경의 높은 구축 비용과 직렬로 실행되어야 된다는 제약 사항 등을 고려하면 사용되는 키의 길이를 두 배로 하는 것으로 충분한 안전성을 보장할 수 있다. 반면에 쇼어 알고리즘을 이용한 공격의 경우 알고리즘의 안전성을 보장하는 수학적 난제에 적용이 가능하다. 공개키 표준인 RSA의 경우 큰 수의 소인수 분해 문제에 적용 가능하며 ECC의 경우에는 타원 곡선에서 정의된 이산 로그 문제에 적용이 가능하다. 이에 대응하기 위해서는 문제 자체를 변경하여야 하며 현재 사용되고 있는 RSA, ECC는 더 이상 안전하지 않다.

RSA, ECC의 안전성을 보장하는 난제들은 기존의 폰 노이만 컴퓨터를 이용하였을 때 지수 시간(exponential time) 내 해결이 가능하다. 이는 입력 길이가 증가함에 따라 문제 해결에 필요한 연산 시간이 지수적으로 증가함을 의미한다. 즉, 충분한 길이의 입력을 사용하여 문제의 어려움을 담보할 수 있다. 하지만, P. Shor는 양자 컴퓨팅 환경을 이용하는 양자 알고리즘을 이용하여 이러한 문제들을 다항 시간(polynomial time) 내에 해결할 수 있음을 증명하였고, 이를 쇼어 알고리즘이라고 한다. 동일한 기술 수준으로 이용 가능한 입력 길이를 고려해봤을 때, 더 이상 안전성이 담보될 수 없는 것이다.



[그림 1-3] Qubit timeline[1)]

양자 컴퓨터의 개발은 초기에는 양자 컴퓨터의 기본 데이터 단위가 되는 큐비트를 얼마나 많이 구현할 수 있는가에 달려있었다. 하지만 현재는 이러한 큐비트들의 양자적 특징을 온전히 이용할 수 있도록 오류를 최소화하는 방향으로 연구가 진행되고 있다. 2018년 미국의 정보 기술 연구 및 자문 회사 가트너(Gartner)는 양자 컴퓨팅 환경의 발전 속도 추세를 고려하여 2019년 양자 우월성에 도달할 것임을 성공적으로 예측하였다. 또한 가트너는 2030년 RSA, ECC가 더 이상 공개키 시스템으로 이용될 수 없을 것이라고 예측하고 있다. 이외에도 양자 전문가들 또한 짧게는 15년에서 길게는 30년 안에 양자 컴퓨터가 상용화될 것임을 예측하고 있다. 하지만 국가기록물 등 장기간 암호화가 필요한 경우가 존재하며, 많은 기업들 역시 장기간 보호할 수 있는 암호를 원한다. 또한 암호 시스템의 변경에는 상당한 시간적, 금전적 비용이 소요된다. 이러한 이유로 양자 컴퓨팅 환경과 기존 컴퓨터 환경에서의 안전성을 보장하는 준수한 성능의 표준화 암호의 필요성이 제기되었다.

미국 국립표준기술연구소(NIST, National Institute of Standards and Technology)에서는 양자 컴퓨팅 환경에 내성을 갖는 암호인 양자 내성

1) Gatner. (2018). Top 10 Strategic Technology Trends for 2019 (40p). https://www.gartner.com/en/doc/3891569-top-10-strategic-technology-trends-for-2019 (Accessed May 2020).

암호(PQC, Post-Quantum Cryptosystem)의 표준화를 위한 공모전을 시작했다.[2] 전자 서명과 KEM(Key Encapsulation Mechanism)/PKE(Public Key Encryption) 두 분야를 아울러 64개의 후보가 제출되었으며, Round 1을 거쳐 26개의 후보만이 Round 2를 진행 중이다. 양자 내성 암호의 경우, 양자 컴퓨팅 환경에서도 쉽게 해결할 수 없는 수학적인 난제를 기반으로 한다. 공모전에 출품된 암호 알고리즘들의 안전성을 보장하고 있는 난제는 크게 분류되어질 수 있다. 그 중 격자 문제를 이용하는 암호 알고리즘들이 있으며 이를 격자 기반 암호(LBC, Lattice Based Cryptography)라고 한다. 격자 기반 암호 알고리즘 들은 NIST Round 2 후보 알고리즘들 중 과반에 가까운 비율을 차지하는 알고리즘으로 주목을 받고 있으며 활발한 연구가 진행되고 있다.

NIST는 후보 알고리즘들에게 기존 컴퓨팅 환경과 양자 컴퓨팅 환경 모두로부터 AES(128bit, 192bit, 256bit)수준의 안전성을 가지는 것 이외에도 제한된 자원을 가지는 경량 디바이스 등에서의 적합성 그리고 부채널 분석 저항을 가질 것 등을 권고하고 있다. 이는 IoT(Internet of Things) 환경 등 앞으로 점차 다양화될 통신 환경을 고려한 것으로 보인다. 신뢰할 수 있는 표준 공개키 시스템의 선정을 위해서는 공모전의 대상이 되고 있는 암호 알고리즘 군들의 연구가 안정성과 더불어 성능 등 다양한 측면에서 연구되어야 한다는 것이다.

격자 문제를 이용하는 LizarMong은 NIST Round 1 후보 알고리즘 중 하나인 Lizard 알고리즘을 계승한 알고리즘이다. Lizard를 근간하여 격자 기반 문제를 위한 최신 연구들을 집약한 알고리즘으로 현재의 NIST Round 2 후보 알고리즘들과 비교하였을 때 가장 빠른 속도를 가지는 암호 알고리즘이다. 본 논문에서는 LizarMong을 대상으로 한 최적화 병렬 구현 방법을 제안한다. 이러한 최적화 구현 기법은 격자 기반 문제의 구현 기법으로서 참고하여 다양한 통신 환경을 위한 구현이나 기존 표준인 RSA가 적용되어 있으나 성능 등의 문제로 대체될 수 없는 상황에 도움이

2) NIST. (2017). Post-Quantum Cryptography Standardization Call for Propsals. https://csrc.nist.gov/Projects/post-quantum-cryptography/Post-Quantum-Cryptography-Standardization (Accessed May 2020).

될 수 있을 것으로 기대된다.

1.2 연구 기여

양자 내성 암호의 한 분류인 격자 기반 암호는 이미 오랜 기간 연구되어 안전성이 검증되었지만 실용성의 측면에서 현재 사용하고 있는 RSA 및 ECC에 밀려 사용되지 못했던 암호이다. 하지만 양자 컴퓨팅 환경이 도래함에 따라 기존의 RSA 및 ECC가 더 이상 안전성을 제공하지 않게 되면서 격자 기반 암호는 다른 알고리즘 군과 비교하였을 때 가장 큰 가능성을 가지는 대안이 되고 있다. 비록 RSA, ECC 보다는 못하지만 뛰어난 성능을 가지고 있으며 오랜 기간 연구되어 역사가 깊다는 점이 크게 작용하는 것이다. 하지만 그럼에도 기존의 표준인 RSA를 대체하기 위해서, NIST가 다양한 환경에서 구현될 수 있도록 권고했다는 점을 고려하면 구현에 대한 다각화된 연구가 필요한 것이다. 본 논문에서는 격자 기반 암호인 LizarMong의 병렬화된 최적화 구현 기법을 제안하고 이에 따른 결과를 제시한다. LizarMong의 128비트 안전성을 제공하는 *Comfort*버전과 256비트 안전성을 제공하는 *Strong*버전의 핵심 연산들을 알아본 뒤 AVX2를 이용한 최적화 병렬 구현 기법을 제안한다. LizarMong이 사용하는 데이터 단위는 8비트이며 AVX2는 기본적으로 8비트 곱셈, Shift 연산 등을 지원하지 않는다. 본 논문에서는 이를 고려하여 곱셈을 덧셈과 뺄셈으로 변경하며, 16비트 Shift를 우회적으로 사용하여 해결한다. 또한 제안하는 기법은 샘플링에 필요한 난수를 최적화하며, 데이터의 구조를 병렬 구현에 효율적이도록 변경하여 AVX2 명령어를 적용한다.

1.3 논문 구성

본 논문의 구성은 다음과 같다. 2장에서는 필요한 배경 지식을 설명하고

LizarMong의 주요 연산들에 대해서 알아본다. 3장에서는 주요 연산들에 대한 최적화 구현을 제안한다. 4장에서는 성능을 평가한다. 5장에서는 결론을 내리고 향후 연구에 대해 논의한다.

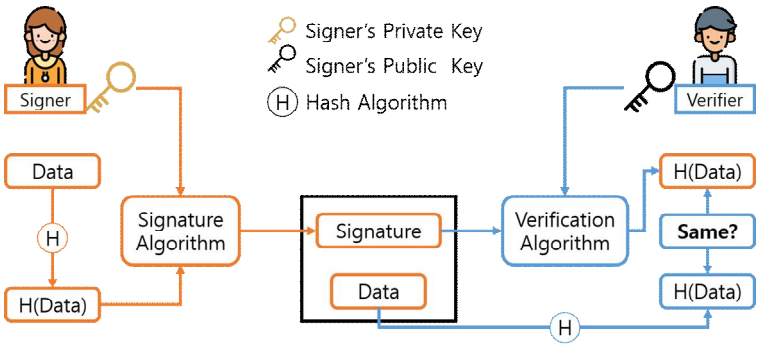
Ⅱ. 배경 지식

2.1 공개키 암호화 방식

현대의 암호 시스템은 송수신자가 동일한 키를 사용하는 대칭키 암호화 방식과 서로 다른 키를 사용하는 공개키 암호화 방식으로 분류된다. 대칭키 암호 알고리즘의 경우 통신 양자가 키를 획득하는 것이 어렵다는 단점이 있지만 키를 공유하였을 경우 빠른 암복호화 속도를 가진다는 장점이 있다. 공개키 암호화 방식의 경우 개인키, 공개키 두 가지 키를 가지고 운용된다. 공개키는 모두에게 공개되기 때문에 안전하게 키를 보관해야하는 대칭키 암호 알고리즘과 달리 키 관리가 간편하다는 장점이 있다. 하지만 메시지의 크기가 제한되거나 암호화 또는 복호화 속도가 느린 등 공개키 암호화 방식의 기반 난제에 관련하여 제약 조건이 존재한다. 따라서 대칭키 암호화 방식의 경우 주로 데이터의 암호화를 위해 사용되는 반면 공개키 암호화 방식의 경우 다음과 같이 사용된다.

2.1.1 전자서명(DSA, Digital Signature Algorithms)

공개키 암호화 방식을 통해 메시지를 인증하고 데이터 변조를 확인하며 부인 방지의 역할을 할 수 있는 활용 방법이다. 송신자는 송신하려는 메시지에 자신이 가진 개인키를 이용하여 서명 알고리즘을 실행한다. 서명 알고리즘은 보내려는 메시지 전체를 해시 함수에 넣어 일정한 크기의 해시값을 만들어 낸 뒤 이를 개인키로 암호화한다. 암호화된 결과를 서명이라고 한다. 수신자는 서명과 메시지를 받은 다음 알려진 송신자의 공개키를 이용하여 서명을 복호화하여 해시값을 복구한다. 복구된 값과 자신이 받은 메시지를 직접 해시 함수에 넣어 나온 결과와 비교한다. 이 값이 같다면 서명과 검증 작업이 성공적으로 이루어진 것이다.



[그림 2-1] DSA 알고리즘 개요

공개키로 복호화가 가능하다는 것은 서명이 송신자의 개인키로 암호화 되었다는 것을 의미하므로 메시지 인증과 부인 방지의 기능을 한다. 또한, 데이터가 변경될 경우 해시값은 완전히 달라진다. 따라서 해시값이 동일한 경우 데이터의 변조가 일어나지 않았음을 확인할 수 있다.

2.1.2 공개키 암호화(PKE, Public Key Encryption)

송신자는 수신자의 공개키를 이용하여 메시지를 암호화한 뒤 수신자에게 보낸다. 암호화된 메시지는 수신자의 개인키로만 복호화가 가능하기 때문에 기밀성이 보존될 수 있다. 하지만 인터넷 환경 등 종단 간 암호화가 필요한 환경에서는 큰 데이터의 지속적인 암호화가 필요할 수 있다. 이러한 경우 대역폭의 제약, 암복호화 연산량의 부담을 갖는 PKE 방식은 적합하지 않다.

정의 1. 공개키 암호화는 $PKE=(KeyGen, Encrypt, Decrypt)$로 표현하며 다음과 같이 정의한다.

- $KeyGen(1^k)=(pk, sk)$. 패러미터 1^k를 입력받아 공개키 pk와 비밀키 sk를 생성한다.
- $Encrypt(pk, M)\rightarrow C$. 공개키 pk를 이용하여 메시지 M을 암호화하여 암호문 C를 생성한다.
- $Decrypt(sk, C)\rightarrow M$. 비밀키 sk를 이용하여 암호문 C를 복호화하여 메시지 M을 생성한다.

2.1.3 키 캡슐화(KEM, Key Encapsulation Mechanisms)

이러한 문제는 대칭키 암호화 방식을 적용하여 보완할 수 있다. 송수신자 간의 특정한 대칭키의 합의를 위해 공개키 암호화 방식을 이용하는 것이다. 하지만 이러한 경우 키의 길이를 고려하여 패딩을 추가하는 등의 작업이 필요하며 안전한 패딩 방식이 별도로 필요하다. 또한 특정한 값을 키로 사용하는 모든 경우의 안전성을 검사하는 것이 어렵다. 따라서 랜덤한 값을 공유한 뒤 이를 키로 이용하는 방식을 키 캡슐화라고 한다. 키 캡슐화의 경우, 랜덤한 바이트를 뽑은 뒤 값을 공유한다 그런 다음 이 바이트를 해시하여 키로서 이용한다. 따라서 패딩 작업이 필요하지 않으며 특정 값을 사용하는 것이 아니므로 안전성 검증도 용이하게 된다.

정의 2. 키 캡슐화는 $KEM = (KeyGen, Encapsulate, Decapsulate)$로 표현하며 다음과 같이 정의한다.

- $KeyGen(1^k) = (pk, sk)$. 패러미터 1^k를 입력받아 공개키 pk와 비밀키 sk를 생성한다.
- $Encapsulate(pk, k) \rightarrow C$. 공개키 pk를 이용하여 키 k를 암호화하여 암호문 C를 생성한다.
- $Decapsulate(sk, C) \rightarrow k$. 비밀키 sk를 이용하여 암호문 C를 복호화하여 키 k을 생성한다.

2.2 양자 컴퓨팅 환경

2.2.1 양자 알고리즘(Quantum Algorithm)

양자 컴퓨터는 양자역학적인 현상을 띄는 큐비트(qubit)를 이용하여 자료를 처리하는 컴퓨터를 이야기 한다. 기존의 폰 노이만 구조의 컴퓨터

는 비트(bit)라고 하는 정보 단위를 사용하며, 각 비트는 0 또는 1의 값을 나타낼 수 있다. 반면에 양자 컴퓨터는 정보 단위로서 큐비트를 사용하며, 각 큐비트는 두 값이 공존하는 형태로 존재할 수 있다. 양자역학에서는 이를 중첩(superposition) 상태라고 부른다. 또한 큐비트는 다른 큐비트와 연결되어 있는 상태로 다른 큐비트에 의존하여 값이 결정될 수 있다. 이를 얽힘(Entanglement) 상태라고 한다. 이러한 큐비트의 양자역학적 성질을 이용하여 현대의 암호의 안전성에 영향을 주는 두 알고리즘이 쇼어 알고리즘과 그로버 알고리즘이다. 쇼어 알고리즘의 경우 공개키 알고리즘인 RSA와 ECC가 기반하고 있는 난제인 소인수 분해와 이산 로가리즘 문제에서 주기를 빠르게 찾을 수 있도록 도와준다. 두 문제는 모두 유한 아벨군(Abelian group)에서 정의되는데. RSA의 소인수 분해의 경우 숨겨진 부분군을 찾는 문제(HSP, Hidden Subgroup Problem)로 변형될 수 있다. HSP는 또한 주기를 찾는 문제로 변형될 수 있으며, 쇼어가 제안한 양자 푸리에 변환을 이용하면 주기를 $O((\log N)^2(\log\log N)(\log\log\log N))$ 내에 찾을 수 있다. 이는 다항 시간 내 문제를 해결할 수 있는 것으로 입력 길이를 증가시키더라도 약간의 연산량의 증가로 해결할 수 있다. 그러므로 현재 널리 사용되고 있는 RSA, ECC는 대체 알고리즘이 필요한 상황이다. 그로버 알고리즘은 효율적인 검색 알고리즘으로 값에 대한 검색을 $O(\sqrt{N})$ 내에 해결하도록 하며 이는 대칭키 암호의 키의 경우의 수를 모두 찾는 전사 공격(brute-force attack)에 이용될 수 있다. 하지만 쇼어 알고리즘과는 다르게 키의 길이를 2배로 하는 것으로 기존의 안전성을 유지할 수 있다고 알려져 있다.

2.2.2 양자 내성 암호(Post-Quantum Algorithm)

공개키 암호 알고리즘에 대한 위험성이 제기됨에 따라 양자 환경에서도 안전성을 제공하는 암호에 대한 요구가 곳곳에서 발생했다. NIST에서는 이를 위해 표준화를 위한 공모전을 진행하였다. 요구하는 조건으로는 기존 폰 노이만 구조 컴퓨터에서의 안전성과 양자 컴퓨터에서의 안전성을 가지는 것이다.

DSA와 KEM/PKE 두 분야에 대해서 공모전이 진행 중이며 Round 2를 거쳐 26개의 후보 알고리즘이 남아있는 상황이다.

[표 2-1] NIST 후보 알고리즘 분류

Algorithms	Signatures	KEM/PKE	Overall
Lattice-based	3	9	12
Code-based	0	7	7
Multi-variate	4	0	4
Symmetric-based	2	0	2
Other	0	1	1
Total	**9**	**17**	**26**

후보 알고리즘이 기반하고 있는 수학적 난제는 양자 환경에서의 알려진 공격법이 없다. 격자 기반, 코드 기반, 다변수 기반 등 다양한 난제에 기반한 알고리즘들이 공모전을 진행하고 있으며 안전성 검증을 받고 있다. NIST에서는 추가적으로 구현에 대한 다양한 벤치마킹을 요구하고 있으며 수학적인 안전성 외 부채널 분석에 대한 안전성 등을 요구하고 있다. 이는 새로운 통신 환경과 늘어난 공격 면적을 고려한 것으로 보이며 표준화 알고리즘으로서 범용성을 가질 것을 요구한 것으로 생각된다. 이에 따라 학계에서는 안전성 검증과 더불어 다각도에서의 구현 및 부채널 분석에 대한 안전성을 연구하고 있다. NIST 표준화 공모전은 2021년 Round 3를 거쳐 2023년 표준화 알고리즘이 선정될 계획이다.

2.3 LBC(Lattice-Based Cryptosystem)

2.3.1 격자

격자란 특정한 점들이 배열된 n차원 유클리드 공간 속 이산 집합이다.

정의 3. 격자 $L \in R^n$과 기저 $b_1,...,b_n \in R^n$은 다음과 같이 정의된다.

$$L(b_1,...,b_n)=\left\{\sum_{i=1}^{n}a_ib_i|a_i\in Z\right\}.$$

격자 $L\in R^n$로 표기하며 이 때, R^n의 이산 부분군이 된다. 격자 기반 암호는 이러한 격자에서 정의되는 문제를 기반으로 설계된 암호 알고리즘이다.

2.3.2 격자 문제

가장 기본이 되는 격자 위 문제로는 SVP(Shortest Vector Problem)와 CVP(Closest Vector Problem)이 있다. SVP는 0이 아닌 가장 짧은 벡터를 찾는 문제로, 벡터의 크기 $|v|$의 0이 아닌 가장 작은 벡터를 격자 내에서 찾는 것이다. CVP는 격자와 상관없는 하나의 벡터가 주어지고 그 벡터에서 가장 가까운 격자 내 벡터를 찾는 것이다. 즉, 벡터 w가 주어졌을 때, $|w-v|$의 값이 최소가 되는 벡터 v를 격자 내에서 찾는 것이다. 최소값을 컴퓨터로 구하기 위해서는 모든 벡터의 크기를 계산해야 하므로 어려운 문제로 알려져 있다.

LWE(Learning With Errors) 문제는 격자 기반 암호 시스템에서 가장 널리 이용하는 문제로 행렬 A와 벡터 $b=As+e$가 주어졌을 때, 벡터 s를 찾아내야 한다. 임의의 오류 e가 포함되어 있어서 이 문제는 어렵다고 알려져 있다. 하지만 이 문제를 계산하기 위해서 이차 행렬의 곱셈이 필요하므로 구현 측면에서의 어려움을 가지고 있다.

RLWE(Ring-Learning With Error) 문제는 SVP 문제를 기반으로 하며 SVP, SIVP보다 어려운 문제로 알려져 있다. RLWE는 행렬이 하나의 열로 표현되며 나머지 열의 경우 하나의 열을 이용하여 구할 수 있다. 이러한 경우 다항식의 곱셈으로 기존 이차 행렬의 곱셈을 해결할 수 있으므로 구현적인 측면에서 더욱 효율적이다.

LWR(Learning With Rounding) 문제는 LWE 문제의 변형된 형태이며 라운딩 연산을 통해 하위 비트를 잘라냄으로써 오류가 더해진 효과를 낸다.

2.4 LizarMong

2.4.1 설계 원리

LizarMong은 RLizard를 바탕으로 만들어진 알고리즘이다. RLizard는 NIST의 PQC 공모전 Round 1의 후보 알고리즘인 Lizard의 환(ring) 버전 알고리즘이다. RLizard의 $KeyGen$ 함수는 LWE 문제의 환 버전 문제인 RLWE(Ring-LWE) 문제를 이용하였으며 $Encrypt, Decrypt$ 함수는 LWR 문제의 환 버전 RLWR(Ring-LWR) 문제를 이용하였다. 기존의 LWE, LWR 문제의 경우 N차원 공간에 대하여 N^2의 계수를 저장할 공간을 필요로 하며 곱셈 역시 이를 이용하여 이루어진다. 환 버전에서는 N개의 계수만이 저장되며 이에 따른 연산 역시 줄어들 수 있어 효율적이다. 환 위에서 연산이 되는 경우 다항 환이 사용되며 LizarMong 알고리즘의 경우 $R_q := Z_q[X]/(X^n+1)$을 사용한다. 사용된 환은 현재까지 알려진 공격이 없으며 연산을 용이하게 한다는 장점을 가지고 있다.

2.4.2 표기법

$\{0,1\}^n$는 n개의 0 또는 1이 나열된 비트열을 말한다. 예를 들어, $\{0,1\}^{256}$의 경우 32바이트를 의미한다. 16진수의 표기를 위해서 숫자 앞에 0x를 사용한다. 예를 들어, 0x01의 경우 16진수로 표현된 1을 의미한다. SHAKE256($seed$,$outlen$)은 스펀지 구조 해시 함수로 임의의 입력과 임의의 출력이 가능한 함수이다. seed를 입력 받아 outlen 길이의 바이트를 출력한다. $HWT_n(h)$는 $\{-1,0,1\}^n$의 부분 집합에서 0이 아닌 값이 h개가 되도록 하는 집합들을 균일하게 뽑는 분포이다. ψ_{cb}는 중심이 0이고 표준 편차가 $\sqrt{cb/2}$인 분포로 -1, 0, 1이 뽑힐 확률이 각각 0.25, 0.5, 0.25이다. ‖는 연접을 의미하며 데이터를 이어 붙이는 것을 말한다. 환에서의 곱셈을 * 로 표기하며 스칼라 곱의 경우 · 으로 표기한다. « , » 는 비트 연산자

Shift를 의미하며 각각 Left Shift, Right Shift 연산을 의미한다. $\lfloor n \rceil$은 반올림을 의미하며 다항식에 사용될 경우 각 계수의 반올림을 의미한다. H', G는 출력의 길이가 각각 sd/8, d/8 바이트인 SHAKE256를 말하며 s와 d는 패러미터에 따라 결정된다. H는 h_r개의 0이 아닌 값을 갖도록 출력하는 SHAKE256이다. LizarMong은 128비트 안전성을 갖는 $Comfort$버전과 256비트 안전성을 갖는 $Strong$버전이 있다.

2.4.3 알고리즘

다음은 LizarMong KEM의 $KeyGen, Encapsulate, Decapsulate$의 상세 알고리즘이다.

LizarMong_KEM.KeyGen
Input: 1^k
Output: pk, sk
1. $Seed_a \leftarrow \{0,1\}^{256}$
2. $\boldsymbol{a} \leftarrow$ SHAKE256$(Seed_a, n/8)$
3. $\boldsymbol{s} \leftarrow HWT_n(h_s), \boldsymbol{u} \leftarrow \{0,1\}^n$
4. $\boldsymbol{e} \leftarrow \psi_{cb}$
5. $\boldsymbol{b} \leftarrow -\boldsymbol{a}^* \boldsymbol{s} + \boldsymbol{e}$
6. $pk \leftarrow Seed_a \| \boldsymbol{b}$
7. $sk \leftarrow \boldsymbol{s} \| \boldsymbol{u}$
8. return pk, sk

[알고리즘 2-1] LizarMong_KEM.KeyGen

LizarMong_KEM.Encapsulation

Input: pk

Output: c, K

1. $\delta \leftarrow R^2$
2. $\boldsymbol{r} \leftarrow H(\delta), \mathrm{d} \leftarrow H'(\delta)$
3. $\delta' \leftarrow eccENC(\delta)$
4. $\boldsymbol{c_{1a}} \leftarrow \lfloor (p/q) \cdot \boldsymbol{a}^* \boldsymbol{r} \rceil, \boldsymbol{c_{1b}} \leftarrow \lfloor (k/q) \cdot (q/2) \cdot \delta' + \boldsymbol{b}^* \boldsymbol{r} \rceil$
5. $\boldsymbol{c_1} \leftarrow \boldsymbol{c_{1a}} \| \boldsymbol{c_{1b}}$
6. $\mathrm{K} \leftarrow G(\boldsymbol{c_1}, \mathrm{d}, \delta)$
7. $\mathrm{c} \leftarrow \boldsymbol{c_1} \| \mathrm{d}$
8. return c, K

[알고리즘 2-2] LizarMong_KEM.Encapsulation

LizarMong_KEM.Decapsulation

Input: pk, sk, c

Output: K

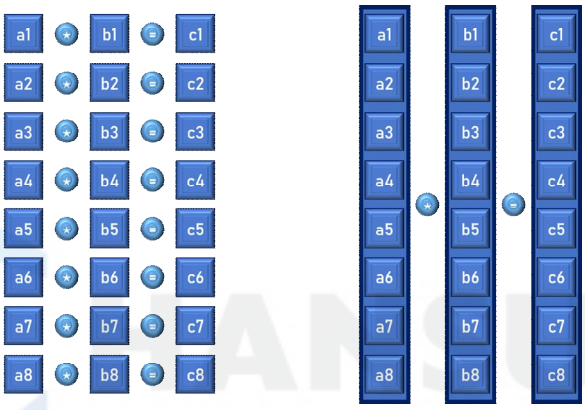
1. $\boldsymbol{c_{1a}}, \boldsymbol{c_{1b}}, \mathrm{d} \leftarrow \mathrm{c}$
2. $\hat{\boldsymbol{c}}_{\boldsymbol{1b}} \leftarrow \boldsymbol{c_{1b}} \ll (\log p - \log k)$
3. $\hat{\delta}' \leftarrow \lfloor (2/p) \cdot (\hat{\boldsymbol{c}}_{\boldsymbol{1b}} + \boldsymbol{c_{1a}}^* \boldsymbol{s}) \rceil$
4. $\hat{\delta} \leftarrow eccDEC(\hat{\delta}')$
5. $\hat{\mathrm{r}} \leftarrow H(\hat{\delta}), \hat{\mathrm{d}} \leftarrow H'(\hat{\delta}), \hat{\delta}'' \leftarrow eccENC(\hat{\delta})$
6. $\hat{\boldsymbol{c}} \leftarrow \lfloor (p/q) \cdot \boldsymbol{a}^* \hat{\boldsymbol{r}} \rceil \| \lfloor (k/q) \cdot ((q/2) \cdot \hat{\delta}'' + \boldsymbol{b}^* \hat{\boldsymbol{r}} \rceil \| \mathrm{d}$
7. if $\mathrm{c} \neq \hat{\mathrm{c}}$ then
8. $\quad \mathrm{K} \leftarrow G(\boldsymbol{c}, \mathrm{d}, \boldsymbol{u})$
9. else
10. $\quad \mathrm{K} \leftarrow G(\boldsymbol{c}, \mathrm{d}, \hat{\delta})$
11. endif
12. return K

[알고리즘 2-3] LizarMong_KEM.Decapsulation

2.5 SIMD(Single Instruction Multiple Data)

2.5.1 AVX2(Advanced Vector Extenstion 2)

하나의 명령어를 이용하여 다수의 데이터를 동시에 처리하는 것을 SIMD(Single Instruction Multiple Data)라고 한다.



[그림 2-2] SIMD 개요

주로 컴퓨터 비전 분야 등에서 사용되는 기능이지만 각 비트, 바이트 혹은 워드에 동일한 연산을 반복하는 암호 분야에도 SIMD의 적용이 가능하다. CUDA와 같이 GPU 상에서 동작하는 병렬처리 명령어가 존재하지만 GPU에 종속성을 갖는 단점이 존재한다. AVX 시리즈는 CPU상에서 돌아가며 Intel 아키텍처의 대부분에서 지원하는 병렬처리를 위한 명령어 집합이다. Intel 프로세서는 SIMD 연산을 수행하는 확장된 명령어 집합인 AVX를 지원한다. SIMD를 위한 확장 명령어 집합은 AVX, AVX2, AVX512 순으로 발달해 왔다. AVX의 경우 128비트 레지스터를 지원하며 Sandy Bridge 프로세서에 탑재되어 2010년부터 지원되고 있다. AVX2의 경우 256비트 레지스터를 지원하며 Haswell 프로세서에 탑재되어 있으며

2013년부터 지원되고 있다. AVX512의 경우 512비트 레지스터를 지원하며 2016년 이후의 프로세서에 탑재되고 있으나 2019년 발표된 Ice Lake 프로세서부터 대부분의 병렬 연산자를 사용할 수 있기 때문에 아직 상용화가 되었다고 보기 어렵다. 본 논문에서는 범용성을 고려하여 AVX2를 사용하며 NIST Round2 후보 알고리즘들 역시 병렬 구현을 위해 AVX2 구현을 사용하고 있다. AVX2의 경우 병렬 연산을 위한 256비트의 CPU 레지스터를 사용할 수 있는데 이는 한 번의 명령으로 8비트 데이터 32개, 16비트 데이터 16개를 동시에 처리할 수 있는 수준이다. LizarMong에서는 8비트 데이터를 단위로 사용하기 때문에 이론적으로는 한 번에 32개의 데이터를 처리할 수 있다. 하지만 컴퓨터 그래픽 기술의 발전 등으로 8비트 데이터를 사용하는 사례는 줄어들고 있으며, 이에 따라 8비트의 병렬 연산을 위한 연산자를 지원하지 않는다. 특히 곱셈과 같은 기본 연산이 16비트부터 지원되기 때문에 이를 고려한 구현이 필요하다. AVX2의 명령어 접근을 위해 인텔에서는 라이브러리 Intrinsics를 제공하고 있다.

2.5.2 Intrinsics

Intrinsics 라이브러리는 컴파일러가 직접 접근할 수 있는 내장 함수로 별도의 설치가 필요하지 않으며 단순히 include하는 방식으로 사용할 수 있다. Intrinsics 라이브러리는 AVX2 사용을 위해서 새로운 자료형을 제공한다. 그 중 256비트 레지스터를 위한 자료형으로 __m256, __m256d, __m256i가 있다. __m256은 32비트 부동소수점 표현을 위한 자료형이고 __m256d는 64비트 부동소수점을 표현하기 위한 자료형이다. __m256i는 정수의 표현을 위한 자료형으로 32개의 8비트 정수, 16개의 16비트 정수, 8개의 32비트 정수 그리고 4개의 64비트 정수가 들어갈 수 있다. LizarMong에서는 소수점을 이용하지 않기 때문에 __m256i를 사용한다.

또한 레지스터 간 병렬 연산을 위한 명령어를 제공하는데, 명령어의 경우 아래와 같은 형태로 구성된다.

$$\_mm256\_<op>\_<suffix>(<type><param1>,\ <type><param2>,\ \ldots\ )$$

_mm256의 경우 연산 결과의 자료형을 표기하며 대부분의 경우 256비트 레지스터를 입력 받아 동일한 256비트 결과를 출력하지만 비교 연산과 같은 특정한 연산의 경우 128비트를 출력하기도 한다. 128비트를 출력하는 경우 _mm128이 된다. <op>는 명령어를 의미한다. 덧셈을 의미하는 add, 곱셈을 의미하는 mul 등이 있다. <suffix>는 입력으로 받은 레지스터가 구성하고 있는 데이터의 형태를 의미하며 이를 고려하여 명령어를 실행한다. 예를 들어, epi32의 경우 32비트로 묶인 정수 데이터가 레지스터를 구성하고 있다는 의미이며 8개의 32비트 정수 데이터를 병렬로 처리한다. 패러미터로 들어가는 <type>과 <params>의 경우 입력 데이터를 의미하며 자료형과 입력 변수를 입력한다. 이러한 자료형과 명령어를 사용하여 Intel 아키텍처의 CPU 상에서 SIMD 연산을 사용할 수 있으며 병렬처리를 구현할 수 있다.

Ⅲ. 연구 내용

3.1 주요 연산 최적화

3.1.1 비밀(s) 추출 및 정렬 (기존)

RLWE 문제의 정의에 따르면, 문제에 이용하는 비밀 s와 오류 e는 본래 서로 다른 분포에서 추출되어야 한다. 하지만 동일한 분포에서 추출하여도 안전성이 증명되었으나, 최근 부채널 분석 공격 방법인 오류 주입 공격으로부터 동일한 분포를 사용하는 것이 더 이상 안전하지 않다는 것이 밝혀졌다. LizarMong에서는 s, e의 값의 추출에 다른 분포를 이용하여 이러한 공격을 방지한다.

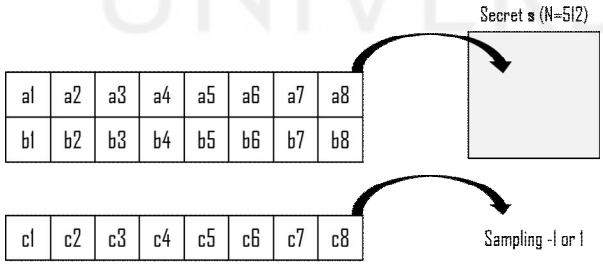
LizarMong에서는 비밀 s의 추출에 분포 $HWT_n(h)$가 사용된다. −1, 1의 개수에 가중치를 두어 h개의 가중치를 갖는 n길이의 비밀 s를 추출하게 된다. 추출 함수에서는 먼저 h*4개의 무작위 바이트를 추출한다. 이렇게 추출된 바이트는 두 가지 목적으로 사용된다. 먼저 -1, 0, 1을 추출하는 데 사용된다. 이러한 경우 추출된 바이트의 하위 2비트에서 1을 빼는 기법을 이용한다. 하위 2비트의 수 표현 범위는 0에서 2의 범위를 표현하게 되며 1을 빼면 목표로 하는 -1에서 1의 범위를 표현하게 된다. 다음으로, 추출된 -1, 0, 1이 n차원 공간을 표현하는 배열의 몇 번째 원소에 해당하는지 표현하는데 이용한다. 각 바이트는 0에서 255의 범위를 가진다. 하지만 n은 $Comfort$와 $Strong$에서 각각 512, 1024를 갖는다. 이는 부호 비트를 제외하면 9비트, 10비트를 이용해서 표현 가능하며 따라서 하나의 바이트로는 표현할 수 없다. 이를 해결하기 위해 두 바이트를 연접하여 사용한다. 두 바이트를 연접하면 부호 비트를 제외하고 0에서 2^{16}까지의 수를 표현할 수 있으므로 n차 배열 원소들의 인덱스를 무작위로 추출하는 것이 가능하다. 무작위로 추출한 인덱스와 -1, 1값을 맵핑하면 h개의 쌍이 나오며 n차 배열에서 h개의 쌍에 해당하지 않는 $(n-h)$개 값의 경우

0으로 간주할 수 있다. 예를 들어 $Comfort$의 경우 n이 512, h가 128로, 512개 원소 중 128개가 맵핑 쌍으로 표현될 수 있으며 -1 또는 1의 값을 갖는다. 나머지 384개 원소의 경우 0으로 간주한다.

LizarMong에서는 비밀 $\boldsymbol{s}$를 하나의 배열에 정렬한다. 1의 값을 갖는 맵핑 쌍을 정렬하여 앞쪽에 -1의 값을 갖는 맵핑 쌍을 정렬하여 뒤쪽에 두며 -1 맵핑 쌍이 시작되는 지점의 인덱스를 저장하여 구분한다. 맵핑 쌍만을 정렬하여 저장하는 것이므로 배열의 크기 역시 h가 된다.

3.1.2 비밀($\boldsymbol{s}$) 추출 및 정렬 (변경)

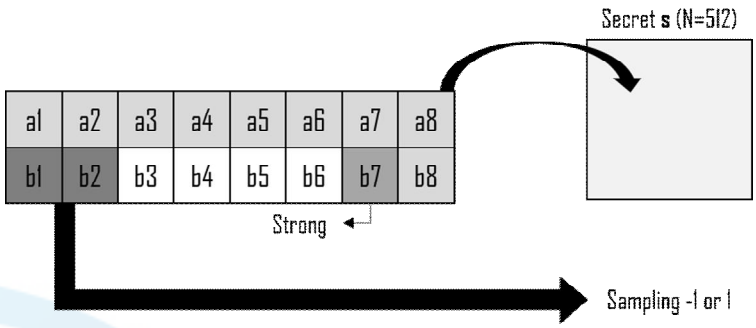
LizarMong에서는 비밀 $\boldsymbol{s}$의 추출을 위해 분포 $HWT_n(h)$를 사용하였다. 대부분의 값이 0으로 존재하는 희소 다항식의 표현을 위해서 0이 아닌 값의 인덱스만을 이용하여 표현하였으며 -1, 1의 값을 구분하기 위해 구분용 인덱스를 두었다. 하나의 값 추출을 위해 세 개의 무작위 바이트가 필요했으며 아래 그림과 같이 표현할 수 있다.



[그림 3-1] LizarMong 비밀 추출

먼저 두 바이트를 이용하여, 추출할 값이 $N-1$차 다항식인 비밀 $\boldsymbol{s}$의 몇 번째 인덱스에 포함될 것인지를 정한다. $Comfort$과 $Strong$의 N의 값은 각각 512, 1024이므로 인덱스를 표현하기 위해 각각 9개, 10개의 비트가 필요했다. 그런 다음 나머지 한 바이트를 이용하여 추출할 값이 -1 인지 1인지를 결정한다. 따라서 비밀 $\boldsymbol{s}$의 0을 제외한 한 원소를 추출하기 위해

3개의 무작위 바이트가 사용되었으며 h개의 원소가 필요하므로 기본 3· h개의 무작위 바이트가 필요하다. 여기서 LizarMong은 무작위 바이트로부터 발생하는 인덱스의 중복을 고려하여 h개의 무작위 바이트를 더 추출하며 4· h개의 무작위 바이트가 사용되었다. 무작위 바이트의 추출을 최적화하는 기법은 아래와 같다.



[그림 3-2] LizarMong 비밀 추출 최적화

인덱스를 표현하기 위한 무작위 비트는 *Comfort* 기준 9비트 이므로 하나의 무작위 바이트와 또 하나의 무작위 비트로 표현할 수 있다. *Strong*의 경우 10비트가 필요하므로 하나의 무작위 바이트와 두 개의 무작위 비트로 표현할 수 있다. 여기에 -1 과 1의 값을 추출하기 위해서는 2개의 무작위 비트면 충분하다. 2개의 비트로 표현될 수 있는 수는 0x00, 0x01, 0x02, 0x03이다. 모든 경우의 수를 비트 연산자 And를 이용하여 0x02와 연산하면 0.5의 확률로 0 또는 2의 결과가 나오며, 여기에 1을 빼 준다면 -1 또는 1의 값이 나오게 된다. 이를 이용하면 3개의 무작위 바이트가 아닌 2개의 무작위 바이트만으로 0이 아닌 비밀 s의 원소를 추출할 수 있다. 그림과 같이 *Comfort*의 경우 a1:a8 비트와 LSB인 b8 비트를 이용하여 인덱스를 표현하며, 기존에 사용되지 않던 b1비트와 b2비트를 이용하여 -1, 1을 추출한다. *Strong*의 경우에는 b7 비트를 추가로 이용하면 인덱스의 범위를 모두 표현할 수 있다. 이러한 방식을 적용하게 되면 하나의 원소 추출에 필요한 무작위 바이트가 3개에서 2개로 줄어든다. 기존의

LizarMong 알고리즘과 같이 중복을 고려하더라도 3· h개의 무작위 바이트면 비밀 $\boldsymbol{s}$의 추출이 가능하다. 무작위 바이트를 쪼개어 활용하는 데에 필요한 연산들이 있지만, 고품질의 무작위 비트열을 뽑는 것이 저전력 디바이스에서는 어려운 등 환경에 따라 무작위 비트열 추출에 큰 비용이 들 수 있으므로 필요한 무작위 바이트의 수를 줄이는 것은 의미가 있으며 어떤 난수 생성기를 사용하느냐에 따라 큰 성능 개선을 기대할 수 있다.

비밀 $\boldsymbol{s}$의 경우 동일한 인덱스가 뽑혔을 경우 추출된 값을 버리게 된다. 따라서 매 추출마다 값의 중복을 검사하게 된다. 동일한 인덱스가 추출될 확률은 $Comfort$, $Strong$의 경우 모두 적지만, 중복을 허용하지 않기 때문에 매번 검사하는 과정을 병렬처리하기 어렵다. 이에 비밀 $\boldsymbol{s}$를 미리 뽑아 놓고 이를 참조하여 사용하는 경우를 생각할 수 있지만 더 큰 비용이 들었다. 따라서 비밀 $\boldsymbol{s}$ 추출에는 병렬화를 적용하지 않으며 사용되는 난수의 수만 줄일 수 있었다.

3.1.3 오류($\boldsymbol{e}$) 추출 (기존)

RLWE 문제의 정의에 따르면, 오류를 정규 분포에서 추출한다. 하지만 정규 분포의 사용은 구현이 어려우며 성능적인 측면에서도 큰 부담을 갖는다. 따라서 많은 격자 암호 알고리즘에서는 정규 분포의 사용을 정규 분포에 근사하는 이항 분포를 사용하는 방식으로 변형하여 사용하고 있다. 변형하는 대표적인 방법으로 사전에 계산된 값을 참조하는 LUT(Look Up Table)방법이 있으며 이를 CDT(Cumulative Distribution Table)라고 한다. 정규 분포의 확률과 근사하도록 오류 $\boldsymbol{e}$를 추출할 수 있는 테이블을 이용하는 방식이다. 정규 분포를 계산하고 추출하는 방식과 달리, 난수를 인덱스로 테이블을 참조하면 정규 분포에 근사한 확률로 추출된 오류 $\boldsymbol{e}$를 얻을 수 있다. 이를 구현하는 방식은 다음과 같다. 먼저, 확률이 사전 계산된 테이블을 준비한다. 추출할 수의 범위에 따라 각 수가 뽑힐 확률을 정하고 이에 정밀도를 고려하여 테이블을 준비할 수 있다. 예를 들면 0,

1, 2에 대한 범위에 맞추어 확률 0.25, 0.5, 0.25를 정한다. 만약 정밀도를 100으로 가정한다면, $\{25, 75, 100\}$의 CDT를 준비할 수 있다. 이 테이블에 난수 x를 대입하면 x가 속해있는 범위에 따라 값을 추출할 수 있다. 예를 들어, 난수 30을 이용하면, 30은 $[25, 75)$에 속하므로 1을 추출할 수 있다. 이와 같은 방식으로 추출된 난수에 따라 정규 분포의 확률에 근사하는 값을 추출할 수 있다. 추출하려는 값의 범위와 정밀도를 결정할 수 있으며, 중심 극한 정리에 따르면, 값의 범위와 정밀도를 충분히 크게 한다면, 이항 분포는 정규 분포에 근사할 수 있다. 이를 통해 많은 격자 기반 암호 알고리즘들은 정규 분포를 구현하지 않고 효율적으로 RLWE 문제를 사용할 수 있었다. 하지만 CDT 방법에 대한 부채널 분석 기법으로서 전력 분석 방법 및 캐시 공격이 제안됨에 따라 LizarMong에서는 중심 이항 분포를 사용한다. 중심 이항 분포는 높은 정밀도를 사용할 경우 구현에 적합하지 않지만, KEM 모델에서의 오류 $\boldsymbol{e}$ 추출에는 높은 정밀도를 필요로 하지 않으며, KEM 환경에서의 안전성 역시 검증되었다. 중심 이항 분포를 ψ_{cb}라고 표기하며 ψ_{cb}는 평균이 0, 표준 편차가 $\sqrt{cb/2}$가 되는 분포이다. cb의 경우 LizarMong의 $Comfort$와 $Strong$에서 모두 1의 값을 가진다. 따라서 LizarMong에서는 -1, 0, 1의 값이 0.25, 0.5, 0.25에 확률에 근사하도록 추출되는 분포가 사용된다. LizarMong에서는 최소한의 난수를 사용하여 ψ_1을 구현하기 위해, $\{0,1\}^8$의 무작위 비트열을 사용하여 4개의 오류 $\boldsymbol{e}$를 추출한다. 8개의 비트를 인접한 비트끼리 묶어 4개의 쌍으로 표시한다. 그리고 묶인 쌍의 원소들끼리 뺄셈을 진행한다. 4개의 쌍에 대한 뺄셈 결과는 아래와 같이 나타낼 수 있다.

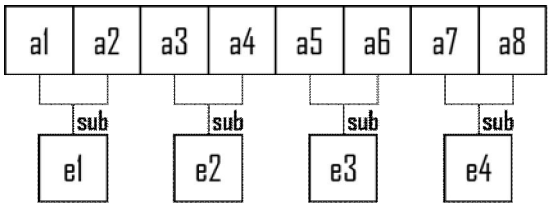
[표 2-2] 두 비트의 뺄셈에 대한 경우의 수

Bit 1	Bit 2	Bit 1 - Bit 2
0	0	0
0	1	-1
1	0	1
1	1	0

2개의 비트를 뺄셈하여 나올 수 있는 경우의 수는 -1, 0, 1이다. 각 경우의 수가 나올 확률 역시 무작위 비트열의 난수성이 보장된다면 0.25, 0.5, 0.25가 될 것이다. 따라서 무작위 비트열 하나를 이용하여 4개의 추출을 할 수 있다. 무작위 비트열을 두 비트씩 사용하기 위해서 비트 연산자가 사용된다. 먼저 비트열과 0x01에 비트 연산자 And를 사용하여 최하위비트(LSB, Least Significant Bit)를 추출한다. LSB는 0 혹은 1이 될 수 있다. 다음으로 비트 연산자 Shift를 이용한다. » 를 이용하여 비트열의 일곱 번째 비트를 여덟 번째 비트로 이동시킬 수 있으며 기존의 여덟 번째 비트는 제거되고 일곱 번째 비트로 새롭게 갱신된다. 이후 재차 0x01과의 And 연산을 통해 LSB를 추출하면 두 개의 무작위 비트열을 획득할 수 있다. 이러한 방식으로 하나의 오류 $\boldsymbol{e}$를 추출하게 되며 $\{0,1\}^8$에 이를 4번 반복하여 4개의 오류 $\boldsymbol{e}$를 추출할 수 있다.

3.1.4 오류($\boldsymbol{e}$) 추출 (변경)

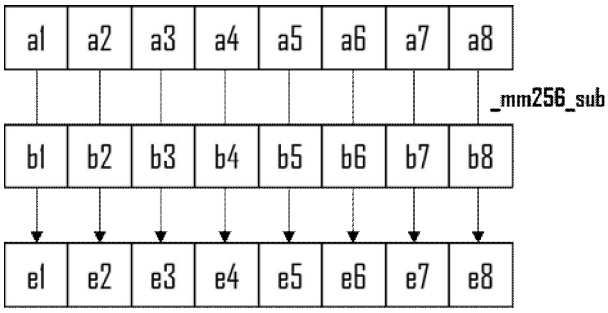
LizarMong에서는 오류 $\boldsymbol{e}$의 추출을 위해 비밀 $\boldsymbol{s}$의 추출과 다른 분포인 분포 ψ_1를 사용하였다. 최소한의 무작위 비트열을 사용하여 구현하기 위해 인접하는 두 비트에 대해 뺄셈을 하여 구현한다. 이에 대한 그림은 아래와 같다.



[그림 3-3] LizarMong 오류 추출

인접하는 두 비트에 대한 뺄셈으로 오류 다항식 $\boldsymbol{e}$의 한 원소를 추출한다. 이러한 방식으로 하나의 바이트를 이용하여 4개의 오류를 추출할 수 있다. 이를 병렬화로 구현하기 위해 intrinsic을 활용할 수 있다. 효율적인

병렬 처리를 위하여 아래 그림과 같은 방법을 생각할 수 있다.



[그림 3-4] LizarMong 오류 추출 최적화

LizarMong에서 구현된 방식은 인접하는 두 비트 간의 뺄셈을 이용하여 오류 e를 추출하는 것이었다. 하지만 난수발생기가 충분한 난수성을 제공하며 사용되지 않은 비트를 사용한다면 분포의 확률과 안전성에는 차이가 없다. 따라서 하나의 무작위 바이트의 인접한 비트들을 활용하는 기존의 방법 대신 두 개의 무작위 바이트를 병렬로 뺄셈하는 방식을 이용한다. 기존의 방법의 경우 총 두 번의 비트 연산자 Shift를 사용하여 바이트 당 두 개의 피연산자를 추출할 수 있었다. 이렇게 추출된 피연산자는 병렬 연산을 위해 다시 레지스터에 저장되어야 했다. 하지만 제안하는 방식은 각 바이트에서 추출된 피연산자들이 레지스터에 저장된 상태로 별도의 저장을 하지 않고도 병렬 연산이 가능하다. 기존의 인접한 비트 간의 뺄셈을 병렬화 적용하였을 때는 전처리를 위한 비용이 더욱 커져 오히려 성능이 감소하였다. 두 바이트를 사용하는 경우 동일한 안전성을 제공하면서 병렬화의 이점을 취할 수 있었다. 또한 바이트에 Shift 연산을 적용하는 설계 원리를 따라 병렬화 구현을 진행하면서 AVX2가 8비트 Shift 연산자를 지원하지 않는 제약이 존재하였다. 하지만 최상위 비트를 최하위 비트로 끌어오기 때문에 Right Shift 연산이 적용되면서 생기는 더미 비트들이 연산에 참여하지 않는다는 점을 확인할 수 있었고 이를 이용하여 AVX2가 지원하

는 16비트 Shift 연산자를 적용하여 병렬화 구현을 적용할 수 있었다.

3.1.5 다항식의 차원 축소 (Reduction) (기존)

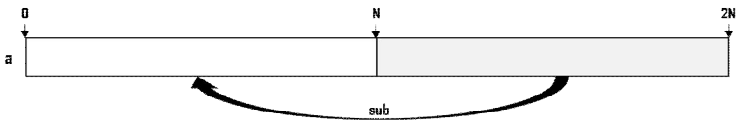
두 $N-1$차 다항식 환 $\mathbf{a}$, $\mathbf{b}$에 대한 다항식의 덧셈은 아래 식과 같이 표현될 수 있다.

$$\mathbf{a}+\mathbf{b}=\sum_{n=0}^{N-1}(a_n+b_n)x^n$$

이러한 경우 결과 다항식 역시 N차 다항식으로 패러미터의 범위를 초과하지 않는다. 하지만 다항식의 곱셈이 일어나게 되면 환의 패러미터의 범위를 초과하는 값을 가질 수 있다. 이러한 값을 지정된 범위 내로 줄이는 과정이 필요한데, 이러한 과정을 Reduction이라고 한다. 예를 들어, N차원의 다항식을 표기하기 위해 $N+1$개의 계수가 필요할 것이다. N차원의 두 다항식의 곱셈에 대한 결과는 $2N+2$개의 계수를 갖게 되므로 $2N+1$차원의 다항식이 된다. 이를 다시 N차원의 다항식으로 줄이는 과정인 것이다. 다항식 간의 덧셈의 경우 차원에 대한 Reduction이 필요하지 않지만 곱셈의 경우 차원을 지정된 범위 내로 줄이는 과정이 필요하다. LizarMong에서는 환 $R_q := Z_q[X]/(X^n+1)$을 사용하기 때문에, 동치 $X^n \equiv -1$를 이용하여 Reduction이 가능하다. 예를 들어, 두 다항식을 곱셈한 결과인 다항식 $\mathbf{x}= \{5x^0+3x^1,...,x^{n-1},2x^n+x^{n+1}\}$에 대하여, $X^n \equiv -1$을 이용한 두 개의 동치 $2x^n \equiv -2x^0$, $x^{n+1} \equiv -x^1$를 적용하면 $\mathbf{x'}=\{3x^0+2x^1,...,x^{n-1}\}$로 결과 다항식을 N차원의 범위 내로 제한할 수 있으며 두 결과 다항식은 동치 $\mathbf{x} \equiv \mathbf{x'}$이다.

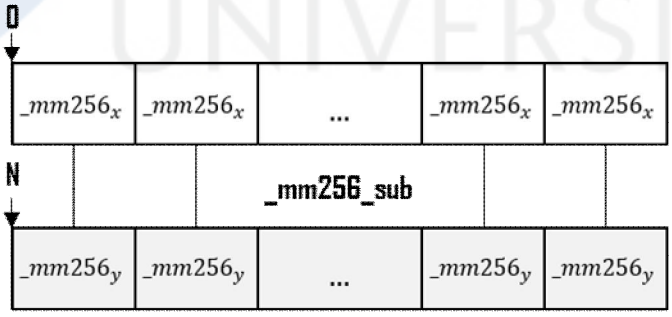
3.1.6 다항식의 차원 축소 (Reduction) (변경)

다항식의 차원 축소의 경우 병렬화의 구현이 간단하게 이루어질 수 있다. 환 $R_q := Z_q[X]/(X^n+1)$에 대한 차원 축소의 경우 뺄셈을 통해 이루어질 수 있으며 아래 그림과 같이 표현될 수 있다.



[그림 3-5] LizarMong 차원 축소

$N-1$차 다항식 **a**의 경우 동일한 $N-1$차 다항식과의 곱셈을 통해 $2N-1$차 다항식으로 확장될 수 있다. 확장되는 차수의 항은 동치 $X^n \equiv -1$에 따라 차원 축소가 일어나는데, $N+\alpha$차 항의 경우 α차 항과의 뺄셈으로 일어날 수 있다. 따라서 초기 N개의 항과 나머지 N개 항의 뺄셈을 병렬로 진행하여 성능 최적화를 할 수 있다.



[그림 3-6] LizarMong 차원 축소 병렬화

확장된 다항식의 초기 N개 원소를 첫 번째 피연산자로 두고 나머지 N개 원소를 두 번째 피연산자로 둔 후 뺄셈을 통해 구현할 수 있다. 결과값은 따로 저장하지 않고 초기 N개 원소에 덮어씌운 뒤 나머지 N개는 사용하지 않는다. 이러한 방식으로 병렬화 적용하여 초기 N개에 차원 축소 결과

를 얻을 수 있다.

3.1.7 희소 삼진 다항식의 곱셈 (기존)

환 $R_q := Z_q[X]/(X^n+1)$에서의 곱셈은 일반적인 다항식의 곱셈을 먼저 진행한 뒤 그 결과에 대하여 상술한 차원 축소를 통해 범위를 제한한다. LizarMong에서는 차원이 동일한 두 $N-1$차 다항식의 곱셈을 사용하며, 두 $N-1$차 다항식 환 $\mathbf{a}$, $\mathbf{b}$에 대한 다항식의 곱셈은 아래 식과 같이 표현될 수 있다.

$$\mathbf{a}*\mathbf{b} = \sum_{n=0}^{2N-1}\sum_{k=0}^{n}(a_n \cdot \ b_{n-k})x^n$$

이후 $X^n \equiv -1$를 이용하여 차원 축소를 실행한다.

$$reduction(\mathbf{c}) = \sum_{n=0}^{N-1}(c_n - c_{n+N})x^n$$

이를 통해, 두 다항식 환을 곱셈이 일어날 수 있다. LizarMong에서는 이러한 다항식 환의 곱셈을 사용하지만 곱해지는 다항식의 특성을 고려하는 곱셈 기법을 사용한다.

LizarMong의 비밀 $\boldsymbol{s}$값은 대부분의 원소가 0이며 h개의 -1 또는 1의 값이 존재하는 희소 다항식이다. 비밀 $\boldsymbol{s}$값을 n차 배열에 모두 저장하는 것이 아닌 h개의 맵핑 쌍만을 h차 배열에 정렬하여 저장하는 이유는 메모리 낭비의 문제도 있지만 비밀 $\boldsymbol{s}$를 구성하는 원소들의 특성에 맞는 곱셈을 적용할 수 있기 때문이다. 비밀 $\boldsymbol{s}$는 -1, 0, 1의 원소만으로 이루어지는데 이를 균형 삼진 다항식이라고 한다. 또한 다항식의 대부분의 원소가 0으로 이루어지는데 이를 희소 다항식이라고 한다. 이러한 경우 다항식 환의 곱셈을 변형하여 적용할 수 있으며 아래 알고리즘과 같이 표현될 수

있다.

Sparse Polynomial Multiplication

Input: $a = a[0] \cdot x^0, \ldots, a[n-1] \cdot x^{n-1}$
$s = x^{r[0]}, \ldots, x^{r[indicator-1]}, -x^{r[indicator]}, \ldots, -x^{r[h_s]}$

Output: $b = a \cdot s = b[0] \cdot x^0, \ldots, b[n-1] \cdot x^{n-1}$

```
13. for i = 0 to h_s − 1 do
14.     for j = 0 to n − 1 do
15.         b[r[i] + j] −= (2 · MSB of (i − indicator) − 1) · a[j]
16.     end for
17. end for
18. for i = 0 to n − 1 do
19.     b[i] −= b[n + i]
20. end for
```

[알고리즘 3-1] 희소 다항식 곱셈

먼저 두 다항식 a, s를 입력으로 이용한다. 다항식 a는 $N-1$차의 다항식이며 다항식 s는 분포 $HWT_n(h)$로부터 추출된 비밀 다항식이다. 비밀 s는 희소 다항식으로 대부분의 값이 0이기 때문에 효율적인 저장을 위해 맵핑 쌍을 저장을 하며, 이 값은 배열 r로 표현되고 있다. $indicator$은 배열 r이 표현하고 있는 인덱스와 맵핑된 값이 1에서 -1로 바뀌는 지점을 이야기한다. 따라서 $indicator$를 넘어가는 시점에서 비밀 다항식 s의 부호가 음수로 바뀐다. 이렇듯 다항식 s는 새롭게 정렬된 맵핑 쌍으로 입력을 받게 된다. 이러한 입력을 받아 희소 다항식의 곱셈을 시작한다. 균형 삼진 다항식의 원소들을 고려하면 다항식 a와 s가 곱해질 때 -1, 0, 1의 중간값이 발생할 수 있다. 이 중 0의 경우 계산 결과에 영향을 주지 않으므로 0을 제외한 -1, 1의 값만을 고려하며 따라서 h개의 맵핑 쌍만을 계산하게 된다. h개의 맵핑 쌍은 인덱스 $indicator$를 기준으로 정렬이 되어 있으므로 이를 통해 -1 또는 1의 값을 추출할 수 있다. 만약 배열 r의 인덱스 i가 $indicator$보다 작다면 양수인 경우에 속하며 $(i-indicator)$의

값은 음수가 나올 것이다. 음수의 MSB의 경우 1이 나오게 되며 따라서 $(2 \cdot MSB \text{ of } (i-indicator)-1)$의 값은 1이 나오게 된다. 이 중간값을 계수에서 빼주면 음수 계수를 갖는 다항식 s의 원소들이 처리된다. 동일한 방법으로 인덱스 i가 $indicator$보다 크게 되면 음수로 $(i-indicator)$값이 양수가 나온다. 양수의 MSB는 0이며 $(2 \cdot MSB \text{ of } (i-indicator)-1)$의 값은 -1이 된다. 이 중간값을 계수에서 빼주면 양수 계수를 갖는 다항식 s의 원소들이 처리된다. 다항식 s에 존재하는 h개의 맵핑쌍이 모두 처리되었고 나머지 $n-h$개 원소의 경우 값이 0으로 중간값을 무시할 수 있다. 따라서 모든 원소에 대한 곱셈이 완료되었다. 다음으로, 차원 축소를 진행한다. 곱셈 결과 b의 차수가 $2N-1$까지 증가하였으므로 이에 대해서 $X^n \equiv -1$에 대한 차원 축소를 진행하여 곱셈을 처리한다.

3.1.8 희소 삼진 다항식의 곱셈 (변경)

두 다항식의 곱셈의 경우, 계수간의 곱셈이 일어나며, 동일한 차수를 가진 항 간의 덧셈으로 이루어진다. 하지만 intrinsic에서는 LizarMong에서 사용하는 8비트 곱셈을 지원하는 명령어가 존재하지 않아서 이를 이용한 병렬화 구현은 적합하지 않다. 그렇지만 LizarMong에서 곱셈에 사용되는 다항식 삼진 다항식으로 계수가 1또는 -1로만 이루어지는 특수한 경우의 다항식이기 때문에 계수간의 곱셈을 제거하고 동일한 차수 항의 덧셈만으로 구현될 수 있다. 하지만 그러한 경우에도 부호처리를 위한 두 번의 곱셈이 필요하다. 자료형태를 고려하여 부호 처리를 두 반복문으로 대체하면 곱셈을 제거할 수 있으며 intrinsic을 사용한 병렬처리가 가능하게 된다. intrinsic을 사용한 새로운 알고리즘은 아래와 같다.

Sparse Polynomial Multiplication for AVX2

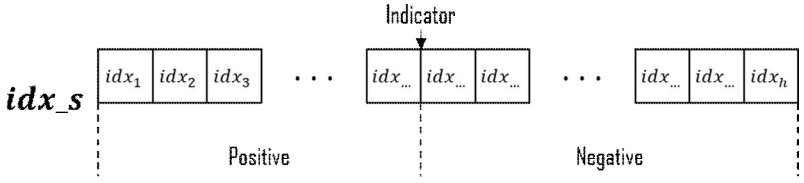
Input: $a = a[0] \cdot x^0, \ldots, a[n-1] \cdot x^{n-1}$
$s = x^{r[0]}, \ldots, x^{r[indicator-1]}, -x^{r[indicator]}, \ldots, -x^{r[h_s]}$

Output: $b = a \cdot s = b[0] \cdot x^0, \ldots, b[n-1] \cdot x^{n-1}$

1. for $i = 0$ to $indicator - 1$ do
2. for $j = 0$ to $n-1$ do
3. $b[r[i]+j] -= a[j]$
4. end for
5. end for
6. for $i = indicator$ to h_s do
7. for $j = 0$ to $n-1$ do
8. $b[r[i]+j] += a[j]$
9. end for
10. end for
11. for $i = 0$ to $n-1$ do
12. $b[i] -= b[n+i]$
13. end for

[알고리즘 3-2] AVX2 병렬화를 위한 변형 곱셈

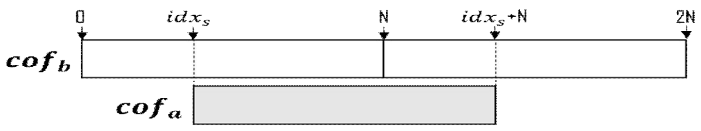
먼저 입력으로 두 다항식 a와 s가 주어진다. 다항식 s의 경우 기존 알고리즘과 동일하게 h개의 인덱스와 값의 맵핑 쌍으로 표현되며 인덱스 $indicator$를 기준으로 양의 부호와 음의 부호로 정렬되어 있으며 아래와 같다.



[그림 3-7] 비밀 다항식 s를 표현하는 정렬된 데이터

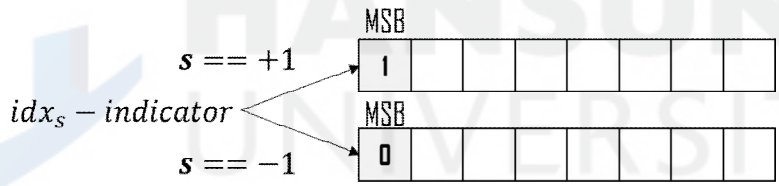
이렇듯 다항식 s를 표현하는 정렬된 데이터 idx_s를 사용하여 곱셈을 진

행한다.



[그림 3-8] idx_s를 활용한 곱셈 중간값 발생

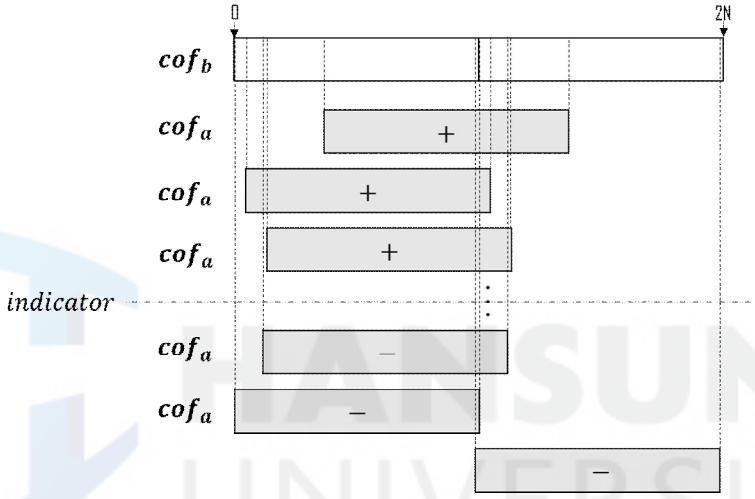
다항식 s의 각 원소는 곱셈 과정에서 N개의 중간값을 발생시킨다. 발생되는 중간값의 계수는 다항식 s의 계수와 곱해지는 다항식 a의 계수에 의해서 결정되는데 다항식 s의 계수가 1 또는 -1이기 때문에 중간값의 절대값은 다항식 a의 계수만으로 결정된다. 이 계수는 결과 다항식 b의 계수로 저장된다. 중간값의 계수가 양수인 +1인 경우, 다항식 a의 계수는 더해진다. 계수가 음수인 -1인 경우, 다항식 a의 계수는 빼진다.



[그림 3-9] 중간값의 부호 결정

기존의 경우 이 부호를 결정하기 위해 MSB를 이용하였다. 정렬된 데이터는 $indicator$를 기점으로 양수에서 음수로 변한다. 즉, idx_s가 가리키는 인덱스가 구분을 위한 인덱스 $indicator$보다 작을 경우 idx_s와 맵핑된 값은 양수가 된다. 반대로 idx_s가 가리키는 인덱스가 더 크다면 인덱스와 맵핑된 값은 음수가 된다. 이를 처리하기 위해 $idx_s - indicator$의 값을 확인하는 방식을 사용하는데, 맵핑된 값이 양수라면 $idx_s - indicator$은 음수로 MSB가 1이 된다. 반대의 경우 MSB는 0이 된다. 이러한 방법으로 조건문을 사용하지 않고 양수, 음수를 판별할 수 있다. [그림 3-9]와 같이 판별 결과에 따라 중간값으로 사용될 다항식 a는 더해지거나 빼지게 된다. 부

호의 설정을 위해서 $(2\cdot MSB)-1$의 식을 이용하는데, 상술한 내용에 따라 양수의 경우 +1 음수의 경우 -1의 값이 나오게 된다. 이렇듯 부호의 생성에 곱셈을 사용하고 이를 다항식 a에 적용하기 위해 곱셈을 사용하게 되어 총 두 번의 곱셈이 사용된다. 하지만 intrinsic에서는 8비트 곱셈을 위한 명령어를 지원하지 않기 때문에 병렬화를 적용하기 어렵다.



[그림 3-10] 정렬화된 데이터를 이용한 중간값 덧셈

제안하는 기법에서는 두 번의 반복문을 사용함으로써 곱셈을 사용하지 않고 두 번의 반복문만을 사용하도록 한다. $indicator$보다 작은 경우 다항식 a를 결과 다항식 b에 더하며 $indicator$보다 같거나 큰 경우 다항식 a를 결과 다항식 b에 뺀다. 이 때, 더하는 초기 인덱스는 정렬된 데이터 idx_s가 가리키는 인덱스이다. 미리 정렬된 데이터를 이용하므로 추가적인 데이터 변환은 필요하지 않다. 이러한 방식으로 곱셈을 사용하지 않을 수 있으며 intrinsic을 사용한 8비트 AVX2 명령어의 호출이 가능하다. 이 중간값들을 결과 다항식 b에 누적한 뒤 마지막에 범위에 맞게 차원 축소의 과정을 거치면 연산 결과 다항식 b를 얻을 수 있다.

3.1.9 공개키 압축 및 암호문 압축 (기존)

공개키의 크기를 줄이기 위해, LizarMong은 Keccak을 이용한다. Keccak은 해시 함수 표준인 SHA-3 공모전에서 선정된 알고리즘의 이름이다. NIST에는 모든 후보 알고리즘이 동일한 Keccak 구현을 사용할 수 있도록 제공하고 있다. LizarMong 역시 이 Keccak 함수를 사용한다.

Keccak 함수는 임의의 값을 입력받아 고정된 길이를 출력하는 함수로 임의의 입력을 축소하는 absorb 부분과 고정된 크기로 출력하는 squeeze 부분으로 구분된다. LizarMong KEM에서는 RLWE 문제에서 공개키 중 하나인 다항식 **a**의 축소를 위해 Keccak 알고리즘을 사용하며, 나머지 하나의 공개키인 다항식 **b**의 경우 안전성의 문제로 축소하지 않는다. 따라서, 공개키를 전송할 때, 다항식 **a**를 복구할 수 있는 $seed_a$만을 전송하여 공개키의 크기를 줄인다.

RLWE 문제를 이용한 암호화 알고리즘들은 복호화가 느리다는 특징을 갖는다. LizarMong에서는 효율적인 복호화를 위해 암호문의 크기를 줄이는 방법을 이용하는데 이를 암호문 압축이라고 한다. 암호문 압축으로 암호문의 크기를 줄이면 더 긴 메시지를 전송할 수 있어 대역폭에 유리하지만, 복호화 성공률을 떨어지게 된다. 이를 해결하기 위해 대부분의 후보 알고리즘은 오류 정정 기법을 사용한다. LizarMong에서는 오류 정정 기법 중 XE5를 사용하며 이를 통해 복호화 실패 확률을 무시할 정도로 줄인다.

3.1.10 공개키 압축 및 암호문 압축 (변경)

공개키 압축을 위해서 LizarMong은 Keccak 함수를 사용한다. 다항식 **a**를 공개키로서 보내는 것이 아닌, 다항식 **a**로 확장이 가능한 $Seed_a$를 보내고 Keccak 함수를 사용하여 확장한다. Keccak 함수의 경우 표준 해시

함수로서 미리 구현된 오픈 소스 라이브러리를 제공한다[3]. 제공하는 라이브러리에 포함되는 함수들 중 lib/low/KeccakP-1600-times4/AVX2/ 폴더에 존재하는 KeccakP-1600-times4-SIMD256.c를 사용한다. Keccak을 사용하는 NIST Round 2의 모든 격자 기반 암호들은 AVX2 구현을 위해 이를 사용하고 있다. 암호문 압축을 위해서 LizarMong은 Reconciliation 기법으로 오류 정정 기법인 XE5를 사용한다. *Strong*의 경우 출력 길이가 길어져 출력을 반으로 쪼개어 각각 XE5에 적용한다. 본 논문에서는 XE5에 대한 병렬 구현을 다루지 않으며 오류 정정 기법에 대한 병렬 구현을 후속 연구로서 남긴다.

3) https://github.com/XKCP/XKCP

Ⅵ. 평가

4.1 성능 평가

성능 평가를 위한 환경으로 AVX2 확장 명령어를 지원하는 Intel(R) Core(TM) i5-8250U CPU 1.60GHz를 사용하였다. 공정한 평가를 위해 모든 성능 평가에는 컴파일 옵션으로 최적화를 위한 -O3 옵션과 AVX2의 명령어 집합을 사용하기 위한 -mavx2만을 사용하였다. 무작위 비트열 생성을 위한 난수발생기는 리눅스 환경에서 제공하는 유사 난수 생성기 dev/urandom을 사용하였다. 기존의 LizarMong의 두 버전 *Comfort*, *Strong* [표 4-1], [표 4-2], [표 4-3]은 제안하는 기법들이 개별적으로 적용되었을 때의 속도를 측정한 결과이다. [표 4-4]는 제안하는 기법들이 동시에 적용되었을 때의 속도를 측정한 결과이다. 측정 단위는 cycle을 사용한다.

[표 4-1] KeyGen 속도 측정 결과 (cycle)

Type	Operation	Original	Opt_AVX2	%
Comfort	s_sampling	84,652	78,815	6.9%
	e_sampling	84,652	83,955	0.1%
	multiplication	84,652	76,150	10.0%
	reduction	84,652	84,331	0.2%
	keccakx4_pk	84,652	81,068	4.2%
			total	21.4%
Strong	s_sampling	111,633	107,553	3.7%
	e_sampling	111,633	110,902	0.1%
	multiplication	111,633	102,813	7.9%
	reduction	111,633	111,302	0.0%
	keccakx4_pk	111,633	107,085	4.1%
			total	15.8%

먼저 비밀 s 추출의 경우, 두 버전 모두에서 동일한 성능 향상이 있었다. 이 연산은 AVX2를 이용한 병렬처리 명령어를 이용하지 않는 최적화 연산

이다. 이에 따라, 개별적인 적용에도 직관적인 성능 향상을 확인할 수 있었다. 사용되는 무작위 비트열의 수를 줄였기 때문에, 만약 비용이 큰 고품질의 난수발생기를 사용한다면 더 높은 성능 향상을 기대할 수 있다. 오류 **e** 추출과 차원 축소의 경우 연산이 개별적으로 적용되었을 때 큰 성능 향상을 기대하기 어려웠다. 이에 대한 이유는 두 명령어 모두 병렬 처리를 위해 AVX2 명령어를 사용하기 때문이다. AVX2 명령어를 사용하기 위해서는 전용 레지스터를 선언하는 등의 전처리 작업이 필요하다. 이러한 전처리 작업의 경우 한 번만 이루어지면 되기 때문에 제안하는 기법들이 동시에 적용될 경우에는 성능 향상을 기대할 수 있을 것이다. 곱셈 연산의 경우 더 큰 차원을 이용하는 *Strong* 버전에서 좀 더 큰 성능 향상이 있었다. 곱셈 연산 역시 AVX2 명령어를 사용하기 때문에 전처리에 대한 비용이 포함된 결과이다. Keccak 함수의 경우 *Strong*에서 더 큰 성능 향상이 있었는데 입력은 동일하지만 더 큰 출력을 *Strong*에서 갖기 때문으로 보인다.

[표 4-2] Encapsulation 속도 측정 결과 (cycle)

Type	Operation	Original	Opt_AVX2	%
Comfort	s_sampling	43,786	42,877	2.1%
	e_sampling	-	-	-
	multiplication	43,786	43,989	0.0%
	reduction	43,786	44,122	0.0%
	keccakx4_pk	43,786	41,778	4.6%
			total	6.7%
Strong	s_sampling	86,263	84,847	1.6%
	e_sampling	-	-	-
	multiplication	86,263	78,643	8.8%
	reduction	86,263	84,405	0.0%
	keccakx4_pk	86,263	80,398	6.8%
			total	17.2%

비밀 **s** 추출의 경우 두 버전 모두 동일한 성능 향상을 보이고 있다. 또한, *Encapsulation* 알고리즘에는 최적화된 오류 **e** 추출 연산 기법을 적용할 연산이 존재하지 않는다. 곱셈의 경우 두 버전에서 큰 차이를 확인할 수 있

다. 그 이유는 *Strong*의 경우 더 큰 차원을 사용하며, 두 버전 모두 두 번의 곱셈 연산을 가지기 때문에 그 가중치가 더 커진 것으로 보인다. 차원 축소 연산의 경우 *KeyGen* 알고리즘과 동일한 양상을 보였다. Keccak의 경우에도 *KeyGen* 알고리즘과 동일한 이유로 *Strong* 버전에서 더 큰 성능 향상을 보인다.

[표 4-3] Decapsulation 속도 측정 결과 (cycle)

Type	Operation	Original	Opt_AVX2	%
Comfort	s_sampling	51,979	51,807	0.0%
	e_sampling	-	-	-
	multiplication	51,979	51,949	0.0%
	reduction	51,979	52,489	-0.1%
	keccakx4_pk	51,979	49,588	4.6%
			total	4.5%
Strong	s_sampling	99,461	101,006	3.7%
	e_sampling	-	-	-
	multiplication	99,461	82,679	7.9%
	reduction	99,461	107.337	0.0%
	keccakx4_pk	99,461	95,938	4.1%
			total	15.8%

Decapsulation 알고리즘의 경우 *Encapsulation* 알고리즘과 동일한 추세를 보였다.

[표 4-4] 항목별 속도 측정 결과 (cycle)

Algorithms	Original	Opt_AVX2	%
Comfort.KeyGen	31,488	23,903	24.1%
Comfort.Enc	36,805	27,209	26.1%
Comfort.Dec	45,296	32,058	29.2%
Strong.KeyGen	44,845	36,004	19.8%
Strong.Enc	75,473	62,039	17.8%
Strong.Dec	91,721	69,984	23.7%

4))성능평가용 코드

*Comfort*의 경우 공개키, 개인키 쌍 생성을 위한 *KeyGen*의 성능이 24.1%

4) https://githhub.com/DragonBeen/AVX2/

증가하였다. 공유할 키를 암호화하는 *Encapsulation*의 경우 26.1% 복호화하는 *Decapsulation*에서는 29.2%의 성능 향상을 보였다. *Strong*의 경우에도 *KeyGen*, *Encapsulation*, *Decapsulation*에 대하여 각각 19.8%, 17.8%, 23.7%의 성능 향상을 보였다. *Comfort*의 경우 전반적으로 *Strong*보다 증가폭이 적었는데, 이는 두 버전에서 사용하는 오류 정정 기법을 원인으로 볼 수 있다. 본 논문에서는 LizarMong이 사용하는 오류 정정 기법 XE5의 최적화를 진행하지 않았다. 오류 정정 기법의 경우 암호문의 커질 경우 더 큰 성능 저하를 일으키기 때문에, *Strong*의 성능에 미치는 비중이 더 커지게 된다. 또한 LizarMong에서는 *Strong*의 메시지 길이를 처리하기 위해 두 번의 오류 정정 기법을 사용한다. 따라서 그 비중이 더욱 커지기 때문이라고 해석할 수 있다. 또한 *KeyGen*의 성능이 두 버전 모두에서 성능이 좋은 것을 확인할 수 있다. 이는 두 가지 이유가 있을 수 있는데, 첫 번째 이유는 *KeyGen*에서는 오류 정정 기법을 사용하지 않기 때문이다. 두 번째 이유는 *Comfort* 기준, *KeyGen*에서 사용되는 약 736바이트의 난수를 약 608바이트로 약 17.4% 줄였기 때문이다. *Encapsulation*의 경우 32바이트의 난수만을 사용하며, *Decapsulation*에서는 난수를 사용하지 않기 때문에 상대적으로 성능 상승폭이 적었다고 볼 수 있다. 최적화 연산이 개별 적용되었을 때와 일괄 적용되었을 때의 성능 향상이 다르게 나타나는데 이 이유는 AVX2 명령어를 사용하기 위한 전처리 작업에 대한 비용을 들 수 있다. 예를 들어, 곱셈을 한 뒤 곱셈 결과에 대하여 차원 축소 과정을 추가적인 레지스터를 선언하지 않고 바로 진행할 수 있기 때문이다.

Ⅴ. 결론

5.1 결론 및 추후 연구

본 논문에서는 격자 기반 암호 LizarMong의 알고리즘에 대하여 분석한 뒤, 각 핵심 연산들을 최적화하였다. 최적화된 핵심 연산들을 일괄적으로 적용한 결과 15%에서 30%까지 성능이 향상된 것을 확인할 수 있었다. 비밀 **s** 추출의 경우, AVX2 명령어를 사용하지 않기 때문에 개별 적용된 경우에도 성능 향상을 보였다. 오류 **e** 추출 등 나머지 연산의 경우 곱셈 연산을 제외하고는 개별로 연산을 진행했을 때 미미한 증가율을 보였다. 이는 AVX2 명령어를 사용하기 위한 전처리 작업의 영향으로 일괄적으로 적용될 경우 전처리 작업을 줄여 성능이 향상되는 것을 확인할 수 있었다. 곱셈 연산의 경우 개별 적용에도 성능 향상을 보였는데, 이는 높은 차원의 다항식 곱셈 연산이 LizarMong 알고리즘에서 큰 비중을 차지하기 때문이다. 이는 *Strong* 버전에서는 *Comfort* 버전보다 더 높은 차원의 다항식 곱셈을 사용하여 성능 향상이 더 큰 것과 동일한 경향성을 보인다. Keccak 연산의 경우 입력 길이는 동일하지만 출력 길이의 차이가 있기 때문에 *Strong* 버전에서 더 큰 비중을 차지한 것을 확인할 수 있었다. LizarMong KEM은 NIST 후보 알고리즘들과 동일한 조건에서 성능 비교를 하였을 때, 가장 빠른 성능을 가진 암호 알고리즘이다. 그러므로 적절한 최적화가 이루어진다면 성능 측면에서 가장 좋은 선택지가 될 것이다. 또한, 양자 내성 암호로서 격자 기반 암호는 기존의 RSA, ECC에 견줄 수 있는 정도의 성능을 가지기 때문에 암호 시스템의 변경 관점에서 보았을 때, 기존의 시스템을 대체할 수 있는 적합한 암호가 될 수 있다.

본 논문에서는 암호문 압축을 위해 사용했던 오류 정정 기법 XE5의 최적화 및 병렬화를 진행하지 않았다. 실험 결과 XE5의 연산 비중이 높을 경우 성능 향상 폭이 눈에 띄게 적었다. 따라서 XE5의 최적화 및 병렬화를 통해 성능을 더욱 끌어올릴 수 있을 것이다. 또한 병렬 구현 시 데이터를 레지스터

단위로 사용하기 때문에 부채널 분석 대응 기법인 셔플링 등에 제약을 줄 수 있다. 이렇듯 최적화 구현이 부채널 분석에 미치는 영향을 고려해야할 필요가 있을 것이다. 마지막으로, 동일한 난수를 반복해서 사용하지 않는 등 안전성에 영향을 미치지 않는 최적화 기법에 대한 가이드 라인이 필요할 것으로 생각된다.

격자 암호 알고리즘에 대하여 성능, 안전성 두 측면에 대한 연구가 충분히 이루어진다면, 현존하는 공개키 암호화 시스템을 적절히 대체할 수 있을 것 기대된다.

참 고 문 헌

1. 국외문헌

Arute, F., Atya, K., Babbush, R. *et al.* (2019). Quantum supremacy using a programmable superconducting processor. Nature 574, 505-510.

Google AI Blog. (2019). Quantum Supremacy Using a Programmable Superconducting Processor. https://ai.googleblog.com/2019/10/quantum-supremacy-using-programmable.html (Accessed May 2020).

IBM. (2020). Quantum Properties. https://www.ibm.com/quantum-computing/learn/what-is-quantum-computing/ (Accessed May 2020).

NIST. (2020). Post-Quantum Cryptography FAQs. https://csrc.nist.gov/projects/post-quantum-cryptography/faqs (Accessed May 2020).

Gatner. (2018). Top 10 Strategic Technology Trends for 2019 (40p). https://www.gartner.com/en/doc/3891569-top-10-strategic-technology-trends-for-2019 (Accessed May 2020).

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. Proceedings 35th annual symposium on foundations of computer science, Ieee, 124-134.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, 212-219.

C. G. Jung, J. H. Lee, Y. J. Ju, Y. B. Kwon, S. W. Kim, Y. H. Paek

(2019). LizarMong: Excellent Key Encapsulation Mechanism Based on RLWE and RLWR. International Conference on Information Security and Cryptology, Lecture Notes in Computer Science, 11975, 208-224.

Hoffstein J., Pipher J., Silverman J.H. (1998). NTRU: A ring-based public key cryptosystem. International Algorithmic Number Theory Symposium, Lecture Notes in Computer Science, 1423, 267-288.

E. Alkim, L. Ducas, T. Pöppelmann, P. Schwabe (2016). Post-quantum Key Exchange—A New Hope. 25th USENIX Security Symposium, 327-343

Regev, O. (2009). On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM), 56(6), 1-40

Lyubashevsky, V., Peikert, C., & Regev, O. (2010). On ideal lattices and learning with errors over rings. Annual International Conference on the Theory and Applications of Cryptographic Techniques, 1-23

S. Kim, S. Hong (2018). Single trace analysis on constant time CDT sampler and its countermeasure. Appl. Sci., 8(10), 1809

M. J. O. Saarinen (2017). HILA5: On Reliability, Reconciliation, and Error Correction for Ring-LWE Encryption. International Conference on Selected Areas in Cryptography, Lecture Notes in Computer Science, 10719, 192-212.

P. Ravi, D. B. Roy, S. Bhasin, A. Chattopadhyay, D. Mukhopadhyay. Number “not used” once-practical fault attack on pqm4 implementations of nist candidates. International Workshop on Constructive Side-Channel Analysis and Secure Design, 232-250.

Z. Liu, H. J. Seo, S. S. Roy, J. Großschädl, H. W. Kim, I. Verbauwhede (2015). Efficient Ring-LWE encryption on 8-bit AVR processors. International Workshop on Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, 9293, 663-682.

T. Güneysu, V. Lyubashevsky T. Pöppelmann (2012). Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems, International Workshop on Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, 7428, 530-547.

ABSTRACT

Optimized implementation of LizarMong using AVX 2

Kwon, Yong Been

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

The development of the quantum computing environment is threatening the classic public-key cryptosystem. It is proved that currently widely used public-key cryptosystem such as RSA or elliptic curves cryptosystem will be analyzed in polynomial time using an algorithm proposed by P. Shor. Many organizations need new algorithms that are safe from not only classic attacks but also quantum attacks. According to these requirements, NIST(National Institute of Standards and Technology) defined cryptographic algorithms that have resistance from these attacks as post-quantum cryptography and initiated a standardization process by announcing a call for proposals. The lattice-based cryptography that accounts for higher than 50% among candidate algorithms has been catching great attention because of its good performance and proved security for a long period. LizarMong is a lattice-based cryptography based on one of the candidate algorithms named Lizard. The well-researched algorithm Lizard and state of the art technique converged into LizarMong. LizarMong has an outstanding performance compared with other NIST candidate algorithms. In this paper, I proposed an optimized LizarMong implementation using AVX2. This

implementation increased performance 15%~30% and minimized the requirement of randomness. According to these, it is expected that LizarMong which have currently the highest performance between NIST candidate algorithms will become a more feasible algorithm.

KEYWORD: Post-quantum cryptography, Lattice-based cryptography, LizarMong, AVX2, Optimized implementation, SIMD