석사학위논문

Grover 알고리즘 적용을 위한 국산 암호 양자 회로 최적화

2021년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

장 경 배

석 사 학 위 논 문
지도교수 서화정

Grover 알고리즘 적용을 위한 국산 암호 양자 회로 최적화

Optimization of Korean Block Cipher Quantum Circuits to Apply Grover's Algorithm

2020년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

장 경 배

석 사 학 위 논 문
지도교수 서화정

Grover 알고리즘 적용을 위한 국산 암호 양자 회로 최적화

Optimization of Korean Block Cipher Quantum Circuits to Apply Grover's Algorithm

위 논문을 공학 석사학위 논문으로 제출함

2020년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

장 경 배

장경배의 공학 석사 학위 논문을 인준함

2020년 12월 일

심사위원장 _____(인)

심 사 위 원 _____(인)

심 사 위 원 _____(인)

국 문 초 록

Grover 알고리즘 적용을 위한 국산 암호 양자 회로 최적화

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
장 경 배

Grover 알고리즘은 $O(2^{n/2})$의 검색만으로 n-bit 개인키를 사용하는 블록 암호의 개인키를 찾아낼 수 있는 가장 효과적인 양자 공격 방법이다. Grover 알고리즘을 적용하기 위해서는 우선 공격 대상 블록 암호가 양자 회로로 구현되어야 한다. 기존 연구들에서는 AES에서 가장 많은 비용이 드는 연산인 Substitute 연산을 양자 회로로 최적화하고 AES 양자 구현에 필요한 자원들을 평가하였다. 양자 구현을 통해 자원을 평가하는 연구는 AES 뿐만 아니라 경량 암호 분야로도 확대되고 있다. 이에 본 논문에서는 국산 경량 블록 암호 HIGHT, CHAM, LEA와 미국 NSA에서 개발한 SPECK을 양자 컴퓨터상에서 최적화 구현하였다. Addition, Rotation 및 XOR을 포함한 블록 암호에 대한 기본 연산은 큐비트, Toffoli 게이트, CNOT 게이트 및 X 게이트 측면에서 최적의 양자 회로를 달성하기 위해 세밀하게 최적화되었다. 최종적으로 국산 경량 블록 암호를 구현하기 위한 양자 자원을 추정하여 다가오는 양자 컴퓨터 시대에 대한 안전성을 분석하였다.

【주요어】 Grover 알고리즘, 양자 회로, 경량 블록 암호, HIGHT, CHAM,

LEA, SPECK, 양자 컴퓨터

목 차

I. 서 론 ····· 1

1.1 양자 컴퓨터 ····· 1
1.2 경량 블록 암호 양자 구현 동향 ····· 3
1.3 IoT(Internet of Things)와 경량 블록 암호 ····· 4

II. 관련 연구 ····· 5

2.1 경량 블록 암호 ····· 5
2.1.1 표기법 ····· 5
2.1.2 HIGHT ····· 5
2.1.3 CHAM ····· 9
2.1.4 LEA ····· 11
2.1.5 SPECK ····· 13

2.2 양자 구현과 알고리즘 ····· 15
2.2.1 양자 게이트 ····· 15
2.1.2 Grover 알고리즘 ····· 16

III. 제안 기법 ····· 18

3.1 HIGHT ····· 19
3.1.1 키 스케줄 ····· 19
3.1.2 라운드 함수 ····· 20
3.2 CHAM ····· 28
3.2.1 키 스케줄 ····· 28
3.2.2 라운드 함수 ····· 30
3.3 LEA ····· 32
3.3.1 키 스케줄 ····· 33
3.3.2 라운드 함수 ····· 34

3.4 SPECK ····· 36
3.4.1 키 스케줄 ····· 36
3.4.2 라운드 함수 ····· 37

IV. 성능 평가 ····· 38

4.1 양자 게이트 비용 ····· 38
4.2 국산 경량 블록 암호 양자 구현 비교 ····· 38
4.2.1 64-bit 평문, 128-bit 키 ····· 39
4.2.2 128-bit 평문, 128-bit 키 ····· 39
4.2.3 LEA 그리고 HIGHT 덧셈 연산 비교 ····· 40
4.2.4 국산 경량 암호 안전성 분석 ····· 40
4.3 소프트웨어, 하드웨어 최적화 암호 양자 구현 비교 ····· 40

V. 결 론 ····· 42

참 고 문 헌 ····· 43

ABSTRACT ····· 46

표 목 차

[표 1-1] IBM 양자 프로세서 개발 로드맵 ····· 2
[표 2-1] 표기법 ····· 5
[표 2-2] HIGHT 보안 파라미터 ····· 6
[표 2-3] CHAM 보안 파라미터 ····· 9
[표 2-4] LEA 보안 파라미터 ····· 11
[표 2-5] SPECK 보안 파라미터 ····· 14
[표 4-1] 국산 암호에 대한 양자 자원 비교 ····· 39
[표 4-2] SPECK 양자 자원 ····· 41
[표 4-3] SIMON 양자 자원 ····· 41

그 림 목 차

[그림 2-1] HIGHT 암호화 구조 ····· 9
[그림 2-2] CHAM 키 스케줄 ····· 11
[그림 2-3] CHAM 라운드 함수(i=홀수) ····· 11
[그림 2-4] LEA 키 스케줄 ····· 13
[그림 2-5] LEA 라운드 함수 ····· 13
[그림 2-6] SPECK 암호화 구조 (m=2) ····· 15
[그림 2-7] X(왼쪽), CNOT(가운데), Toffoli(오른쪽) 게이트 회로 구성 ····· 16
[그림 2-8] Grover 알고리즘 ····· 16
[그림 2-9] 오라클 함수 수행 후 ····· 17
[그림 2-6] Diffusion 연산자 수행 후 ····· 17
[그림 3-1] HIGHT 양자 회로 ····· 27
[그림 3-2] CHAM-64/128 양자 회로 ····· 32
[그림 3-3] LEA-128 양자 회로 ····· 36
[그림 3-4] SPECK-32/64 양자 회로 ····· 38

수 식 목 차

[수식 2-1] 화이트닝 키 WK 생성 ····· 6
[수식 2-2] δ_i 생성 ····· 6
[수식 2-3] 키 스케줄 ····· 6
[수식 2-4] 초기 변환 ····· 7
[수식 2-5] 보조 함수 F_0, F_1 ····· 7
[수식 2-6] 라운드 함수 ····· 7
[수식 2-7] 최종 라운드 함수 ····· 8
[수식 2-8] 최종 변환 ····· 8
[수식 2-9] 키 스케줄 ····· 10
[수식 2-10] 라운드 함수 (i = 홀수) ····· 10
[수식 2-11] 라운드 함수 ($i \neq$ 홀수) ····· 10
[수식 2-12] 상수 δ ····· 12
[수식 2-13] LEA-128 키 스케줄 ····· 12
[수식 2-14] 라운드 함수 ····· 12
[수식 2-15] 키 스케줄 ····· 14
[수식 2-16] 라운드 함수 ····· 14

알고리즘 목 차

[알고리즘 3-1] δ_i 생성 ····· 19
[알고리즘 3-2] HIGHT 키 스케줄 ····· 20
[알고리즘 3-3] HIGHT 초기 변환 ····· 21
[알고리즘 3-4] HIGHT 라운드 함수 ····· 22
[알고리즘 3-5] 보조 함수 F_0 ····· 24
[알고리즘 3-6] 보조 함수 F_1 ····· 25
[알고리즘 3-7] CHAM $RK[i]$ 키 스케줄 ····· 29
[알고리즘 3-8] CHAM $RK[i+8 \oplus 1]$ 키 스케줄 ····· 30
[알고리즘 3-9] CHAM 라운드 함수 ····· 32
[알고리즘 3-10] δ_0에서 $\delta[1]$로의 변환 ····· 34
[알고리즘 3-11] LEA-128 키 스케줄 ····· 34
[알고리즘 3-12] LEA-128 라운드 함수 ····· 35
[알고리즘 3-13] SPECK 키 스케줄 ····· 37
[알고리즘 3-14] SPECK 라운드 함수 ····· 37

Ⅰ. 서 론

1.1 양자 컴퓨터

양자 컴퓨터에는 기존 컴퓨터에서 사용하는 비트(bit)가 아닌 양자 비트 즉, 큐비트(Qubit)를 사용한다. 큐비트의 가장 큰 특징은 중첩과 얽힘이 있다. 중첩이란 큐비트에 0과 1이 확률로서 존재하여 0과 1을 동시에 표현할 수 있다. 얽힘이란 하나의 큐비트의 상태가 멀리 떨어져있는 다른 큐비트의 상태에 관여하는 것이다. 큐빗의 중첩과 얽힘의 성질을 결합하면 해당 큐빗으로 표현 가능한 모든 경우의 수를 계산할 수 있다. 이는 양자 컴퓨터에 엄청난 속도, 병렬성을 제공하여 특정 계산에 대한 방대한 경우의 수를 동시에 처리할 수 있다.

구글, IBM, 마이크로소프트와 같은 글로벌 IT 기업들이 양자 컴퓨터 개발에 있어 우위를 선점하기 위해 전폭적인 투자를 하고 있다. 2019년도 10월 구글은 제일 먼저 양자 우위 달성에 대한 논문을 발표 하였다. 양자 우위란 기존 컴퓨터의 성능을 양자 컴퓨터가 뛰어 넘는 것이다. 난수성을 증명하는 문제에서 기존 컴퓨터는 1만년이 소요되지만 구글의 양자 컴퓨터 Syncamore는 3분 20초만에 문제를 해결하였다. 이전 2018년도, 구글은 72 큐비트 양자 컴퓨터 Bristlcone을 이미 개발하였지만 양자 컴퓨터 개발의 대표적인 어려움인 오류율을 제어하기 어렵다는 문제점이 있었다. 하지만 2019년 54 큐비트의 Syncamore를 출시하면서 오류를 최적으로 증폭하여 교정하는 기술을 추가하였고 양자 우위를 달성하였다.

IBM 또한 양자 컴퓨터 개발에 많은 투자를 하고 있다. IBM은 2020년도 9월, 65 큐비트의 Quantum Hummingbird를 출시하였고 1000 큐비트 이상의 양자 컴퓨터 개발을 목표로 하는 로드맵을 발표하였다. 로드맵은 [표 1-1]과 같다. 양자 컴퓨터는 오류제어를 위해 극저온의 환경에서 동작한다. 1000 큐비트 이상의 양자 컴퓨터를 위한 냉각장치는 현재 시중에서 판매하는 규모

로는 불가능하다. 이에 IBM은 자체적으로 대규모의 냉각장치 Goldeneye를 개발하고 있다. IBM은 현재까지의 개발 경험을 토대로 향후 10년 이내에 100만 큐빗의 대규모 양자 컴퓨터 개발을 목표로 하고 있다.

[표 1-1] IBM 양자 프로세서 개발 로드맵

	2019	2020	2021	2022	2023	2030
Qubit	27	65	127	433	1,121	Over 1 million
Processor	Falcon	Humming bird	Eagle	Osprey	Condor	•

양자 컴퓨터에 양자 알고리즘을 활용한다면 우리가 현재 사용하고 있는 암호들의 보안을 훼손시킬 수 있다. 대표적인 양자 알고리즘으로는 Shor 알고리즘과 Grover 알고리즘(Grover algorithm)이 있다. 1994년, Peter Shor는 양자 알고리즘을 사용하여 소인수 분해 문제와 이산대수 문제를 다항시간 내에 풀 수 있다는 것을 증명하였다. 현대 암호 RSA는 소인수 분해 문제의 어려움에 안전성을 기반하며, ECC는 이산대수 문제의 어려움에 기반하고 있다. 따라서 충분한 큐비트의 양자 컴퓨터가 올바르게 작동한다면 기존 공개키 암호 시스템은 더 이상 사용할 수 없다. 즉, 현재 가장 많이 사용되고 있는 공개키 암호 RSA, ECC(Elliptic Curve Cryptography)는 더 이상 사용할 수 없다.

1996년, Lov Grover는 n개의 분류되지 않은 데이터에서 특정 데이터를 찾아내는 양자 알고리즘을 제안하였다. Shor 알고리즘처럼 기하급수적인 속도 향상을 보여주진 않지만, 2차 속도 향상을 보여준다. 기존 컴퓨터에서 무차별 대입 공격 시 $O(2^n)$의 횟수가 소모될 때, Grover 알고리즘을 사용하면 $O(2^{n/2})$의 횟수 만으로 공격에 성공할 수 있다. 따라서 n-bit 보안 레벨의 대칭키 암호를 $\sqrt{n}$-bit 보안 레벨로 감소시킬 수 있다. 기존 128-bit 대칭키 암호는 약 2^{64}회의 반복으로, 256-bit 대칭키 암호는 2^{128}회의 반복만으로 원본 메시지를 복구할 수 있다. 대규모의 양자 컴퓨터가 개발된다면 기존 보안 레벨을 유지하기 위해서는 키 길이를 두 배로

늘려야한다.

1.2 경량 블록 암호 양자 구현 동향

양자 알고리즘을 활용하여 현대 암호의 안전성을 붕괴시킬 수 있다는 이론은 정립되었다. 양자 알고리즘을 구동시킬 수 있는 대규모의 양자 컴퓨터가 개발된다면 공개키, 대칭키 암호의 안전성이 무너지게 된다.

AES(Advanced Encryption Standard)는 전 세계적으로 가장 많이 사용되고 있는 대칭키 블록 암호이다. 때문에 AES의 양자 최적화 구현과 이에 대한 자원을 측정하는 수많은 연구들이 이루어지고 있다. Grassl은 AES의 첫 번째 양자 회로를 제안하였고 키를 찾는 공격에 Grover 알고리즘을 적용하기 위한 양자 자원들을 분석하였다 AES-128, AES-192, AES-256 모두 Grover 알고리즘 적용을 위해 각 레이어 당 최적의 게이트 연산으로 구현되었다.

Langenberg는 새로운 AES 양자 구현을 제안하였다. 모든 키 길이의 AES를 구현하였으며 이 중, AES-128의 경우 Grassl의 구현 결과보다 Toffoli 게이트의 수를 약 88퍼센트 감소시켰으며 큐비트의 사용 개수 또한 줄어들었다.

최근에는 경량 블록 암호에 대한 양자 구현 연구들도 수행되고 있다. 양자 구현에 있어 가장 중요한 부분은 사용되는 큐비트와 양자 게이트의 수를 최소화 하는 것이다. 양자 컴퓨터는 아직 초기 개발 단계이기 때문에 사용 가능한 큐비트가 충분하지 않으며 복잡한 양자 게이트 연산으로 인해 생기는 오류율 또한 제어하는 것이 어렵다. 따라서 양자 구현 시 필요로 하는 큐비트가 적고 양자 게이트로 구성되는 회로의 복잡도가 낮은 것이 중요하다. Anand는 NSA에서 개발한 SIMON을 양자 회로로 구현하였다. 이를 통해 SIMON을 Grover 알고리즘으로 공격하는데 필요한 양자 자원들을 큐비트, CNOT, Toffoli, X 게이트의 관점에서 최적화 하였다. Schlieper는 NIST(National Institute of Standards and Technology)의 경량 암호 공모전 라운드 2의 후보인 Gilmi를 양자 회로로 구현하였으며, Jang은

SPN(Substitution-Permutation-Network) 구조의 경량 블록 암호 GIFT를 양자 회로로 구현하였다. 이처럼 현재 다가오는 양자 컴퓨터 시대에 대비하여 암호 알고리즘을 양자 회로로 최적화 구현하여 Grover 알고리즘을 적용하는데 필요한 자원을 최소화 하는 연구가 활발히 진행되고 있다. 이에 본 논문에서는 모든 국산 경량 암호를 양자 구현하였다. 최적화 구현을 위해 키 스케줄, 라운드 함수를 병행하였으며 필요에 따라 연산 대상과 순서를 재배열하고 리버스 연산을 활용하여 큐비트를 최소화 하였다. 국산 경량 암호 HIGHT, CHAM, LEA를 구현하였으며 부가적인 비교를 위해 미국 NSA에서 개발한 경량 암호 SPECK을 추가로 구현하였다.

1.3 IoT(Internet of Things)와 경량 블록 암호

4차 산업혁명의 핵심 기술로 지목되는 IoT(Internet of Things) 즉, 사물인터넷 기술이 점차 발전하고 있다. 수많은 웨어러블, 스마트 디바이스가 점점 사람들의 일상생활에 자리 잡고 있으며 이러한 디바이스들을 통해 민감정보를 포함한 방대한 데이터들이 서로 교환되고 있다. 개인의 사생활 보호를 위해서는 이러한 민감 정보들을 모두 암호화하여 통신해야 한다. 암호 알고리즘을 적용하기 위해서는 연산능력, 메모리와 같은 자원들이 요구된다. 하지만 대부분의 IoT 디바이스들은 낮은 연산능력, 작은 메모리를 가지고 있어 암호 알고리즘을 적용시키기에 어려움이 따른다.

IoT 디바이스에 암호 알고리즘 적용 시 발생하는 문제를 해결하기 위해 경량 암호에 대한 연구가 활발히 진행되고 있다. 기존 암호와는 다르게 경량 암호는 저전력 디바이스를 대상으로 설계된 암호이며 디바이스의 제한된 자원을 효율적으로 활용하는 것에 초점을 두고 있다. 또한 경량 암호 또한 양자 컴퓨터의 공격에 내성을 가져야 한다. 헬스케어, 의료용 IoT 디바이스같은 경우, 환자나 사용자의 민감 정보가 저장되며 서로 통신한다. 미래에 양자 컴퓨터가 개발되어 IoT 디바이스 데이터를 해킹한다면 많은 사람들의 개인정보가 노출될 수 있다.

Ⅱ. 관련연구

2.1 경량 블록 암호

본 장에서는 Addition, Rotation, XOR 연산만으로 구성된 국산 ARX 구조 경량 블록 암호 HIGHT, CHAM, LEA 그리고 미국 NSA(National Security Agency)에서 제안한 ARX 구조의 경량 블록 암호 SPECK에 대하여 자세히 설명한다.

2.2.1 표기법

본 논문에서 사용되는 표기법과 의미는 [표 2-1]과 같다.

[표 2-1] 표기법

Notation	Meaning
K	Initial keywords
RK	Round key
$\oplus$	XOR operation
$\boxplus$	Modular addition operation
ROL_i	Rotation left operation(i-bit)
ROR_i	Rotation right operation(i-bit)
$\lll i$	Rotation left operation(i-bit)
$\ggg i$	Rotation right operation(i-bit)

2.2.2 HIGHT

HIGHT는 ARX 연산만을 활용한 국산 경량 블록암호이다. 128-bit 키를 사용하며 64-bit 블록 크기를 사용한다. 암호화를 수행하기 위해 8-bit 로 구성 된 8개의 화이트닝 키와 128개의 라운드 키가 입력 128-bit 키로부터 생성된다. HIGHT는 총 32라운드로 구성되어 있으며 각 라운드마다 32-bit

의 라운드 키를 사용하여 암호화한다. HIGHT의 보안 파라미터는 [표 2-2]와 같다.

[표 2-2] HIGHT 보안 파라미터

Cipher	Block size(bits)	Key size(bits)	Word size(bits)	Keywords	Rounds
HIGHT	64	128	8	16	32

HIGHT의 키 스케줄은 초기 키 값 $K=(K[0],K[1],\cdots,K[15])$를 사용하여 화이트닝 키 WK를 생성한다. WK는 입력 평문의 초기 변환과 암호문 생성을 위한 최종 변환에 사용된다. 화이트닝 키는 다음과 같이 생성된다.

$$WK[i]=K[i+12],\quad 0\le i\le 3$$
$$WK[i]=K[i-4],\quad 4\le i\le 7$$

[수식 2-1] 화이트닝 키 WK 생성

초기 $\delta_0 = (s_6, s_5, s_4, s_3, s_2, s_1, s_0) = (1,0,1,1,0,1,0)$으로부터 라운드 키를 계산하기 위해 δ_i $(1\le i\le 127)$가 다음 수식을 통해 생성된다.

$$s_{i+6}=s_{i+2}\oplus s_{i-1}$$
$$\delta_i=(s_{i+6}, s_{i+5}, s_{i+4}, s_{i+3}, s_{i+2}, s_{i+1}, s_i)$$

[수식 2-2] δ_i 생성

라운드 키는 다음과 같이 생성된다.

for $i=0$ to 7:
  for $j=0$ to 7:
    $RK[16\cdot i+j]=K[j-i \bmod 8]\boxplus\delta_{16\cdot i+j}$
  for $j=0$ to 7:
    $RK[16\cdot i+j+8]=K[j-i \bmod 8]\boxplus\delta_{16\cdot i+j+8}$

[수식 2-3] 키 스케줄

라운드 함수를 수행하기 전에 앞서 생성한 WK를 사용하여 입력 평문 T의 초기 변환을 다음과 같이 수행한다.

$$X_0[i] = T[i],\ i = 1, 3, 5, 7$$
$$X_0[0] = T[0] \boxplus WK[0]$$
$$X_0[2] = T[2] \oplus WK[1]$$
$$X_0[4] = T[4] \boxplus WK[2]$$
$$X_0[6] = T[6] \oplus WK[3]$$

[수식 2-4] 초기 변환

HIGHT의 라운드 함수에서는 XOR, Rotation 연산만으로 구성 된 보조함수 $F_0(X)$와 $F_1(X)$가 $1 \le i \le 32$ 에 대해 사용되며 $F_0(X)$와 $F_1(X)$, 라운드 함수는 다음과 같다.

$$F_0(X) = ROL_1(X) \oplus ROL_2(X) \oplus ROL_7(X)$$
$$F_1(X) = ROL_3(X) \oplus ROL_4(X) \oplus ROL_6(X)$$

[수식 2-5] 보조함수 F_0, F_1

$$X_i[j] = X_{i-1}[j-1],\quad j = 1, 3, 5, 7$$
$$X_i[0] = X_{i-1}[7] \oplus (F_0(X_{i-1}[6]) \boxplus RK[4i-1])$$
$$X_i[2] = X_{i-1}[1] \boxplus (F_0(X_{i-1}[0]) \oplus RK[4i-4])$$
$$X_i[4] = X_{i-1}[3] \oplus (F_0(X_{i-1}[2]) \boxplus RK[4i-3])$$
$$X_i[6] = X_{i-1}[5] \boxplus (F_0(X_{i-1}[4]) \oplus RK[4i-2])$$

[수식 2-6] 라운드 함수

최종 라운드 함수는 아래와 같으며 마지막 라운드(i=32)에선 $F_1(X)$가 같이 사용된다.

$$X_{32}[i] = X_{31}[i], \quad i = 0, 2, 4, 6$$
$$X_{32}[1] = X_{31}[1] \boxplus (F_1(X_{31}[0]) \oplus RK[124])$$
$$X_{32}[3] = X_{31}[3] \oplus (F_0(X_{31}[2]) \boxplus RK[125])$$
$$X_{32}[5] = X_{31}[5] \boxplus (F_1(X_{31}[4]) \oplus RK[126])$$
$$X_{32}[7] = X_{31}[7] \oplus (F_0(X_{31}[6]) \boxplus RK[127])$$

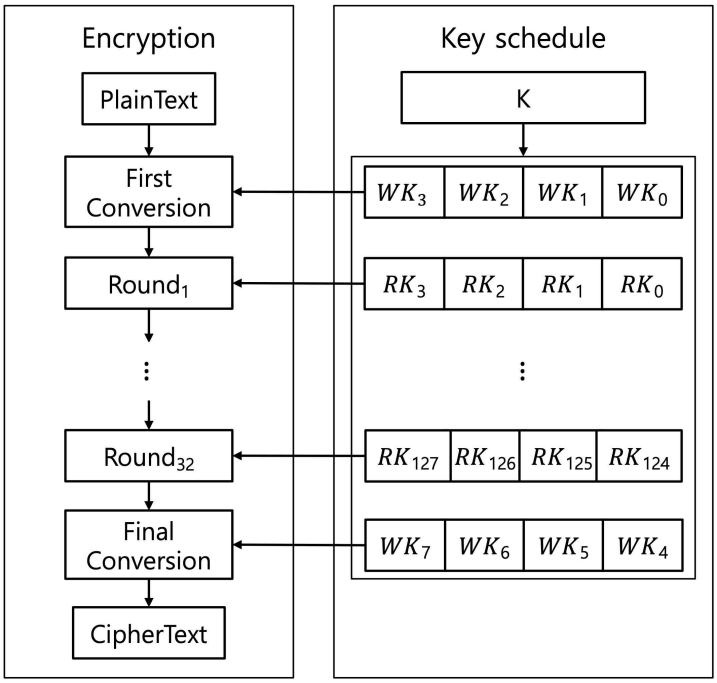
[수식 2-7] 최종 라운드 함수

모든 라운드 함수를 수행한 뒤, 아래 수식의 화이트닝 키를 사용한 최종 변환을 통해 암호문 C가 생성되고 암호화가 종료된다.

$$C[i] = X_{32}[i], \; i = 1, 3, 5, 7$$
$$C[0] = X_{32}[0] \boxplus WK[4]$$
$$C[2] = X_{32}[2] \oplus WK[5]$$
$$C[4] = X_{32}[4] \boxplus WK[6]$$
$$C[6] = X_{32}[6] \oplus WK[7]$$

[수식 2-8] 최종 변환

HIGHT의 키 스케줄, 라운드 함수를 활용한 전체적인 암호화 구조는 [그림 2-1]과 같다.



[그림 2-1] HIGHT 암호화 구조

2.2.3 CHAM

CHAM은 ETRI(Electronics and Telecommunications Research Institute)에서 개발한 국산 경량 블록암호이다. 다음 CHAM-64/128, CHAM-128/128, CHAM-128/256 세 가지 보안 파라미터를 제공하며 [표 2-3]과 같다.

[표 2-3 CHAM 보안 파라미터]

Cipher	Block size(bits)	Key size(bits)	Word size(bits)	Keywords	Rounds
CHAM-64/128	64	128	16	8	80
CHAM-128/128	128	128	32	4	80
CHAM-128/256	128	256	32	8	96

초기 키 값 $K=(K[0],K[1],\cdots,K[m-1])$에 키 스케줄을 수행하여 $(0 \le i < r)$의 i 번째 라운드에서 사용되는 라운드 키

$RK = (RK[0], K[1], \cdots, RK[2m-1])$를 생성하며 아래 수식과 같다. $(0 \le i < r)$

$$RK[i] = K[i] \oplus ROL_1(K[i]) \oplus ROL_8(K[i])$$
$$RK[(i+m) \oplus 1] = K[i] \oplus ROL_1(K[i]) \oplus ROL_{11}(K[i])$$

[수식 2-9] 키 스케줄

키 스케줄을 통해 라운드 키 RK가 생성되고 입력 평문 $T = (X_0[0], X_0[1], X_0[2], X_0[3])$는 암호화 과정에서 다음 두 가지 경우의 라운드 함수를 수행한다. 라운드 i가 홀수인 경우 다음 라운드 함수를 수행한다.

$$X_{i+1}[j] = X_i[j+1], (0 \le j \le 2)$$
$$X_{i+1}[3] = ROL_8((X_i[0] \oplus i) \boxplus (ROL_1(X_i[1]) \oplus RK[i \bmod 2m]))$$

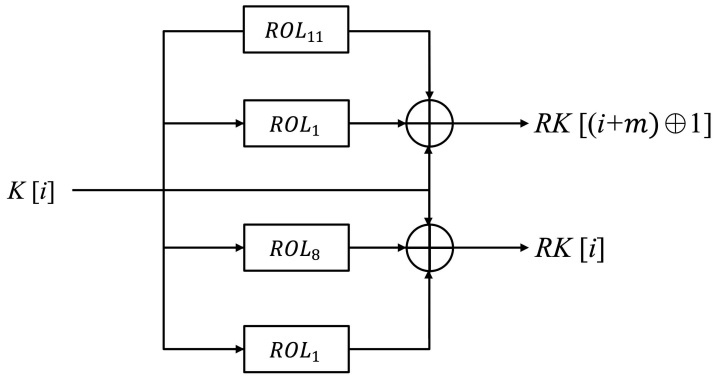
[수식 2-10] 라운드 함수 (i = 홀수)

이 외의 경우 다음 라운드 함수를 수행한다.

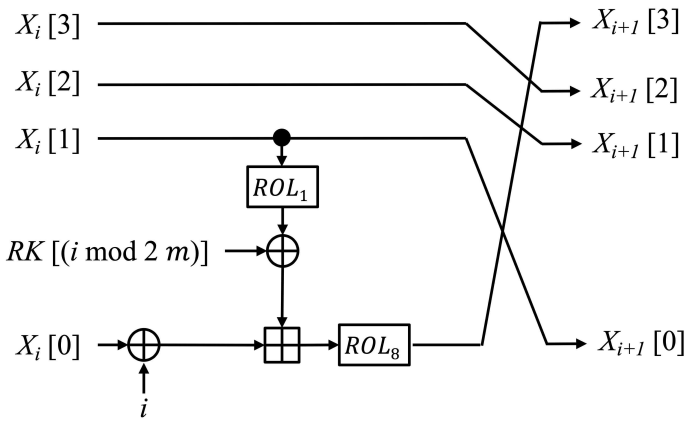
$$X_{i+1}[j] = X_i[j+1], (0 \le j \le 2)$$
$$X_{i+1}[3] = ROL_1((X_i[0] \oplus i) \boxplus (ROL_8(X_i[1]) \oplus RK[i \bmod 2m]))$$

[수식 2-11] 라운드 함수 ($i \neq$ 홀수)

CHAM의 키 스케줄과 라운드 함수 구조는 [그림2-2], [그림2-3]과 같다.



[그림 2-2] CHAM 키 스케줄



[그림 2-3] CHAM 라운드 함수(i=홀수)

2.2.4 LEA

LEA는 3가지 128-bit, 192-bit, 256-bit의 키 크기를 지원하며 128-bit 블록 크기의 국산 경량 블록암호이다. LEA-128, LEA-192, LEA-256은 24, 28, 32라운드로 구성된다. 각 라운드에서 192-bit의 라운드 키를 사용한다. LEA의 파라미터들은 [표 2-4]와 같다.

[표 2-4 LEA 보안 파라미터]

Cipher	Block size(bits)	Key size(bits)	Word size(bits)	Keywords	Rounds

LEA-128	128	128	32	4	24
LEA-192	128	192	32	6	28
LEA-256	128	256	32	8	32

LEA의 키 스케줄에서는 'L', 'E', 'A"를 ASCII 코드로 표현한 상수 δ가 사용되며 파라미터에 따라 키 스케줄 연산에 조금의 차이가 있다. LEA-128의 키 스케줄($0 \le i < 24$)은 다음과 같다. LEA-192, LEA-256의 키 스케줄에 대한 자세한 사항은 [13]에서 확인할 수 있다.

$$\delta[0] = 0xc3efe9db,\ \ \delta[1] = 0x44626b02$$
$$\delta[2] = 0x79e27c8a,\ \delta[3] = 0x78df30ec$$
$$\delta[4] = 0x715ea49e,\ \delta[5] = 0xc785da0a$$
$$\delta[6] = 0xe04ef22a,\ \delta[7] = 0xe5c40957$$

[수식 2-12] 상수 δ

$$K[0] = ROL_1(K[0] \boxplus ROL_i(\delta[i \bmod 4]))$$
$$K[1] = ROL_3(K[1] \boxplus ROL_{i+1}(\delta[i \bmod 4]))$$
$$K[2] = ROL_6(K[2] \boxplus ROL_{i+2}(\delta[i \bmod 4]))$$
$$K[3] = ROL_{11}(K[3] \boxplus ROL_{i+3}(\delta[i \bmod 4]))$$
$$RK_i = (K[0], K[1], K[2], K[1], K[3], K[1])$$

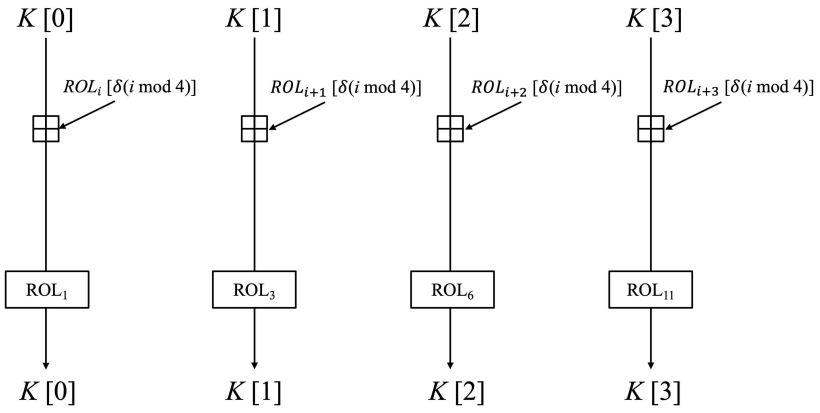
[수식 2-13] LEA-128 키 스케줄

키 스케줄과는 달리 라운드 함수는 파라미터에 상관없이 다음 한 가지 수식을 입력 평문 $T = (X_0[0], X_0[1], X_0[2], X_0[3])$에 대해 수행한다.

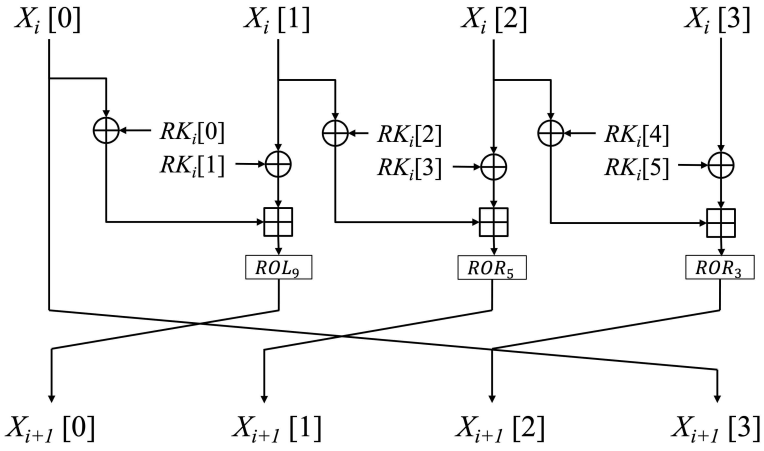
$$X_{i+1}[0] = ROL_9((X_i[0] \oplus RK_i[0]) \boxplus (X_i[1] \oplus RK_i[1]))$$
$$X_{i+1}[1] = ROL_5((X_i[1] \oplus RK_i[2]) \boxplus (X_i[2] \oplus RK_i[3]))$$
$$X_{i+1}[2] = ROL3((X_i[2] \oplus RK_i[4]) \boxplus (X_i[3] \oplus RK_i[5]))$$
$$X_{i+1}[3] = X_i[0]$$

[수식 2-14] 라운드 함수

LEA의 키 스케줄과 라운드 함수 구조는 [그림 2-4], [그림 2-5]와 같다.



[그림 2-4] LEA 키 스케줄



[그림 2-5] LEA 라운드 함수

2.2.5 SPECK

미국 NSA에서는 경량 블록암호 SPECK, SIMON을 개발하였다. SPECK은 소프트웨어 환경을 대상으로 하였으며, SIMON은 하드웨어 환경을 대상으로 최적화되었다. SIMON은 이미 양자 회로로 구현하는 연구가 진행되었다[?]. 따라서 본 논문에서는 오직 SPECK의 양자 회로 최적화 구현을 다룬다. 최종적으로 NSA에서 개발한 SPECK과 국산 경량 블록암호 구현 결과를

비교하여 양자 컴퓨터에 대한 영향을 범용적으로 확인해보고자 한다.

SPECK은 ARX 연산만으로 구성 되었으며 다양한 보안 파라미터를 제공한다. SPECK의 파라미터들은 [표2-5]와 같다.

[표 2-5 SPECK 보안 파라미터]

Block size(bits)	Key size(bits)	Word size(bits)	Keywords	Rounds
32	64	16	4	22
48	72, 96	24	3, 4	22, 23
64	96, 128	32	3, 4	26, 27
96	96, 144	48	2, 3	28, 29
128	1228, 192, 256	64	2, 3, 4	32, 33, 34

키 스케줄에서는 각 라운드 함수에서 사용 될 라운드 키 RK를 생성한다. 초기 키 값 $K=(K[0],l[0],\cdots,l[m-2])$, ($m$ = Keywords)에 아래 키 스케줄을 수행($0 \le i <$ Rounds)하여 라운드 키 $RK=(K[0],K[1],\cdots,K[\text{Rounds}-1])$를 생성한다. 블록 크기가 32-bit 인 경우 α와 β는 각각 7과 2이며, 나머지 블록 크기에 대한 α와 β는 각각 8과 3이다.

$$l[i+m-1]=(K[i] \boxplus ROR_{\alpha}(l[i])) \oplus i$$
$$K[i+1]=ROL_{\beta}(K[i]) \oplus l[i+m-1]$$

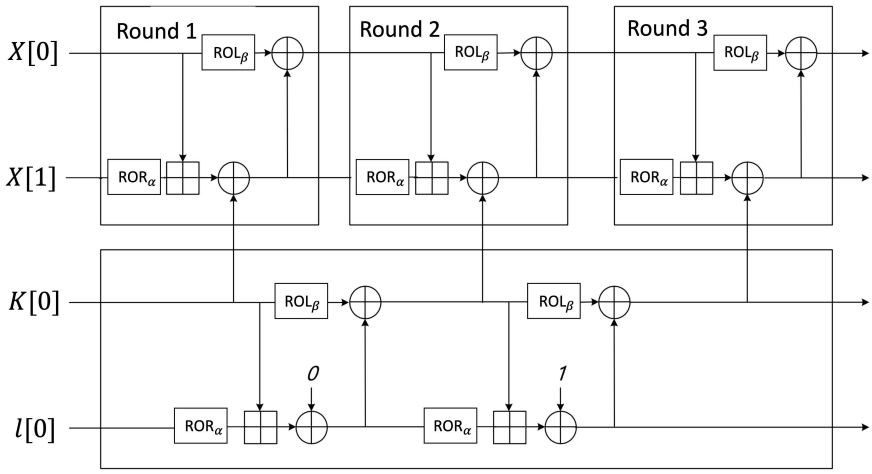
[수식 2-15] 키 스케줄

블록 크기가 $2n$인 SPECK의 라운드 함수($0 \le i <$ Rounds)에서는 n-bit 라운드 키가 $2n$-bit의 입력 평문에 사용되어 $2n$-bit의 암호문이 생성된다.

$$X[1]=(ROR_{\alpha}(X[1]) \boxplus X[0]) \oplus RK[i]$$
$$X[0]=ROL_{\beta}(X[0]) \oplus X[1]$$

[수식 2-16] 라운드 함수

$m=2$인 경우의 SPECK의 암호화 구조는 [그림 2-6]과 같다.

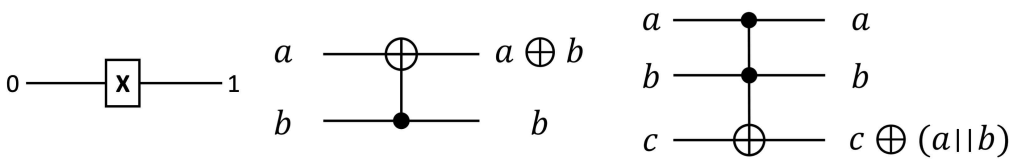


[그림 2-6] SPECK 암호화 구조 ($m=2$)

2.2 양자 구현과 알고리즘

2.2.1 양자 게이트

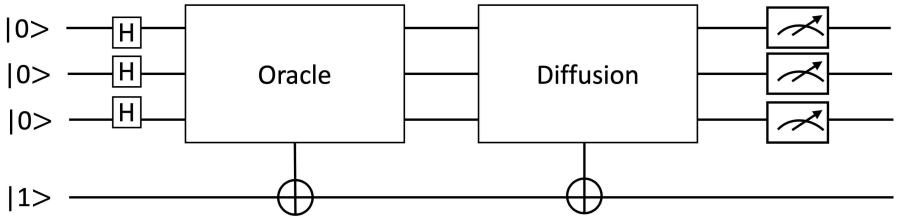
같은 기존 컴퓨터의 NOT, XOR, AND 연산을 양자 컴퓨터에서는 양자 게이트로 대체할 수 있으며 대표적으로 X, CNOT, Toffoli 게이트가 있다. X 게이트는 기존 컴퓨터의 NOT 연산을 대체하며 단일 큐빗을 대상으로 한다. 단일 큐빗에 X 게이트를 통과시키면 해당 큐빗의 상태 값을 반전시킬 수 있다. CNOT 게이트는 기존 XOR 연산을 대체하며 두 개의 큐빗을 대상으로 한다. CNOT 게이트를 통과 시키면 한 큐빗의 상태가 1인 경우, 다른 큐빗의 상태를 반전시킨다. Toffoli 게이트는 기존 AND 연산을 대체하며 세 개의 큐빗을 대상으로 한다. Toffoli 게이트를 통과 시키면 두 큐빗이 모두 1인 경우, 나머지 큐빗의 상태를 반전시킨다. X, CNOT, Toffoli 게이트의 회로 구성은 [그림 2-7]과 같다.



[그림 2-7] X(왼쪽), CNOT(가운데), Toffoli(오른쪽) 게이트 회로구성

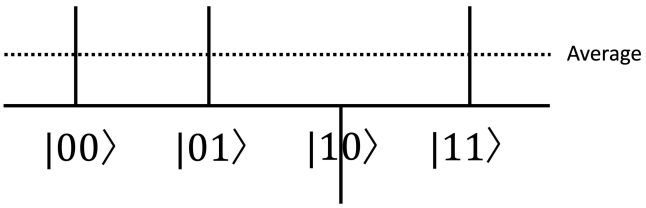
2.2.2 Grover 알고리즘

Grover 알고리즘은 분류되지 않은 데이터들 중 특정 데이터를 찾는 양자 알고리즘이다. 기존 전수 조사로 해답을 찾기 위해 $O(2^n)$의 탐색이 필요한 경우, Grover 알고리즘을 적용하면 $O(2^{n/2})$의 횟수 만에 해답을 찾아낼 수 있다. Grover 알고리즘은 크게 오라클 함수와 diffusion 연산자로 구성되며 [그림 2-8]과 같다.



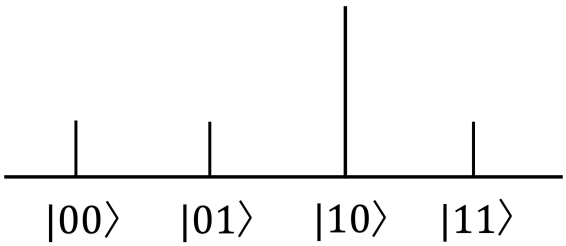
[그림 2-8] Grover 알고리즘

오라클 함수 $f(x)$는 입력 x가 해답인 경우 1을 반환하며 그렇지 않은 경우, 0을 반환한다. 또한 $f(x)=1$ 인 경우 입력 x의 값이 반전된다. [그림2-8]은 2-큐빗을 입력 받을 때 해답이 10인 경우 오라클 함수를 수행한 뒤 상태이다.



[그림 2-9] 오라클 함수 수행 후

오라클 함수를 수행했다면 다음은 diffusion 연산자를 수행한다. diffusion 연산자는 오라클 함수에서 반전 된 해답의 관측 확률을 증폭시킨다. diffusion 연산자의 수행 단계는 다음과 같다. 제일 먼저 모든 경우의 수의 관측확률의 평균을 계산한 뒤, 평균 확률에서 각 데이터의 확률의 차이 값을 계산한다. [그림 2-9]는 diffusion 연산자를 수행한 뒤 상태이다.



[그림 2-10] Diffusion 연산자 수행 후

Grover 알고리즘은 오라클 함수와 diffusion 연산자를 반복 수행한다. 반복 수행으로 해답 데이터의 관측 확률은 점점 증가하게 되고 해답이 아닌 데이터의 관측 확률은 감소하게 된다. 따라서 반복 횟수를 최적화하여 올바른 정답이 관측 될 확률을 증가시킨다.

Ⅲ. 제안 기법

본 장에서는 HIGHT, CHAM, LEA, SEPCK에 Grover 알고리즘을 적용하기 위한 최적화 양자 회로 구현 기법에 대해 살펴본다.

본 논문에서 구현한 모든 블록 암호들은 Addition, Rotation, XOR 연산을 기반으로 하는 ARX 구조이다. 양자 회로구현 시, Rotation 연산은 큐비트의 위치를 서로 바꾸는 Swap 게이트로 수행할 수 있다. 하지만 이는 큐비트를 재 라벨링 해주는 것으로 대체할 수 있기 때문에 Swap 게이트에 대한 비용은 무시할 수 있다. 예를 들어 $a=(a_0,a_1,a_2,a_3)$ 와 $b=(b_0,b_1,b_2,b_3)$에 대하여 Rotation연산인 $a \ggg 1$ 을 수행한 뒤, XOR 연산 $b=b\oplus a$ 를 수행한다. 이때 Rotation 연산이 수행되면 a_0가 최하위 큐비트기 때문에 $a=(a_3,a_0,a_1,a_2)$이 된다. 따라서 XOR 연산을 수행 시, 재 라벨링한 큐비트의 위치에 따라 CNOT 게이트를 다음과 같이 통과시켜 준다. CNOT(a_3,b_0), CNOT(a_0,b_1), CNOT(a_1,b_2), CNOT(a_2,b_3), 이러한 방법으로 Rotation 연산에 대한 추가 게이트 없이 큐비트 재 라벨링을 수행하고 CNOT 게이트를 통해 최종적으로 Rotation + XOR 연산을 수행할 수 있다. Rotation, XOR 연산과는 달리 Addition 연산은 Toffoli 게이트가 사용하는 더 복잡한 회로를 사용한다. 본 구현에서는 Cuccaro의 Ripple-carry addition 덧셈회로를 사용하였다. Ripple-carry addition 회로에는 캐리 값 연산을 위한 2개의 추가 큐비트가 사용된다. 하지만 블록 암호의 Addition은 모듈러 Addition이므로 최상위 캐리를 고려하지 않아도 된다. 따라서 초기 캐리 큐빗인 c_0만을 사용하였으며, 이는 연산 후에 0으로 초기화되기 때문에 다음 Addition 연산에서 재사용함으로써 큐비트의 수를 최적화 하였다. 블록 암호의 암호화 과정은 크게 키 스케줄과 라운드 함수, 크게 2단계로 나뉘어 수행된다. 큐비트의 수를 최소화하기 위하여 라운드 키를 생성하고 라운드 함수를 수행하는데 있어 on-the-fly 방식을 활용하였다. on-the-fly 방식은 암호화에 필요한 모든 라운드 키를 생성해두는 것이 아닌, 라운드 함수

수행 전에 해당 라운드 함수의 라운드 키만을 생성하는 방식이다.

3.1 HIGHT

본 장에서는 제안하는 HIGHT의 양자 구현 최적화 기법에 대하여 살펴본다.

3.1.1 키 스케줄

HIGHT에서 사용하는 큐비트의 수를 줄이기 위해, 모든 라운드 키를 한 번에 생성해두지 않고 라운드 함수와 키 스케줄을 병행하여 수행한다. 2.2.2을 보면 HIGHT의 키 스케줄에서는 $\delta_0 = (s_6, s_5, s_4, s_3, s_2, s_1, s_0)$가 제일먼저 사용되고 $(1 \le i \le 127)$에 대하여 새로운 δ_i를 생성한다. 이때 기존 큐비트들을 재사용함으로써 큐비트 수를 최적화 하였다. 예를 들어 δ_1을 생성할 때 새로운 s_7가 생성된다. 그리고 s_0는 새로운 δ_1에 포함되지 않는다. 그러므로 s_7을 위한 새로운 큐비트를 할당하지 않고 기존 s_0 큐비트에 연산을 한 뒤 새로운 값 s_7으로 취급한다. 이에 대한 자세한 과정은 [알고리즘 3-1]에 나타나 있다.

Algorithm 1 Generate δ_i of HIGHT

Input: $\delta_{i-1} = (s_{(i+5)\%7}, s_{(i+4)\%7}, s_{(i+3)\%7}, s_{(i+2)\%7}, s_{(i+1)\%7}, s_{i\%7}, s_{(i-1)\%7})$

Output: δ_i

1: $s_{(i-1)\%7} \leftarrow \mathrm{CNOT}(s_{(i+2)\%7}, s_{(i-1)\%7})$

2: **return** $\delta_i = (s_{(i-1)\%7}, s_{(i+5)\%7}, s_{(i+4)\%7}, s_{(i+3)\%7}, s_{(i+2)\%7}, s_{(i+1)\%7}, s_{i\%7})$

[알고리즘 3-1] δ_i 생성

앞서 언급하였듯이, 큐비트의 개수를 줄이기 위해 키 스케줄을 라운드

함수와 병행하여 수행한다. 모든 라운드 키를 한 번에 생성하는 것이 아닌 하나의 라운드 키를 생성한 뒤 바로 라운드 함수에서 사용한다. 라운드 키 생성을 위한 새로운 큐비트를 할당하지 않고 초기 키 $K=(K[0],K[1],...,K[15])$에 키 스케줄을 수행하여 라운드 키 $RK=(RK[0],RK[1],\cdots,RK[15])$로 사용한다. 이에 대한 자세한 과정은 [알고리즘 3-2]에 나타나 있다.

Algorithm 2 i-th key schedule of HIGHT

Input: $\delta_{16\,\cdot\,(i-1)}, K$

Output: K

1: **for** $j=0$ to 7 **do**
2: Generate $\delta_{16\,\cdot\,(i-1)+j}$
3: $RK[16\cdot(i-1)+j] \leftarrow \mathrm{ADD}(\delta_{16\,\cdot\,(i-1)+j}, K[j-i+1 \bmod 8])$
4: Use $RK[16\cdot(i-1)+j]$ in round function
5: Reverse : $ADD(\delta_{16\,\cdot\,(i-1)+j}, K[j-i+1 \bmod 8])$
6: **end for**
7: **for** $j=0$ to 7 **do**
8: Generate $\delta_{16\,\cdot\,(i-1)+j+8}$
9: $RK[16\cdot(i-1)+j+8] \leftarrow \mathrm{ADD}(\delta_{16\,\cdot\,(i-1)+j+8}, K[(j-i+1 \bmod 8)+8])$
10: Use $RK[16\cdot(i-1)+j+8]$ in round function
11: Reverse : $ADD(\delta_{16\,\cdot\,(i-1)+j+8}, K[(j-i+1 \bmod 8)+8])$
12: **end for**
13: **return** $K=(K[0],K[1],\cdots,K[15])$

[알고리즘 3-2] HIGHT 키 스케줄

라운드 키 RK는 라운드 함수에서 사용 후 리버스 연산을 통해 K로 초기화 되고 다음 라운드 키 $RK=(RK[16],RK[17],\cdots,RK[31])$를 생성하는 키 스케줄에서 다시 사용된다.

3.1.2 라운드 함수

HIGHT는 라운드 함수를 수행하기 전, 화이트닝 키 WK를 사용하여 입력 평문 T에 대한 초기 변환을 수행한다. 이에 대한 양자 회로 구현은 [알고리즘 3-3]과 같다.

Algorithm 3 Initial conversion of HIGHT

Input: T, K

Output: X_0

1: $X_0[i] \leftarrow T[i],\ \ i = 1, 3, 5, 7$

2: $X_0[0] \leftarrow$ ADD($K[12]$, $T[0]$)

3: $X_0[2] \leftarrow$ CNOT($K[13]$, $T[2]$)

4: $X_0[4] \leftarrow$ ADD($K[14]$, $T[4]$)

5: $X_0[6] \leftarrow$ ADD($K[15]$, $T[6]$)

6: **return** $X = (X_0[0], X_0[1], \cdots, X_0[7])$

[알고리즘 3-3] HIGHT 초기 변환

제안하는 양자 구현에서는 라운드 함수의 연산 대상과 순서를 적절히 배치함으로써 최적화 하였다. 제안하는 라운드 함수의 양자 구현은 [알고리즘 3-4]와 같으며 이에 대해 자세히 살펴보고자 한다.

Algorithm 4 Round function of HIGHT

Input: Round i, X_{i-1}, RK

Output: X_i

First operation:
  $X_{i-1}[0] \leftarrow F_0(X_{i-1}[0])$
  $X_{i-1}[0] \leftarrow \text{CNOT}(RK[4i-4], X_{i-1}[0])$
  $X_i[2] \leftarrow \text{ADD}(X_{i-1}[0], X_{i-1}[1])$
  $X_{i-1}[0] \leftarrow \text{CNOT}(RK[4i-4], X_{i-1}[0])$ (reverse)
  $X_{i-1}[0] \leftarrow F_0\_\text{reverse}(X_{i-1}[0])$
Second operation:
  $X_{i-1}[2] \leftarrow F_0(X_{i-1}[2])$
  $X_{i-1}[2] \leftarrow \text{ADD}(RK[4i-3], X_{i-1}[2])$
  $X_i[4] \leftarrow \text{CNOT}(X_{i-1}[2], X_{i-1}[3])$
  $X_{i-1}[2] \leftarrow \text{ADD}(RK[4i-3], X_{i-1}[2])$ (reverse)
  $X_{i-1}[2] \leftarrow F_0\_\text{reverse}(X_{i-1}[2])$
Third operation:
  $X_{i-1}[4] \leftarrow F_0(X_{i-1}[4])$
  $X_{i-1}[4] \leftarrow \text{CNOT}(RK[4i-2], X_{i-1}[4])$
  $X_i[6] \leftarrow \text{ADD}(X_{i-1}[4], X_{i-1}[5])$
  $X_{i-1}[4] \leftarrow \text{CNOT}(RK[4i-2], X_{i-1}[4])$ (reverse)
  $X_{i-1}[4] \leftarrow F_0\_\text{reverse}(X_{i-1}[4])$
Fourth operation:
  $X_{i-1}[6] \leftarrow F_0(X_{i-1}[6])$
  $X_{i-1}[6] \leftarrow \text{ADD}(RK[4i-1], X_{i-1}[6])$
  $X_i[0] \leftarrow \text{ADD}(X_{i-1}[6], X_{i-1}[7])$
  $X_{i-1}[6] \leftarrow \text{CNOT}(RK[4i-1], X_{i-1}[6])$ (reverse)
  $X_{i-1}[6] \leftarrow F_0\_\text{reverse}(X_{i-1}[6])$
Last operation:
  $X_i[j] = X_{i-1}[j-1], \ j = 1, 3, 5, 7$
return $X_i = (X_i[0], X_i[1], \cdots, X_i[7])$

[알고리즘 3-4] HIGHT 라운드 함수

라운드 키 RK는 생성된 뒤, 바로 사용되고 리버스 연산을 통해 K로 초기화 된다. 때문에 라운드 함수의 연산을 라운드 키 RK의 생성 순서에

맞춰 수행하였다. 또한 큐비트 사용을 최소화하기 위해 새로운 X_i를 위한 큐비트를 할당하지 않고 기존 X_{i-1}에 연산을 수행하였다. [수식 2-6]의 첫 번째 줄과 세 번째 줄 그리고 [알고리즘 3-4]의 두 번째 줄을 보면 [수식 2-6]의 첫 번째 줄에서 $X_{(i-1)}[0]$은 라운드 함수에서 제일 먼저 계산되고 세 번째 줄에서 다시 사용된다. 그러므로 $X_{(i-1)}[0]$의 값은 변경되어서는 안 된다. 그러나 세 번째 줄의 $X_{(i-1)}[1]$는 한번 사용된 후 더 이상 사용되지 않는다. 그러므로 새로운 값인 $X_i[2]$를 기존 $X_{i-1}[1]$에 연산하여 만들어주면 큐비트를 절약할 수 있다. 이와 같은 방식으로 라운드 함수의 연산을 추가 큐비트 없이 모두 수행할 수 있다. 이에 대한 자세한 과정은 [알고리즘 3-4]에서 확인할 수 있다.

라운드 함수에서 사용되는 보조 함수 F_0, F_1 최적화를 통해 사용 큐비트를 대폭 줄일 수 있었으며 CNOT 게이트 사용 또한 감소하였다. F_0, F_1은 입력 값에 Rotation 연산 한 값을 모두 XOR 연산하는 보조 함수이다. 입력 값이 8-bit 이기 때문에 만약 생성되는 값을 위한 큐비트를 할당한 뒤, Rotation 값을 XOR 한다면 $8\times3=24$ 개의 CNOT 게이트와 8개의 새로운 큐빗이 필요하다. 본 논문에서는 보조 함수를 효율적으로 최적화 하였다. $F_{0,}F_1$의 출력 값을 위한 큐비트를 할당하지 않고 입력 값을 뒤섞어 줌으로써 원하는 값을 생성한다. 그 결과, F_0를 추가 큐빗 없이 21개의 CNOT 게이트로 구현하였으며 F_1는 추가 큐빗 없이 24개의 CNOT 게이트로 구현하였다. 보조 함수 $F_{0,}F_1$ 구현에 대한 자세한 내용은 [알고리즘 3-5], [알고리즘 3-6]에 나타나 있다.

Algorithm 5 $F_0(X[i])$ of HIGHT

Input: $X[i]$

Output: $X_{new}[i]$

1: CNOT($X[i][4], X[i][5]$), CNOT($X[i][3], X[i][4]$)
2: CNOT($X[i][2], X[i][3]$), CNOT($X[i][2], X[i][0]$)
3: CNOT($X[i][4], X[i][2]$), CNOT($X[i][5], X[i][2]$)
4: $X_{new}[i][4] \leftarrow X[i][2]$
5: CNOT($X[i][7], X[i][5]$)
6: $X_{new}[i][6] \leftarrow X[i][5]$
7: CNOT($X[i][6], X[i][4]$)
8: $X_{new}[i][5] \leftarrow X[i][4]$
9: CNOT($X[i][0], X[i][3]$), CNOT($X[i][1], X[i][3]$)
10: $X_{new}[i][2] \leftarrow X[i][3]$
11: CNOT($X[i][6], X[i][1]$), CNOT($X[i][7], X[i][1]$)
12: $X_{new}[i][0] \leftarrow X[i][1]$
13: CNOT($X[i][7], X[i][0]$)
14: $X_{new}[i][1] \leftarrow X[i][0]$
15: CNOT($X[i][5], X[i][6]$), CNOT($X[i][4], X[i][6]$)
16: CNOT($X[i][3], X[i][6]$), CNOT($X[i][1], X[i][6]$)
17: $X_{new}[i][7] \leftarrow X[i][6]$
18: CNOT($X[i][6], X[i][7]$), CNOT($X[i][5], X[i][7]$)
19: CNOT($X[i][1], X[i][7]$), CNOT($X[i][0], X[i][7]$)
20: $X_{new}[i][3] \leftarrow X[i][7]$
21: **return** $X_{new} = (X_{new}[0], X_{new}[1], \cdots, X_{new}[7])$

[알고리즘 3-5] 보조함수 F_0

Algorithm 6 $F_1(X[i])$ of HIGHT

Input: $X[i]$

Output: $X_{new}[i]$

1: CNOT($X[i][3], X[i][4]$), CNOT($X[i][1], X[i][4]$)
2: $X_{new}[i][7] \leftarrow X[i][4]$
3: CNOT($X[i][2], X[i][3]$), CNOT($X[i][0], X[i][3]$)
4: $X_{new}[i][6] \leftarrow X[i][3]$
5: CNOT($X[i][1], X[i][2]$), CNOT($X[i][7], X[i][2]$)
6: $X_{new}[i][5] \leftarrow X[i][2]$
7: CNOT($X[i][0], X[i][1]$), CNOT($X[i][6], X[i][1]$)
8: $X_{new}[i][4] \leftarrow X[i][1]$
9: CNOT($X[i][7], X[i][0]$), CNOT($X[i][5], X[i][0]$)
10: $X_{new}[i][3] \leftarrow X[i][0]$
11: CNOT($X[i][6], X[i][7]$), CNOT($X[i][4], X[i][7]$)
12: CNOT($X[i][3], X[i][7]$), CNOT($X[i][2], X[i][7]$)
13: CNOT($X[i][0], X[i][7]$), CNOT($X[i][5], X[i][7]$)
14: $X_{new}[i][2] \leftarrow X[i][7]$
15: CNOT($X[i][0], X[i][6]$), CNOT($X[i][7], X[i][6]$)
16: CNOT($X[i][4], X[i][6]$), CNOT($X[i][1], X[i][6]$)
17: $X_{new}[i][1] \leftarrow X[i][6]$
18: CNOT($X[i][7], X[i][5]$), CNOT($X[i][6], X[i][5]$)
19: CNOT($X[i][3], X[i][5]$), CNOT($X[i][0], X[i][5]$)
20: $X_{new}[i][0] \leftarrow X[i][5]$
21: **return** $X_{new} = (X_{new}[0], X_{new}[1], \cdots, X_{new}[7])$

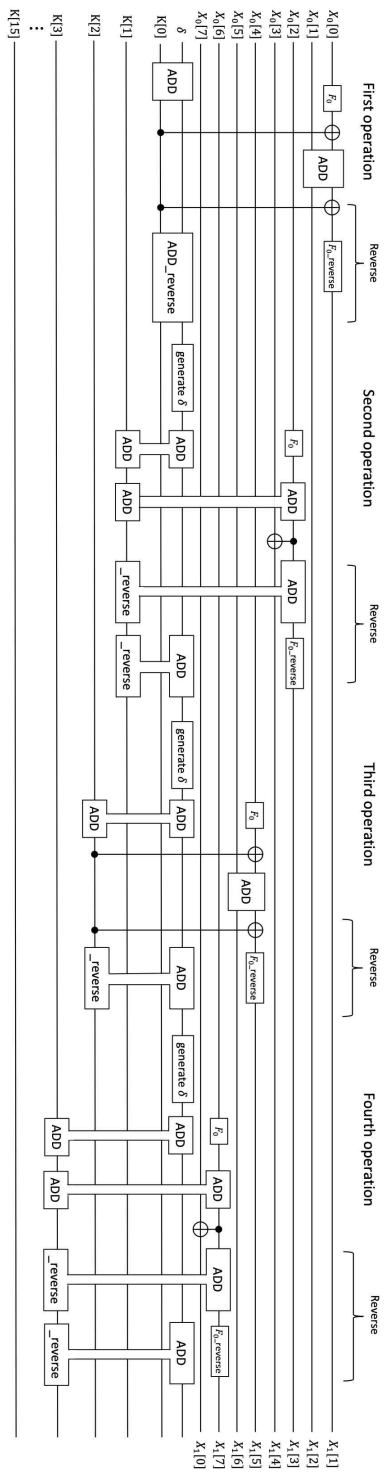
[알고리즘 3-6] 보조함수 F_1

라운드 함수에서 F_0, F_1을 사용 후에는 다시 기존 입력 값으로 되돌려 주어야 한다. 이를 위해 F_0, F_1에 대한 리버스 함수 F_0_reverse와 F_1_reverse가 필요하다. 이는 기존 F_0, F_1 연산을 거꾸로 수행하여 간단히 구현할 수 있다.

HIGHT의 마지막 32번째 라운드 함수에서는 [수식 2-7]이 수행된다. 이는 [알고리즘 3-4]와 비교하여 가장 큰 차이점은 F_1을 사용하는 것이다. 그러므로 [알고리즘 3-4]와 거의 유사하기 때문에 본

논문에서는 생략하였다.

모든 라운드 함수가 수행되면 [수식 2-8]의 최종 변환이 수행되고 암호화가 종료된다. 이 또한 초기 변환인 [알고리즘 3-3]과 유사하기 때문에 생략하였다. 전체적인 HIGHT의 양자 구현 회로는 [그림 3-1]에 나타나 있다.



[그림 3-1] HIGHT 양자 회로

3.2 CHAM

본 장에서는 제안하는 CHAM의 양자 구현 최적화 기법에 대하여 살펴본다.

3.2.1 키 스케줄

CHAM의 키 스케줄은 HIGHT의 보조 함수 F_0, F_1와 동일한 구조를 가지고 있다. 초기 키 K에 Rotation 연산한 값끼리 XOR 연산하여 라운드키 RK를 생성한다. 그러므로 HIGHT의 보조 함수를 최적화 한 기법이 CHAM의 키 스케줄에도 동일하게 적용된다. K 값을 저장하고 있는 큐비트끼리 CNOT 게이트를 수행하여 라운드 키 RK를 생성한다. [수식 2-9]의 키 스케줄에서 첫 번째 줄의 연산은 추가 큐비트 없이 최적화하였다. 이는 [알고리즘 3-7]에 자세히 설명되어 있다. [수식 2-9]의 두 번째 줄의 연산은 첫 번째 줄 연산만큼은 아니지만 3개의 추가 큐빗을 사용하여 최적화하였다. 이에 대한 자세한 사항은 [알고리즘 3-8]에 나타나 있다. HIGHT의 보조 함수와 동일하게 CHAM의 키 스케줄에서 K는 라운드 키 RK로 사용된 후, 다시 원래 K값으로 돌려주어야 한다. 따라서 CHAM의 키 스케줄에서도 연산 순서를 거꾸로 수행하는 리버스 연산이 필요하다.

Algorithm 7 $RK[i]$ key schedule of CHAM

Input: $K[i]$

Output: $RK[i]$

1: CNOT($K[i][0], K[i][8]$), CNOT($K[i][1], K[i][9]$)
2: CNOT($K[i][2], K[i][10]$), CNOT($K[i][3], K[i][11]$)
3: CNOT($K[i][4], K[i][12]$), CNOT($K[i][5], K[i][13]$)
4: CNOT($K[i][6], K[i][14]$), CNOT($K[i][15], K[i][6]$)
5: **for** $j = 0$ to 5 **do**
6: CNOT($K[i][j+9], K[i][j]$), $RK[i][j+1] \leftarrow K[i][j]$
7: **end for**
8: CNOT($K[i][7], K[i][6]$), $RK[i][7] \leftarrow K[i][6]$
9: CNOT($K[i][8], K[i][15]$), $RK[i][0] \leftarrow K[i][15]$
10: CNOT($K[i][8], K[i][7]$), $RK[i][8] \leftarrow K[i][7]$
11: **for** $j = 0$ to 6 **do**
12: CNOT($K[i][j], K[i][j+8]$), $RK[i][j+9] \leftarrow K[i][j+8]$
13: **end for**
14: **return** $RK[i] = (RK[i][0], RK[i][1], \cdots, RK[i][15])$

[알고리즘 3-7] CHAM $RK[i]$ 키 스케줄

Algorithm 8 $RK[i+8\oplus 1]$ key schedule of CHAM

Input: $K[i]$

Output: $RK[i+8\oplus 1]$

1: CNOT($K[i][15]$, temp$[0]$), CNOT($K[i][9]$, temp$[0]$)
2: CNOT($K[i][5]$, temp$[1]$), CNOT($K[i][15]$, temp$[1]$)
3: CNOT($K[i][4]$, temp$[2]$), CNOT($K[i][10]$, temp$[2]$)
4: CNOT($K[i][14]$, $K[i][15]$), CNOT($K[i][4]$, $K[i][15]$)
5: CNOT($K[i][9]$, $K[i][4]$), CNOT($K[i][3]$, $K[i][4]$)
6: $RK[i+8\oplus 1][15]\leftarrow K[i][15]$, $RK[i+8\oplus 1][4]\leftarrow K[i][4]$
7: CNOT($K[i][14]$, $K[i][9]$), CNOT($K[i][8]$, $K[i][9]$)
8: CNOT($K[i][13]$, $K[i][14]$), CNOT($K[i][3]$, $K[i][14]$)
9: $RK[i+8\oplus 1][9]\leftarrow K[i][9]$, $RK[i+8\oplus 1][14]\leftarrow K[i][14]$
10: CNOT($K[i][8]$, $K[i][3]$), CNOT($K[i][2]$, $K[i][3]$)
11: CNOT($K[i][13]$, $K[i][8]$), CNOT($K[i][7]$, $K[i][8]$)
12: $RK[i+8\oplus 1][3]\leftarrow K[i][3]$, $RK[i+8\oplus 1][8]\leftarrow K[i][8]$
13: CNOT($K[i][12]$, $K[i][13]$), CNOT($K[i][2]$, $K[i][13]$)
14: CNOT($K[i][7]$, $K[i][2]$), CNOT($K[i][1]$, $K[i][2]$)
15: $RK[i+8\oplus 1][13]\leftarrow K[i][13]$, $RK[i+8\oplus 1][2]\leftarrow K[i][2]$
16: CNOT($K[i][12]$, $K[i][7]$), CNOT($K[i][6]$, $K[i][7]$)
17: CNOT($K[i][1]$, $K[i][12]$), CNOT($K[i][1]$, $K[i][12]$)
18: $RK[i+8\oplus 1][7]\leftarrow K[i][7]$, $RK[i+8\oplus 1][12]\leftarrow K[i][12]$
19: CNOT($K[i][6]$, $K[i][1]$), CNOT($K[i][0]$, $K[i][1]$)
20: CNOT($K[i][11]$, $K[i][6]$), CNOT($K[i][5]$, $K[i][6]$)
21: $RK[i+8\oplus 1][1]\leftarrow K[i][1]$, $RK[i+8\oplus 1][6]\leftarrow K[i][6]$
22: CNOT($K[i][10]$, $K[i][11]$), CNOT($K[i][0]$, $K[i][11]$)
23: $RK[i+8\oplus 1][11]\leftarrow K[i][11]$
24: CNOT(temp$[0]$, $K[i][10]$), CNOT(temp$[1]$, $K[i][0]$)
25: $RK[i+8\oplus 1][10]\leftarrow K[i][10]$, $RK[i+8\oplus 1][0]\leftarrow K[i][0]$
26: CNOT(temp$[2]$, $K[i][5]$)
27: $RK[i+8\oplus 1][5]\leftarrow K[i][5]$
28: **return** $RK[i+8\oplus 1]=(RK[i+8\oplus 1][0],\cdots,RK[i+8\oplus 1][15])$

[알고리즘 3-8] CHAM $RK[i+8\oplus 1]$ 키 스케줄

3.2.2 라운드 함수

CHAM의 라운드 함수 [수식 2-10]의 두 번째 줄을 보면, 입력 X_i 값 중 $X_i[1], X_i[2], X_i[3]$은 새로운 값인 X_{i+1}로 사용된다. 그러나 $X_i[0]$은 연산

후 다시 사용되지 않는다. 그러므로 두 번째 줄의 $X_i[0]$ 큐비트는 새로운 값인 $X_{i+1}[3]$을 저장하는 큐비트로 사용한다. 두 번째 줄에서 상수 i를 $X_i[0]$에 XOR 연산한다. 이는 i의 비트가 1인 위치에 따라 $X_i[0]$에 X 게이트를 통과시키는 것으로 대체할 수 있다. 예를 들어 만약 $i=3=(0,0,\cdots,1,1)$ 라면, X 게이트를 $X_3[0][0], X_3[0][1]$ 큐비트에 수행한다. 이를 통해 상수 i를 위한 큐비트를 할당하지 않고 라운드 함수를 수행할 수 있다.

CHAM의 라운드 함수 [수식 2-10], [수식 2-11]에 대한 양자 구현은 [알고리즘 3-9]와 같으며 CHAM-64/128에 대한 전체적인 양자 회로는 [그림 3-2]와 같다.

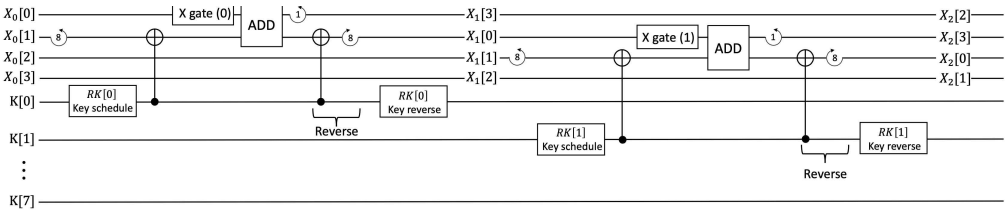
Algorithm 9 Round function of CHAM

Input: RK, X_i, Round i

Output: X_{i+1}

if i is odd **then**
  $X_i[1] \leftarrow X_i[1] \lll 1$
  $X_i[1] \leftarrow \text{CNOT}(RK[i \bmod 2m], X_i[1])$
  $X_i[0] \leftarrow \text{X gate}(i, X_i[0])$
  $X_i[0] \leftarrow \text{ADD}(X_i[1], X_i[0])$
  $X_i[1] \leftarrow \text{CNOT}(RK[i \bmod 2m], X_i[1])$ (reverse)
  $X_i[1] \leftarrow X_i[1] \ggg 1$ (reverse)
  $X_{i+1}[3] \leftarrow X_i[0] \lll 8$
  $X_{i+1}[j] \leftarrow X_i[j+1], \quad j = 0, 1, 2$
else
  $X_i[1] \leftarrow X_i[1] \lll 8$
  $X_i[1] \leftarrow \text{CNOT}(RK[i \bmod 2m], X_i[1])$
  $X_i[0] \leftarrow \text{X gate}(i, X_i[0])$
  $X_i[0] \leftarrow \text{ADD}(X_i[1], X_i[0])$
  $X_i[1] \leftarrow \text{CNOT}(RK[i \bmod 2m], X_i[1])$ (reverse)
  $X_i[1] \leftarrow X_i[1] \ggg 8$ (reverse)
  $X_{i+1}[3] \leftarrow X_i[0] \lll 1$
  $X_{i+1}[j] \leftarrow X_i[j+1], \quad j = 0, 1, 2$
end if
return $X_{i+1} = (X_{i+1}[0], X_{i+1}[1], X_{i+1}[2], X_{i+1}[3])$

[알고리즘 3-9] CHAM 라운드 함수



[그림 3-2] CHAM-64/128 양자 회로

3.3 LEA

본 장에서는 제안하는 LEA의 양자 구현 최적화 기법에 대하여 살펴본다.

3.3.1 키 스케줄

[수식 2-13]의 LEA-128 키 스케줄을 보면, 첫 번째 라운드 키 RK_0를 초기 키 K 그리고 상수 δ를 사용하여 생성한다. 그런데 다음 라운드 키 RK_1 생성을 위해서는 RK_0가 필요하다. 따라서 LEA 또한 라운드 함수와 키 스케줄을 병행하여 수행한다. 또한 상수 δ 덧셈에서 큐비트의 사용을 최적화 하였다. LEA에서는 최대 8개인 δ[0~8]이 필요한데 하나의 δ 값을 위한 큐비트만을 할당하고 재사용하는 방식을 선택하였다. 사용되는 δ의 값과 순서는 정해져 있다. 따라서 사전에 필요한 δ값을 X 게이트만을 활용해 만들 수 있다. 동작 과정은 다음과 같다. δ[0]가 가장 먼저 사용되기 때문에 선언한 32 큐비트의 δ를 δ[0]의 32 큐비트 값 중 1인 위치에 X 게이트를 통과시킨다. 예를 들어 δ[0]가 만약 0x00000002라면 2번째 큐빗에만 X 게이트를 통과시켜 0에서 1로 반전시켜 값을 완성한다. 사용된 후에는 다음에 사용되는 δ[1]값으로 교체해야 한다. 하지만 기존 컴퓨터와는 다르게 양자 컴퓨터는 새로운 값으로 덮어씌우는 것이 불가능하다. 따라서 기존 값에서 X 게이트를 통과시켜 다음 값을 완성한다. 만약 δ[1]이 0x00000003 이라면 첫 번째 큐비트에만 X 게이트를 통과시켜 값을 완성할 수 있다. 이와 같은 방식으로 키 스케줄에서 사용되는 모든 상수 δ 덧셈을 하나의 32 큐비트 배열만으로 수행할 수 있다. 이해를 돕기 위한 제안하는 δ[0]에서 δ[1]로 바뀌는 과정은 [알고리즘 3-10]과 같으며 LEA-128의 키 스케줄은 [알고리즘 3-11]과 같다.

Algorithm 10 Change $\delta[0]$ to $\delta[1]$

Input: $\delta_{0 \sim 31}$ $(0xc3efe9db)$

Output: $\delta_{0 \sim 31}$ $(0x44626b02)$

1: X gate(δ_0), X gate(δ_3)
2: X gate(δ_4), X gate(δ_6), X gate(δ_7)
3: X gate(δ_9)
4: X gate(δ_{15})
5: X gate(δ_{16}), X gate(δ_{18}), X gate(δ_{19})
6: X gate(δ_{23})
7: X gate(δ_{24}), X gate(δ_{25}), X gate(δ_{26})
8: X gate(δ_{31})
9: **return** $\delta[1] = (\delta_0, \delta_1, \cdots, \delta_{31})$

[알고리즘 3-10] $\delta[0]$에서 $\delta[1]$로의 변환

Algorithm 11 i-th key schedule of LEA-128

Input: $\delta, K(i=1)$ or $RK_{i-2}(i>1)$

Output: RK_{i-1}

1: $K[0] \leftarrow \mathrm{ADD}(\delta[(i-1) \bmod 4] \lll (i-1), K[0])$
2: $K[0] \leftarrow K[0] \lll 1$
3: $K[1] \leftarrow \mathrm{ADD}(\delta[(i-1) \bmod 4] \lll i, K[1])$
4: $K[1] \leftarrow K[1] \lll 3$
5: $K[2] \leftarrow \mathrm{ADD}(\delta[(i-1) \bmod 4] \lll (i-1), K[2])$
6: $K[2] \leftarrow K[2] \lll 6$
7: $K[3] \leftarrow \mathrm{ADD}(\delta[(i-1) \bmod 4] \lll (i-1), K[3])$
8: $K[3] \leftarrow K[3] \lll 11$
9: **return** $RK_{i-1} = (K[0], K[1], K[2], K[1], K[3], K[1])$

[알고리즘 3-11] LEA-128 키 스케줄

3.3.2 라운드 함수

라운드 함수의 구현에 있어 연산 대상과 순서를 적절히 배치함으로써 추가 큐빗을 전혀 사용하지 않았다. [수식 2-14]의 새로운 값 X_{i+1}을 위한 큐비트를 할당하지 않고 X_i에 연산한 뒤, X_{i+1}로 취급한다. 연산은

[수식 2-14]의 세 번째 줄부터 시작한다. $X_i[3]$은 라운드 함수에서 더 이상 사용되지 않기 때문에 $X_{i+1}[2]$ 값으로 사용된다. 세 번째 줄 연산이 끝나면 두 번째 줄의 $X_i[2]$는 더 이상 사용되지 않는다. 따라서 $X_i[2]$는 $X_{i+1}[1]$ 값으로 사용된다. 이에 대한 자세한 과정은 [알고리즘 3-12]에 나타나 있다. LEA-128의 전체적인 양자 회로는 [그림 3-3]과 같다.

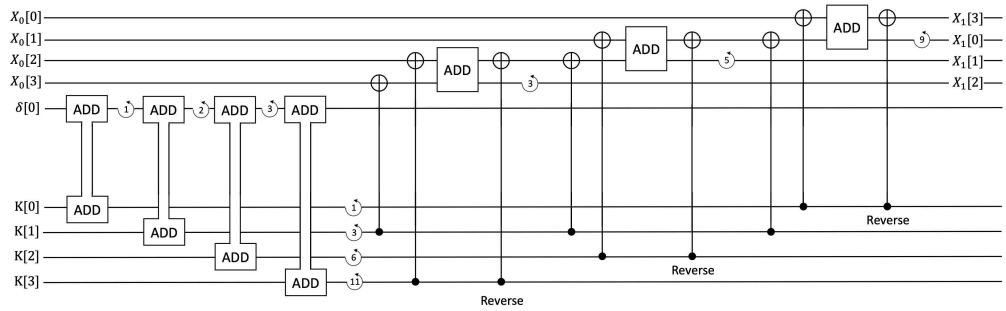
Algorithm 12 Round function of LEA-128

Input: RK, X_i, Round i

Output: X_{i+1}

1: $X_i[3] \leftarrow \text{CNOT}(RK_i[5], X_i[3])$
2: $X_i[2] \leftarrow \text{CNOT}(RK_i[4], X_i[2])$
3: $X_i[3] \leftarrow \text{ADD}(X_i[2], X_i[3])$
4: $X_i[2] \leftarrow \text{CNOT}(RK_i[4], X_i[2])$ (reverse)
5: $X_{i+1}[2] \leftarrow X_i[3] \lll 3$
6: $X_i[2] \leftarrow \text{CNOT}(RK_i[3], X_i[2])$
7: $X_i[1] \leftarrow \text{CNOT}(RK_i[2], X_i[1])$
8: $X_i[2] \leftarrow \text{ADD}(X_i[1], X_i[2])$
9: $X_i[1] \leftarrow \text{CNOT}(RK_i[2], X_i[1])$ (reverse)
10: $X_{i+1}[1] \leftarrow X_i[2] \lll 5$
11: $X_i[1] \leftarrow \text{CNOT}(RK_i[1], X_i[1])$
12: $X_i[0] \leftarrow \text{CNOT}(RK_i[0], X_i[0])$
13: $X_i[1] \leftarrow \text{ADD}(X_i[0], X_i[1])$
14: $X_i[0] \leftarrow \text{CNOT}(RK_i[0], X_i[0])$ (reverse)
15: $X_{i+1}[0] \leftarrow X_i[1] \lll 9$
16: $X_{i+1}[3] \leftarrow X_i[0]$
17: **return** $X_{i+1} = (X_{i+1}[0], X_{i+1}[1], X_{i+1}[2], X_{i+1}[3])$

[알고리즘 3-12] LEA-128 라운드 함수



[그림 3-3] LEA-128 양자 회로

3.4 SPECK

본 장에서는 제안하는 SPECK의 양자 구현 최적화 기법에 대하여 살펴본다.

3.4.1 키 스케줄

[수식 2-15]에서 초기 키 $K=(K[0],l[0],\cdots,l[m-2])$가 $K=(K[0],K[1],\cdots,K[r-1])$로 확장된다 ($r=$ Rounds). 제안하는 구현에선 큐비트를 최소화하기 위해 첫 번째 라운드에서 사용되는 $K[0]$에 다음 라운드 키의 값을 연산하여 사용한다. 이를 위해 한 번에 모든 라운드 키를 생성하지 않고 라운드 함수와 병행하여 키 스케줄을 수행한다. 그 결과 $K[0]$가 처음부터 끝까지 라운드 키로 사용된다. 이러 방식으로 큐비트를 재활용하여 모든 라운드 함수를 초기 키 큐비트만으로 수행한다. $1 \leq i < r$에 대해 수행되며 키 스케줄에 대한 자세한 과정은 [알고리즘 3-13]에 나타나있다.

Algorithm 13 $RK[i]$ key schedule of SPECK

Input: $K=(K[0], l[0], \cdots, l[m-2])$

Output: $RK[i]$

1: $l[(i-1)\%(m-1)] \leftarrow l[(i-1)\%(m-1)] \ggg \alpha$
2: $l[(i-1)\%(m-1)] \leftarrow \mathrm{ADD}(K[0], l[(i-1)\%(m-1)])$
3: $l[(i-1)\%(m-1)] \leftarrow \mathrm{X\ gate}((i-1), l[(i-1)\%(m-1)])$
4: $K[0] \leftarrow K[0] \lll \beta$
5: $K[0] \leftarrow \mathrm{CNOT}(l[(i-1)\%(m-1)], K[0])$
6: $RK[i] \leftarrow K[0]$
7: **return** $RK[i]$

[알고리즘 3-13] SPECK 키 스케줄

3.4.2 라운드 함수

[수식 2-16] 의 라운드 함수는 입력 평문 X와 라운드 키 RK만을 사용한다. X를 업데이트하면서 진행되며 모든 라운드 함수가 수행되면 암호화가 종료된다. 이에 대한 자세한 과정은 [알고리즘 3-14]에 나타나있으며 SPECK-32/64에 대한 전체적인 양자 회로는 [그림 3-4]와 같다.

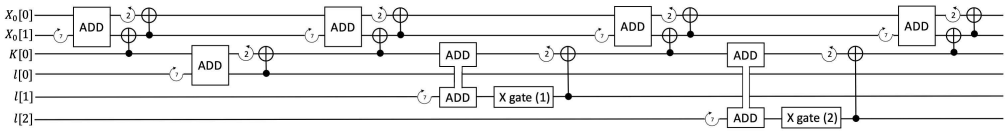
Algorithm 14 Round function of SPECK

Input: RK, X_i, Round i

Output: X_{i+1}

1: $X_i[1] \leftarrow X_i[1] \ggg \alpha$
2: $X_i[1] \leftarrow \mathrm{ADD}(X_i[0], X_i[1])$
3: $X_{i+1}[1] \leftarrow \mathrm{CNOT}(RK_i[i], X_i[1])$
4: $X_i[0] \leftarrow X_i[0] \lll \beta$
5: $X_{i+1}[0] \leftarrow \mathrm{CNOT}(X_i[1], X_i[0])$
6: **return** $X_{i+1} = (X_{i+1}[0], X_{i+1}[1])$

[알고리즘 3-14] SPECK 라운드 함수



[그림 3-4] SPECK-32/64 양자 회로

Ⅳ. 성능 평가

본 논문에서는 성능 평가를 위해 제안 기법을 IBM의 오픈소스 양자 프로그래밍 툴인 ProjectQ를 활용하여 구현하였다. ProjectQ는 양자 시뮬레이션과 양자 자원 측정 기능을 제공한다. 이를 통해 구현 시 사용한 큐비트와 양자 게이트의 개수 그리고 양자 회로의 Depth를 측정할 수 있었다.

4.1 양자 게이트 비용

구현 시 사용된 양자 게이트 중 가장 비싼 자원은 Toffoli 게이트이다. Toffoli 게이트는 6개의 CNOT 게이트와 7개의 T 게이트로 구성되어 있다. 즉 1개의 Toffoli 게이트가 6개의 CNOT 게이트보다 양자 자원을 더 많이 소모한다. 그러므로 성능 평가 시 Toffoli 게이트의 사용 개수가 가장 중요한 요소가 된다. 반면에 X 게이트는 구현 시 사용된 양자 게이트 중 가장 낮은 비용의 양자 게이트이며 전체적으로 사용된 개수가 고려하지 않아도 될 만큼 적다.

4.2 국산 경량 블록 암호 양자 구현 비교

HIGHT, CHAM, LEA 모두 ARX 구조의 경량 블록 암호지만 연산 횟수 또는 구성이 각각 다르다. [표 4-1]에서 국산 암호에 대한 양자

자원들을 분석하였으며 동일한 보안 파라미터 기준 높은 비용에 대해서 짙은 회색으로 표현되어 있다.

[표 4-1] 국산 암호에 대한 양자 자원 비교

Block Cipher	Plaintext (bit)	Key (bit)	Qubit	Toffoli gate	CNOT gate	X gate	Depth
CHAM	64	128	196	2,400	12,285	240	7,807
	128	128	268	4,960	26,885	240	19,880
	128	256	396	5,952	32,277	304	23,856
HIGHT	64	128	201	6,272	20,523	4	16,447
LEA	128	128	289	10,416	28,080	352	26,329
	128	192	353	15,624	39,816	398	39,453
	128	256	417	17,856	45,504	465	45,058

4.2.1 64-bit 평문, 128-bit 키

제일 먼저, CHAM-64/128과 HIGHT-64/128에 대해 양자 자원 관점에서 분석한다. 큐비트, Toffoli 게이트, CNOT 게이트의 자원이 모두 HIGHT-64/128이 CHAM-64/128이 보다 많이 사용된다.
CHAM-64/128의 경우, 라운드 함수 한번동안 16-bit 덧셈이 단 한번 수행된다. 반면에 HIGHT-64/128은 8-bit 덧셈이 리버스 연산을 포함하여 14번 수행된다. 그 결과, HIGHT가 라운드 횟수가 더 많은 CHAM 보다 많은 양자 자원이 요구됨을 보였다. 오직 X 게이트만이 HIGHT가 더 적은데 이는 CHAM에서 상수 i의 XOR 연산을 X 게이트로 대체하였기 때문이다.

4.2.2 128-bit 평문, 128-bit 키

두 번째, CHAM-128/128과 LEA-128/128에 대한 양자 자원을 분석한다. 큐비트, Toffoli 게이트, CNOT 게이트, X 게이트 모두 LEA-128/128에서 더 많이 사용된다. 큐비트 관점에서는 LEA는 상수 δ 덧셈을 위해 32개의 큐비트를 추가로 할당하기 때문에 더 많은 큐비트를

사용한다. 그리고 CNOT 게이트와 Toffoli 게이트 측면에서 LEA는 대부분 덧셈 연산으로 구성되어 있다. X 게이트는 LEA에서 δ 덧셈에서 큐비트를 최적화 하기 위해 X 게이트를 추가로 사용하였다. 종합해보았을 때 LEA는 모든 측면에서 CHAM보다 많은 양자 자원을 요구한다.

4.2.3 LEA 그리고 HIGHT 덧셈 연산 비교

마지막으로, LEA와 HIGHT의 덧셈 연산에 대하여 비교 한다. LEA-128은 키 스케줄에서 4번의 덧셈, 라운드 함수에서 3번의 덧셈으로 총 7번의 덧셈을 수행한다. HIGHT는 키 스케줄에서 4번, 라운드 함수에서 4번 총 8번의 덧셈을 수행한다. 기존 컴퓨터와는 다르게 양자 컴퓨터에서는 이전 값으로 되돌려 주기 위해서는 반드시 앞서 수행한 연산을 그대로 반복하는 리버스 연산을 수행해야 한다. LEA의 덧셈 값은 계속해서 사용되기 때문에 리버스 연산이 필요하지 않다. 반면에 HIGHT는 큐비트의 사용을 최소화하기 위해 덧셈 값을 사용 후 리버스 연산으로 다시 원래 값으로 돌려준다. 때문에 6번의 덧셈 리버스 연산이 수행되어 CNOT 게이트와 Toffoli 게이트의 사용이 증가하였다.

4.2.4 국산 경량 암호 안전성 분석

전체적으로 CHAM이 가장 적은 양자 자원을 요구한다. 이러한 결과는 기존 컴퓨터에 대한 보안 강도가 양자 컴퓨터의 공격 자원에 영향을 끼치지 않는다는 것을 보여준다. 또한 현대 경량 암호가 양자 컴퓨터의 공격으로부터 좋은 선택지는 아님을 알 수 있다.

4.3 소프트웨어, 하드웨어 최적화 암호 양자 구현 비교

본 장에서는 소프트웨어에 최적화된 암호와 하드웨어에 최적화된 암호를 양자 자원의 관점에서 분석한다. 소프트웨어에 최적화된 SPECK은 Addition, XOR, Rotation 연산으로 구성되어 있는 반면, SIMON은 AND, XOR, Rotation 연산으로 구성되어 있다. 주요 차이점은 Addition 과 AND 연산이다. [표 4-2]과 [표 4-3]에 SPECK과 SIMON 구현 시 필요한 양자 자원이 나타나있으며 X 게이트를 제외하고 SIMON보다 SPECK에 많이 요구되는 것을 볼 수 있다. 따라서 하드웨어 최적화 암호가 소프트웨어 최적화 암호보다 양자 컴퓨터의 공격에 더 취약하다는 것을 알 수 있다.

[표 4-2] SPECK 양자 자원

Plaintext (bit)	Key (bit)	Qubit	Toffoli gate	CNOT gate	X gate	Depth
32	64	97	1,290	3,706	42	3,313
48	72	121	1,982	5,606	42	4,969
48	96	145	2,074	5,866	45	5,203
64	96	161	3,162	8,890	54	8,009
64	128	193	3,286	9,238	57	8,323
96	96	193	5,172	14,436	60	12,923
96	144	241	5,360	14,960	64	13,397
128	128	257	7,942	22,086	75	19,797
128	192	321	8,192	22,784	80	20,427
128	256	385	8,444	23,484	81	21,061

[표 4-3] SIMON 양자 자원

Plaintext (bit)	Key (bit)	Qubit	Toffoli gate	CNOT gate	X gate	Depth
32	64	96	512	2,816	448	946
48	72	120	864	3,312	792	1,062
48	96	144	864	4,800	768	1,597
64	96	160	1,344	5,184	1,248	1,674
64	128	192	1,408	7,396	1,216	2,643
96	96	192	2,496	9,792	2,400	4,785
96	144	240	2,592	10,080	2,448	3,282
128	128	256	4,352	17,152	4,224	8,427
128	192	320	4,416	17,472	4,224	5,656
128	256	384	4,608	26,624	4,352	8,848

Ⅳ. 결론

양자 구현 최적화의 관점은 구현에 필요한 큐비트와 양자 게이트를 최소화하는 것이다. 본 논문에서는 제안하는 기법을 통해 리버스 연산을 활용하거나 연산 대상 및 순서를 알맞게 배치함으로써 큐비트와 양자 게이트의 사용을 최소화 하였다. 구현한 국산 암호 양자 회로를 기반으로 Grover 알고리즘 적용에 필요한 양자 자원들이 결정된다. 그 결과 CHAM에 가장 적은 양자 자원이 요구되는 것을 확인하였다. 또한 SPECK과 SIMON을 양자 자원 측면에서 평가한 결과, 하드웨어에 최적화된 암호의 연산은 양자 컴퓨터에서 쉽게 구현할 수 있어 비교적 낮은 양자 자원으로도 Grover 알고리즘 적용이 가능하였다. 다가오는 양자 컴퓨터 시대에 대비하여 국산 경량 블록 암호의 양자 회로 최적화 구현을 통해 Grover 알고리즘 적용에 필요한 자원들을 최소화 하였고 양자 컴퓨터는 아직 초기 개발 단계이기 때문에 요구되는 양자 자원은 안전성의 지표가 될 수 있다. 미래의 보안 통신을 위해서는 현재 암호들의 안전성을 위협하는 양자 컴퓨터의 공격에 대해 분석하고 이에 대한 대비가 필요할 것이다.

참 고 문 헌

1. 국외문헌

Atzori, L., Iera, A., Morabito, G. (2010) The Internet of Things: A survey. Comput. Netw. 54, 2787-2805.

Biryukov, A., Perrin, L.P. (2017) State of the Art in Lightweight Symmetric Cryptography; Technical Report 2017/511, Cryptology ePrint Archive; University of Luxembourg: Luxembourg.

Hankerson, D., Menezes, A.J., Vanstone, S. (2006) Guide to Elliptic Curve Cryptography; Springer Science & Business Media: Berlin/Heidelberg.

Shor, P.W. (1994) Algorithms for quantum computation: Discrete logarithms and factoring. In Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science, 124-134.

Grover, L.K. (1996) A fast quantum mechanical algorithm for database search. In Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, Philadelphia, 212-219.

Grassl, M., Langenberg, B., Roetteler, M., Steinwandt, R. (2016) Applying Grover's algorithm to AES: Quantum resource estimates. In Post-Quantum Cryptography; Springer: Berlin/Heidelberg, 29-43.

Langenberg, B., Pham, H., Steinwandt, R. (2019) Reducing the Cost of Implementing AES as a Quantum Circuit; Technical Report 2019/854, Cryptology ePrint Archive; University of Luxembourg: Luxembourg.

Jaques, S., Naehrig, M., Roetteler, M., Virdia, F. (2020) Implementing Grover oracles for quantum key search on AES and LowMC. In

Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer: Berlin/Heidelberg, 280-310.

Anand, R., Maitra, A., Mukhopadhyay, S. (2020) Grover on SIMON. arXiv 2020, arXiv:2004.10686.

Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L. (2013) The SIMON and SPECK Families of Lightweight Block Ciphers. IACR Cryptol. ePrint Arch. 2013, 404-449.

Gambetta, J, (2020) IBM's Roadmap For Scaling Quantum Technology, IBM Research Blog.

Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.S., Lee, C., Chang, D., Lee, J., Jeong, K. et al. (2006) HIGHT: A new block cipher suitable for low-resource device. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Springer: Berlin/Heidelberg, 46-59.

Koo, B., Roh, D., Kim, H., Jung, Y., Lee, D.G., Kwon, D. (2017) CHAM: A Family of Lightweight Block Ciphers for Resource-Constrained Devices. In Proceedings of the International Conference on Information Security and Cryptology(ICISC'17).

Roh, D., Koo, B., Jung, Y., Jeong, I.W., Lee, D.G., Kwon, D., Kim, W.H. (2019) Revised Version of Block Cipher CHAM. In Proceedings of the International Conference on Information Security and Cryptology, Springer: Cham,Switzerland, 1-19.

Hong, D., Lee, J.K., Kim, D.C., Kwon, D., Ryu, K.H., Lee, D.G. (2013) LEA: A 128-bit block cipher for fast encryption on common processors. In Proceedings of the International Workshop on Information Security Applications, Springer: Cham, Switzerland, 3-27.

Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P. (2004) A new quantum ripple-carry addition circuit. arXiv 2004, arXiv:quant-ph/0410184.

Draper, T.G. (2000) Addition on a quantum computer. arXiv 2000, arXiv:quant-ph/0008033.

Draper, T.G., Kutin, S.A., Rains, E.M., Svore, K.M. (2004) A logarithmic-depth quantum carry-lookahead adder. arXiv 2004, arXiv:quant-ph/0406142.

ABSTRACT

Optimization of Korean Block Cipher Quantum Circuits to Apply Grover's Algorithm

Jang, Kyung-Bae

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

Grover search algorithm can be used to find the n-bit secret key at the speed of $O(2^{n/2})$ which is most effective quantum attack method for block ciphers. In order to evaluate the Grover search algorithm, the target block cipher should be efficiently implemented in quantum circuits. Recently, many research works evaluated required quantum resources of AES block ciphers by optimizing the expensive substitute layer. Research on lightweight block ciphers, an active field of research, is also gradually taking place. In this paper, we present optimized implementations of every Korean made lightweight block ciphers for quantum computers, which include HIGHT, CHAM, and LEA, and NSA made lightweight block ciphers, namely SPECK. Primitive operations for block ciphers, including Addition, Rotation, and XOR, are finely optimized to achieve the optimal quantum circuit, in terms of qubits, Toffoli gate, CNOT gate, and X gate. Korean block ciphers were optimized with a quantum circuit. Finally, we analyzed the security of the upcoming quantum computer era by estimating quantum resources to implement the Korean lightweight block ciphers.

【Keyword】 Grover search algorithm, Quantum circuit, Lightweight block cipher, HIGHT, CHAM, LEA, SPECK, Quantum computer