mod 2^n상에서의 덧셈에 대한 양자 덧셈기 자원 비교

임세진, 장경배, 양유진, 오유진, 서화정

한성대학교 IT융합공학부

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

01. 서론

- 양자컴퓨터 상에서의 암호의 보안강도 추정 필요

  →NIST는 AES와 SHA2, SHA3에 대한 양자 공격 회로에 요구되는 비용을 기준으로, A 암호를 양자 공격 회로를 통해 해킹하는 비용과 비교하여 해당 암호의 보안강도를 추정함

- 이를 위해서 암호를 양자회로로 구현해야하며, 회로를 최적화하여 구현에 사용되는 양자 자원을 최소화할수록 해킹 비용도 감소하게 됨
  - 암호 알고리즘에 사용되는 덧셈과 같은 기본 연산도 양자회로로 구현되어야 함
  - 암호 알고리즘에서 사용되는 덧셈 연산은 주로 최상위 비트 캐리를 고려하지 않는 mod 2^n상에서의 덧셈임 (ex. 32-bit + 32-bit = 32-bit)

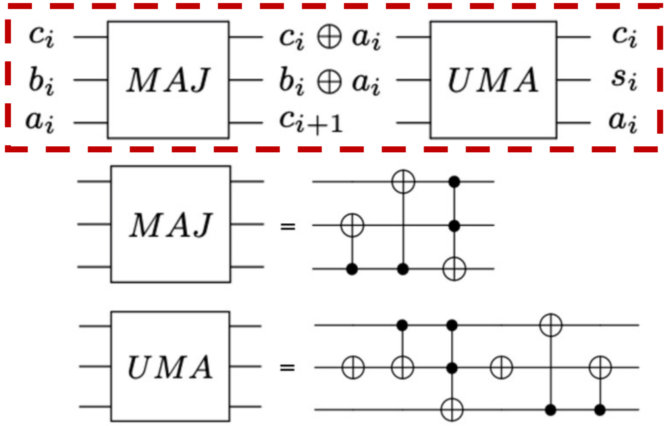
- 본 논문에서는 **다양한 양자 덧셈기를 mod 2^n상에서 구현**하고, **덧셈에 사용되는 양자 자원에 대한 비교를 수행**하고자 함

  ➔ 이러한 비교는 Trade off 관계인 큐비트 수와 회로 깊이 측면에서 회로의 최적화 방향에 따른 덧셈기 식별에 유용할 것으로 사료됨

02. 양자 덧셈기

- 양자 덧셈기는 다항 깊이를 가지는 Ripple-carry 덧셈기와 대수 깊이를 가지는 Carry-lookahead 덧셈기로 나눌 수 있음 (Carry-lookahead 덧셈기는 캐리 연산을 병렬로 수행함으로써 대수 깊이를 갖게 됨)
- **덧셈 결과를 저장하는 위치에 따라** in-place 구현과 out-of-place 구현이 있음
  - In-place : 피연산자 중 하나에 저장 (주로 b에 저장)
  - Out-of-place : 입력받은 결과용 큐비트에 저장 (피연산자의 값은 덧셈 전과 동일하게 유지)
- 일반적으로 n-bit 크기의 두 피연산자의 덧셈 결과는 최상위 캐리 비트를 고려하여 $(n + 1)$-bit가 되지만, 본 논문에서는 mod 2^n의 덧셈을 다루고자 하므로, 덧셈기 회로 알고리즘을 일부 수정함

02. 양자 덧셈기



Algorithm 1 : Cuccaro adder (Addition mod 2^n)

Input: $A_i = a_i,\ B_i = b_i,\ X = 0$

Output: $A_i = a_i,\ B_i = s_i,\ X = 0$

for $i = 1$ to $n-2$: $B_i \oplus = A_i$

$X \oplus = A_1$

$X \oplus = A_0 B_0$; $A_1 \oplus = A_2$

$A_1 \oplus = X B_1$; $A_2 \oplus = A_3$

for $i = 2$ to $n-3$:

  $A_i \oplus = A_{i-1} B_i$; $A_{i+1} \oplus = A_{i+2}$

$A_{n-3} \oplus = A_{n-4} B_{n-3}$; $B_{n-1} \oplus = A_{n-2}$

$B_{n-1} \oplus = A_{n-3} B_{n-2}$; for $i = 1$ to $n-3$: $NOT(B_i)$

$B_1 \oplus = X$; for $i = 2$ to $n-2$: $B_i \oplus = A_{i-1}$

$A_{n-3} \oplus = A_{n-4} B_{n-3}$

for $i = n-4$ down to 2:

  $A_i \oplus = A_{i-1} B_i$; $A_{i+1} \oplus = A_{i+2}$; $NOT(B_{i+1})$

$A_1 \oplus = X B_1$; $A_2 \oplus = A_3$; $NOT(B_2)$

$X \oplus = A_0 B_0$; $A_1 \oplus = A_2$; $NOT(B_1)$

$X \oplus = A_1$

for $i = 0$ to $n-1$: $B_i \oplus = A_i$

알고리즘 1. 수정된 Cuccaro mod 2^n 덧셈기 알고리즘 ($n \geq 4$)

Cuccaro 덧셈기 (CDKM)

- 0으로 초기화된 하나의 보조 큐비트 x를 초기 캐리 비트로 사용하여 덧셈 수행
- In-place 형태의 Ripple-carry 덧셈기로, MAJ 게이트와 UMA 게이트로 구성됨
  - MAJ 게이트에서 수행한 연산을 UMA 게이트에서 해제하여 초기값으로 되돌리고, 덧셈 결과를 b에 저장함
  - 이때 보조 큐비트도 초기 값으로 되돌아가기 때문에 덧셈 연산 이후에도 재사용 가능
- 논문에서 제안한 덧셈기 알고리즘을 **mod 2^n 회로로 수정**
  - n-bit 덧셈 대신 **($n-1$)-bit 덧셈**을 기존 알고리즘대로 구현
  - 최상위 캐리 비트 저장용인 보조 큐비트 z 사용 X → z에 적용되는 연산을 **최상위 비트인 B_{n-1}**에 적용
  - $B_{n-1} \oplus A_{n-1}$ 연산 추가 ← ($n-1$)-bit 덧셈 시에는 최상위 비트의 합은 구해지지 않기 때문
    - 해당 **연산을 삽입하는 위치에 따라** 연산이 병렬로 수행되지 못해 **depth가 커질 수 있어** 맨 마지막에 하위 비트 연산들과 함께 수행되도록 함

02. 양자 덧셈기

Algorithm 2 : mod 2^n Draper adder (out-of-place)

Input: $A_i = a_i$, $B_i = b_i$, $Z_i = 0$, $ancilla_k = 0$
Output: $A_i = a_i$, $B_i = b_i$, $Z_i = s_i$, $ancilla_k = 0$

```
1: for i = 0 to n-2: Z_{i+1} ⊕= A_i B_i
2: for i = 1 to n-2: B_i ⊕= A_i
3: P, G, C, P^{-1}-rounds < n → (n-1) >
4: for i = 0 to n-2: Z_i ⊕= B_i
5: Z_0 ⊕= A_0 ;  for i = 1 to n-2: B_i ⊕= A_i
6: Z_{n-1} ⊕= A_{n-1} ;  Z_{n-1} ⊕= B_{n-1}
```

알고리즘 2. 수정된 out-of-place Draper 덧셈기 알고리즘

Algorithm 3 : mod 2^n Draper adder (in-place)

Input: $A_i = a_i$, $B_i = b_i$, $ancilla1_{i-1} = 0$, $ancilla2_k = 0$
Output: $A_i = a_i$, $B_i = s_i$, $ancilla1_{i-1} = 0$, $ancilla2_k = 0$

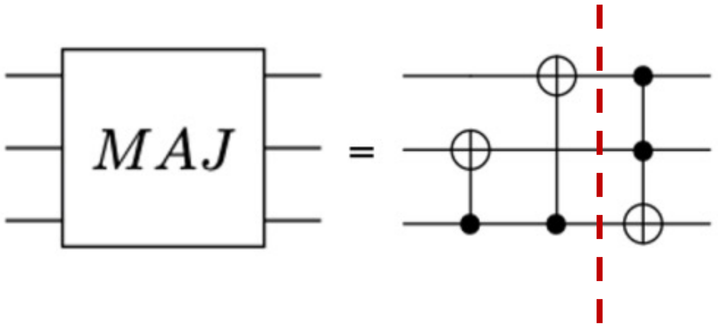
```
1: for i = 0 to n-2: ancilla1_i ⊕= A_i B_i
2: for i = 0 to n-1: B_i ⊕= A_i
3: P, G, C, P^{-1}-rounds < n → (n-1) >
4: for i = 1 to n-1: B_i ⊕= ancilla1_{i-1}
5: for i = 0 to n-2: NOT(B_i)
6: for i = 1 to n-2: B_i ⊕= A_i
7: line 3 reverse
8: for i = 1 to n-2: B_i ⊕= A_i
9: for i = 0 to n-2: ancilla1_i ⊕= A_i B_i
10: for i = 0 to n-2: NOT(B_i)
```

알고리즘 3. 수정된 in-place Draper 덧셈기 알고리즘

Draper 덧셈기

- 대표적인 Carry-lookahead 덧셈기로, in-place와 out-of-place 회로를 모두 제시
- **캐리 연산을 병렬로 수행하기 위해 보조 큐비트가 사용**됨 → 재사용 가능
  - in-place : $-2+2n-w(n-1)-\lfloor \log(n-1) \rfloor$ 개
  - out-of-place : in-place보다 1개의 보조 큐비트를 더 사용 (이 중 n개는 덧셈 결과를 저장하기 위한 큐비트이므로 이를 제외한 나머지만 재활용 가능)
- 캐리를 미리 계산하는 작업은 **하위 비트의 캐리 값에 따른 캐리 발생 여부를 판단하는** P (Propogate) 연산과, **하위 비트의 캐리 값에 관계없이 반드시 캐리가 발생하는지 여부를 판단하는** G (Generate) **연산**을 통해 수행
- Draper 덧셈기에서는 **P, G, C,** P^{-1} 라운드를 통해 , 연산을 수행하여 캐리를 병렬로 계산할 수 있도록 알고리즘을 구성

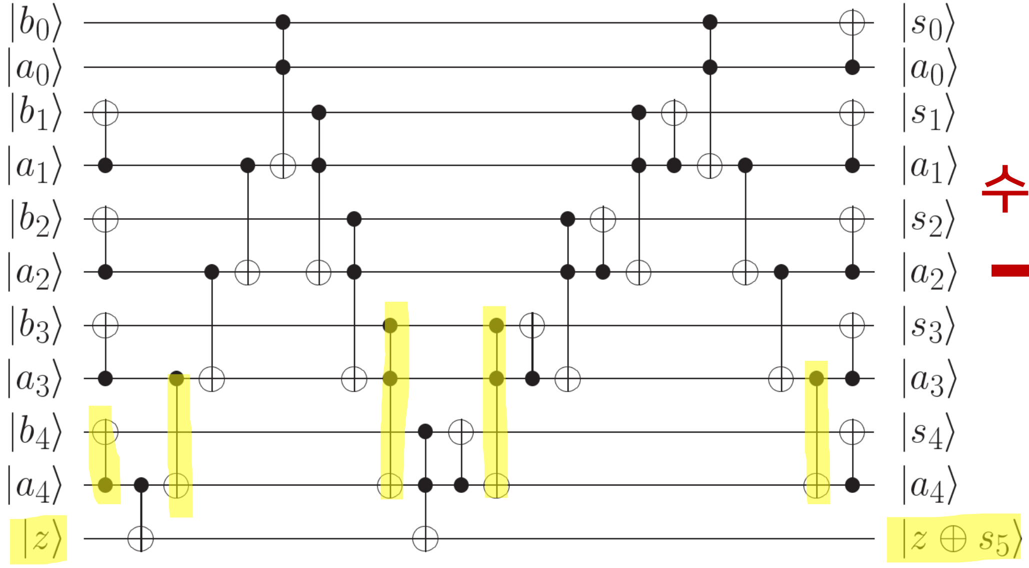
02. 양자 덧셈기



Takahashi 덧셈기

- 보조 큐비트 없이 덧셈을 수행하도록 개선한 Ripple-carry 덧셈기
  - Cuccaro 덧셈기의 MAJ 게이트를 두개의 CNOT 게이트로 구성된 부분과 Toffoli 게이트 부분으로 나누어 적용함

- Cuccaro 덧셈기와 동일한 방식으로 알고리즘 수정



수정 결과

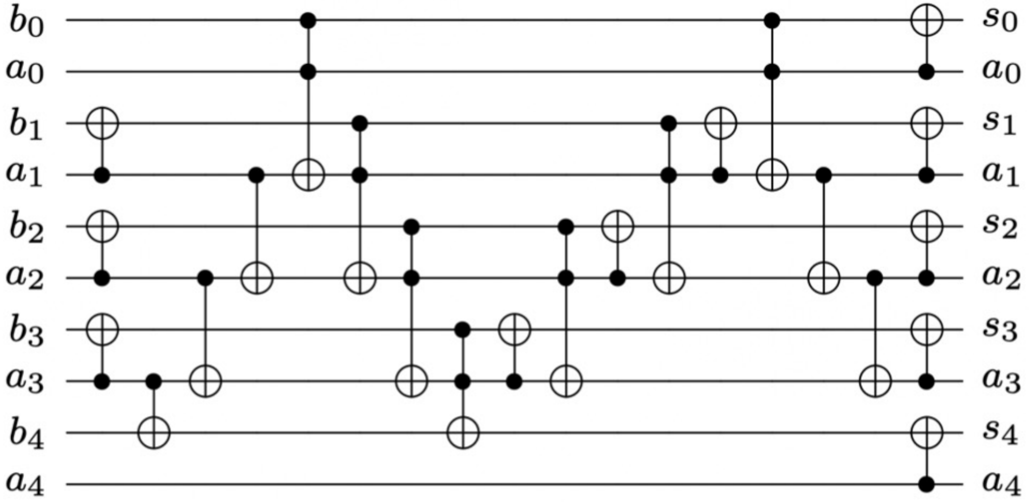


그림 2. mod 2^5에 대해 수정된 Takahashi 덧셈기 회로

03. mod 2^n에서의 양자 덧셈기 자원 비교

- mod 2^n상의 수정된 양자 덧셈기 회로에 사용되는 양자 자원에 대한 비교
  - 3가지 양자 덧셈기를 통한 비트 덧셈에 사용되는 큐비트 수, CNOT 게이트 수, Toffoli 게이트 수, Toffoli-depth, Full-depth
  - 정확한 수치를 확인할 수 있도록 수식으로 정리함

Adder		#Qubit	#CNOT	#Toffoli	Toffoli-depth	Full-depth
Cuccaro [2]		$2n+1$	$5n-7$	$2n-3$	$2n-3$	$2n+2$
Draper [3]	in	$-2+4n-w(n-1)$ $-\lfloor \log(n-1) \rfloor$	$4n-5$	$-12+10n$ $-6w(n-1)$ $-6\lfloor \log(n-1) \rfloor$	$7+2\lfloor \log(n-1) \rfloor$ $+2\left\lfloor \log\frac{n-1}{3} \right\rfloor$	$14+2\lfloor \log(n-1) \rfloor$ $+2\left\lfloor \log\frac{n-1}{3} \right\rfloor$
	out	$-1+4n-w(n-1)$ $-\lfloor \log(n-1) \rfloor$	$3n-2$	$-6+5n-3w(n-1)$ $-3\lfloor \log(n-1) \rfloor$	$4+\lfloor \log(n-1) \rfloor$ $+\left\lfloor \log\frac{n-1}{3} \right\rfloor$	$7+\lfloor \log(n-1) \rfloor$ $+\left\lfloor \log\frac{n-1}{3} \right\rfloor$
Takahashi [4]		$2n$	$5n-9$	$2n-3$	$2n-3$	$5n-8$

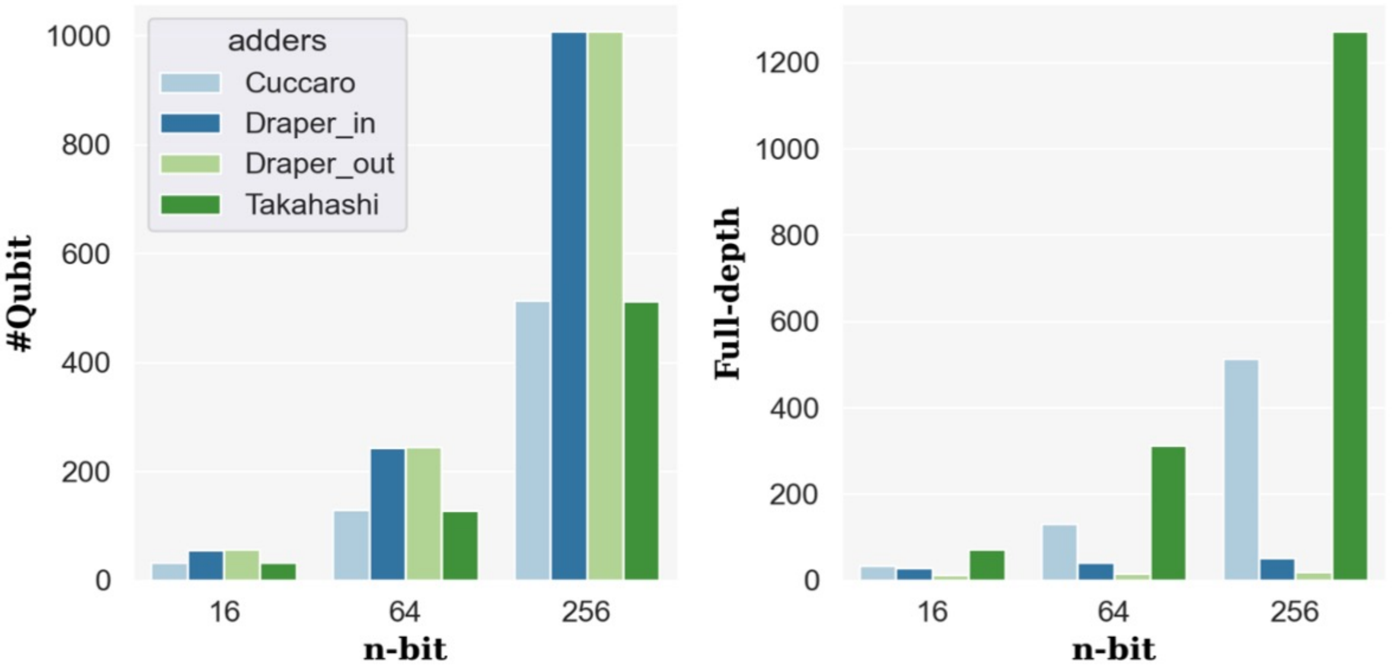
03. mod 2^n에서의 양자 덧셈기 자원 비교

- 그래프를 통한 주요 이점 비교 : Trade off 관계인 큐비트 수와 회로 깊이 측면에서 한눈에 이점을 비교

  → Ripple-carry 덧셈기인 Cuccaro와 Takahashi 덧셈기 : 큐비트를 적게 사용하는 반면, 높은 회로 depth

  → Carry-lookahead 덧셈기인 Draper 덧셈기 : 캐리 병렬 연산을 위해 큐비트 수가 비교적 많이 사용되지만, 병렬 연산으로 인해 낮은 회로 depth

- Takahashi 덧셈기는 보조 큐비트를 사용하지 않는 대신 Cuccaro 덧셈기보다 약 2.5배 높은 전체 깊이를 가지는 것을 알 수 있음



결론적으로 양자 회로 최적화 구현 시에 **큐비트 수를 최소화하는 경우에는 Ripple-carry 덧셈기**를, **회로 depth를 최소화하는 경우에는 Carry-lookahead 덧셈기**를 사용하는 것이 성능 개선에 유리할 것으로 판단

04. 결론

- 본 논문에서는 암호 알고리즘을 양자 회로로 구현하는 과정에서 주로 사용되는 상의 양자 덧셈기에 대해 살펴보고, 수정된 알고리즘 및 회로를 제시하였으며, 사용되는 양자 자원에 대한 비교를 수행하였음
- 해당 연구는 양자 회로의 최적화 방식에 따라 이점을 줄 수 있는 덧셈기를 식별하는데 유용할 것으로 사료됨

Q & A