Grover on SPARKLE*

Yujin Yang, Kyungbae Jang, Gyeongju Song,
Hyunji Kim, and Hwajeong Seo**

IT Department, Hansung University, Seoul, South Korea,
{yujin.yang34, starj1023, thdrudwn98,
khj1594012, hwajeong84}@gmail.com

Abstract. Quantum computers that take advantage of quantum mechanics efficiently model and solve certain hard problems. In particular, quantum computers are considered a major threat to cryptography in the near future. In this current situation, analysis of quantum computer attacks on ciphers is a major way to evaluate the security of ciphers. Several studies of quantum circuits for block ciphers have been presented. However, quantum implementations for Authenticated Encryption with Associated Data (AEAD) are not actively studied.
In this paper, we present a quantum implementation for authenticated ciphers of SPARKLE, a finalist candidate of the National Institute of Standards and Technology (NIST) Lightweight Cryptography (LWC) project. We apply various techniques for optimization by considering trade-off between qubits and gates/depth in quantum computers. Based on proposed quantum circuit, we estimate the cost of applying key search using Grover's algorithm, which degrades the security of symmetric key ciphers. Afterward, we further explore the expected level of post-quantum security for SPARKLE on the basis of post-quantum security requirements of NIST.

Keywords: Quantum Computer · Grover Algorithm · Lightweight Block Cipher · SPARKLE · Authenticated Encryption with Associated Data.

1 Introduction

Quantum computers utilizing quantum mechanics can efficiently model the certain hard problems and solve them in polynomial-time [1]. As these quantum computers develop, the safety of ciphers based on these hard problems is threatened. It is well known that the Shor algorithm can solve the factorization and discrete logarithm problems of Rivest-Shamir-Adleman (RSA) and Elliptic Curve

* This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services, 50%) and this work was partly supported by Institute for Information & communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (<Q|Crypton>, No.2019-0-00033, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity, 50%)
** Corresponding Author

Cryptography (ECC), which are public key cryptosystems, in polynomial-time [2]. In response to the serious security threat to the public key cryptography system, NIST is working on a post-quantum cryptography project to standardize a new cryptographic system that can replace RSA and ECC algorithms.

Grover's Algorithm is well known for reducing the security level of the symmetric key cryptography since it can reduce the complexity of searching for a secret key by as much as square root. Although less of a threat than the Shor algorithm, the Grover algorithm makes it impossible to assert the full security of ciphers.

Accordingly, studies are being conducted to implement ciphers as quantum circuits and to analyze threats to the Grover algorithm. Starting with the most famous symmetric key cipher (i.e. AES) [3,4,5,6], the field of research has expanded to lightweight ciphers [7,8,9,10,11] in recent years. However, the only quantum implementation for AEAD is the quantum implementation for KNOT by Baksi et al [12].

In this paper, we present a quantum implementation of AEAD among the authenticated cipher family of SPARKLE [13], the final candidate of the NIST LWC project. We apply various techniques, such as parallelization and compact reversible operations, to our quantum circuit in order to optimize the quantum circuit by considering the trade-off between qubits and gates/depth. Based on our optimized quantum circuit, we estimate the cost of applying key search by using Grover's algorithm. Finally, we evaluate the post-quantum security level of SPARKLE based on the post-quantum security requirements of the NIST. The source code of the proposed method is provided as open-source[1]

1.1 Our Contribution and Organization

Our contributions are as follows:

1. We report the first quantum implementation of all parameters of SCHWAEMM, which is AEAD of lightweight block cipher SPARKLE.
2. We optimize the quantum circuit of SCHWAEMM by reducing the number of qubits using inverse operations and fake padding. By implementing quantum additions in parallel, the depth of the quantum circuit is reduced, significantly.
3. We estimate the quantum resources for applying Grover's algorithm on SCHWAEMM and evaluate the post-quantum security based on the proposed quantum circuit.

The organization of this paper is as follows. Section 2 presents the background of SPARKLE, quantum implementation, and the Grover search algorithm. In Section 3, the proposed quantum implementation is presented. In Section 4, we evaluate the proposed quantum circuit for SCHWAEMM. Section 5 estimates Grover key search cost for the proposed SCHWAEMM quantum circuit. Further, the NIST post-quantum security level for SCHAEMM is evaluated based on the estimated cost. Finally, Section 6 concludes the paper.

[1] https://github.com/YangYu34/SPARKLE_SCHWAEMM.git.

2 Background

2.1 SPARKLE

SPARKLE is a lightweight cipher designed based on the sponge structure, which is the final candidate for NIST lightweight cipher standardization. SPARKLE consists of ESCH that belongs to hash function family, and SCHWAEMM that belongs to authenticated cipher family.

SPARKLE Permutation SPARKLE permutation, a cryptographic permutation, consists of ARX-box (Addition, Rotation, and XOR) Alzette and linear diffusion layer which has a linear layer of Feistel structure. This permutation is key because it is used in all steps of SCHWAEMM except for Padding Data.

The process of SCHWAEMM consists of 5 steps and is given as:

Padding Data This process pads associated data A and message M according to the given block sizes (e.g. 128, 192, and 256). The padding function $Pad_r(M)$ used when $|M| < r$ performs $M||00000001||0^i$ operation ($i = (-|M| - 1)\ mod\ r$). It can reduce collision resistance by using domain extensions.

State Initialization This process initializes the internal state S. Since S is used throughout the encryption phase, initializing S is important. The sequentially connected value of key K and nonce N becomes S. According to the predetermined size, the left value of S is defined as S_L and the right value of S is defined as S_R. Only state initialization is completed when S is put into the SPARKLE function and then SPARKLE permutation is performed. At this time, it should be noted that the number of SPARKLE permutation rounds varies depending on which operation is performed, depending on the parameter. The number of SPARKLE permutation rounds depends on the parameter.

Processing of Associated Data This process puts S and associated data A that has padded as inputs into feedback function ρ_1, which is calculated on equation (1) In order to prevent trivial collision, if A is the last block, XOR operation is performed with the predefined constant $const_A$ to S_R. After that, S is renewed by performing XOR operation and SPARKLE permutation.

$$\rho_1 : (S, D) \mapsto (S_2||(S_2 \oplus S_1)) \oplus D \tag{1}$$

Encrypting This process is to encrypt the messages M. First of all, S is updated by performing SPARKLE permutation in the same way as in the previous step. At this time, $const_M$, which is predefined together with $const_A$, is used as a constant which is XORed to S_R. After that, S and M are puts to ρ_2 function that is $\rho_2 : (S, M) \mapsto S \oplus M$ in order to encrypt messages. S is put in $trunc_t$ function that returns the t leftmost bits of the input data in order to delete the padded data. As a result, ciphertext C is generated with the same size as M.

Finalization This process generates the final return value which consists of ciphertext C, and an authentication tag T. T is the result of XOR operation between S_R and key K. The final return value is the concatenation of C and T.

Figure 1 shows the process of SCHWAEMM-128/128. In the case of 128, 192, and 256 parameters, the overall structure is the same. Only the size of instances is different. On the other hand, in 256/128, the $W_{c,r}$ process is added to the same structure. $W_{c,r}$, which performs rate-whitening, is ORed with S_R corresponding to the inner part with the outer part.

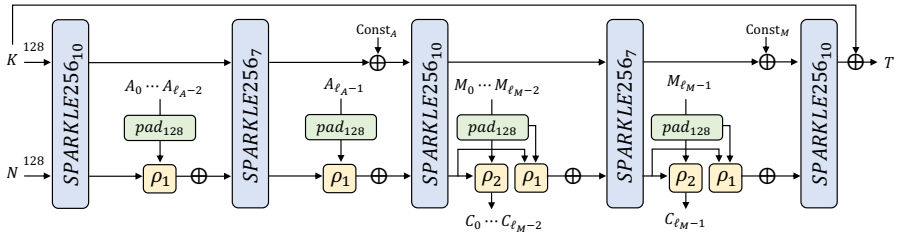


Fig. 1: Process of SCHWAEMM128-128.

2.2 Quantum Gates

Quantum gates are the fundamental building blocks of quantum computing. In this section, we describe the quantum gates needed to implement cryptographic operations of ciphers. Quantum gates that are essential and quite frequently used in implementing quantum circuits of ciphers are shown in Figure 2.

The X gate shown in Figure 2(a), also known as the NOT gate, is equivalent to the classical bit flip and is the simplest reversible logic gate. The NOT gate returns 1-output for 1-input and inverts the received qubit.

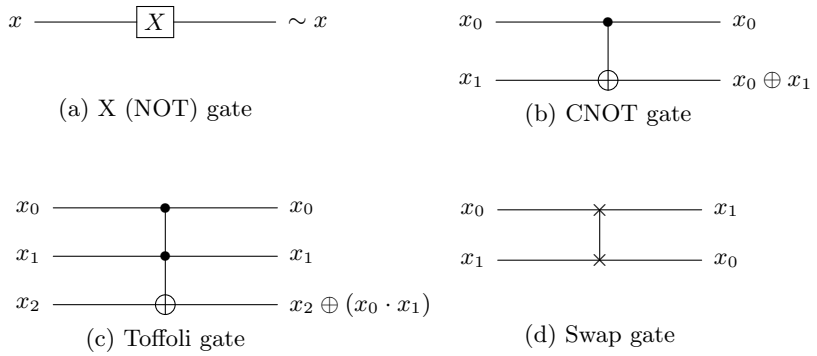
The CNOT gate shown in Figure 2(b) stands for a controlled-not gate and returns a 1-qubit output for a 2-qubit input. If the control qubit is $|1\rangle$, the target qubit is negated. In other words, the target qubit is determined by the control qubit. The CNOT gate is able to generate entangled (i.e. non-separable) states contrary to the single-qubit gates.

The Toffoli gate shown in Figure 2(c) stands for a controlled-controlled-not gate. It returns 1-output for 3-inputs. The target qubit is flipped if the two control qubits are $|1\rangle$.

The Swap Gate shown in Figure 2(d) returns a 2-qubit output by exchanging values for a 2-qubit input.

2.3 Grover's Algorithm for key search

1. Through the use of Hadamard gates, n-qubit key is prepared in superposition $|\psi\rangle$. It has the same amplitude at all state of the qubits then:



(a) X (NOT) gate (b) CNOT gate (c) Toffoli gate (d) Swap gate

Fig. 2: Quantum gates.

$$|\psi\rangle = H^{\otimes n} |0\rangle^{\otimes n} = \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle \tag{2}$$

2. The cipher implemented as a quantum circuit is placed in oracle. In oracle $f(x)$, the plaintext is encrypted using the key of superposition state. Consequently, ciphertexts in regard to all of the key values are generated. By comparing to the known ciphertext, the sign of the solution key is changed to a negative number. The sign is changed to negative by the condition $(f(x) = 1)$, and this condition applies to all states.

$$f(x) = \begin{cases} 1 \text{ if } Enc(key) = c \\ 0 \text{ if } Enc(key) \neq c \end{cases} \tag{3}$$

$$U_f(|\psi\rangle |-\rangle) = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle |-\rangle \tag{4}$$

3. Finally, the diffusion operator serves to amplify the amplitude of the negative sign state. The diffusion operator requires no special skills to implement because it is usually generic.

3 Quantum Implementation of SPARKLE

3.1 SPARKLE Permutation

ARX-box Alzette Addition, XOR, and Rotation operations in quantum are essential in the ARX-box Alzette implementation. The XOR operation is simply implemented using the CNOT gate. Rotation operation can be implemented using Swap gates, but we implement logical swap that only changes the index of qubits. Thus, no quantum resources are used for rotation operations in our quantum implementation.

In classical computers, addition operations do not have that high overhead compared to XOR and AND operations. In contrast, in quantum computers, the overhead for implementing quantum addition is relatively high. Quantum addition is implemented as a combination of several Toffoli, CNOT, and X gates. More quantum resources are required than for XOR (i.e. CNOT) and AND (i.e. Toffoli) operations. Accordingly, several quantum adders for optimizing addition on a quantum computer have been presented [14,15,16,17].

We adopt an improved quantum adder with a ripple-carry approach [17] (i.e. CDKM adder). This quantum adder is efficiently implemented when $n \geq 4$. Since there are only additions with $n = 32$ in our work, we can implement an improved quantum adder, effectively. This quantum adder is further improved for modular additions where the carry value does not need to be computed. As a result, an optimal quantum adder that requires only one ancilla qubit is used in the ARX-box Alzette quantum circuit. For n-qubit quantum addition, $(2n-3)$ Toffoli gates, $(5n-7)$ CNOT gates, and $(2n-6)$ X gates are used, with a depth of $(2n+3)$. Details of the quantum adder implementation can be found in our source code and [17].

In Alzette, *AddConstant*, which XORs the constant c to 32-qubit x, is implemented with only X gates, not CNOT gates. Since the constant c is a known value, we perform X gates on 32-qubit x according to the position where the bit of c is 1. For example, if the condition is $c = 3$, it performs X gates on $x[0]$ and $x[1]$ qubits of 32-qubit x. This saves qubits and CNOT gates by using only X gates. A quantum implementation of the ARX-box Alzette is described in Algorithm 1.

Algorithm 1 Quantum implementation of Alzette.

Input: 32-qubit x and y, Constant c, Adder carry ac
Output: x, y

1: $x \leftarrow ADD((y \ggg 31), x, ac)$
2: $y \leftarrow \text{CNOT32}((x \ggg 24), y)$
3: $x \leftarrow AddConstant(x, c)$
4: $x \leftarrow ADD((y \ggg 17), x, ac)$
5: $y \leftarrow \text{CNOT32}((x \ggg 17), y)$
6: $x \leftarrow AddConstant(x, c)$
7: $x \leftarrow ADD(y, x, ac)$
8: $y \leftarrow \text{CNOT32}((x \ggg 31), y)$
9: $x \leftarrow AddConstant(x, c)$
10: $x \leftarrow ADD((y \ggg 24), x, ac)$
11: $y \leftarrow \text{CNOT32}((x \ggg 16), y)$
12: $x \leftarrow AddConstant(x, c)$
13: **return** x, y

Linear Diffusion Layer In the Feistel round phase of linear diffusion layer denoted by $\mathcal{L}_{n_b}$, $t_x \leftarrow (t_x \oplus (t_x \ll 16) \lll 16)$, it should be performed. However,

implementing this formula as-is will result in unnecessary gate usage. To prevent this, 16 CNOT gates are not used by XORing t_{xL}, which is the left 16 qubits of t_x, to t_{xR}. Next, the 16-qubit rotation process can be implemented using the SWAP gate since the left and right values are simply swapped. However, the left and right of t_x are used interchangeably in order to reduce the resources. In other words, the following XOR operation is divided into 16-qubit and performed using t_{xL} as the right value of t_x and t_{xR} as the left value of t_x. For example, if $y_0 \oplus t_x$ is implemented as a circuit, the operation is performed in this way: $y_{0R} \oplus t_{xL}$, $y_{0L} \oplus t_{xR}$.

Compute() and Uncompute() are meta functions provided by ProjectQ, which, if Compute() designs to perform certain function, Uncompute() reverses the actions performed in Compute(). Instead of allocating qubits to t_x and t_y, recycling x_0, y_0 by utilizing Compute() and Uncompute() saves two qubits.

$\mathcal{L}_4$, $\mathcal{L}_6$, and $\mathcal{L}_8$ are used for SPARKLE256, SPARKLE384, and SPARKLE512, respectively. Details of the implementation is found in Algorithms 2. Since the quantum circuit design of $\mathcal{L}_b$($\mathcal{L}_4$, $\mathcal{L}_6$, and $\mathcal{L}_8$) differs only in the number of variables and the overall structure is the same, only the quantum circuit of $\mathcal{L}_4$ is shown in Algorithm 2.

SPARKLE Permutation The SPAKRLE permutation quantum circuit is implemented by combining the previously introduced ARX-box Alzette and the linear diffusion layer. The quantum circuit implementation for SPARKLE256$_r$ is described and the same applies to other parameters. Within a round of SPARKLE256$_r$, Alzette runs 4 times, and we execute them in parallel. The quantum adder we adopted requires one carry qubit ac for addition. Since the carry qubit is initialized to 0 after addition, it can be used in all Alzette boxes. However, this imposes a sequential structure of the circuit, which significantly increases the depth. To solve this, we allocate four carry qubits ($ac_{0\sim3}$) to operate four Alzettes, simultaneously. Without a doubt, it would be more efficient to reduce the depth by a quarter instead of using only three more qubits. Algorithm 3 describes a quantum implementation of SPARKLE256$_r$.

3.2 SCHWAEMM

In this section, the quantum circuit implementation of SCHWAEMM-128/128 is described in detail and can be extended with other parameters. We also describe our quantum implementation by assuming that the input associated data A and the message M have a length of 32 bits.

Padding Data The padded data of Associated data A and Message M are used only when performing XOR operations with internal state S in the ρ_1 function. That is, A and M do not need to maintain the padding state because they are used as volatile in the encryption stage. Since only meaningful data (mostly the non-padded part) is used when performing the actual operation, we can get the value we want by only operating on the non-padded part and the data worth

Algorithm 2 Quantum implementation of $\mathcal{L}_4$.

Input: 128-qubit $x_{0\sim3}$, and $y_{0\sim3}$
Output: x, y

1: // `Feistel round`
2: **Transform** x_0: ▷ Compute()
3: $x_0 \leftarrow \text{CNOT32}(x_1, x_0)$
4: $x_{0L} \leftarrow \text{CNOT16}(x_{0R}, x_{0L})$
5: $y_{2R} \leftarrow \text{CNOT16}(x_{0L}, y_{2R})$
6: $y_{2L} \leftarrow \text{CNOT16}(x_{0R}, y_{2L})$
7: $y_2 \leftarrow \text{CNOT32}(y_0, y_2)$
8: $y_{3R} \leftarrow \text{CNOT16}(x_{0L}, y_{3R})$
9: $y_{3L} \leftarrow \text{CNOT16}(x_{0R}, y_{3L})$
10: $y_3 \leftarrow \text{CNOT32}(y_1, y_3)$
11: Reverse(**transform** x_0) ▷ Uncompute()

12: **Transform** y_0: ▷ Compute()
13: $y_0 \leftarrow \text{CNOT32}(y_1, y_0)$
14: $y_{0L} \leftarrow \text{CNOT16}(y_{0R}, y_{0L})$
15: $x_{2R} \leftarrow \text{CNOT16}(y_{0L}, x_{2R})$
16: $x_{2L} \leftarrow \text{CNOT16}(y_{0R}, x_{2L})$
17: $x_2 \leftarrow \text{CNOT32}(x_0, x_2)$
18: $x_{3R} \leftarrow \text{CNOT16}(y_{0L}, x_{3R})$
19: $x_{3L} \leftarrow \text{CNOT16}(y_{0R}, x_{3L})$
20: $x_3 \leftarrow \text{CNOT32}(x_1, x_3)$
21: Reverse(**transform** y_0) ▷ Uncompute()

22: // `Branch permutation`
23: $(x_0, x_2) \leftarrow \text{SWAP32}(x_0, x_2)$
24: $(x_1, x_3) \leftarrow \text{SWAP32}(x_1, x_3)$
25: $(y_0, y_2) \leftarrow \text{SWAP32}(y_0, y_2)$
26: $(y_1, y_3) \leftarrow \text{SWAP32}(y_1, y_3)$
27: $(x_0, x_1) \leftarrow \text{SWAP32}(x_0, x_1)$
28: $(y_0, y_1) \leftarrow \text{SWAP32}(y_0, y_1)$
29: **return** x, y

operating on. We call this technique *fake padding*. The implementation of fake padding is easily extended according to the lengths of input A and M. Also, the implementation details are described assuming that A and M lengths are 32-bit.

We do not allocate padding qubits to A and M in order to compute only useful data in ρ_1. This saves 32-qubit and reduces the number of 64 CNOT gates. Because it reduces the number of 64 CNOT gates, which are more expensive than X gate, and 32 qubits, although the X gate is increased by 1, therefore it is a reasonable trade-off.

State Initialization Key K and nonce N are initialized differently depending on parameters, and most parameters except 256/128 have the characteristic as

Algorithm 3 Quantum implementation of SPAKRLE256$_r$.

Input: 128-qubit $x_{0\sim3}$, and $y_{0\sim3}$, Adder carry $ac_{0\sim3}$, Constant $c_{0\sim7}$
Output: x, y

for $i = 0$ to r **do**
  $y_0 \leftarrow AddConstant(y_0, c_{(i\%8)})$
  $y_1 \leftarrow AddConstant(y_1, i)$

  // Parallel Azlettes
  $(x_0, y_0) \leftarrow$ Alzette(x_0, y_0, c_0, ac_0)
  $(x_1, y_1) \leftarrow$ Alzette(x_1, y_0, c_1, ac_1)
  $(x_2, y_2) \leftarrow$ Alzette(x_2, y_0, c_2, ac_2)
  $(x_3, y_3) \leftarrow$ Alzette(x_3, y_0, c_3, ac_3)

  // Linear Diffusion Layer
  $(x_{0\sim3}, y_{0\sim3}) \leftarrow \mathcal{L}_4(x_{0\sim3}, y_{0\sim3})$
end for
return x, y

$|K| = |N|$. S is allocated as many qubits as $|N| + |K|$x because of $S = N||K$. N and K is put into S in sequence using CNOT gates when initializing S. There are no additional qubits because it is simply an operation of putting N and K into S. After that, if SPARKLE permutation is performed for n_s of rounds suitable for each parameter, the state initialization operation is completed.

Processing of Associated Data As mentioned in the padding data phase, non-padded A is used when calculating $\rho_1(S_L, A)$. In the same way as the existing operation, A is XORed to S_L using CNOT gates. In the data padding step, a padding function Pad_r is performed that appends with the single 1 and 0s mentioned in Section 2.1 to A to fill the r-bit block. Because XORing 0s is no change even, there is no need to implement these with quantum gates, but single 1 must be XORed. Further, the CNOT gate can be replaced with an X gate because XOR operation of the single 1 has the same value as using the NOT operation. When A is XORed to S_L, X gate is used for a single XOR operation on S_L. As a result, we efficiently implement it using only a single X gate instead of 32 CNOT gates.

AddConstant, which XORing the constant $const_A$, is implemented with only X gates as in Alzette. The quantum circuit of processing of associated data is described in Algorithm 4.

Encrypting and Finalization In encrypting, the result of XORing M and S_L is the ciphertext C (i.e. $trunc_t(\rho_2(S_L, M))$). Since S_L and M are later used in finalization, in-place computation (i.e. $S_L = S_L \oplus M$ or $M = M \oplus S_L$) is not possible. It has to be implemented as $C = S_L \oplus M$. This is not an overhead in classical computers, but in quantum computers we have to allocate new 32 qubits for C. This is inevitable, so we reduce the use of CNOT gates by using

Algorithm 4 Quantum implementation of processing of associated data.

Input: State S, Associated data A, Constant of A $const_A$
Output: S

$S_R \leftarrow AddConstant(const_A, S_R)$

// $\rho_1(S_L, A)$
$S_L \leftarrow \text{SWAP64}(S_{L1}, S_{L2})$
$S_{L1} \leftarrow \text{CNOT64}(S_{L2}, S_{L1})$
$S_L \leftarrow \text{CNOT32}(A, S_L)$
$S_L \leftarrow \text{X}(S_L)$
$S_L \leftarrow \text{CNOT128}(S_R, S_L)$

// `SPRAKLE Permutation`
$S \leftarrow \text{SPARKLE256}_{10}(S)$
return S

X gates to copy the value of M to the newly allocated ciphertext C. Then, we compute ciphertext C by XORing S_L to C using CNOT gates.

In finalization, AddConstant, which XORs the constant $const_M$ of encryption, is also implemented using only X gates as before. Operations of ρ_1 and SPARKLE permutation have the same mechanism as the previous step, except for using M instead of A. Authentication tag T is generated by XORing key K to S_R. In this case, in-place computation (i.e. $S_R = S_R \oplus K$) is possible. Rather than creating a T, K is XORed to S_R and S_R is used instead of T. Consequently, no additional qubits are allocated, and less quantum resources are used. Finally, the ciphertext C is appended with the authentication tag T and returned. Details of the quantum circuit implementation are shown in Algorithm 5.

Figure 3 shows a diagram of the quantum circuit for the SCHWAEMM128-128/128.

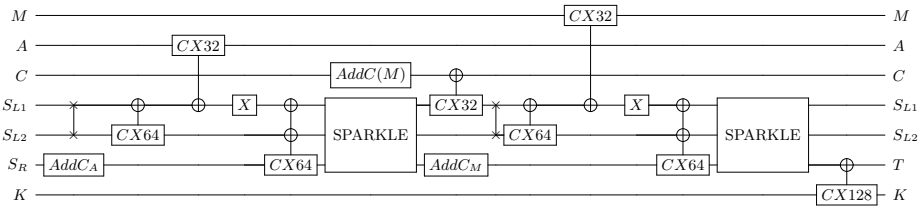


Fig. 3: The quantum circuit of SCHWAEMM128-128 (A and M are 32-qubit).

Algorithm 5 Quantum implementation of encrypting and finalization.

Input: State S, Message M, Constant of M $const_M$, Key K, Ciphertext C
Output: $C||S_R$

Encrypting:
  $//C \leftarrow trunc_t(\rho_2(S_L,\, M)))$
  $C \leftarrow$ Allocate new qubits of length $|M|$
  $AddConstant(M(\text{classical}),\, C)$
  $C \leftarrow \text{CNOT32}(S_L, C)$

Finalization:
  $S_R \leftarrow AddConstant(const_M,\, S_R)$

  $//\ \rho_1(S_L,\, M)$
  $S_L \leftarrow \text{SWAP64}(S_{L1}, S_{L2})$
  $S_{L1} \leftarrow \text{CNOT64}(S_{L2},\, S_{L1})$
  $S_L \leftarrow \text{CNOT32}(M, S_L)$
  $S_L \leftarrow \text{X}(S_L)$
  $S_L \leftarrow \text{CNOT128}(S_R, S_L)$

  //`SPARKLE Permutation`
  $S \leftarrow \text{SPARKLE256}_{10}(S)$

  $//S_R \oplus K$
  $S_R \leftarrow \text{CNOT128}(K, S_R)$
return $C||S_R$

4 Performance

In this section, we evaluate the performance of proposed SCHWAEMM quantum circuits by estimating the quantum resources. Large-scale quantum computers in which proposed quantum circuits can operate have not yet been developed. Thus, we implement and simulate quantum circuits using ProjectQ, a quantum programming tool on a classical computer.

ProjectQ's internal library, `ClassicalSimulator`, is limited to simple quantum gates (e.g. X, CNOT, and Toffoli), enabling simulation using numerous qubits. This feature allows ClassicalSimulator to verify implementations by classically computing the output for a specific input to the implemented SCHWAEMM quantum circuit. Another internal library, `ResourceCounter`, is used for estimation of detailed quantum resources. Unlike ClassicalSimulator, ResourceCounter does not run quantum circuits and counts only quantum gates and circuit depth. There is no limit to quantum gates.

In order to proceed with the standardized evaluation in all parameters, the encryption is based on the case where Associated Data A and Message M are 32-bit. Table 1 shows the quantum resources required to implement SCHWAEMM quantum circuits at the NCT (NOT (X), CNOT, Toffoli) level.

Table 1: Quantum resources required for SCHWAEMM quantum circuits.

Cipher	#CNOT	#X	#Toffoli	Toffoli depth	#qubits	Depth
SCHWAEMM-128/128	102,976	35,951	29,280	2,440	612	8,598
SCHWAEMM-256/128	170,872	59,873	48,312	2,684	870	9,591
SCHWAEMM-192/192	170,808	59,873	48,312	2,684	870	9,591
SCHWAEMM-256/256	249,056	87,062	70,272	2,928	1,128	10,605

The actual implementation of the Toffoli gate consists of a combination of Clifford gates and T gates. There are several options for decomposing the Toffoli gate [18,19,20]. In this work, we decompose into 7 T gates + 8 Clifford gates following the method of [18] (which is frequently adopted in quantum-related studies). Simply, one Toffoli gate has 7 T gates, 8 Clifford gates, a T-depth of 4, and a full depth of 8. Table 2 shows detailed quantum resources required for SCHWAEMM quantum circuits at the Clifford + T level.

Table 2: Quantum resources required for SCHWAEMM quantum circuits in detail.

Cipher	#CNOT	#1qCliff	#T	T-depth	#qubits	Full depth
SCHWAEMM-128/128	278,656	94,511	204,960	9,760	612	59,687
SCHWAEMM-256/128	460,744	156,497	338,184	10,736	870	65,783
SCHWAEMM-192/192	460,680	156,497	338,184	10,736	870	65,783
SCHWAEMM-256/256	670,688	227,606	491,904	11,712	1,128	71,906

5 Cost Estimation for Grover Key Search

Grover's algorithm (Section 2.3) consists of an oracle that finds a solution and a diffusion operator that amplifies the amplitude of the solution. The diffusion operator requires no special technique to implement, and its overhead is negligible. Thus, in most cases [3,5,12,11], the cost of Grover's algorithm is estimated with only the quantum resources required for oracle. Quantum resources for oracle are determined by the efficiency of the quantum circuit implemented. Following this, we also estimate the cost for the oracle only.

In oracle, our proposed SCHWAEMM quantum circuit works twice as illustrated in Figure 4. In the intermediate process, there is a process to verify that the ciphertext matches the known ciphertext, but for simplicity of comparison, this is excluded from the count. Since the quantum circuit operates sequentially, the cost for oracle is calculated as (Table 2 $\times$ 2) excluding the number of qubits. As can be seen, the performance of Grover key search depends on the efficiency of

the implemented quantum circuit. Table 3 shows the quantum resources required for Grover's oracle on SCHWAEMM.

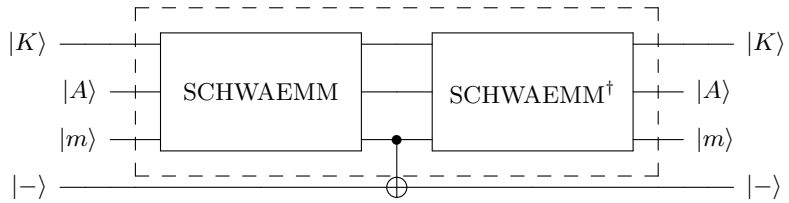


Fig. 4: Grover's oracle on SCHWAEMM.

Table 3: Quantum resources required for Grover's oracle on SCHWAEMM.

Cipher	#CNOT	#1qCliff	#T	T-depth	#qubits	Full depth
SCHWAEMM-128/128	557,312	189,022	409,920	19,520	613	119,374
SCHWAEMM-256/128	921,488	312,994	676,368	21,472	871	131,566
SCHWAEMM-192/192	921,360	312,994	676,368	21,472	871	131,566
SCHWAEMM-256/256	1,341,376	455,212	983,808	23,424	1,129	143,812

Grover key search recovers with high probability after sufficiently increasing the amplitude of the solution key through numerous iterations of the oracle and diffusion operator. When recovering an n-bit key (i.e., 2^n search space), the Grover algorithm is well known to iterate $\sqrt{2^n}$ times. However, after Grover's algorithm was presented, the authors of [21] suggested that $\lfloor \frac{\pi}{4} \cdot \sqrt{2^n} \rfloor$ iterations are optimal through a detailed analysis of Grover's iterations. Following this, Grover key search cost is calculated as (Table $3 \cdot \lfloor \frac{\pi}{4} \cdot \sqrt{2^n} \rfloor$) excluding the number of qubits. Table 4 shows the quantum resources required for Grover's key search on SCHWAEMM. Additionally, Table 4 shows the Grover attack cost specified by NIST according to the key size.

According to the NIST security requirements [22], the Grover attack cost is 2^{170}, 2^{233}, and 2^{298} in the order of AES-128, AES-192, and AES-256, and the security level gradually increases starting from Level 1. The attack cost for AES estimated by NIST (calculated as Total gates $\times$ Total depth) follows Grassl et al.'s quantum circuit implementation for AES [3]. NIST's post-quantum security requirements specify that ciphers should be comparable to or higher than the Grover attack cost for AES in order to claim security from quantum computers. If the Grover attack cost for SCHWAEMM is evaluated based on NIST's post-quantum security requirements, it is exposed to attack at a lower cost than AES with the same key size. Since an attack is possible with fewer quantum resources, an appropriate security level cannot be achieved.

However, it should be pointed out that the AES attack cost estimated by NIST is the result of 2016, and implementations for optimizing quantum circuits for AES have recently been proposed [5,4,23,6]. Among them, Jaques et al. significantly reduced the attack costs for AES-128, 192, and 256 to 2^{157}, 2^{221}, and 2^{285} [5]. NIST noted that the estimated cost in post-quantum security requirements should be conservatively evaluated if the results of significantly reducing the quantum attack cost are presented. Thus, compared with the attack cost for AES estimated in [5], SCHWAEMM achieves an appropriate security level according to the key size.

Table 4: Quantum resources required for Grover's key search on SCHWAEMM.

Cipher	Total gates	Total depth	Cost	NIST security
SCHWAEMM-128/128	$1.732 \cdot 2^{83}$	$1.431 \cdot 2^{80}$	$1.239 \cdot 2^{164}$	2^{170}
SCHWAEMM-256/128	$1.431 \cdot 2^{84}$	$1.577 \cdot 2^{80}$	$1.128 \cdot 2^{165}$	2^{170}
SCHWAEMM-192/192	$1.431 \cdot 2^{116}$	$1.577 \cdot 2^{112}$	$1.128 \cdot 2^{229}$	2^{233}
SCHWAEMM-256/256	$1.041 \cdot 2^{149}$	$1.723 \cdot 2^{144}$	$1.795 \cdot 2^{293}$	2^{298}

6 Conclusion

We present quantum circuit implementations of the AEAD instances SCHWAEMM of the lightweight cipher SPARKLE in this work. This is the first implementation of SCHWAEMM as a quantum circuit. Our implementation has been optimized by applying various techniques to minimize the cost. In this implementation, we focus a bit more on reducing qubit complexity than on depth complexity. We estimate the cost of our quantum circuit needed to run the Grover's algorithm and evaluated the security level of SCHWAEMM against the post-quantum security requirements criteria of NIST. Future work is to analyze the cost of Grover's attack for other final candidate algorithms of the NIST LWC project and evaluate the post-quantum security strength.

References

1. D. S. Abrams and S. Lloyd, "Nonlinear quantum mechanics implies polynomial-time solution for NP-complete and #P problems," *Physical Review Letters*, vol. 81, no. 18, p. 3992, 1998. 1
2. P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999. 2
3. M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: Quantum resource estimates," in *Post-Quantum Cryptography* (T. Takagi, ed.), (Cham), pp. 29–43, Springer International Publishing, 2016. 2, 12, 13

4. B. Langenberg, H. Pham, and R. Steinwandt, "Reducing the cost of implementing the advanced encryption standard as a quantum circuit," *IEEE Transactions on Quantum Engineering*, vol. 1, pp. 1–12, 01 2020. 2, 14
5. S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing Grover Oracles for quantum key search on AES and LowMC," in *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II* (A. Canteaut and Y. Ishai, eds.), vol. 12106 of *Lecture Notes in Computer Science*, pp. 280–310, Springer, 2020. 2, 12, 14
6. J. Zou, Z. Wei, S. Sun, X. Liu, and W. Wu, "Quantum circuit implementations of AES with fewer qubits," in *Advances in Cryptology – ASIACRYPT 2020* (S. Moriai and H. Wang, eds.), (Cham), pp. 697–726, Springer International Publishing, 2020. 2, 14
7. K. Jang, S. Choi, H. Kwon, H. Kim, J. Park, and H. Seo, "Grover on korean block ciphers," *Applied Sciences*, vol. 10, no. 18, p. 6407, 2020. 2
8. K. Jang, G. Song, H. Kim, H. Kwon, H. Kim, and H. Seo, "Efficient implementation of PRESENT and GIFT on quantum computers," *Applied Sciences*, vol. 11, no. 11, 2021. 2
9. K. Jang, S. Choi, H. Kwon, and H. Seo, "Grover on speck: quantum resource estimates," *Cryptology ePrint Archive*, 2020. 2
10. K. Jang, G. Song, H. Kwon, S. Uhm, H. Kim, W.-K. Lee, and H. Seo, "Grover on PIPO," *Electronics*, vol. 10, no. 10, p. 1194, 2021. 2
11. S. Bijwe, A. K. Chauhan, and S. K. Sanadhya, "Quantum search for lightweight block ciphers: GIFT, SKINNY, SATURNIN," *Cryptology ePrint Archive*, 2020. 2, 12
12. A. Baksi, K. Jang, G. Song, H. Seo, and Z. Xiang, "Quantum implementation and resource estimates for Rectangle and Knot," *Quantum Inf. Process.*, vol. 20, no. 12, p. 395, 2021. 2, 12
13. C. Beierle, A. Biryukov, L. C. dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, Q. Wang, and A. Biryukov, "Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family," *NIST round*, vol. 2, 2019. 2
14. T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore, "A logarithmic-depth quantum carry-lookahead adder," *arXiv preprint quant-ph/0406142*, 2004. 6
15. Y. Takahashi, S. Tani, and N. Kunihiro, "Quantum addition circuits and unbounded fan-out," *arXiv preprint arXiv:0910.2530*, 2009. 6
16. T. G. Draper, "Addition on a quantum computer," *arXiv preprint quant-ph/0008033*, 2000. 6
17. S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," *arXiv preprint quant-ph/0410184*, 2004. 6
18. M. Amy, D. Maslov, M. Mosca, M. Roetteler, and M. Roetteler, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, p. 818–830, Jun 2013. 12
19. A. Fedorov, L. Steffen, M. Baur, M. P. da Silva, and A. Wallraff, "Implementation of a Toffoli gate with superconducting circuits," *Nature*, vol. 481, no. 7380, pp. 170–172, 2012. 12
20. T. Ralph, K. Resch, and A. Gilchrist, "Efficient Toffoli gates using qudits," *Physical Review A*, vol. 75, no. 2, p. 022313, 2007. 12
21. M. Boyer, G. Brassard, P. Høyer, and A. Tapp, "Tight bounds on quantum searching," *Fortschritte der Physik*, vol. 46, p. 493–505, Jun 1998. 13

22. NIST., "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 13
23. M. Almazrooie, A. Samsudin, R. Abdullah, and K. N. Mutter, "Quantum reversible circuit of AES-128," *Quantum Information Processing*, vol. 17, p. 1–30, may 2018. 14