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

***Abstract*—Secure communication is an important aspect Internet of Things (IoT) applications in order to avoid cyber-security attacks and privacy issue. 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) had initialized a competition to standardize lightweight hash functions targeting constrained devices, which can be used in IoT applications. The communication in IoT involves various hardware platforms, from low-end microcontrollers to high-end cloud servers with accelerators like GPU. In this paper, we show that with carefully crafted implementation techniques, all the finalist hash function candidates in NIST standardization can achieve high throughput on GPU. This research output can be used in IoT gateway devices and cloud servers to perform data integrity check in high speed. On top of that, we also present the first implementation of these hash functions on a quantum computer (IBM ProjectQ). The efficient implementation of these hash functions on GPU and quantum computer is useful in evaluating their strength against brute-force attack, which is important to protect the secure communication in IoT.**

***Index Terms*—Lightweight Cryptography, Graphics Processing Units (GPU), Hash Function, Quantum Computer.**

I. Introduction

INTERNET of things (IoT) is an emerging technology that inspired many innovative applications in recent years. Together with other important technologies like artificial intelligence (AI) and cloud computing, we can create various smart applications that greatly enhance the quality of our life. For instance, smart home [1], smart laboratory [2], and smart city [3] become possible due to the advancement of IoT and other relevant technologies. Since the IoT applications are associated with many sensitive data, protecting the communication in IoT is of utmost importance [4]. One of the important criteria of secure IoT communication is the ability to check the integrity of sensor data 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 United States (US) had initiated a worldwide competition [5] to standardize lightweight cryptography (LWC) that targets applications in constrained system. The selection criteria of LWC includes small memory requirement and fast computation, which is very useful for IoT applications. This standardization is currently in final round [6], wherein four hash functions and nine authenticated encryption with associated data (AEAD) algorithms are being reviewed.

Communication within an IoT system is usually heavy due to the large number of connected sensor nodes, and the complex communication protocols between sensor nodes, gateway devices and cloud server. Moreover, IoT communication involves various platforms, including low-end microcontrollers, mid-end gateway devices, and high-end cloud servers. In view of that, the efficient implementation of hash functions on various platforms is critical to provide integrity check without severely affecting the system response time. Although 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 the NIST finalist candidates (lightweight hash functions) can achieve very high throughput.

Brute-force attack is a common way to evaluate the strength of a hash function without exploiting the weakness in the underlying algorithm. This process is important to understand the security of the selected hash functions in order to protect IoT system in future. To achieve this, we present the first implementation of NIST finalist hash functions in quantum computer, which is a contemporary computing system that potentially faster than many existing computer systems. Note that efficient implementation techniques presented in this paper on both GPU and quantum computer, can be used to perform brute-force attack.

The contributions of this paper are summarized below:

1) The first efficient implementation of PHOTON-Beetle, ASCON, Xoodyak, and SPARKLE on GPU platforms is presented in this paper. The proposed techniques include table-based implementation with warp shuffle instruction and various memory optimization techniques on GPU platforms. The performance of these implementations is evaluated on a high end GPU platform (RTX 3080). The hash throughput of our implementation is up-to 1,000 Gbps, which is fast enough to handle the massive traffic in IoT system.
2) The first implementation of PHOTON-Beetle, ASCON, Xoodyak, and SPARKLE hash functions on quantum computers. Hash functions are 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 IBM [8].
3) To the purpose of reproduction, we shared the GPU implementation codes to the public domain available at: https://github.com/benlwk/lwcnist-finalists

II. Background

This section describes how the cryptographic hash functions are used to check data integrity in IoT communication. It

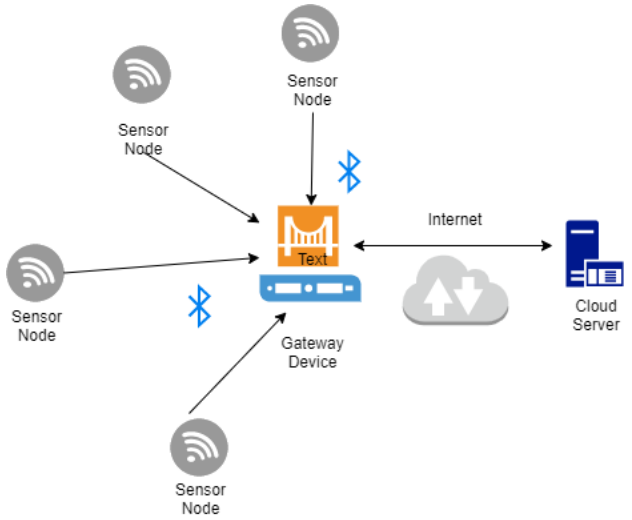


Fig. 1: A typical IoT communication architecture.

also provides overview of the selected hash functions and implementation platforms.

A. Secure Communication in IoT Applications

Referring to Figure 1, IoT system usually consists of three communicating parties: sensor nodes, gateway device and cloud server. Sensor nodes are usually placed ubiquitously to collect important sensor data. Due to this requirement, sensor nodes are implemented with low power microcontrollers and powered by battery. On the other hand, 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 utilized wireless technology like Bluetooth Low Energy (BLE) or Zigbee. It is not directly connected to the Internet. On the other hand, the cloud server communicates with the gateway devices through Internet connection, which is usually protected through TLS protocol.

Data integrity is an important security aspect as it ensures that the collected sensor data is not modified maliciously during the communication process from sensor nodes to the cloud server. By employing cryptographic hash function, any malicious modification on 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 strengthen the security of IoT communication. On top of that, the hash function was also used to construct the mutual authentication protocol [9] or hash-based message authentication code (HMAC) to achieve confidentiality and authenticity. The role of hash-based signature in IoT systems was also investigated in prior work [10].

Although hash functions are generally regarded as lightweight, the efficient implementation is still important due to the massive traffic in IoT communication. For instance, the gateway device may need to perform data integrity check (i.e., recomputing the hash value) on every sensor data it received. This may introduce a huge burden to the gateway device and potentially degrade its response time, causing unwanted communication delay. 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. [11]. Hence, the efficient implementation of hash functions on GPU platforms is crucial in securing the future IoT communication system, especially for the applications that have a large number of sensor nodes.

B. Lightweight Hash Functions

TABLE I: 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

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 are selected in our implementation. More detailed descriptions can be found in the respective specifications submitted to NIST for standardization [12], [13], [14], [15]. Notations used in describing operations in these four hash functions are presented in Table I.

TABLE II: 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 [12] uses 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 make use of a 4-bit S-Box described in Table II.

Ascon [13] consists of the authenticated ciphers (Ascon-128 and Ascon-128a), the hash function (Ascon-Hash, Ascon-XOF), and a new variant Ascon-80pq with increased resistance against quantum key-search [16]. Ascon is designed based on substitution-permutation network (SPN) that make use of a 5-bit S-Box described in Table III and a linear layer explained in Equation (1):

Algorithm 1 PHOTON permutation function.

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

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

TABLE III: 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

Xoodyak [14] make use of the Xoodoo permutation, which is inspired by the Keccak-*p* permutation function. The Xoodoo permutation consists of five simple steps; it is illustrated in Algorithm 2. Xoodyak can can be used as hash function or extendable output function (XOF), but not as AEAD.

SPARKLE [15] 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.

C. Overview of GPU Architecture

GPU is a massively parallel architecture that consists of hundreds to thousands of cores. To achieve high throughput implementation, 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. GPU has a deep memory architecture that needs to be carefully used in order to achieve high performance. Global memory is the DRAM in GPU. It provides 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). Registers is the fastest memory in GPU, but it is limited to thread-level access and small in size (64K registers per streaming-multiprocessor). To exchange data across

Algorithm 2 Xoodoo permutation function.

A[48] ▷ 384-bit state represented in 48 bytes
C_i ▷ Round constant at round *i*
`//The sequence of steps is as follow:`
θ :
    $P \leftarrow A_0 + A_1 + A_2$
    $E \leftarrow P \lll (1,5) + P \lll (1,14)$
    $A_y \leftarrow A_y + E$ for $y \in \{0,1,2\}$
ρ_{west} :
    $A_1 \leftarrow A_1 \lll (1,0)$
    $P \leftarrow A_2 \lll (0,11)$
ι :
    $A_0 \leftarrow A_0 + C_i$
χ :
    $B_0 \leftarrow \bar{A_1} \cdot A_2$
    $B_1 \leftarrow \bar{A_2} \cdot A_0$
    $B_2 \leftarrow \bar{A_0} \cdot A_1$
    $A_y \leftarrow \bar{A_y} + A_y$ for $y \in \{0,1,2\}$
ρ_{east} :
    $A_1 \leftarrow A_1 \lll (0,1)$
    $A_2 \leftarrow A_2 \lll (2,8)$

Algorithm 3 Alzette ARX-box in the Sparkle permutation function.

$x[8]$ ▷ 256-bit state represented in eight 32-bit words
c ▷ Round constant
$x \leftarrow x + (y \ggg 31)$
$y \leftarrow y \oplus (x \ggg 24)$
$x \leftarrow x \oplus c$
$x \leftarrow x + (y \ggg 17)$
$y \leftarrow y \oplus (x \ggg 17)$
$x \leftarrow x \oplus c$
$x \leftarrow x + (y \ggg 0)$
$y \leftarrow y \oplus (x \ggg 31)$
$x \leftarrow x \oplus c$
$x \leftarrow x + (y \ggg 24)$
$y \leftarrow y \oplus (x \ggg 16)$
$x \leftarrow x \oplus c$

Algorithm 4 Linear diffusion layer $L_6(x)$ in the Sparkle permutation function.

$t \leftarrow y_0 \oplus y_1 \oplus y_2$
$t \leftarrow t \oplus (t \lll 16)) \lll 16)$
$(x_3, x_4, x_5) \leftarrow (x_3 \oplus x_0 \oplus t, x_4 \oplus x_1 \oplus t, x_5 \oplus x_2 \oplus t)$
$(x_0, x_1, x_2, x_3, x_4, x_5) \leftarrow (x_4, x_5, x_3, x_0, x_1, x_2)$

different threads, we need to rely on shared memory or the warp shuffle instructions. A more detail explanation on the GPU architecture and its programming model can be found in [17].

D. Quantum Computers for Brute-force Attack

A preimage attack on hash functions is to find a message that outputs a specific hash value. The preimage resistance means that it is difficult to find the preimage x for a given y in the hash function $h(x) = y$. Grover search algorithm is a quantum algorithm that is optimal for preimage attacks on hash functions [18]. Compared to the preimage attack, which requires 2^n searches (worst case) on a classical computer, Grover preimage attack finds the preimage with a high probability with only $2^{\frac{n}{2}}$ searches. The steps for a Grover preimage attack are as follows.

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

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

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. preimage) into a negative sign. Since $(-1)^1$ is -1, the sign becomes negative only when $f(x) = 1$ and applies to all states.

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

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

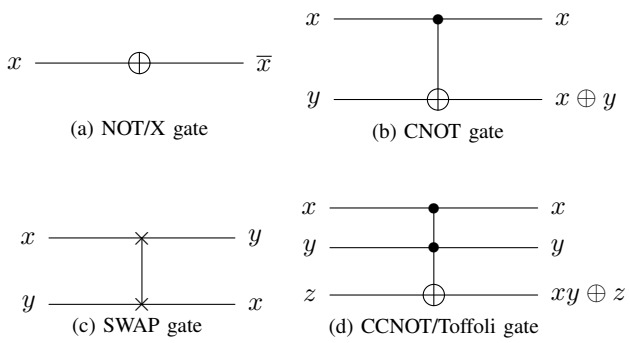
3) Lastly, the probability increased by amplifying the amplitude of the negative sign state in the diffusion operator.

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 $2^{\frac{n}{2}}$). That is, the classical preimage attack that requires 2^n searches is reduced to $2^{\frac{n}{2}}$ searches by using the 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.

E. Quantum gates

Quantum computing is reversible for all changes except measurement. Reversible means 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) = $\overline{x}$, Inverting the input qubit.
2) CNOT gate : CNOT(x, y) = $(x, x \oplus y)$, One of the two qubits acts as a control qubit. If the control qubit x is 1, y is inverted.



(a) NOT/X gate (b) CNOT gate (c) SWAP gate (d) CCNOT/Toffoli gate

Fig. 2: Quantum gates.

3) SWAP gate : SWAP(x, y) = (y, x), Changing the state of two qubits x, y.
4) CCNOT/Toffoli gate : Toffoli(x, y, z) = $(x, y, x \cdot y \oplus z)$, Using two control qubits. 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 selected hash functions in 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 on 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 the implementation performance. Hence, optimizing the access to this pre-computed table in PHOTON-beetle is the key to achieve high throughput performance in GPU. Algorithm 5 describes the PHOTON permutation function implemented with pre-computed table.

The pre-computed table in PHOTON permutation function only consumes 128 32-bit words, so it can be cached in the shared memory for a 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 lightly to experience bank conflict, which is not an optimal solution.

To improve the performance, we proposed another technique to store Table in registers and access it through warp shuffle instruction, which is illustrated in Algorithm 6. 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 distributed into 32 threads equally. 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. 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

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 ← 0;
9:     for  j = 0 to 7 do                  ▷ Use Pre-computed table
10:        v ← v ⊕ Table[j*16 + X(j,(j+i)%8)]
11:    end for
12:    for  j = 1 to 8 do
13:        X(8 - j, i) ← v · (1 << 4) − 1
14:        v ← v ≫ 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%32];
tb1 = Table[tid%32 + 32];
tb2 = Table[tid%32 + 64];
tb3 = Table[tid%32 + 96];

// 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)%D]);
  v ^= __shfl(tb0, 16 + X[1][(1+c)%D]);
  v ^= __shfl(tb1, X[2][(2+c)%D]);
  v ^= __shfl(tb1, 16 + X[3][(3+c)%D]);
  v ^= __shfl(tb2, X[4][(4+c)%D]);
  v ^= __shfl(tb2, 16 + X[5][(5+c)%D]);
  v ^= __shfl(tb3, X[6][(6+c)%D]);
  v ^= __shfl(tb3, 16 + X[7][(7+c)%D]);
  ...
}

}
```

proposed warp shuffle version can eliminate the adverse effect of bank conflict and improves the throughput of PHOTON-beetle hash function. We have implemented the reference version, shared memory version, and the warp shuffle version in this paper to compare their performance.

B. Ascon

Ascon permutation function operates on 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 our implementation of one round Ascon permutation, which is repeated for 12 rounds. We follow the bit-sliced approach in implementing the S-box (lines 8 - 12) without using any shared memory like the case in PHOTON-Beetle. The linear layer in Ascon permutation function can also be implemented using simple logical and shift operations (lines 17 - 21). Note that NVIDIA GPU does not come with a native rotate instruction. Rotate operations are 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] ⊕ S[1] · S[2]
9: T[1] ← S[1] ⊕ S[2] · S[3]
10: T[2] ← S[2] ⊕ S[3] · S[4]
11: T[3] ← S[3] ⊕ S[4] · S[0]
12: T[4] ← S[4] ⊕ S[0] · 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 is using a permutation (Xoodoo) similar to 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, round constants are stored in constant memory since it is accessible to all threads. 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 do not store them into the shared memory, as it is not going to provide any performance gain. Our GPU implementation of Xoodoo permutation function follow Algorithm 2 closely. We do not repeat it here.

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. Similar to Xoodyak, the round constants are stored in constant memory instead of shared memory. The implementation of SPARKLE-256 permutation function is illustrated in Algorithm 8.

Algorithm 8 Implementation of SPARKLE-256 permutation function.

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

```
// Ascon S-Box starts
```
for $j = 0 to 11$ **do**
    rc ← $C[j \gg 1]$
    S[*j*] ← S[*j*] + S[*j*+1]⋙ 31
    S[*j*+*1*] ← S[*j*] ⊕ S[*j*+1]⋙ 24
    S[*j*] ← S[*j*] ⊕ rc
    S[*j*] ← S[*j*] + S[*j*+1]⋙ 17
    S[*j*+*1*] ← S[*j*] ⊕ S[*j*+1]⋙ 17
    S[*j*] ← S[*j*] ⊕ rc
    S[*j*] ← S[*j*] + S[*j*+1]
    S[*j*+*1*] ← S[*j*] ⊕ S[*j*+1]⋙ 31
    S[*j*] ← S[*j*] ⊕ rc
    S[*j*] ← S[*j*] + S[*j*+1]⋙ 24
    S[*j*+*1*] ← S[*j*] ⊕ S[*j*+1]⋙ 16
    S[*j*] ← S[*j*] ⊕ rc
end for
```
// Ascon S-Box ends

// Linear layer starts
```
tx = x0 = S[0]
ty = y0 = S[1]
for j = 2 to 6 step 2 **do**
    tx ← tx ⊕ S[j]
    ty ← ty ⊕ S[j+1]
end for
tx ← (tx ⋙ 16) ⊕ (tx · 0xFFFF)
ty ← (ty ⋙ 16) ⊕ (ty · 0xFFFF)
for j = 2 to 6 step 2 **do**
    S[j-2] = S[j+6]⊕ S[j]⊕ ty
    S[j+6] = S[j]
    S[j-1] = S[j+7] ⊕ S[j+1]⊕ tx
    S[j+7] = S[j+1]
end for
S[4] = S[6] ⊕ x0 ⊕ ty
S[6] = x0
S[5] = S[7] ⊕ y0 ⊕ tx
S[7] = y0
//Linear layer ends
```

## IV. Development of Implementation Techniques on Quantum Computer

### A. PHOTON-Beetle

The PHOTON permutation function (Algorithm 1) operates on a 256-qubit state organized in an 4-qubit array with $8\times8$ dimension. The PHOTON permutation function, which consists of AddConstant, SubCells, ShiftRows, and MixColumnSerial, is implemented as a quantum circuit as follows.

In AddConstant, 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] $\oplus$ 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 4-qubit S-box $\times$ 64 to 256-qubit state. When implementing an S-box in classical computing, lookup table is a common choice. However, in quantum computing, this approach is quite inefficient. To solve this, we use the LIGHGTER-R tool [19] to convert Table II into ANF (Algebraic Normal Form). The LIGHTER-R can find reversible implementations of 4-bit SBox. The implementation works in place, thus no additional qubits are allocated. The PHOTON S-box quantum circuit of ANF is shown in Figure 3. LIHGTER-R is described in detail in [19].

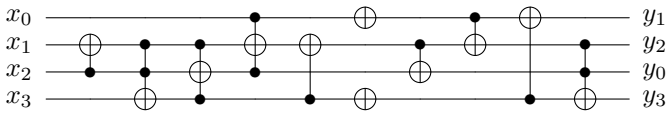



Fig. 3: Quantum circuit for PHOTON S-box.

In ShiftRow, the arrangement of qubits is changed, which can be done only with Swap gates. We used Swap gates for the convenience in the implementation, but we do not count them as quantum resources. This is because Swap gates can be replaced by relabeling of qubits [20], [21], [22] (called logical swap). Algorithm 9 describes Shiftrows implemented as a quantum circuit. SWAP4 means a Swap operation in units of 4 qubits.

**Algorithm 9** Quantum circuit for ShiftRows.

**for** $i$ = 1 to 7 **do**
    **for** $j$ = 0 to $i - 1$ **do**
        **for** $k$ = 0 to 7 **do**
            SWAP4$(X[i, k], X[i, k + 1])$
        **end for**
    **end for**
**end for**

In MixColumnSerial, 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 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 [23]. When the constant $C = 2$, $C \cdot X \bmod x^4+x+1$ 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 use them according to the value of C in matrix multiplication of MixColumnSerial.

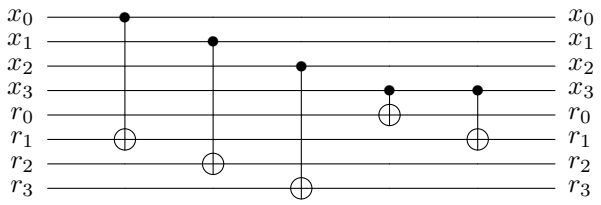


Fig. 4: $C \cdot X \bmod x^4 + x + 1$ ($C = 2$).

B. Ascon

Ascon permutation function consists of AddConstant, Substitution layer (Table III), 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 III as a quantum circuit. In PHOTON, we converted Table II to ANF using LIGHTER-R, but since Ascon uses 5-bit S-box, LIGHTER-R (only for 4-bit S-box) cannot be applied. Therefore, we implement the S-box of ANF (Figure 5) specified in the Ascon paper [13].

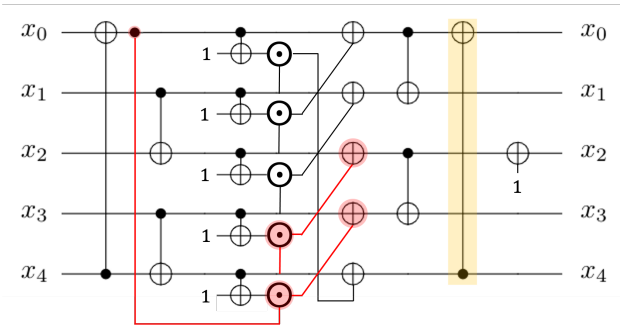


Fig. 5: Ascon S-box in ANF.

Substitution layer and Linear diffusion layer operate on a 320-qubit state, represented in a 5×64-qubit array $x_{i(i=0,\ldots,4)}$. When compute 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 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 replace it with additional qubits allocated from Linear diffusion layer. In 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 $x_0[0] \oplus x_0[19] \oplus x_0[28]$, 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. By utilizing the reverse operation and taking into account the Linear diffusion layer (Equation 1), we design an efficient S-box quantum circuit. 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 Linear diffusion layer using CNOT gates. Then, to compute x_2, x_3, reverse operation (except for LD) is performed to obtain x_4, x_0 before S-box. Finally, we compute x_2 and x_3 without temp qubits using x_4 and x_0 before S-box.

C. Xoodyak

Xoodoo permutation function operates on 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 rotation in 32-bit units in 128-bit state, and b means rotation in 1-bit units in 32-bit state. We use RotateCNOT to XOR P to A_0, A_1, A_2 based on a logical swap for P, and 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}, rotation operations can be replaced with the logical swap as in RotateCNOT, but for the convenience of implementation, we use Swap gates. ι, which adds constant C_i to A_0, is performed using only NOT gates in the same way as AddConstant of PHOTON permutation function. The most quantum gates and qubits are used for the non-linear layer χ. Toffoli gates (high cost) are used to replace AND operations on A_0, A_1, A_2 and the results are stored in newly allocated B_0, B_1 and B_2. We reduce the use of qubits by avoiding allocation for B_2. After computing $B_0 = \bar{A}_1 \cdot A_2$, $B_1 = \bar{A}_2 \cdot A_0$, 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. Lastly, ρ_{east} is performed using Swap gates.

D. SPARKLE

This section describes only the Sparkle384 permutation implementation technique. This technique works similarly 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 use an improved the quantum ripple-carry adder [24]. The ripple-carry adder stores the result of the addition of $A + B$ in B, keeps A as it is (i.e. $\mathrm{ADD}(A, B) = (A, A + B)$. This adder uses the MAJ and UMA modules shown in Figure 7 and uses additional qubits r_0 and r_1 except for A and B as shown in Figure 8. Since ARX-box uses modular addition ignoring the highest carry, we allocate only a single qubit for r_0. Since this r_0 is initialized to 0 after addition, it can be reused in subsequent additions.

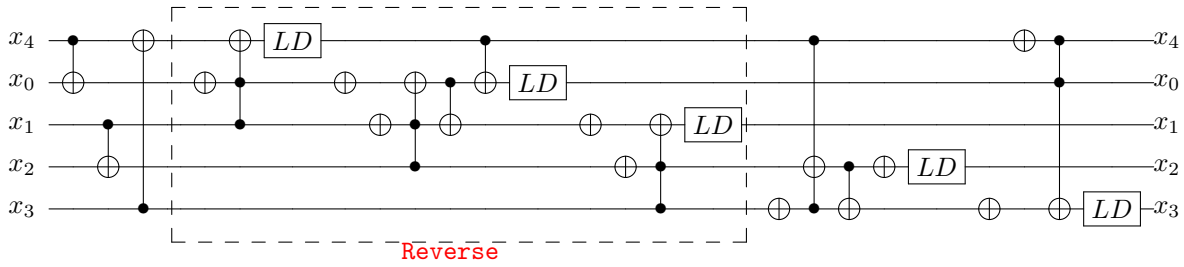


Fig. 6: Ascon S-box quantum circuit (LD : performing linear diffusion of Equation 1).

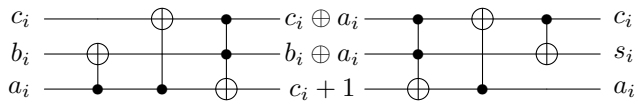


Fig. 7: MAJ quantum circuit (left) and UMA circuit (right).

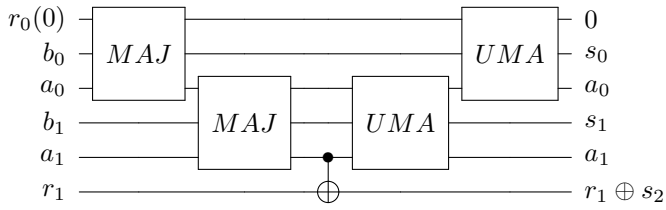


Fig. 8: Improved ripple carry adder for $i = 2$.

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 are 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 Xoodoo permutation, but this can be implemented, simply. Algorithm 13 describes RotateCNOT32. For RotateADD32, the structure in Figure 8 is redesigned in 32-qubit units and ignores the r_1 qubit line. Similar to RotateXOR32, a_is are relabeled according to the rotated result (i.e. logical swaps).

In the linear layer $L_6(x)$, t for $y_0 \oplus y_1 \oplus y_2$ is used. In the classical computing, the use of such temp storage (t) is not a problem. However, in the quantum computing, 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 solve this by designing a quantum circuit for $L_6(x)$ as in Algorithm 14. In 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, value preparation is reversed to return to the original y_2. In 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 selected NIST lightweight hash functions on two different platforms: GPU and quantum computer. The GPU implementation was performed on a workstation equipped with a Intel i9-10900K CPU and an RTX 3080 GPU. The quantum computer implementation was performed on a MacBook Pro equipped with a Intel i7 CPU.

A. Results of Implementation on GPU

For all hash function implementations on GPU, this paper targets a high throughput implementation. To achieve this, all subsequent experiments are conducted by launching P blocks in parallel, each block consists of 512 threads. Within each thread, we perform one hash operation with different length (MLEN) that range 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 (Giga-bit per second (Gbps)) is calculated with as follow:

$$Throughput = \frac{8 \times P \times 512 \times MLEN}{Timeelapsed} \quad (5)$$

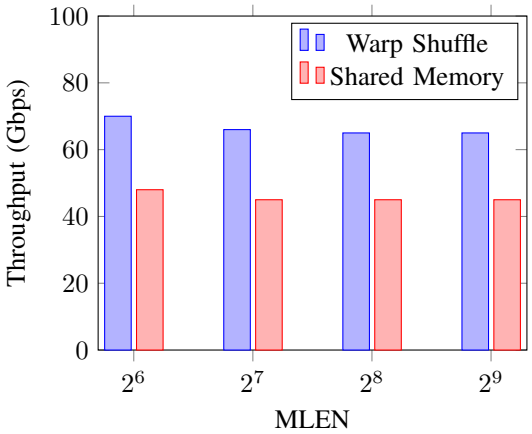


Fig. 9: Throughput of PHOTON-Beetle with various message length.

Figure 9 shows the throughput achieved by PHOTON-Beetle in our GPU implementation. The shared memory version is always slower than the proposed warp shuffle version by approximately 40%. This is because in the PHOTON round

Algorithm 10 Quantum circuit for R_i in Xoodoo permutation.

θ :
  $P \leftarrow$ 128-qubit allocation
  $P \leftarrow \text{CNOT128}(A_0, P)$
  $P \leftarrow \text{CNOT128}(A_1, P)$
  $P \leftarrow \text{CNOT128}(A_2, P)$
  $A_0 \leftarrow \text{RotateCNOT}(P, A_0)$
  $A_1 \leftarrow \text{RotateCNOT}(P, A_1)$
  $A_2 \leftarrow \text{RotateCNOT}(P, A_2)$
ρ_{west} :
  $\text{SWAP32}(A_1[64:96], A_1[96:128])$
  $\text{SWAP32}(A_1[32:64], A_1[64:96])$
  $\text{SWAP32}(A_1[0:32], A_1[32:64])$
  for j = 0 to 10 **do**
    for k = 0 to 30 **do**
      $\text{SWAP}(A_2[31-k], A_2[30-k])$
      $\text{SWAP}(A_2[63-k], A_2[62-k])$
      $\text{SWAP}(A_2[95-k], A_2[94-k])$
      $\text{SWAP}(A_2[127-k], A_2[126-k])$
    end for
  end for
ι :
  $\text{RoundConstantXOR}(A_0, C_i)$
χ :
  $B_0 \leftarrow$ 128-qubit allocation
  $B_1 \leftarrow$ 128-qubit allocation
  $A_1 \leftarrow \text{NOT128}(A_1)$
  $B_0 \leftarrow \text{Toffoli128}(A_1, A_2, B_0)$
  $A_1 \leftarrow \text{NOT128}(A_1)$ // reverse
  $A_2 \leftarrow \text{NOT128}(A_2)$
  $B_1 \leftarrow \text{Toffoli128}(A_2, A_0, B_1)$
  $A_2 \leftarrow \text{NOT128}(A_2)$ // reverse
  $A_0 \leftarrow \text{NOT128}(A_0)$
  $A_2 \leftarrow \text{Toffoli128}(A_0, A_1, A_2)$
  $A_0 \leftarrow \text{NOT128}(A_0)$ // reverse
  $A_0 \leftarrow \text{CNOT128}(B_0, A_0)$
  $A_1 \leftarrow \text{CNOT128}(B_1, A_1)$
ρ_{east} :
  for j = 0 to 30 **do**
    $\text{SWAP}(A_1[31-j], A_1[30-j])$
    $\text{SWAP}(A_1[63-j], A_1[62-j])$
    $\text{SWAP}(A_1[95-j], A_1[94-j])$
    $\text{SWAP}(A_1[127-j], A_1[126-j])$
  end for
  for j = 0 to 1 **do**
    $\text{SWAP32}(A_2[64:96], A_2[96:128])$
    $\text{SWAP32}(A_2[32:64], A_2[64:96])$
    $\text{SWAP32}(A_2[0:32], A_2[32:64])$
  end for
  for j = 0 to 7 **do**
    for k = 0 to 30 **do**
      $\text{SWAP}(A_2[31-k], A_2[30-k])$
      $\text{SWAP}(A_2[63-k], A_2[62-k])$
      $\text{SWAP}(A_2[95-k], A_2[94-k])$
      $\text{SWAP}(A_2[127-k], A_2[126-k])$
    end for
  end for

Algorithm 11 Quantum circuit for RotateCNOT.

for i = 0 to 31 **do**
  $//A = A + (P \lll (1,5))$
  $A[(5+i)\%32] \leftarrow \text{CNOT}(P[96+i], A[(5+i)\%32])$
  $A[(32+((5+i)\%32)] \leftarrow \text{CNOT}(P[i], A[32+((5+i)\%32)])$
  $A[(64+((5+i)\%32)] \leftarrow \text{CNOT}(P[32+i], A[64+((5+i)\%32)])$
  $A[(96+((5+i)\%32)] \leftarrow \text{CNOT}(P[64+i], A[96+((5+i)\%32)])$
  $//A = A + (P \lll (1,14))$
  $A[(5+i)\%32] \leftarrow \text{CNOT}(P[96+i], A[(14+i)\%32])$
  $A[(32+((14+i)\%32)] \leftarrow \text{CNOT}(P[i], A[32+((14+i)\%32)])$
  $A[(64+((14+i)\%32)] \leftarrow \text{CNOT}(P[32+i], A[64+((14+i)\%32)])$
  $A[(96+((14+i)\%32)] \leftarrow \text{CNOT}(P[64+i], A[96+((14+i)\%32)])$
end for

Algorithm 12 Quantum circuit for ARX-box in Sparkle permutation.

$x \leftarrow \text{RotateADD32}(y, x, r_0, 31)$
$y \leftarrow \text{RotateCNOT32}(x, y, 24)$
$x \leftarrow \text{RoundConstantXOR}(x, c)$
$x \leftarrow \text{RotateADD32}(y, x, r_0, 17)$
$y \leftarrow \text{RotateCNOT32}(x, y, 17)$
$x \leftarrow \text{RoundConstantXOR}(x, c)$
$x \leftarrow \text{ADD32}(y, x, r_0)$
$y \leftarrow \text{RotateCNOT32}(x, y, 31)$
$x \leftarrow \text{RoundConstantXOR}(x, c)$
$x \leftarrow \text{RotateADD32}(y, x, r_0, 24)$
$y \leftarrow \text{RotateCNOT32}(x, y, 16)$
$x \leftarrow \text{RoundConstantXOR}(x, c)$

Algorithm 13 Quantum circuit for RotateCNOT32(a, b, n).

for i = 0 to 31 **do**
  $b[i] \leftarrow \text{CNOT}(a[(n+i)\%32], b[i])$
end for

Algorithm 14 Quantum circuit for $L_6(x)$.

Value preparation :
  $//y_2 = y_0 \oplus y_1 \oplus y_2$
  $y_2 \leftarrow \text{CNOT32}(y_0, y_2)$
  $y_2 \leftarrow \text{CNOT32}(y_1, y_2)$
  $//y_2 = y_2 \oplus (y_2 \lll 16)$
  $y_2 \leftarrow \text{CNOT16}(y_2[0:16], y_2[16:32])$
end

$//x_3 = x_3 \oplus x_0 \oplus (y_2 \lll 16)$
$x_3[0:16] \leftarrow \text{CNOT16}(y_2[16:32], x_3[0:16])$
$x_3[16:32] \leftarrow \text{CNOT16}(y_2[0:16], x_3[16:32])$
$x_3 \leftarrow \text{CNOT32}(x_0, x_3)$

$//x_4 = x_4 \oplus x_1 \oplus (y_2 \lll 16)$
$x_4[0:16] \leftarrow \text{CNOT16}(y_2[16:32], x_4[0:16])$
$x_4[16:32] \leftarrow \text{CNOT16}(y_2[0:16], x_4[16:32])$
$x_4 \leftarrow \text{CNOT32}(x_1, x_4)$

$//x_5 = x_5 \oplus x_2 \oplus (y_2 \lll 16)$
$x_5[0:16] \leftarrow \text{CNOT16}(y_2[16:32], x_5[0:16])$
$x_5[16:32] \leftarrow \text{CNOT16}(y_2[0:16], x_5[16:32])$
$x_5 \leftarrow \text{CNOT32}(x_2, x_5)$

//Back from $y_0 \oplus y_1 \oplus y_2$ to y_2
Reverse(Value preparation)

function, shared memory used to store the pre-computed table is accessed in a random manner, which may introduce a lot of bank conflicts. On the contrary, the warp shuffle version stores the pre-computed table into registers, which is not affected by any random access pattern. Hence, the throughput of warp shuffle version consistently outperformed the shared memory version. The highest throughput achieved by PHOTON-Beetle in our implementation range between 70 Gps to 63 Gbps for different MLEN.

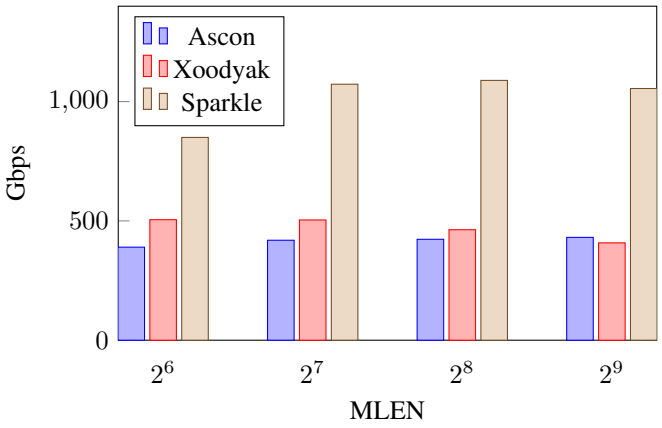


Fig. 10: Throughput of Ascon, Xoodyak, and Sparkle with various message length.

Compared to PHOTON-Beetle, the other three candidates can achieve a much higher throughput. Referring to Figure 10, Sparkle is able to achieve very high throughput across different MLEN, range between 850 Gbps to 1000 Gbps. Xoodyak and Ascon performs at the similar level, achieving throughput that range between 400 Gbps to 500 Gbps. The throughput achieved by these three candidates are an order of magnitude higher than PHOTON-Beetle. The main reason is that PHOTON-Beetle uses byte-wise operations, which is efficient in constrained devices (e.g., 8-bit microcontroller), but not efficient in GPU with 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 GPU. Hence, the throughput achieved by these three candidates are much higher compared to PHOTON-Beetle.

B. Results of Implementation on Quantum Computer

All hash functions implemented in this paper are optimized for qubits and quantum gates in the reversible computing environment of quantum computers. Table IV shows the logical resources for all hash functions implemented as quantum circuits. We estimate the security strength of all hash functions through the post-quantum security requirements presented by NIST [25]. NIST presented the following requirements for the security strength of post-quantum cryptosystems.

- 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).
- 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).

The attack cost for block cipher is estimated as D (Total gates $\times$ Depth) based on Grassl's implementation of AES quantum circuit [26]. For the case of AES-128, it is estimated as 2^{170}(D) quantum gates. For hash functions, there is no attack cost estimated by quantum gates (only for classical gates). As an alternative, we estimate the attack cost D for hash functions by applying the estimation method in block cipher [27], [28] to SHA-256 and SHA-3. Amy et al. [29] presented techniques to estimate the cost of quantum preimage attacks for SHA-256 and SHA3. We follow the method in [29] by estimating the cost of attacks on the 256-bit input message as well. Quantum resources for SHA-2 and SHA3 are shown in Table V. In [29], resources are analyzed at T+Clifford level by decomposing Toffoli gates. SHA3 is also analyzed as a resource at the NCT (NOT, CNOT, and Toffoli) level, but SHA-256 is not, so it was extrapolated based on the T+Clifford level.

In oracle, the hash function is executed twice due to (hashing + reverse). The resources of Table IV $\times$ 2 and Table V $\times$ 2 are used except for qubits. Resources of using a single multi-controlled NOT gate to compare the generated hash value to a known hash value is omitted for simplicity. The optimal number of Grover search iterations is $\lfloor \frac{\pi}{4} 2^{\frac{n}{2}} \rfloor$. For a 256-bit input message, oracle is repeated $\lfloor \frac{\pi}{4} 2^{128} \rfloor$ times. Finally, resources for attack are estimated as Table IV, V $\times 2 \times \lfloor \frac{\pi}{4} 2^{128} \rfloor$ as shown in Table VI. Since total gates is the sum of all gates at the T+Clifford level, Tables IV and V of the NCT level are decomposed into the T+Clifford level and estimated as shown in Table VI. We decompose the Tofffoli gate into 7 T gates + 9 Clifford gates [30], as in [29].

It can be seen that attack costs for 256-bit hash functions PHOTON-Beetle, Sparkle, Xoodyak, and Ascon (ESCH-256) are lower than those for SHA-256 and SHA3, which is the security requirements of NIST. Note that the number of qubits is not counted in D. This should be taken into account. Increasing the number of rounds of permutation, which is the most costly in hash functions, will be one of the ways to satisfy the post-quantum security strength.

TABLE IV: Quantum resources required for Lightweight hash functions.

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	769	37,200	113,360	10,559	95,033
ESCH384	1,025	71,424	217,792	20,248	182,421

Input message length = 256 bits

VI. Conclusion

High throughput data integrity check is essential to protect the communication in IoT systems. In this paper, we proposed

TABLE V: Quantum resources required for SHA-256 and SHA-3.

Algorithm	Qubits	Toffoli gates	CNOT gates	X gates	Depth
SHA-256 [29] (Extrapolation)	2,402	57,184	133,984	·	528,768
SHA-3 [29]	3,200	84,480	332,697,60	85	10,128

Input message length = 256 bits

TABLE VI: Comparison of performance.

Algorithm	Total gates	Depth	D	Security requirements
PHOTON	$1.859 \cdot 2^{147}$	$1.292 \cdot 2^{140}$	$1.201 \cdot 2^{288}$	$1.244 \cdot 2^{296}$ (SHA-256) or $1.574 \cdot 2^{295}$ (SHA3)
ASCON	$1.71 \cdot 2^{148}$	$1.907 \cdot 2^{139}$	$1.63 \cdot 2^{288}$	
Xoodyak	$1.797 \cdot 2^{146}$	$1.164 \cdot 2^{138}$	$1.046 \cdot 2^{285}$	
ESCH256	$1.077 \cdot 2^{148}$	$1.139 \cdot 2^{145}$	$1.227 \cdot 2^{293}$	

Input message length = 256 bits, D = Total gates $\times$ Depth

techniques to optimize the four lightweight hash functions finalists in the NIST standardization (PHOTON-Beetle, Ascon, Xoodyak and Sparkle). All four candidates can achieve high hashing throughput (70 Gbps to 1000 Gbps) on a GPU platform, which can be used to perform high performance data integrity check in IoT systems. The implementation of these four hash functions on quantum computer was analyzed using IBM ProjectQ. Further, we estimated the cost of the Grover preimage 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 the IoT communication (high throughput integrity check) as well as analyze the vulnerabilities of these hash functions against brute-force attack [31].

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 of Things Journal*, vol. 8, no. 12, pp. 9664–9671, 2020.

[2] M. Chen, Y. Liu, J. C. Tam, H.-y. Chan, X. Li, C. Chan, and W. J. Li, "Wireless ai-powered iot sensors for laboratory mice behavior recognition," *IEEE Internet of Things Journal*, 2021.

[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. 18 611–18 621, 2019.

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

[5] L. Bassham, Ç. Çalık, K. McKay, and M. S. Turan, "Submission requirements and evaluation criteria for the lightweight cryptography standardization process," *US National Institute of Standards and Technology*, 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. Abd 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. 207 610–207 618, 2020.

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

[9] 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 of Things Journal*, 2021.

[10] 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 of Things Journal*, 2020.

[11] 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 of Things Journal*, vol. 6, no. 3, pp. 4448–4461, 2018.

[12] 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

[13] 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

[14] J. Daemen, S. Hoffert, S. Mella1, 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

[15] 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

[16] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2," *Submission to the CAESAR Competition*, 2016.

[17] 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

[18] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[19] V. A. Dasu, A. Baksi, S. Sarkar, and A. Chattopadhyay, "Lighter-r: Optimized reversible circuit implementation for sboxes," in *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, 2019, pp. 260–265.

[20] L. Schlieper, "In-place implementation of quantum-gimli," 2020.

[21] 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. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/4776

[22] M. Žnidarič, O. Giraud, and B. Georgeot, "Optimal number of controlled-not gates to generate a three-qubit state," *Physical Review A*, vol. 77, no. 3, Mar 2008. [Online]. Available: http://dx.doi.org/10.1103/PhysRevA.77.032320

[23] I. L. Markov and M. Saeedi, "Constant-optimized quantum circuits for modular multiplication and exponentiation," 2015.

[24] S. Cuccaro, T. Draper, S. Kutin, and D. Moulton, "A new quantum ripple-carry addition circuit," 11 2004.

[25] NIST., "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[26] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying grover's algorithm to aes: quantum resource estimates," 2015.

[27] R. Anand, S. Maitra, A. Maitra, C. S. Mukherjee, and S. Mukhopadhyay, "Resource estimation of grovers-kind quantum cryptanalysis against fsr based symmetric ciphers," Cryptology ePrint Archive, Report 2020/1438, 2020, https://ia.cr/2020/1438.

[28] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, *Implementing Grover Oracles for Quantum Key Search on AES and LowMC*, 05 2020, pp. 280–310.

[29] 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.

[30] M. Amy, D. Maslov, M. Mosca, 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, 06 2012.

[31] P. Liu, S. Li, and Q. Ding, "An energy-efficient accelerator based on hybrid cpu-fpga devices for password recovery," *IEEE Transactions on Computers*, vol. 68, no. 2, pp. 170–181, 2018.