Received March 31, 2022, accepted May 24, 2022, date of publication June 3, 2022, date of current version June 9, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3179970

Efficient Implementation of Lightweight Hash Functions on GPU and Quantum Computers for IoT Applications

WAI-KONG LEE[1], (Member, IEEE), KYUNGBAE JANG[2], GYEONGJU SONG[2], (Student Member, IEEE), HYUNJI KIM[2], (Graduate Student Member, IEEE), SEONG OUN HWANG[1], (Senior Member, IEEE), AND HWAJEONG SEO[2]

[1]Department of Computer Engineering, Gachon University, Seongnam 13120, South Korea
[2]Department of IT Engineering, Hansung University, Seoul 02876, South Korea

Corresponding author: Hwajeong Seo (hwajeong84@gmail.com)

The work of Wai-Kong Lee was supported by the Brain Pool Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information & Communications Technology (ICT) (2019H1D3A1A01102607. 30%). The work of Seongoun Hwang was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (2020R1A2B5B01002145, 20%). The work of Kyungbae Jang, Gyeongju Song, and Hyunji Kim was supported in part by the Institute for Information and Communications Technology Planning and Evaluation (IITP) Grant through the Korea Government (MSIT) (<Q|Crypton>, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity) under Grant 2019-0-00033 (20%); and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant through the Korea Government (MSIT) [Development of Fast Design and Implementation of Cryptographic Algorithms based on Graphics Processing Unit (GPU)/Application-Specific Integrated Circuit (ASIC)] under Grant 2021-0-00540 (20%). The work of Hwajeong Seo was financially supported by Hansung University (10%).

ABSTRACT Secure communication is important for Internet of Things (IoT) applications, to avoid cyber-security attacks. One of the key security aspects is data integrity, which can be protected by employing cryptographic hash functions. Recently, US National Institute of Standards and Technology (NIST) announced a competition to standardize lightweight hash functions, which can be used in IoT applications. IoT communication involves various hardware platforms, from low-end microcontrollers to high-end cloud servers with GPU accelerators. Since many sensor nodes are connected to the gateway devices and cloud servers, performing high throughput integrity check is important to secure IoT applications. However, this is a time consuming task even for high-end servers, which may affect the response time in IoT systems. Moreover, no prior work had evaluated the performance of NIST candidates on contemporary processors like GPU and quantum computers. In this study, we showed that with carefully crafted implementation techniques, all the finalist hash function candidates in the NIST standardization competition can achieve high throughput (up-to 1,000 Gbps) on a RTX 3080 GPU. This research output can be used by IoT gateway devices and cloud servers to perform data integrity checks at high speed, thus ensuring a timely response. In addition, this is also the first study that showcase the implementation of NIST lightweight hash functions on a quantum computer (ProjectQ). Besides securing the communication in IoT, these efficient implementations on a GPU and quantum computer can be used to evaluate the strength of respective hash functions against brute-force attack.

INDEX TERMS Graphics processing units (GPU), hash function, lightweight cryptography, quantum computer.

I. INTRODUCTION

Internet of Things (IoT) is an emerging field of technology that has inspired many innovative applications in recent years. Combined with other important technologies like Artificial Intelligence (AI) and cloud computing, IoT involves various smart applications that can greatly enhance the quality of our lives. For instance, smart homes [1], smart laboratories [2], and smart cities [3] will become possible with advances in IoT and other relevant technologies. Since many IoT applications

The associate editor coordinating the review of this manuscript and approving it for publication was Abdullah Iliyasu.

involve the use of sensitive data, protecting communications in IoT is of utmost importance [4]. One of the important criteria used to secure IoT communication is the ability to check the integrity of the sensor data being communicated. This can be achieved through the use of cryptographic hash functions like SHA-2 and SHA-3. In 2018, the National Institute of Standards and Technology (NIST) of the United States (US) initiated a worldwide competition [5] to standardize Light-Weight Cryptography (LWC), targeting applications in constrained systems. The LWC selection criteria included requirements for small memory and fast computation, which is useful for IoT applications. This standardization is currently in its final round [6]. Four hash functions and nine authenticated encryption with associated data (AEAD) algorithms are being reviewed.

Communication within an IoT system is usually heavy because of the large number of connected sensor nodes, and the complex communication protocols between sensor nodes, gateway devices, and cloud server. In addition, IoT communication involves various platforms, including low-end microcontrollers, mid-end gateway devices, and high-end cloud servers. Considering these factors, it is critical that the hash functions be efficiently implemented on various platforms to provide integrity checks, so that they do not severely affect the system response time. Although a LWC can achieve good performance in constrained platforms [7], its performance in mid-end gateway devices and high-end cloud servers is unknown. In this paper, we show that with carefully designed implementation techniques, all of the NIST finalist candidates (lightweight hash functions) can achieve very high throughput.

Brute-force attacks are a common way of evaluating the strength of a hash function without exploiting a weakness in the underlying algorithm. This process is important to better understand the security of the selected hash functions and protect IoT systems in the future. To achieve this, we present the first implementation of the NIST finalist hash functions in a quantum computer, which is a contemporary computing system that is potentially faster than many existing computer systems. Estimating the cost of quantum brute-force attacks on hash functions or block ciphers in response to the upcoming post-quantum era is an active research field [8]–[15]. The cost of an attack depends on how efficiently the quantum circuit for the target algorithm is implemented. Keeping this in mind, in this work, we implement efficient quantum circuits for the selected NIST hash functions and estimate the quantum resources required for the attack.

The proposed efficient implementation techniques can be used in two specific environments. Firstly, we showed that by using our GPU implementation techniques, we can help to secure the high throughput IoT communication. Secondly, we also demonstrate how to efficiently utilize quantum computers to estimate the security level of the hash functions that are used in IoT applications. These hash functions are evaluated according to the security strength estimation for symmetric key cryptography [16] suggested by NIST.

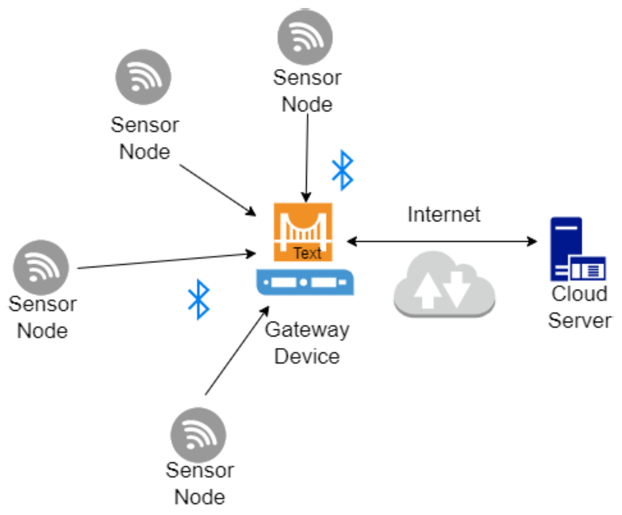


FIGURE 1. A typical IoT communication architecture.

Contributions of this paper are summarized below:

1) The first efficient implementations of PHOTON-Beetle, ASCON, Xoodyak, and SPARKLE on GPU platforms are presented in this paper. Proposed techniques include table-based implementation with warp shuffle instruction and various memory optimization techniques on GPU platforms. The performance of these implementations was evaluated on a high-end GPU platform (RTX 3080). The hash throughput of proposed implementation was up-to 1,000 Gbps, which is fast enough to handle the massive traffic of an IoT system.
2) We report the first implementation of PHOTON-Beetle, ASCON, Xoodyak, and SPARKLE hash functions on quantum computers. Hash functions were optimized taking into account the reversible computing environment in quantum computers, which is different from classical computers. The implementation was performed on ProjectQ, a quantum programming tool provided by ETH Zurich and IBM [17].
3) For the purpose of reproduction, we share the GPU implementation codes in the public domain at: https://github.com/benlwk/lwcnist-finalists and the quantum circuit implementation codes in the public domain at: https://github.com/starj1023/lwcnist-finalists-QC

II. BACKGROUND

This section describes how cryptographic hash functions are used to check data integrity in IoT communication. It also provides an overview of the selected hash functions and implementation platforms.

A. SECURE COMMUNICATION IN IoT APPLICATIONS

Referring to Figure 1, an IoT system consists of three communicating parties: sensor nodes, gateway device, and cloud server. Sensor nodes are usually placed ubiquitously to collect important sensor data. Because of this requirement, sensor nodes are designed with low power microcontrollers and powered by battery. Gateway devices are placed at a strategic location to obtain the IoT data from sensor nodes. These gateway devices need to handle connections from a lot of sensor nodes, so they are usually implemented with a more powerful processor and connected to a continuous power source. The communication between gateway device and sensor nodes utilizes wireless technology, like Bluetooth Low Energy (BLE) or Zigbee. In other words, the sensor nodes are usually not directly connected to the Internet. On the other hand, the cloud server communicates with the gateway devices through an internet connection, which is usually protected through TLS protocol.

Data integrity is important for security because it ensures the collected sensor data is not maliciously modified during the communication process, from sensor nodes to the cloud server. With the use of a cryptographic hash function, any malicious modification of the communicated sensor data can be easily detected. This allows us to verify the integrity of the sensor data on the gateway or server side, which greatly strengthens the security of IoT communication. On top of that, the hash function is also used to construct a mutual authentication protocol [18] or Hash-based Message Authentication Code (HMAC) to ensure confidentiality and authenticity. The role of a hash-based signature in IoT systems was also investigated in a prior work [19].

Although hash functions are generally considered lightweight, efficient implementation is still important because of the massive amount of traffic in IoT communication. For instance, the gateway device may need to perform a data integrity check (i.e., recomputing the hash value) on all sensor data it receives. This can impose a huge burden on the gateway device and potentially degrade its response time, causing unwanted communication delay. Note that the gateway device may still need to perform other computations like data summarizing and edge computing; performing integrity check on many sensor nodes on a regular basis is definitely a demanding task. To mitigate this potential performance bottleneck, we can offload the data integrity check to an accelerator (e.g., GPU), following the strategy proposed by Chang *et al.* [20]. Hence, efficient implementation of hash functions on GPU platforms is crucial to secure future IoT communication systems, especially applications that have a large number of sensor nodes. Besides that, the security level of hash functions is usually allows estimated based on classical computers. Due to the emergence of quantum computers, their security level must be re-examined against this new processor architecture.

B. LIGHTWEIGHT HASH FUNCTIONS

In March 2021, NIST announced that four hash function candidates (PHOTON-Beetle, Ascon, Xoodyak, and Sparkle) had successfully advanced into the final round. Another five AEAD candidates (Elephant, GIFT-COFB, Grain128-AEAD, ISAP, Romulus, and TinyJambu) also advanced into the final round. Note that PHOTON-Beetle, Ascon, and Sparkle can also be configured to operate as AEAD. This sub-section provides an overview of the four finalist hash functions that were selected for implementation in the present study. More detailed descriptions can be found in the respective specifications submitted to NIST for standardization [21]–[24]. Notations used to describe operations in these four hash functions are presented in Table 1.

TABLE 1. Notations of logical operations.

Symbol	Operation
$\oplus$	Bitwise sum (XOR)
$\cdot$	Bitwise product (AND)
$\odot$	Matrix multiplication
$\bar{A}$	Bitwise complement of A
$\ggg$	Right rotate
$\lll$	Left rotate
$\gg$	Right shift
$\ll$	Left shift

TABLE 2. The PHOTON S-box.

x	0	1	2	3	4	5	6	7
$S(x)$	c	5	6	b	9	0	a	d
x	8	9	a	b	c	d	e	f
$S(x)$	3	e	f	8	4	7	1	2

PHOTON-Beetle [21] uses the PHOTON permutation function and sponge-based mode Beetle to construct the hash function. The main computation lies on the PHOTON permutation function, which is described in Algorithm 1. PHOTON permutation makes use of a 4-bit S-Box described in Table 2.

Algorithm 1 PHOTON Permutation Function

```
1: X[64]                                ▷ 512-bit state represented in 8 × 8 bytes
2: RC[12] ←{1,3,7,14,13,11,6,12,9,2,5,10}
3: IC[8] ←{0,1,3,7,15,14,12,8}
4: for i = 0 to 7 do                                        ▷ AddConstant
5:     X[i,0] ← X[i,0] ⊕ RC[k] ⊕ IC[i];
6: end for
7: for i = 0 to 7, j = 0 to 7 do                             ▷ SubCells
8:     X[i,j] ← S(X[i,j]);
9: end for
10: for i = 0 to 7, j = 0 to 7 do                            ▷ ShiftRows
11:     X[i,j] ← X[i,(j+i)%8];
12: end for
13: for i = 0 to 7, j = 0 to 7 do                     ▷ MixColumnSerial
14:     M ← Serial [2,4,2,11,2,8,5,6]
15:     $X \leftarrow M^8 \odot X$;
16: end for
```

Ascon [22] consists of authenticated ciphers (Ascon-128 and Ascon-128a), a hash function (Ascon-Hash, Ascon-XOF), and a new variant Ascon-80pq with increased resistance against quantum key-search [25]. The Ascon design is based on a substitution-permutation network (SPN) that makes use of the 5-bit S-Box described in Table 3, and a linear

TABLE 3. The Ascon S-box.

x	0	1	2	3	4	5	6	7
$S(x)$	4	b	1f	14	1a	15	9	2
x	8	9	a	b	c	d	e	f
$S(x)$	1b	5	8	12	1d	3	6	1c
x	10	11	12	13	14	15	16	17
$S(x)$	1e	13	7	e	0	d	11	18
x	18	19	1a	1b	1c	1d	1e	1f
$S(x)$	10	c	1	19	16	a	f	17

layer explained in Equation (1):

$$\begin{aligned} x_0 &\leftarrow \sum(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\ x_1 &\leftarrow \sum(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\ x_2 &\leftarrow \sum(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\ x_3 &\leftarrow \sum(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\ x_4 &\leftarrow \sum(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \end{aligned} \tag{1}$$

Xoodyak [23] make use of the Xoodoo permutation, which was inspired by the Keccak-p permutation function. The Xoodoo permutation consists of five simple steps, illustrated in Algorithm 2. Xoodyak can be used as a hash function or extendable output function (XOF), but not as AEAD.

Algorithm 2 Xoodoo Permutation Function

```
1: A[48]                    ▷ 384-bit state represented in 48 bytes
2: $C_i$                        ▷ Round constant at round $i$
//The sequence of steps is as follow:
3: $\theta$ :
4:     $P \leftarrow A_0 + A_1 + A_2$
5:     $E \leftarrow P \lll (1, 5) + P \lll (1, 14)$
6:     $A_y \leftarrow A_y + E$ for $y \in \{0, 1, 2\}$
7: $\rho_{west}$ :
8:     $A_1 \leftarrow A_1 \lll (1, 0)$
9:     $P \leftarrow A_2 \lll (0, 11)$
10: $\iota$ :
11:     $A_0 \leftarrow A_0 + C_i$
12: $\chi$ :
13:     $B_0 \leftarrow \bar{A_1} \cdot A_2$
14:     $B_1 \leftarrow \bar{A_2} \cdot A_0$
15:     $B_2 \leftarrow \bar{A_0} \cdot A_1$
16:     $A_y \leftarrow \bar{A_y} + A_y$ for $y \in \{0, 1, 2\}$
17: $\rho_{east}$ :
18:     $A_1 \leftarrow A_1 \lll (0, 1)$
19:     $A_2 \leftarrow A_2 \lll (2, 8)$
```

SPARKLE [24] is an SPN based cryptographic primitive that can be used for authenticated encryption and hashing. The Sparkle permutation function consists of an Alzette ARX-box and a linear diffusion layer. The Alzette ARX-box, described in Algorithm 4, is a Feistel-like 64-bit block cipher, to provide quick diffusion.

Algorithm 3 Alzette ARX-Box in the Sparkle Permutation Function

```
1: $x[8]$         ▷ 256-bit state represented in eight 32-bit words
2: $c$                                          ▷ Round constant
3: $x \leftarrow x + (y \ggg 31)$
4: $y \leftarrow y \oplus (x \ggg 24)$
5: $x \leftarrow x \oplus c$
6: $x \leftarrow x + (y \ggg 17)$
7: $y \leftarrow y \oplus (x \ggg 17)$
8: $x \leftarrow x \oplus c$
9: $x \leftarrow x + (y \ggg 0)$
10: $y \leftarrow y \oplus (x \ggg 31)$
11: $x \leftarrow x \oplus c$
12: $x \leftarrow x + (y \ggg 24)$
13: $y \leftarrow y \oplus (x \ggg 16)$
14: $x \leftarrow x \oplus c$
```

Algorithm 4 Linear Diffusion Layer $L_6(x)$ in the Sparkle Permutation Function

```
1: $t \leftarrow y_0 \oplus y_1 \oplus y_2$
2: $t \leftarrow t \oplus (t \lll 16)) \lll 16)$
3: $(x_3, x_4, x_5) \leftarrow (x_3 \oplus x_0 \oplus t, x_4 \oplus x_1 \oplus t, x_5 \oplus x_2 \oplus t)$
4: $(x_0, x_1, x_2, x_3, x_4, x_5) \leftarrow (x_4, x_5, x_3, x_0, x_1, x_2)$
```

C. OVERVIEW OF THE GPU ARCHITECTURE

A GPU is a massively parallel architecture consisting of hundreds to thousands of cores. To achieve high throughput, every core is assigned the same instruction, but operates on a different piece of data. This is essentially a Single Instruction Multiple Data (SIMD) parallel computing paradigm. The GPU has a deep memory architecture that needs to be carefully used in order to achieve high performance. The DRAM is the global memory in the GPU. It tends to be large in size but very slow in access speed. Shared memory is a user-managed cache that can be used to cache temporary data or look-up table; it is faster than global memory but small in size (e.g., 96KB). The register is the fastest memory in a GPU, but it is limited to thread-level access and small in size (64K registers per streaming-multiprocessor). To exchange data across different threads, we need to rely on shared memory or warp shuffle instructions. A more detail explanation of the GPU architecture and its programming model can be found in [26].

D. QUANTUM COMPUTERS FOR BRUTE-FORCE ATTACK

Pre-image attack on hash functions involves finding a message that outputs a specific hash value. Pre-image resistance indicates that it is difficult to find a pre-image x for a given y in the hash function $h(x) = y$. Grover search algorithm is a quantum algorithm that is optimal for pre-image attacks on hash functions [27]. Compared to the pre-image attack, which requires 2^n searches (worst case) on a classic computer, Grover pre-image attack finds pre-image with a high probability with only $2^{\frac{n}{2}}$ searches. The steps for Grover pre-image attack are as follows.

1) n-qubit message is prepared in superposition state $|\psi\rangle$ using Hadamard gates. This ensures that all qubits have the same amplitude.

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

2) A hash function implemented as a quantum circuit is located in oracle $f(x)$ and is defined as follows. Oracle operator U_f turns the solution (i.e., pre-image) into a negative sign. Since $(-1)^1$ is -1, the sign becomes negative only when $f(x) = 1$ and applies to all states.

$$f(x) = \begin{cases} 1 \text{ if } h(x) = y \\ 0 \text{ if } h(x) \neq y \end{cases} \tag{3}$$

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

3) The probability is increased by amplifying the amplitude of the negative sign state in the diffusion operator.

Grover algorithm repeats steps 2 and 3 to increase the probability of measuring a solution. The optimal number of Grover iterations is $\lfloor\frac{\pi}{4}2^{\frac{n}{2}}\rfloor$ (about $2^{\frac{n}{2}}$) [28]. That is, the classical pre-image attack which requires 2^n searches is reduced to $2^{\frac{n}{2}}$ searches by using Grover search algorithm. What is important in this attack is to efficiently implement the hash function $h(x)$ as a quantum circuit. Since the diffusion operator has a typical structure, there is no special technique to implement.

The advent of large-scale quantum computers proved to be a threat to the cryptographic community, as it is one of the best cryptanalysis tools available. Cryptanalysis, which has been performed on classical computers so far, needs to be performed on quantum computers as well in order to provide sufficient confidence to the underlying hash functions. This is evident from the effort of NIST in estimating the post-quantum security strength according to the cost of applying the Grover algorithm for symmetric key cryptography [16].

E. QUANTUM GATES

Quantum computing is reversible for all changes except measurement. Reversible represents that the initial state must be re-produced using only the output state. There are quantum gates with reversible properties that can replace classical gates. Figure 2 shows representative quantum gates used in quantum computing.

1) NOT / X gate: NOT$(x) = \overline{x}$, This inverts the input qubit.
2) CNOT gate: CNOT$(x, y) = (x, x \oplus y)$, One of two qubits acts as a control qubit. If the control qubit x is set to 1, y is inverted.

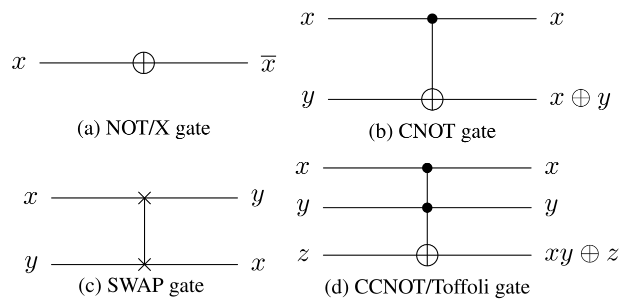


FIGURE 2. Quantum gates.

3) SWAP gate: SWAP$(x, y) = (y, x)$, This changes the state of two qubits x, y.
4) CCNOT / Toffoli gate: Toffoli$(x, y, z) = (x, y, x \cdot y \oplus z)$, Two control qubits are used. When both control qubits x and y are 1, z is inverted.

III. DEVELOPMENT OF IMPLEMENTATION TECHNIQUES ON GPU

This section describes the optimization techniques developed to implement the selected hash functions in the GPU. Note that in order to achieve high throughput, we adopt a coarse grain parallel method, wherein many parallel threads are initiated and each thread computes one hash value independently.

A. PHOTON-BEETLE

The PHOTON permutation function (Algorithm 1) operates in a 256-bit state organized in an 8-bit array (X) with 8×8 dimension. The SubCells, ShiftRows and MixColumnSerial operations can be combined and pre-computed in a table, which greatly improves implementation performance. In PHOTON-beetle, optimizing the access to this pre-computed table is the key to achieving high throughput performance in GPU. Algorithm 5 describes the PHOTON permutation function implemented with the pre-computed table.

The pre-computed table in the PHOTON permutation function only consumes 128 32-bit words, so it can be cached in the shared memory for faster access speed. A closer look into Algorithm 5 reveals that the access pattern to Table is influenced by the state in PHOTON (X, line 9). Since the value in state X is random, the access to Table is also random. If Table is stored in shared memory, the access pattern is very likely to experience bank conflict, which is not an optimal solution.

To improve the performance and avoid bank conflicts, we propose another technique to store Table in registers and access it through warp shuffle instruction, which is illustrated in Algorithm 6. In this proposed technique, each thread in a warp (32 threads) stores four values from Table into four registers (tb0, tb1, tb2, and tb3), so that the 128 values from Table are equally distributed into 32 threads. To access the values from Table, we can read one of the registers (tb0, tb1, tb2, or tb3), which is stored in one of the 32 threads.

Algorithm 5 PHOTON Permutation Function With Pre-Computed Table

```
1:  X[64]                 ▷ 256-bit state represented in 8 × 8 bytes
2:  RC[12] ←{1,3,7,14,13,11,6,12,9,2,5,10}
3:  IC[8] ←{0,1,3,7,15,14,12,8}
4:  for i = 0 to 7 do                                  ▷ AddConstant
5:      X[i,0] ← X[i,0] ⊕ RC[k] ⊕ IC[i];
6:  end for
7:  for i = 0 to 7 do
8:      $v \leftarrow 0$;
9:      for j = 0 to 7 do                    ▷ Use Pre-computed table
10:         $v \leftarrow v \oplus$ Table[j*16 + X(j,(j+i)%8)]
11:     end for
12:     for j = 1 to 8 do
13:         X(8 - j, i) ← $v \cdot (1 \ll 4) - 1$
14:         $v \leftarrow v \gg 4$;
15:     end for
16: end for
```

Algorithm 6 Snippets of Table Implementation (line 8 - 10 in Algorithm 5) Using Warp Shuffle

```
  // tid is the thread ID. Each thread
  // stores four values from Table
  tb0 = Table[tid
  tb1 = Table[tid
  tb2 = Table[tid
  tb3 = Table[tid

  // unrolled r
  for(c = 0; c < D; c++){ // for all col.
    v = 0;
    // Retrieve the values in row-wise
    v ^= __shfl(tb0, X[0][(0+c)
    v ^= __shfl(tb0, 16 + X[1][(1+c)
    v ^= __shfl(tb1, X[2][(2+c)
    v ^= __shfl(tb1, 16 + X[3][(3+c)
    v ^= __shfl(tb2, X[4][(4+c)
    v ^= __shfl(tb2, 16 + X[5][(5+c)
    v ^= __shfl(tb3, X[6][(6+c)
    v ^= __shfl(tb3, 16 + X[7][(7+c)
    \ldots
  }

}
```

For instance, $\_\_shfl(tb0, X[0][(0{+}c)\%D])$ allows us to access tb0 stored in the thread indexed by $X[0][(0 + c)\%D]$. The proposed warp shuffle version can eliminate the adverse effect of bank conflict and improves the throughput of the PHOTON-beetle hash function. In this study we implemented the reference version, shared memory version, and the warp shuffle version to compare their performance.

B. ASCON

The Ascon permutation function operates in a 320-bit state, represented in a 5×64-bit array. The S-box in Ascon can be implemented in a bit-sliced manner, which is very efficient in both high-end processors and constrained devices. Algorithm 7 shows the implementation of one round of Ascon permutation, which is repeated for 12 rounds. We proposed to utilize the bit-sliced approach when implementing the S-box (lines 8 - 12) without using any shared memory, as was the case in PHOTON-Beetle. The linear layer in the Ascon permutation function can also be implemented using simple logical and shift operations (lines 17 - 21). Note that the NVIDIA GPU does not come with a native rotate instruction. Rotate operations were replaced with two shifts and one XOR instruction.

Algorithm 7 Implementation of Ascon Permutation Function

```
1:  S[5]        ▷ 320-bit state represented in five 64-bit words
2:  T[5]                                 ▷ 320-bit temporary state
3:  C                                             ▷ Round constant
4:  S[2] ← S[2] ⊕ C                           ▷ Add round constant
5:  S[0] ← S[0] ⊕ S[4]
6:  S[4] ← S[4] ⊕ S[3]
7:  S[2] ← S[2] ⊕ S[1]

// Ascon S-Box starts
8:  T[0] ← S[0] ⊕ $\overline{S[1]} \cdot S[2]$
9:  T[1] ← S[1] ⊕ $\overline{S[2]} \cdot S[3]$
10: T[2] ← S[2] ⊕ $\overline{S[3]} \cdot S[4]$
11: T[3] ← S[3] ⊕ $\overline{S[4]} \cdot S[0]$
12: T[4] ← S[4] ⊕ $\overline{S[0]} \cdot S[1]$

// Ascon S-Box ends
13: T[1] ← T[1] ⊕ T[0]
14: T[0] ← T[0] ⊕ T[4]
15: T[3] ← T[3] ⊕ T[2]
16: T[2] ← T[2] ⊕ T[2]

// Linear diffusion layer starts
17: S[0] ← T[0] ⊕ (T[0]⋙ 19)⊕ (T[0]⋙ 28)
18: S[1] ← T[1] ⊕ (T[1]⋙ 61)⊕ (T[1]⋙ 39)
19: S[2] ← T[2] ⊕ (T[2]⋙ 1)⊕ (T[2]⋙ 6)
20: S[3] ← T[3] ⊕ (T[3]⋙ 10)⊕ (T[3]⋙ 17)
21: S[4] ← T[4] ⊕ (T[4]⋙ 7)⊕ (T[4]⋙ 41)
//Linear diffusion layer ends
```

C. XOODYAK

Xoodyak uses a permutation (Xoodoo) similar to the Keccak hash function. Unlike the other three selected hash functions, Xoodoo does not have any S-box or ARX-box layer. In our GPU implementation, the round constants are used by all threads, so they were stored in constant memory. Unlike the pre-computed Table in PHOTO-Beetle at each round, these Xoodoo round constants are only read once and consumed by every thread, so it is highly possible to be cached at the L1 cache. Hence, we did not store them in the shared memory, as it wouldn't have provided any performance gain. Our GPU implementation of the Xoodoo permutation function follows Algorithm 2 closely. We do not repeat it here.

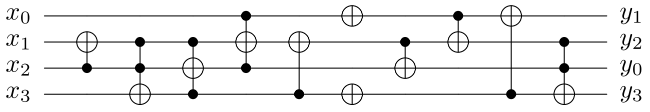


FIGURE 3. Quantum circuit for PHOTON S-box.

D. SPARKLE

The SPARKLE permutation consists of an ARX-box layer followed by a linear layer. The Alzette ARX-box in SPARKLE can be executed efficiently using only logical operations (see Algorithm 4. Due to the same reason in Xoodyak, the round constants in SPARKLE are stored in constant memory instead of shared memory or registers. The implementation of the SPARKLE-256 permutation function is illustrated in Algorithm 8.

IV. DEVELOPMENT OF IMPLEMENTATION TECHNIQUES ON QUANTUM COMPUTER

A. PHOTON-BEETLE

The PHOTON permutation function (Algorithm 1) operates in a 256-qubit state organized in a 4-qubit array with 8 × 8 dimensions. The PHOTON permutation function, which consists of AddConstant, SubCells, ShiftRows, and MixColumnSerial, was implemented as a quantum circuit as follows.

In AddConstant, the predetermined constants RC and IC are XORed with each other. In this case, it can be implemented using only NOT gates, and the overlapping parts are omitted. For example, when $k = 1$ and $i = 1$, in $X[1, 0] \oplus$ RC[1] ⊕ IC[1] (i.e. $X[1, 0] \oplus 3 \oplus 1$), two NOT gates are performed on the first qubit of $X[1, 0]$, so it is omitted and the NOT gate is performed only on the second qubit of $X[1, 0]$. Subcells apply the 4-qubit S-box × 64 to the 256-qubit state. When implementing an S-box in classical computing, a lookup table is a common choice. However, in quantum computing, this approach is quite inefficient. Therefore, a quantum circuit that computes the output for the input of the SBox should be implemented. Quantum circuit implementations for SBox sometimes incur additional qubits or increase circuit cost. To solve this, we use the LIGHTER-R tool [29] to convert Table 2 into ANF (Algebraic Normal Form). The LIGHTER-R can find reversible implementations of the 4-bit SBox. The implementation works in place, thus no additional qubits are allocated. Since the most cost in the PHOTON permutation function is used in SBox, efficient implementation of SBox is important. The PHOTON S-box quantum circuit of ANF is shown in Figure 3. LIGHTER-R is described in detail in [29].

In ShiftRow, the arrangement of qubits is changed, which can only be done with Swap gates. For convenience we used Swap gates in the implementation, but we did not count them as quantum resources. This is because Swap gates can be replaced by relabeling qubits [30]–[32] (called a logical swap). Algorithm 9 describes Shiftrows implemented as a quantum circuit. SWAP4 means a Swap operation in units of 4 qubits.

Algorithm 8 Implementation of SPARKLE-256 Permutation Function

```
1: S[5]                    ▷ 256-bit state represented in five 64-bit words
2: rc, tx, ty, x0, y0                                  ▷ Temporary variables
3: $C$                                                     ▷ Round constant
4: S[1] ← S[1] ⊕ $C_{i\%8}$ ▷ Add round constant at $i$-th round
5: S[3] ← S[3] ⊕ i

// Ascon S-Box starts
6: for $j = 0 to 11$ do
7:     rc ← $C[j \gg 1]$
8:     S[j] ← S[j] + S[j+1]⋙ 31
9:     S[j+1] ← S[j] ⊕ S[j+1]⋙ 24
10:    S[j] ← S[j] ⊕ rc
11:    S[j] ← S[j] + S[j+1]⋙ 17
12:    S[j+1] ← S[j] ⊕ S[j+1]⋙ 17
13:    S[j] ← S[j] ⊕ rc
14:    S[j] ← S[j] + S[j+1]
15:    S[j+1] ← S[j] ⊕ S[j+1]⋙ 31
16:    S[j] ← S[j] ⊕ rc
17:    S[j] ← S[j] + S[j+1]⋙ 24
18:    S[j+1] ← S[j] ⊕ S[j+1]⋙ 16
19:    S[j] ← S[j] ⊕ rc
20: end for
// Ascon S-Box ends

// Linear layer starts
21: tx = x0 = S[0]
22: ty = y0 = S[1]
23: for j = 2 to 6 step 2 do
24:     tx ← tx ⊕ S[j]
25:     ty ← ty ⊕ S[j+1]
26: end for
27: tx ← (tx ⋙ 16) ⊕ (tx · 0xFFFF)
28: ty ← (ty ⋙ 16) ⊕ (ty · 0xFFFF)
29: for j = 2 to 6 step 2 do
30:     S[j-2] = S[j+6]⊕ S[j]⊕ ty
31:     S[j+6] = S[j]
32:     S[j-1] = S[j+7] ⊕ S[j+1]⊕ tx
33:     S[j+7] = S[j+1]
34: end for
35: S[4] = S[6] ⊕ x0 ⊕ ty
36: S[6] = x0
37: S[5] = S[7] ⊕ y0 ⊕ tx
38: S[7] = y0
//Linear layer ends
```

In MixColumnSerial, the matrix multiplication in $GF(2^4)$ is used. For the general multiplication, Tofffoli gates replace AND operations. Since constant multiplications are used in this matrix multiplication, only CNOT gates are used, where the gates have a lower cost than the Tofffoli gates. We already know the modulus $x^4 + x + 1$, thus we can implement the multiplication circuit for each constant using only CNOT gates [33]. When the constant $C = 2$, $C \cdot X$

Algorithm 9 Quantum Circuit for ShiftRows

```
1: for $i = 1$ to 7 do
2:     for $j = 0$ to $i - 1$ do
3:         for $k = 0$ to 7 do
4:             SWAP4($X[i, k], X[i, k + 1]$)
5:         end for
6:     end for
7: end for
```

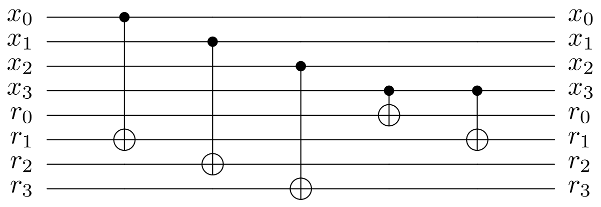


FIGURE 4. $C \cdot X \bmod x^4 + x + 1$ $(C = 2)$.

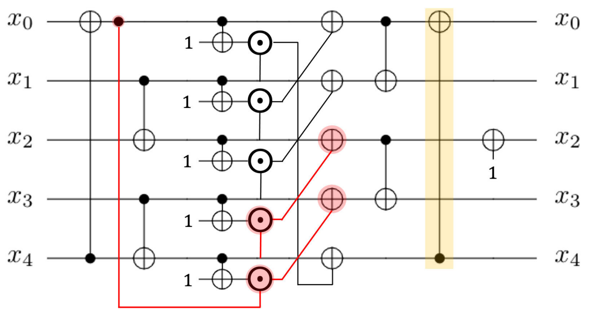


FIGURE 5. Ascon S-box in ANF.

mod $x^4 + x + 1$ is shown in Figure 4. Since X has to be used continuously, the product is stored in the newly allocated qubits r_0, r_1, r_2, r_3. We prepare modular multiplication quantum circuits for $C(0 \sim 15)$ and used them according to the value of C in the matrix multiplication of MixColumnSerial.

B. ASCON

The Ascon permutation function consists of AddConstant, a Substitution layer (Table 3), and a Linear diffusion layer (Equation 1). AddConstant adds a round constant to the state and is implemented using only NOT gates, as in PHOTON. For the Substitution layer, it is inefficient to implement an S-box in the form of Table 3 as a quantum circuit. In PHOTON, we converted Table 2 to ANF using LIGHTER-R, but since Ascon uses 5-bit S-box, LIGHTER-R (only suitable for a 4-bit S-box) could not be applied. Therefore, we implemented the S-box in ANF (Figure 5) as specified in the Ascon paper [22]. The notation $\odot$ indicates an AND operation.

The Substitution layer and Linear diffusion layer operate in a 320-qubit state, represented in a 5×64-qubit array $x_{i(i=0,\ldots,4)}$. When computing x_0 in the S-box, we need the final x_4 (yellow highlight in Figure 5). It is efficient to compute in the order x_4, x_0, x_1, x_2, x_3. Generating the final x_4, x_0, x_1 is not a problem. However, in order to obtain x_2 and x_3, the values of x_4 and x_0 before the S-box are required (red highlight in Figure 5). One way to solve this is to store the values (x_4 and x_0 before S-box) in temp qubits. However, we replaced it with additional qubits allocated from the Linear diffusion layer. In the Linear diffusion layer, to compute x_0, values of $x_0 \ggg 19$ and $x_0 \ggg 28$ are needed, simultaneously. If the first qubit $x_0[0]$ is updated to $x_0[0] \oplus x_0[19] \oplus x_0[28]$, and the original $x_0[0]$ value disappears. Since $x_0[45]$ and $x_0[36]$ cannot be computed, new qubits are allocated to store the updated value. To reduce the number of qubits, we present an S-box quantum circuit using newly allocated qubits in Linear diffusion layer. This approach allows the substitution layer and the linear diffusion layer to share temporary qubits. As a result, the use of qubits is minimized by allocating temporary qubits that should be allocated to the substitution layer and the linear diffusion layer only to the linear diffusion layer. We design an efficient S-box quantum circuit by utilizing the reverse operation and taking into account the Linear diffusion layer (Equation 1). Figure 6 shows the structure of the proposed S-box quantum circuit. In this quantum circuit, 1-qubit of each register operates the S-box and transfers the value to the temp qubit of the Linear diffusion layer using CNOT gates. Then, to compute x_2, x_3, a reverse operation (except for LD) is performed to obtain x_4, x_0 before S-box. Finally, we computed x_2 and x_3 without temp qubits using x_4 and x_0 before S-box.

C. XOODYAK

The Xoodoo permutation function operates in a 384-qubit state, represented in a 3×128-qubit array (A_0, A_1, A_2), and each 128-qubit is arranged in a 4×32 array. Algorithm 10 describes each step of the Xoodoo permutation implemented as a quantum circuit.

For the mixing layer θ, we need to allocate a new 128-qubit P for $P = A_0 + A_1 + A_2$. Then XOR A_0, A_1, A_2 to P using $3 \times$ CNOT128. CNOT128 means CNOT gates operating in units of 128 qubits. In $\lll (a, b)$ of θ, a means a rotation in 32-bit units in a 128-bit state, and b means a rotation in 1-bit units in a 32-bit state. We used RotateCNOT to XOR P to A_0, A_1, A_2 based on a logical swap for P. RotateCNOT is shown in Algorithm 11. In this way, the rotation operation can be performed without using Swap gates. In ρ_{west} and ρ_{east}, the rotation operations can be replaced with a logical swap as in RotateCNOT, but for the convenience of implementation, we used Swap gates. ι, which adds the constant C_i to A_0, is performed using only NOT gates in the same way as AddConstant in the PHOTON permutation function. Most of the quantum gates and qubits are used for the non-linear layer χ. Toffoli gates (high cost) were used to replace AND operations on A_0, A_1, A_2 and the results were stored in newly allocated B_0, B_1 and B_2. However, we reduced the use of qubits by avoiding allocation for B_2. After computing $B_0 = \bar{A_1} \cdot A_2$, $B_1 = \bar{A_2} \cdot A_0$, the reverse operations return the values of A_1 and A_2. Then $A_2 = A_2 + \bar{A_0} \cdot A_1$ (i.e., replace $A_2 = A_2 + B_2$) avoids allocating qubits for B_2. When A_2 is completed, B_0 and B_1 can be XORed to A_0 and A_1 with CNOT128. The reverse

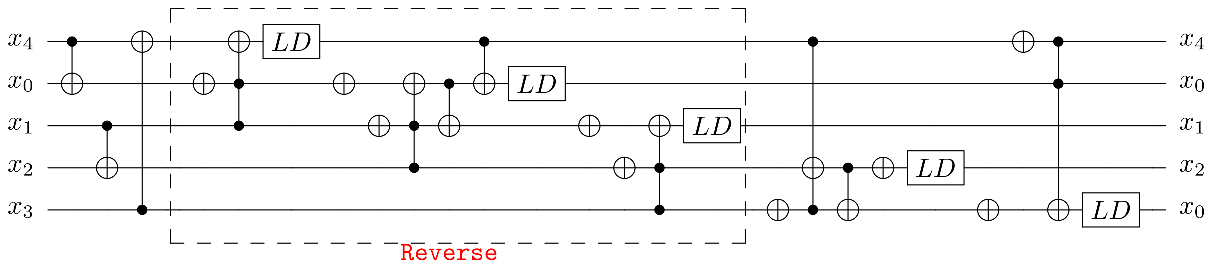


FIGURE 6. **Ascon S-box quantum circuit (*LD* : performing Linear diffusion of Equation 1).**

operation for CNOT gates does not have a large overhead in the gate and depth of the quantum circuit. We save 128-qubit every round in permutation with less overhead for gate and depth. Lastly, ρ_{east} is performed using Swap gates.

D. SPARKLE

This section only describes the Sparkle384 permutation implementation technique. This same technique works on Sparkle512. Sparkle permutations consist of an ARX-box layer followed by a linear layer. For additions in ARX-box, a quantum adder is required. For this, we used an improved quantum ripple-carry adder, called the CDKM adder [34]. The ripple-carry adder stores the result of the addition of $A+B$ in B, keeps A as it is (i.e. $\mathrm{ADD}(A, B, r) = (A, A+B, r)$). The ripple-carry adder allocates two carry qubit (r) for addition. However, since the ARX-box uses modular addition ignoring the highest carry, we only allocated a single qubit for r_0. Since this r_0 is initialized to 0 after the addition, it can be reused in subsequent additions. However, we design parallel addition by using a few more qubits, which greatly reduces the depth. In a round, Sparkle384 operates ARX-box 6 times and Sparkle512 operates 8 times. Since these ARX-boxes are independent of each other, parallel addition is possible. For this, we do not use only r_0, but $r_{0\sim5}$ for SPARKLE-384 and $r_{0\sim7}$ for Sparkle512. Implementation details can be found in our source code.

Algorithm 12 describes an ARX-box implemented as a quantum circuit. For additions and XORs using rotated input (e.g. $x + (y \ggg 31)$, $y \oplus (x \ggg 24)$), resources for rotation were not used by using RotateCNOT and RotateADD based on logical swap. RotateCNOT32 and RotateADD32, which are based on logical swaps and operate in 32-qubit units, are similar to RotateCNOT in the Xoodoo permutation, but this can be implemented, simply. Algorithm 13 describes RotateCNOT32. For RotateADD32, a CDKM adder in units of 32 qubits works. Similar to RotateXOR32, a_i were relabeled according to the rotated result (i.e., logical swaps).

In the linear layer $L_6(x)$, t for $y_0 \oplus y_1 \oplus y_2$ was used. In classic computing, using temp storage (t) like this is not a problem. However, in quantum computing, the qubits for t must be newly allocated, and since they cannot be recycled, they must be allocated every $L_6(x)$, which is very inefficient. We solved this by designing a quantum circuit for $L_6(x)$ as in Algorithm 14. Algorithm 14 computes $y_2 = y_0 \oplus y_1 \oplus y_2$ (value preparation), and XORs y_2 to x_3, x_4 and x_5 (lines 8∼19). CNOT16 and CNOT32 indicate CNOT operations in units of 16 and 32 qubits. In the last step, the value preparation is reversed to return to the original y_2. In the linear diffusion layer, $L_6(y)$ is also performed on y. Since $L_6(y)$ differs from $L_6(x)$ only in operands and the implementation technique is the same, the quantum circuit for $L_6(y)$ is omitted.

V. EXPERIMENTAL RESULTS AND DISCUSSIONS

This section presents the implementation of the selected NIST lightweight hash functions on two different platforms: a GPU and a quantum computer. The GPU implementation was performed on a workstation equipped with an Intel i9-10900K CPU and an RTX 3080 GPU. The quantum computer implementation was performed on ProjectQ, which enables quantum programming and simulation.

A. RESULTS OF IMPLEMENTATION ON GPU

This study focused on achieving a high throughput for all of the hash functions implemented on the GPU. To achieve this, all experiments were conducted by launching P blocks in parallel, with each block consisting of 512 threads. Within each thread, we performed one hash operation with different lengths (MLEN) that ranged from 64 bytes to 512 bytes. This represents the common sizes of IoT sensor data typically found in sensor nodes that are built on constrained devices with only a few KB of RAM available. The throughput (Gigabit per second (Gbps)) was calculated as follows:

$$Throughput = \frac{8 \times P \times 512 \times MLEN}{Timeelapsed} \tag{5}$$

Figure 7 shows the throughput achieved by PHOTON-Beetle in our GPU implementation. The shared memory version was always slower than the proposed warp shuffle version by approximately 40%. This is because in the PHOTON round function, the shared memory used to store the pre-computed table is accessed in a random manner, which may introduce a lot of bank conflicts. In contrast, the warp shuffle version stores the pre-computed table in registers, which are not affected by any random access pattern. Hence, the throughput of the warp shuffle version consistently outperformed the shared memory version. The highest throughput achieved by PHOTON-Beetle in our

Algorithm 10 Quantum Circuit for Xoodoo Permutation

```
1: $\theta$ :
2:     $P \leftarrow$ 128-qubit allocation
3:     $P \leftarrow$ CNOT128($A_0$, $P$)
4:     $P \leftarrow$ CNOT128($A_1$, $P$)
5:     $P \leftarrow$ CNOT128($A_2$, $P$)
6:     $A_0 \leftarrow$ RotateCNOT($P$, $A_0$)
7:     $A_1 \leftarrow$ RotateCNOT($P$, $A_1$)
8:     $A_2 \leftarrow$ RotateCNOT($P$, $A_2$)
9: $\rho_{west}$ :
10:    SWAP32($A_1[64:96]$, $A_1[96:128]$)
11:    SWAP32($A_1[32:64]$, $A_1[64:96]$)
12:    SWAP32($A_1[0:32]$, $A_1[32:64]$)
13:    for $i = 0$ to 10 do
14:        for $j = 0$ to 30 do
15:            SWAP($A_2[31-j]$, $A_2[30-j]$)
16:            SWAP($A_2[63-j]$, $A_2[62-j]$)
17:            SWAP($A_2[95-j]$, $A_2[94-j]$)
18:            SWAP($A_2[127-j]$, $A_2[126-j]$)
19:        end for
20:    end for
21: $\iota$ : RoundConstantXOR($A_0$, $C_i$)
22: $\chi$ :
23:    $B_0 \leftarrow$ 128-qubit allocation
24:    $B_1 \leftarrow$ 128-qubit allocation
25:    $A_1 \leftarrow$ NOT128($A_1$)
26:    $B_0 \leftarrow$ Toffoli128($A_1$, $A_2$, $B_0$)
27:    $A_1 \leftarrow$ NOT128($A_1$) // reverse
28:    $A_2 \leftarrow$ NOT128($A_2$)
29:    $B_1 \leftarrow$ Toffoli128($A_2$, $A_0$, $B_1$)
30:    $A_2 \leftarrow$ NOT128($A_2$) // reverse
31:    $A_0 \leftarrow$ NOT128($A_0$)
32:    $A_2 \leftarrow$ Toffoli128($A_0$, $A_1$, $A_2$)
33:    $A_0 \leftarrow$ NOT128($A_0$) // reverse
34:    $A_0 \leftarrow$ CNOT128($B_0$, $A_0$)
35:    $A_1 \leftarrow$ CNOT128($B_1$, $A_1$)
36: $\rho_{east}$ :
37:    for $i = 0$ to 30 do
38:        SWAP($A_1[31-i]$, $A_1[30-i]$)
39:        SWAP($A_1[63-i]$, $A_1[62-i]$)
40:        SWAP($A_1[95-i]$, $A_1[94-i]$)
41:        SWAP($A_1[127-i]$, $A_1[126-i]$)
42:    end for
43:    for $i = 0$ to 1 do
44:        SWAP32($A_2[64:96]$, $A_2[96:128]$)
45:        SWAP32($A_2[32:64]$, $A_2[64:96]$)
46:        SWAP32($A_2[0:32]$, $A_2[32:64]$)
47:    end for
48:    for $i = 0$ to 7 do
49:        for $j = 0$ to 30 do
50:            SWAP($A_2[31-j]$, $A_2[30-j]$)
51:            SWAP($A_2[63-j]$, $A_2[62-j]$)
52:            SWAP($A_2[95-j]$, $A_2[94-j]$)
53:            SWAP($A_2[127-j]$, $A_2[126-j]$)
54:        end for
55:    end for
```

Algorithm 11 Quantum Circuit for RotateCNOT

```
1: for $i = 0$ to 31 do
2:     //$A = A + (P \lll (1, 5))$
3:     $A[(5+i)\%32] \leftarrow$ CNOT($P[96+i]$, $A[(5+i)\%32]$)
4:     $A[32+((5+i)\%32)] \leftarrow$ CNOT($P[i]$, $A[32+((5+i)\%32)]$)
5:     $A[64+((5+i)\%32)] \leftarrow$ CNOT($P[32+i]$, $A[64+((5+i)\%32)]$)
6:     $A[96+((5+i)\%32)] \leftarrow$ CNOT($P[64+i]$, $A[96+((5+i)\%32)]$)
7:     //$A = A + (P \lll (1, 14))$
8:     $A[(5+i)\%32] \leftarrow$ CNOT($P[96+i]$, $A[(14+i)\%32]$)
9:     $A[32+((14+i)\%32)] \leftarrow$ CNOT($P[i]$, $A[32+((14+i)\%32)]$)
10:    $A[64+((14+i)\%32)] \leftarrow$ CNOT($P[32+i]$, $A[64+((14+i)\%32)]$)
11:    $A[96+((14+i)\%32)] \leftarrow$ CNOT($P[64+i]$, $A[96+((14+i)\%32)]$)
12: end for
```

Algorithm 12 Quantum Circuit for ARX-Box in Sparkle Permutation

```
1: $x \leftarrow$ RotateADD32($y$, $x$, $r_0$, 31)
2: $y \leftarrow$ RotateCNOT32($x$, $y$, 24)
3: $x \leftarrow$ RoundConstantXOR($x$, $c$)
4: $x \leftarrow$ RotateADD32($y$, $x$, $r_0$, 17)
5: $y \leftarrow$ RotateCNOT32($x$, $y$, 17)
6: $x \leftarrow$ RoundConstantXOR($x$, $c$)
7: $x \leftarrow$ ADD32($y$, $x$, $r_0$)
8: $y \leftarrow$ RotateCNOT32($x$, $y$, 31)
9: $x \leftarrow$ RoundConstantXOR($x$, $c$)
10: $x \leftarrow$ RotateADD32($y$, $x$, $r_0$, 24)
11: $y \leftarrow$ RotateCNOT32($x$, $y$, 16)
12: $x \leftarrow$ RoundConstantXOR($x$, $c$)
```

implementation range was between 70 Gps to 63 Gbps for different MLEN.

Compared to PHOTON-Beetle, the other three candidates achieved a much higher throughput. Referring to Figure 8, Sparkle was able to achieve very high throughput across different MLEN, ranging between 850 Gbps to 1000 Gbps. Xoodyak and Ascon performed at a similar level, achieving throughput that ranged between 400 Gbps to 500 Gbps. The throughput achieved by these three candidates were an order of magnitude higher than PHOTON-Beetle. The main reason for the difference in performance is that PHOTON-Beetle uses byte-wise operations, which is efficient in constrained devices (e.g., a 8-bit microcontroller), but is not efficient in a GPU with a 32-bit architecture. On the other hand, Sparkle, Xoodyak and Ascon are designed based on word-level operations (32-bit or 64-bit), which can be efficiently implemented in a GPU. Hence, the throughput achieved by these three candidates was much higher compared to PHOTON-Beetle.

Algorithm 13 Quantum Circuit for RotateCNOT32(a, b, n)

```
1: for $i = 0$ to 31 do
2:     $b[i] \leftarrow$ CNOT($a[(n+i)\%32]$, $b[i]$)
3: end for
```

Algorithm 14 Quantum Circuit for $L_6(x)$

```
1: Value preparation :
2:     $//y_2 = y_0 \oplus y_1 \oplus y_2$
3:     $y_2 \leftarrow$ CNOT32($y_0, y_2$)
4:     $y_2 \leftarrow$ CNOT32($y_1, y_2$)
5:     $//y_2 = y_2 \oplus (y_2 \lll 16)$
6:     $y_2 \leftarrow$ CNOT16($y_2[0:16], y_2[16:32]$)
7: end

8: $//x_3 = x_3 \oplus x_0 \oplus (y_2 \lll 16)$
9: $x_3[0:16] \leftarrow$ CNOT16($y_2[16:32], x_3[0:16]$)
10: $x_3[16:32] \leftarrow$ CNOT16($y_2[0:16], x_3[16:32]$)
11: $x_3 \leftarrow$ CNOT32($x_0, x_3$)

12: $//x_4 = x_4 \oplus x_1 \oplus (y_2 \lll 16)$
13: $x_4[0:16] \leftarrow$ CNOT16($y_2[16:32], x_4[0:16]$)
14: $x_4[16:32] \leftarrow$ CNOT16($y_2[0:16], x_4[16:32]$)
15: $x_4 \leftarrow$ CNOT32($x_1, x_4$)

16: $//x_5 = x_5 \oplus x_2 \oplus (y_2 \lll 16)$
17: $x_5[0:16] \leftarrow$ CNOT16($y_2[16:32], x_5[0:16]$)
18: $x_5[16:32] \leftarrow$ CNOT16($y_2[0:16], x_5[16:32]$)
19: $x_5 \leftarrow$ CNOT32($x_2, x_5$)

20: //Back from $y_0 \oplus y_1 \oplus y_2$ to $y_2$
21: Reverse(Value preparation)
```

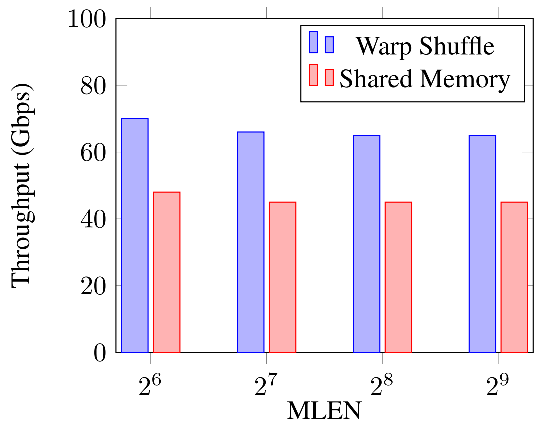


FIGURE 7. Throughput of PHOTON-Beetle with various message length.

B. RESULTS OF IMPLEMENTATION ON A QUANTUM COMPUTER

A large-scale quantum computer capable of implementing the entire quantum circuits proposed in this work is still not available yet. However, simulation and analysis can be performed using quantum programming tools. This is also a common practice found in other work [8]–[11]. For our implementation, we used the quantum programming tool ProjectQ. The implementation of quantum circuits was validated using the `ClassicalSimulator` library and the quantum resources used were analyzed using the `ResourceCounter` library. All of the hash functions implemented in this paper were optimized for qubits and quantum gates in the reversible computing environment of quantum computers. Table 4 shows the quantum resources for the quantum circuit of hash functions. The input message length is fixed to 256-bit (384-bit only for ESCH384). Among the four NIST lightweight hash functions, Xoodyak uses few quantum gates and has the lowest circuit depth. Conversely, Sparkle uses many quantum gates and has the highest circuit depth. This is because the quantum adder used in Sparkle requires many quantum gates and has a high circuit depth. One thing to note is that quantum addition uses a lot of resources, but in this work, the depth is greatly reduced by designing parallel addition. In terms of the number of qubits, Sparkle can be implemented with a relatively small number of qubits.

The quantum resources required to implement a quantum circuit for a hash function can be utilized to evaluate resistance to quantum attacks. At the current level of advancement in quantum computers, the number of available qubits is insufficient. So the number of qubits is related to when it can actually work in a quantum computer. The depth represents the start to the end of the circuit, which is related to the execution time [35].

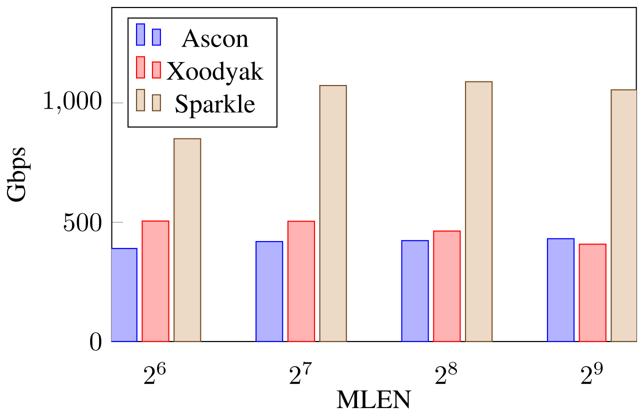


FIGURE 8. Throughput of Ascon, Xoodyak, and Sparkle with various message length.

C. POST-QUANTUM SECURITY STRENGTH

In this section, we estimate the post-quantum security strength of NIST lightweight hash functions using the post-quantum security requirements presented by NIST [16]. In symmetric key cryptography, the security strength is halved when Grover algorithm is applied. However, if the application cost is high, the target cipher or hash function can be evaluated to be resistant to quantum attacks. Therefore,

TABLE 4. Quantum resources required for lightweight hash functions and SHA3-256.

Algorithm	Qubits	Toffoli gates	CNOT gates	X gates	Depth
PHOTON	18,944	18,432	315,328	10,369	3,371
ASCON	35,136	55,296	159,232	97,346	2,487
Xoodyak	14,464	13,824	50,944	27,754	760
ESCH256	774	36,600	129,560	45,359	7,287
ESCH384	1,160	85,888	304,416	106,412	13,038
SHA3-256 [9]	3,200	84,480	33,269,760	85	10,128

Input message length = 256-bit (384-bit only for ESCH384)

TABLE 5. Quantum resources required for lightweight hash functions and SHA3-256 (Clifford + T).

Algorithm	Qubits	T gates	Clifford gates	Total gates
PHOTON	18,944	129,024	491,5858	620,609
ASCON	35,136	387,072	754,242	1,141,314
Xoodyak	14,464	96,768	203,114	299,882
ESCH256	774	256,200	504,319	760,519
ESCH384	1,160	601,216	1,183,820	1,785,036
SHA3-256 [9]	3,200	591,360	34,030,165	34,621,525

Input message length = 256-bit (384-bit only for ESCH384)

the quantum cost required to attack the target symmetric key cryptography is being used to evaluate the post-quantum security strength. NIST presented the following requirements for the security strength of post-quantum cryptosystems.

- Attacks that break the security strength of a block cipher with a 128-bit key must require similar or more resources than those required for an attack against a hash function (e.g. AES-128).
- Attacks that break the security strength of a 256-bit hash function must require similar or more resources than those required for an attack against a hash function (e.g. SHA-256 or SHA3-256).

NIST estimates the quantum attack cost for symmetric key cryptography as D (total gates $\times$ total depth) [16]. For the block cipher AES-128, NIST estimates the cost of quantum attack to be $2^{170}(D)$, citing Grassl's implementation of AES quantum circuits [8]. On the other hand, NIST did not give an estimated cost for hash functions (only for classic gates). Thus, we estimate the attack cost(D) for SHA3-256 [9] following the estimation method in [16]. The attack cost was estimated based on the quantum circuit for SHA3-256 (Table 4). For detailed analysis, we decompose the Toffoli gate into 7 T gates + 9 Clifford gates and 3 T depth, identical to the approach of [9]. X gates and CNOT gates are counted as Clifford gates. Table 5 is a resource analysis at T+Clifford level for NIST lightweight hash functions and SHA3-256.

Now we estimate the cost of Grover's pre-image attack (D in Section II) for NIST lightweight hash functions and

TABLE 6. Quantum resources required for quantum pre-image attacks.

Algorithm	Total gates	Total depth	D	Security requirements
PHOTON	$1.859 \cdot 2^{147}$	$1.292 \cdot 2^{140}$	$1.201 \cdot 2^{288}$	$1.574 \cdot 2^{295}$ (SHA3-256) [9]
ASCON	$1.71 \cdot 2^{148}$	$1.907 \cdot 2^{139}$	$1.63 \cdot 2^{288}$	
Xoodyak	$1.797 \cdot 2^{146}$	$1.166 \cdot 2^{138}$	$1.047 \cdot 2^{285}$	
ESCH256	$1.139 \cdot 2^{148}$	$1.397 \cdot 2^{141}$	$1.592 \cdot 2^{289}$	
ESCH384	$1.337 \cdot 2^{213}$	$1.25 \cdot 2^{206}$	$1.671 \cdot 2^{419}$	

Input message length = 256-bit (384-bit only for ESCH384), D = Total gates $\times$ Total depth

SHA3-256 based on the quantum resources in Table 4. Grover's algorithm consists of an oracle and a diffusion operator, but the cost of the diffusion operator is commonly ignored when estimating the cost [8], [11], [16], [36]. This is because the overhead for the diffusion operator is negligible. We also estimate only oracle as the cost of Grover's algorithm.

In the Grover's oracle, the hash function is executed twice due to (hashing + reverse). Therefore, the resources of Table 5 $\times$ 2 are used except for qubits. Resources using a single multi-controlled NOT gate to compare the generated hash value to a known hash value were omitted for simplicity. The optimal number of Grover search iterations is $\lfloor \frac{\pi}{4} 2^{\frac{n}{2}} \rfloor$. Thus, for a 256-bit input message, the oracle is repeated $\lfloor \frac{\pi}{4} 2^{128} \rfloor$ times ($\lfloor \frac{\pi}{4} 2^{192} \rfloor$ for ESCH384). Finally, the resources for the attack were estimated as Table 5 $\times 2 \times \lfloor \frac{\pi}{4} 2^{128} \rfloor$ and is shown in Table 6 (Table 5 $\times 2 \times \lfloor \frac{\pi}{4} 2^{192} \rfloor$ for ESCH384). Note that the number of qubits was not counted in D. NIST does not consider the number of qubits in estimating the attack cost.

It can be seen that the attack costs for the 256-bit hash functions for PHOTON-Beetle, Sparkle, Xoodyak, and ASCON (ESCH-256) are lower than for SHA3-256($1.574 \cdot 2^{295}$), which is the NIST security requirement. One of the ways to meet the NIST-defined security requirements against quantum computer attacks is to increase the length of the message, a well-known countermeasure. Even if the length is doubled, the security strength against quantum computer attacks is halved, but the originally intended security strength can be obtained. In addition, there are cost difficulties in performing quantum attacks because the number of Grover iterations increases exponentially with the length of the input message (e.g. ESCH384 ($1.538 \cdot 2^{422}$) in Table 6). Lastly, increasing the number of permutation functions, which occupies the most quantum resources in NIST lightweight hash functions, will also be one of the methods to satisfy the post-quantum security strength in terms of cost.

VI. CONCLUSION

Conducting high throughput data integrity checks is essential to protect communications in IoT systems. In this study, we proposed techniques to optimize the four lightweight hash functions finalists in the NIST standardization competition (PHOTON-Beetle, Ascon, Xoodyak and Sparkle). All four candidates achieved high hashing throughput (70 Gbps to 1000 Gbps) on a GPU platform, which can be used to perform high performance data integrity checks in IoT systems. Implementing these four hash functions on a quantum computer was analyzed using ProjectQ. Further, we estimated the cost of a Grover pre-image attack and compared it with NIST's post-quantum security requirements. Our work contributes to the analysis of hash functions by a quantum computer. The output from this article can be used to protect IoT communication (high throughput integrity check) as well as analyze the vulnerabilities of these hash functions against brute-force attack [37].

REFERENCES

[1] A. H. Sodhro, A. Gurtov, N. Zahid, S. Pirbhulal, L. Wang, M. M. U. Rahman, M. A. Imran, and Q. H. Abbasi, "Toward convergence of AI and IoT for energy-efficient communication in smart homes," *IEEE Internet Things J.*, vol. 8, no. 12, pp. 9664–9671, Jun. 2021.

[2] M. Chen, Y. Liu, J. C. Tam, H. Chan, X. Li, C. Chan, and W. J. Li, "Wireless AI-powered IoT sensors for laboratory mice behavior recognition," *IEEE Internet Things J.*, vol. 9, no. 3, pp. 1899–1912, Feb. 2022.

[3] M. A. Rahman, M. M. Rashid, M. S. Hossain, E. Hassanain, M. F. Alhamid, and M. Guizani, "Blockchain and IoT-based cognitive edge framework for sharing economy services in a smart city," *IEEE Access*, vol. 7, pp. 18611–18621, 2019.

[4] S. Sengupta and S. S. Bhunia, "Secure data management in cloudlet assisted IoT enabled e-health framework in smart city," *IEEE Sensors J.*, vol. 20, no. 16, pp. 9581–9588, Aug. 2020.

[5] L. Bassham, C. C. Çalık, K. McKay, and M. S. Turan, "Submission requirements and evaluation criteria for the lightweight cryptography standardization process," U.S. Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep., 2018.

[6] NIST. (2021). *Lightweight Cryptography: Finalist*. [Online]. Available: https://csrc.nist.gov/Projects/lightweight-cryptography/finalists/

[7] M. O. A. Al-Shatari, F. A. Hussin, A. A. Aziz, G. Witjaksono, and X.-T. Tran, "FPGA-based lightweight hardware architecture of the PHOTON hash function for IoT edge devices," *IEEE Access*, vol. 8, pp. 207610–207618, 2020.

[8] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: Quantum resource estimates," in *Post-Quantum Cryptography*. Cham, Switzerland: Springer, 2016, pp. 29–43.

[9] M. Amy, O. D. Matteo, V. Gheorghiu, M. Mosca, A. Parent, and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3," 2016, *arXiv:1603.09383*.

[10] J. Zou, Z. Wei, S. Sun, X. Liu, and W. Wu, "Quantum circuit implementations of AES with fewer qubits," in *Advances in Cryptology—ASIACRYPT*, S. Moriai and H. Wang, Eds. Cham, Switzerland: Springer, 2020, pp. 697–726.

[11] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing Grover oracles for quantum key search on AES and LowMC," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, May 2020, pp. 280–310.

[12] K. Jang, H. Kim, S. Eum, and H. Seo, "Grover on gift," Cryptol. ePrint Arch., Tech. Rep. 2020/1405, 2020.

[13] K. Jang, S. Choi, H. Kwon, and H. Seo, "Grover on speck: Quantum resource estimates," Cryptol. ePrint Arch., Tech. Rep. 2020/640, 2020.

[14] R. Anand, A. Maitra, and S. Mukhopadhyay, "Grover on SIMON," *Quantum Inf. Process.*, vol. 19, no. 9, pp. 1–17, 2020.

[15] M. Rahman and G. Paul, "Grover on KATAN: Quantum resource estimation," *IEEE Trans. Quantum Eng.*, vol. 3, pp. 1–9, 2022.

[16] NIST. (2016). *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[17] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018.

[18] M. Adil, M. A. Jan, S. Mastorakis, H. Song, M. M. Jadoon, S. Abbas, and A. Farouk, "Hash-MAC-DSDV: Mutual authentication for intelligent IoT-based cyber-physical systems," *IEEE Internet Things J.*, early access, May 26, 2022, doi: 10.1109/JIOT.2021.3083731.

[19] S. Suhail, R. Hussain, A. Khan, and C. S. Hong, "On the role of hash-based signatures in quantum-safe Internet of Things: Current solutions and future directions," *IEEE Internet Things J.*, vol. 8, no. 1, pp. 1–17, Jan. 2021.

[20] C.-C. Chang, W.-K. Lee, Y. Liu, B.-M. Goi, and R. C.-W. Phan, "Signature gateway: Offloading signature generation to IoT gateway accelerated by GPU," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4448–4461, Jun. 2019.

[21] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. (2021). *PHOTON-Beetle Authenticated Encryption and Hash Family*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf

[22] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. (2021). *Ascon V1.2 Submission to NIST*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf

[23] J. Daemen, S. Hoffert, S. Mella, M. Peeters, G. Assche, and R. V. Keer. (2021). *Xoodyak, a Lightweight Cryptographic Scheme*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf

[24] C. Beierle, A. Biryukov, L. C. Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, Q. Wang, A. Moradi, and A. R. Shahmirzadi. (2021). *Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing Using the Sparkle Permutation Family*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf

[25] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2," CAESAR Competition, Tech. Rep., 2016.

[26] C. Beierle, A. Biryukov, L. C. Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, Q. Wang, A. Moradi, and A. R. Shahmirzadi. (2021). *Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing Using the Sparkle Permutation Family*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#maximize-instruction-throughput

[27] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory Comput. (STOC)*, 1996, pp. 212–219.

[28] M. Boyer, G. Brassard, P. Hoyer, and A. Tappa, *Tight Bounds on Quantum Searching* (Progress of Physics), vol. 46. Hoboken, NJ, USA: Wiley, Jan. 2005, pp. 187–199.

[29] V. A. Dasu, A. Baksi, S. Sarkar, and A. Chattopadhyay, "LIGHTER-R: Optimized reversible circuit implementation for SBoxes," in *Proc. 32nd IEEE Int. Syst.-Chip Conf. (SOCC)*, Sep. 2019, pp. 260–265.

[30] L. Schlieper, "In-place implementation of quantum-gimli," 2020, *arXiv:2007.06319*.

[31] K. Jang, G. Song, H. Kim, H. Kwon, H. Kim, and H. Seo, "Efficient implementation of PRESENT and GIFT on quantum computers," *Appl. Sci.*, vol. 11, no. 11, p. 4776, May 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/4776

[32] M. Žnidarič, O. Giraud, and B. Georgeot, "Optimal number of controlled-NOT gates to generate a three-qubit state," *Phys. Rev. A, Gen. Phys.*, vol. 77, no. 3, Mar. 2008, Art. no. 032320, doi: 10.1103/PhysRevA.77.032320.

[33] I. L. Markov and M. Saeedi, "Constant-optimized quantum circuits for modular multiplication and exponentiation," 2012, *arXiv:1202.6614*.

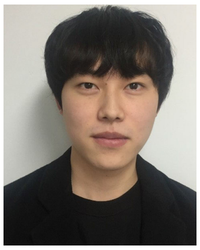
[34] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," 2004, *arXiv:quant-ph/0410184*.

[35] D. Bhattacharjee and A. Chattopadhyay, "Depth-optimal quantum circuit placement for arbitrary topologies," 2017, *arXiv:1703.08540*.

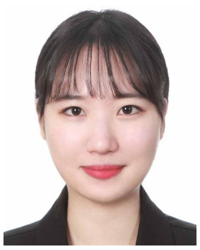
[36] A. Baksi, K. Jang, G. Song, H. Seo, and Z. Xiang, "Quantum implementation and resource estimates for rectangle and knot," *Quantum Inf. Process.*, vol. 20, no. 12, pp. 1–24, Dec. 2021.

[37] P. Liu, S. Li, and Q. Ding, "An energy-efficient accelerator based on hybrid CPU-FPGA devices for password recovery," *IEEE Trans. Comput.*, vol. 68, no. 2, pp. 170–181, Feb. 2019.

WAI-KONG LEE (Member, IEEE) received the B.Eng. degree in electronics and the M.Sc. degree from Multimedia University, in 2006 and 2009, respectively, and the Ph.D. degree in engineering from Universiti Tunku Abdul Rahman, Malaysia, in 2018. He was a Visiting Scholar at Carleton University, Canada, in 2017, at Feng Chia University, Taiwan, in 2016 and 2018, and at OTH Regensburg, Germany, in 2015, 2018, and 2019. He worked in several multi-national companies, including Agilent Technologies (Malaysia) as Research and Development Engineer. He is currently a Postdoctoral Researcher with Gachon University, South Korea. His research interests include cryptography, numerical algorithms, GPU computing, the internet of things, and energy harvesting.



KYUNGBAE JANG received the B.S. and M.S. degrees in IT convergence engineering from Hansung University, in 2019 and 2021, respectively.



GYEONGJU SONG (Student Member, IEEE) received the B.S. degree in IT convergence engineering from Hansung University, in 2021.

HYUNJI KIM (Graduate Student Member, IEEE) received the B.S. degree in IT convergence engineering from Hansung University, in 2020.

SEONG OUN HWANG (Senior Member, IEEE) received the B.S. degree in mathematics from Seoul National University, in 1993, the M.S. degree in information and communications engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, in 2004, South Korea. He was a Software Engineer at LG-CNS Systems Inc., from 1994 to 1996, a Senior Researcher at the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007, and a Professor at the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Professor with the Department of Computer Engineering, Gachon University. His research interests include cryptography, cybersecurity, and artificial intelligence. He is the Editor of *ETRI Journal*.

HWAJEONG SEO received the B.S., M.S., and Ph.D. degrees in computer engineering from Pusan National University. He is currently an Assistant Professor with Hansung University. His research interest includes cryptographic engineering.

• • •