Area-time Efficient Implementation of NIST Lightweight Hash Functions Targeting IoT Applications

Safiullah Khan, *Graduate Student Member, IEEE,* Wai-Kong Lee, *Member, IEEE*, Angshuman Karmakar, Jose Maria Bermudo Mera, Abdul Majeed, and Seong Oun Hwang, *Senior Member, IEEE*

***Abstract*—To mitigate cybersecurity breaches, secure communication is crucial for the Internet of Things (IoT) environment. Data integrity is one of the most significant characteristics of security, which can be achieved by employing cryptographic hash functions. In view of the demand from IoT applications, the National Institute of Standards and Technology (NIST) initiated a standardization process for lightweight hash functions. This work presents field-programmable gate array (FPGA) implementations and carefully worked out optimizations of four Round-3 finalists in the NIST standardization process. A novel compact PHOTON-Beetle implementation is proposed wherein the underlying matrix multiplication is executed in serialized fashion to achieve a small hardware footprint. SPARKLE implementations are carried out by implementing the ARX-box in serialized, parallelized, and hybrid approaches. For Ascon and XOODYAK, the proposed implementations compute certain permutation rounds in one clock cycle in order to explore the trade-off between computation time and hardware area. As a result, this work achieves the smallest hardware footprint for PHOTON-Beetle consuming an area $3.4\times$ smaller than state-of-the-art implementations. Ascon and XOODYAK are implemented in a flexible manner that achieves throughput-to-area (TP/A) ratios $1.8\times$ and $3.9\times$ higher, respectively, compared to implementations found in the literature. In addition, we propose the first FPGA implementations for the SPARKLE hash function. These efficient implementations provide guidelines for choosing a suitable architecture for applications in demand that can be employed in the IoT environment to achieve data integrity for various applications.**

***Index Terms*—Hash Functions, IoT, Lightweight Cryptography, Field-programmable Gate Array (FPGA), National Institute of Standards and Technology (NIST).**

I. INTRODUCTION

Manuscript received April 19, 2021; revised August 16, 2021.

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under Grant RS-2022-00198001, and by the National Research Foundation of Korea (NRF) under Grant 2020R1A2B5B01002145, all funded by the Korean Government through Ministry of Science and ICT (MSIT).

Safiullah Khan is with the Department of IT Convergence Engineering, Gachon University, 13120 Seongnam, South Korea (safi@gachon.ac.kr),

Angshuman Karmakar and Jose Maria Bermudo Mera are with the COSIC Research Group, Katholieke Universiteit Leuven, Belgium (angshuman.karmakar@esat.kuleuven.be; jose.bermudo@esat.kuleuven.be), Angshuman Karmakar is also with Computer Science and Engineering department at Indian Institute of Technology, Kanpur, India. Jose Maria Bermudo Mera is also with PQShield Ltd, UK.

Wai-Kong Lee, Abdul Majeed and Seong Oun Hwang are with the Department of Computer Engineering, Gachon University, 13120 Seongnam, South Korea (waikonglee@gachon.ac.kr; ab09@gachon.ac.kr; sohwang@gachon.ac.kr),

Corresponding author: Seong Oun Hwang; sohwang@gachon.ac.kr

THE Internet of Things (IoT) emerged as a technology that motivated numerous ingenious applications. The IoT incorporates a variety of smart applications that can significantly enhance the quality of our lives when combined with other important technologies such as artificial intelligence (AI) and cloud computing. Common examples like smart cities [1] and smart homes [2] are only possible when the IoT merges with other appropriate technologies. In an IoT environment, sensitive and important data are communicated at high speed [3]. The integrity of the transmitted data must remain intact to achieve secure communications.

Data integrity is highly desirable for IoT applications, which can be achieved by employing cryptographic hash functions. A hash function converts input of an arbitrary length to a fixed-length, deterministic output hash value. It can be used to verify the integrity of messages (e.g., IoT sensor data) communicated over IoT networks. Traditional hash functions like SHA-3 [4], [5] may not be suitable for resource-constrained IoT sensor nodes that demand low computation and memory consumption. Therefore, the National Institute of Standards and Technology (NIST) initiated a competition [6] to select a standard for Lightweight Cryptography (LWC) with applications to resource-constrained systems.

A large number of hashes need to be computed at sensor nodes (generating tags), gateways, and cloud servers (verifying integrity). However, sensor nodes are mostly built on resource-constrained devices like microcontrollers (e.g., Cortex-M0, AVR) or low-power application-specific integrated circuits (ASICs). Although the existing LWC schemes can provide reasonable performance in sensor nodes, they may not work for all IoT applications. For instance, some IoT applications require high throughput (TP) (e.g., video surveillance), while others are more concerned with small hardware areas and low power consumption (e.g., a remote weather station). This implies there is no single hardware architecture that matches all the demands from hash computation in IoT applications.

To address this concern, this work implements the four finalists in the NIST competition in such a way to realize the demands for varying performance for IoT applications. Ascon [7] and XOODYAK [8] are implemented based on a certain number of permutation rounds executed in each clock cycle. This provides flexibility to choose from the area-constrained to high-speed architectures suitable for different IoT applications. Similarly, SPARKLE [9] hardware implementations are based on serialized, parallelized, and hybrid approaches for permu-

tation function, which again address the varying requirements for the IoTs. This work also suggests the implementation of PHOTON-Beetle [10] that consumes the least hardware by serializing the underlying matrix multiplication. Therefore, by employing careful design and optimization strategies, LWC hardware architectures can either produce a high TP, a balanced area-time, or extremely small hardware footprints. As a result, all of these implementation strategies found their own applications in the IoT environments.

A. Contributions

In this work, we propose several architectures for NIST finalist hash functions that achieve high TP, area-time efficiency, or low hardware area consumption. This allows IoT practitioners to select the architectures that best suit their needs. The contributions of this article can be summarized as follows.

- There are few implementations of PHOTON-Beetle in the existing literature that are based on round-based executions. In this work, we explore the possibility of achieving the most area-efficient architecture for PHOTON-Beetle catering to IoT applications. Hence, we propose a novel serialized architecture that re-uses the multiplier hardware architecture to achieve the matrix multiplication necessary for the MixColumnSerial step. We also propose a parallelized version based on implementation of a single permutation round in one clock cycle with an area-optimized S-box architecture. The serialized version can reduce hardware consumption by 40%, compared to the parallelized version, while the critical path delay remains the same.
- This work presents the first FPGA implementations for SPARKLE that explore various possible combinations of parallelized and serialized implementation strategies. The first architecture is a round-based implementation that executes one permutation round-per-clock cycle. The second architecture executes the ARX-box in a serialized fashion to reduce area consumption. The third architecture combines the two design approaches, wherein a certain number of ARX-boxes work in a serialized manner while the others work in parallel. The round-based approach achieved the highest TP and area-time efficiency, while the serialized ARX-box provided the smallest hardware footprint.
- The design space for Ascon and XOODYAK has not been explored fully in the literature as they suggest implementing one round in one clock cycle [11], [12]. This work proposes executing several possible permutation rounds in one clock cycle. Extensive evaluation of the available design space exploration is performed, which is missing from prior works. Trade-offs between different implementation parameters are observed. Implementation results for Ascon showed that by executing four permutations per clock cycle, maximum TP can be achieved. Increasing the number of permutation rounds increases the critical path, which in turn limits the operating frequency. XOODYAK, on the other hand (which is inherently serialized), is also studied based on several permutation rounds for each clock cycle. The flexible implementations for Ascon and XOODYAK outperformed the state-of-the-art implementations with respect to area consumption and TP/A ratios.

The rest of this article is organized as follows. Section II presents the background, literature review, and descriptions of the selected hash functions. Implementation of the hash functions, along with optimized architectures and preliminary results, are given in Section III. Section IV is dedicated to the analysis of the hash functions and comparison with other counterparts. The applications are discussed in Section V. Finally, the article concludes in Section VI.

II. Background and Literature Review

This section describes how secure communication is possible in IoT environments by employing hash functions. Furthermore, an overview of the literature is provided, and a summary of the selected hash functions is accommodated at the end of this section.

A. Secure communication in the IoT environment

The typical IoT architecture comprises three main entities: sensor nodes, gateways, and cloud servers. The sensor nodes are responsible for collecting and transmitting important data to the gateways. They are resource-constrained, introducing additional tasks (e.g., a data-integrity checking mechanism) can pose non-negligible overhead in terms of hardware area and computation time. Therefore, design considerations for secure sensor nodes target a small area and low latency, which require specialized optimization strategies to achieve. Gateway devices capable of handling large amounts of data from IoT sensor nodes usually incorporate a processor with a higher power. This in turn requires an implementation strategy that fulfills the demand for higher TP. Normally, sensor nodes are not connected directly to the Internet. They communicate with gateway devices via wireless technology such as Bluetooth Low Energy (BLE) or Zigbee [13]. On the other hand, gateway devices are connected to cloud servers through Internet communications, often secured by the TLS protocol. Microcontrollers and FPGAs [14] are typically used to implement IoT sensor nodes. On the other hand, gateway devices can be implemented on the FPGA platform or can utilize an FPGA accelerator [15]. The overall architecture of typical IoT communication is shown in Fig. 1.

IoT edge devices are so prevalent that sensitive or confidential information is frequently included in the transmitted data. It is important for the transmitted data to be secured against malicious manipulation in order to guarantee data privacy and confidentiality. As a result, IoT sensor nodes must incorporate cryptographic features. In order to ensure that the gathered sensor data are not maliciously altered during the transmission process from sensor nodes to the cloud server, data integrity is critical. Any malicious change of the conveyed sensor data can be easily identified using a cryptographic hash function. This enables us to validate sensor data integrity at the gateway or on the server, considerably enhancing the security of IoT

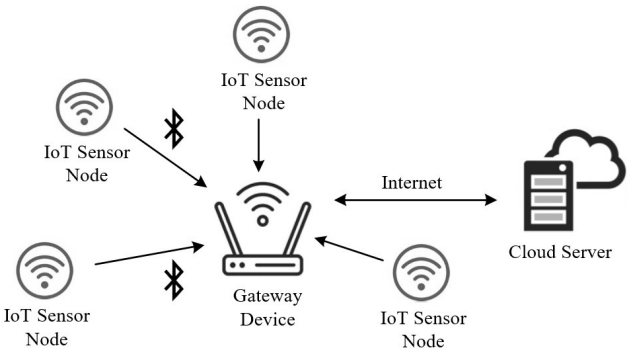


Fig. 1. The typical IoT communication architecture

connectivity. In addition, hash functions are used to create mutual authentication protocols [16] and hash-based message authenticated codes (HMAC) [17].

B. Literature Review

In this subsection, we review several implementations found in the literature on lightweight hash functions. Hardware implementations of PHOTON-Beetle on a Startix-IV device were provided [18]. The focus of that work was parallel architecture implementations with one round-per-clock cycle. A 2×1 multiplexer is employed to select either the initial value or feedback from the internal state. PHOTON-Beetle, Ascon, and XOODYAK implementations were reported in [11] and [12]. The aim of these two articles was to provide a benchmark for all candidates in the NIST LWC standardization process. Performance optimization was not the main focus.

Ascon implementations reported in [19] are based on a basic iterative architecture. For hash function implementation, permutation is performed in 12 clock cycles where one permutation round is executed in one clock cycle. Implementations have been performed for the Artix-7 platform. Implementation of a variant of Ascon was introduced [20]. The authors claimed the permutation function to be more lightweight than the original Ascon because of a modified S-box. The authors have used the Kintex-7 platform for the implementations. Ascon implementations based on unrolled, round-based, and serialized permutation functions were presented in [21]. Although the implementations were performed for the authenticated encryption, the underlying permutation function is the same. Note that our work is the first to provide optimized architectures for SPARKLE hash functions.

C. Lightweight hash functions

NIST announced the finalists for the standardization competition in March 2021. For all the candidates, computation of the hash is based on permutation functions. This subsection provides an overview of four hash functions. More detailed explanations about the respective specifications can be found in the NIST submission to the standardization process [22]. Table I provides the notations employed to describe operations involved in the hash functions.

TABLE I
OPERATIONS INVOLVED IN LIGHTWEIGHT HASH FUNCTIONS.

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

1) PHOTON-Beetle: $\texttt{PHOTON}_{256}$ [23] is the underlying 256-bit permutation for the PHOTON-Beetle family. The permutation is applied to a state comprising 64 elements of four bits each. The state is represented as an 8×8 matrix, X. The permutation consists of 12 rounds where each round includes four steps: AddConstant, SubCells, Shiftrows, and MixColumnSerial. A description of a single round of permutations is in Algorithm 1. The first step, AddConstant, adds a constant to each element of the first column of the internal state. SubCells is simply a substitution box for each four-bit element, while Shiftrows rotates the positions of the elements in the matrix for each row. Finally, MixColumnSerial mixes all the columns by employing matrix multiplication.

Algorithm 1 $\texttt{PHOTON}_{256}(X)$

Require: X' (Updated 8×8 matrix)
Ensure: X (Input 8×8 matrix)
AddConstant (X, k)
1: $RC[12] \leftarrow \{1, 3, 7, 14, 13, 11, 6, 12, 9, 2, 5, 10\}$
2: $IC[8] \leftarrow \{0, 1, 3, 7, 15, 14, 12, 8\}$
3: **for** $i = 0$ to 7 **do**
4: $\quad X[i, 0] \leftarrow X[i, 0] \oplus RC[k] \oplus IC[i]$;
SubCells (X)
5: **for** $i = 0$ to 7, $j = 0$ to 7 **do**
6: $\quad X[i, j] \leftarrow$ S-box $(X[i, j])$;
ShiftRows (X)
7: **for** $i = 0$ to 7, $j = 0$ to 7 **do**
8: $\quad X[i, j] \leftarrow X[i, (j + i)\%8]$;
MixColumnSerial (X)
9: $\quad M \leftarrow$ Serial$[2, 4, 2, 11, 2, 8, 5, 6]$;
10: $\quad X' \leftarrow M^8 \odot X$;
return X'

PHOTON-Beetle-Hash takes as input a message, $M \in \{0,1\}^*$, of arbitrary length, and generates a hash, $H \in \{0,1\}^{256}$. The first 128 bits of the input message along with 128-bit 0's are absorbed by the permutation function (12 rounds) as the initial vector followed by a consecutive absorption rate of $r = 32$. The final message block, if incomplete, is concatenated with the minimum zeros to make it 32-bit, and acts as input to the next permutation. In addition, for the final message block, a small constant is XORed in the capacity part, depending on whether the final block is partial or full, to support the domain separation. The 256-bit hash is generated in two steps of 128-bit each. Fig. 2 shows the architecture of PHOTON-Beetle-Hash. The only recommended size for the hash function is PHOTON-Beetle-

Hash_[32], where message M is parsed into blocks, each of 32-bit after the first 128-bit block. This architecture aims to achieve an extremely small hardware footprint along with excellent throughput and energy efficiency.

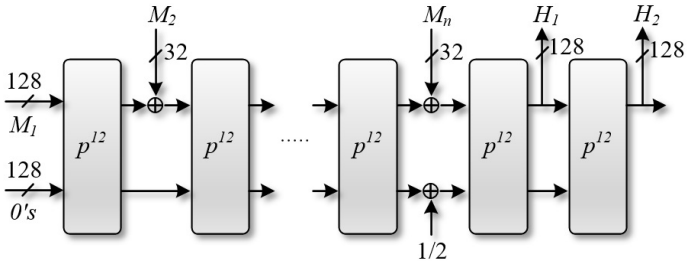


Fig. 2. The PHOTON-Beetle Hash architecture.

2) Ascon: Ascon permutations are applied to an internal state of 320-bit and follow an iterative substitution permutation network (SPN)-based round transformation [24]. The state is updated in three steps: addition of a round constant, a substitution layer, and the linear diffusion layer. The state for Ascon is divided into five 64-bit words where the first word constitutes the outer r-bit part, S_r, and the next four words compose the inner c-bit part, S_c, as shown in (1):

$$S = S_r \parallel S_c = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4 \tag{1}$$

The first step of the permutation is addition of the round constant, in which constant c_r is added to word x_2 of the internal state for each permutation round. (the values for round constants can be found in [25]):

$$x_2 \leftarrow x_2 \oplus c_r \tag{2}$$

The substitution layer can be seen as 64 parallel operations of the five-bit S-box. The S-box is applied to each bit-slice of the five internal registers. For hardware implementations, the S-box for Ascon can be implemented with a few logical operations, making it highly parallelized. The final step is the linear diffusion layer. The linear diffusion layer provides diffusion within each of the 64-bit words. The description of a single round of permutations is given in Algorithm 2.

Algorithm 2 Ascon Permutation.

Require: S' (Updated 320-bit state)
Ensure: S (Input 320-bit state)
AddConstant (S)
1: $S = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4$
2: $x_2 \leftarrow x_2 \oplus c_r$;
S-Box (S)
3: **for** $i = 0$ to 63 **do**
4: $S[i] \leftarrow$ S-box ($S[i]$);
LinearDiffusion (S)
5: **for** $i = 0$ to 4, **do**
6: $x'_i = x_i \oplus (x_i \ggg l_i) \oplus (x_i \ggg r_i)$
7: $S' = x'_0 \parallel x'_1 \parallel x'_2 \parallel x'_3 \parallel x'_4$
return S'

The Ascon hash is based on a mode of operation similar to sponges [26]. Fig. 3 shows the overall architecture for the Ascon hash. The first step in hash computation is initialization. During initialization, the 320-bit internal state is formed by concatenation of the initialization vector (IV) with a certain number of $0's$ to make it 320-bit. The IV in turn is a constant specifying the algorithm parameters, and is pre-defined by the algorithm. The IV contains information about the rate, r, the number of rounds, and the length of the output hash. The permutation for the IV can also be pre-computed because it is independent of the input message. The next step is absorbing the message. The arbitrary length of the message, M, is parsed into blocks of r-bits. The final block of the message is appended with a single 1 and a minimum number of $0's$ to make it a multiple of r in case it is not already a multiple of r. Each message block is XORed with the r-bit of the internal state followed by 12 permutation rounds. Eq. 3 explains the absorbing of the message:

$$S \leftarrow p^{12}((S_r \oplus M_i) \parallel S_c) \tag{3}$$

Finally, during the squeezing step, the hash is extracted from the state in the form of r-bit blocks until the required length of the hash is completed. After each extraction, 12 permutation rounds are applied to the state to generate the next block of the hash. Eq. 4 explains the squeezing step:

$$H_i \leftarrow S_r,\ S \leftarrow p^{12}(S), \qquad 1 \le i \le \lceil l/r \rceil \tag{4}$$

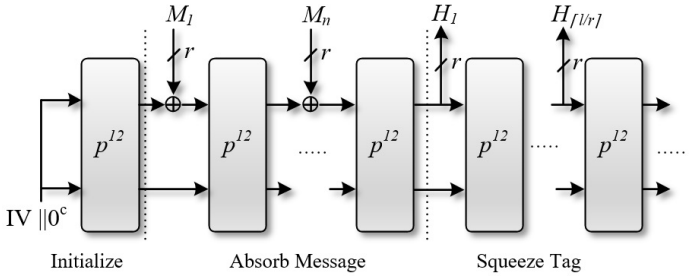


Fig. 3. The Ascon hash architecture.

3) SPARKLE*:* The SPARKLE family comprises permutations SPARKLE256_{n_s}, SPARKLE384_{n_s}, and SPARKLE512_{n_s} with block sizes of 256, 384, and 512 respectively, where n_s is the number of steps taken during the permutation. The permutation consists of two main steps: the *ARX-box*, which is a 64-bit cipher with a 32-bit key, and the linear diffusion layer. Algorithm 3 and Algorithm 4 show the high-level structure for the permutation functions.

The *ARX-box* Alzette, known as A for short, can be realized as a four-round iterated block cipher where each round differs in the constant set for the rotations. After rotation, the 32-bit key is XORed to the left. This provides non-linearity to the permutation. For x and y as inputs, and for c as the key, the *ARX-box* can be realized as follows:

$$\begin{aligned}
&x \leftarrow x + (y \ggg 31),\ y \leftarrow y \oplus (x \ggg 24),\ x \leftarrow x \oplus c\\
&x \leftarrow x + (y \ggg 17),\ y \leftarrow y \oplus (x \ggg 17),\ x \leftarrow x \oplus c\\
&x \leftarrow x + (y \ggg 00),\ y \leftarrow y \oplus (x \ggg 31),\ x \leftarrow x \oplus c\\
&x \leftarrow x + (y \ggg 24),\ y \leftarrow y \oplus (x \ggg 16),\ x \leftarrow x \oplus c
\end{aligned} \tag{5}$$

The linear diffusion layer, $\mathcal{L}_{n_b}$, where n_b is the number of branches, consists of the Feistel round and the branch

Algorithm 3 SPARKLE384$_{n_s}$.

Require: $((x'_0, y'_0), ..., (x'_5, y'_5))$ (Updated 384-bit state)
Ensure: $((x_0, y_0), ..., (x_5, y_5))$ (Input 384-bit state)
$(c_0, c_1) \leftarrow$ (0xB7E15162, 0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56, 0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB, 0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8, 0xC2B3293D)
1: **for all** $s \in [0, n_s - 1]$ **do**
2: $y_0 \leftarrow y_0 \oplus c_{(s \ mod \ 8)}$
3: $y_1 \leftarrow y_1 \oplus (s \ mod \ 2^{32})$
4: **for all** $i \in [0, 5]$ **do**
5: $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
6: $((x'_0, y'_0), ..., (x'_5, y'_5)) \leftarrow \mathcal{L}_6((x_0, y_0), ..., (x_5, y_5))$
return $((x'_0, y'_0), ..., (x'_5, y'_5))$

Algorithm 4 SPARKLE512$_{n_s}$.

Require: $((x'_0, y'_0), ..., (x'_7, y'_7))$ (Updated 512-bit state)
Ensure: $((x_0, y_0), ..., (x_7, y_7))$ (Input 512-bit state)
$(c_0, c_1) \leftarrow$ (0xB7E15162, 0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56, 0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB, 0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8, 0xC2B3293D)
1: **for all** $s \in [0, n_s - 1]$ **do**
2: $y_0 \leftarrow y_0 \oplus c_{(s \ mod \ 8)}$
3: $y_1 \leftarrow y_1 \oplus (s \ mod \ 2^{32})$
4: **for all** $i \in [0, 7]$ **do**
5: $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
6: $((x'_0, y'_0), ..., (x'_7, y'_7)) \leftarrow \mathcal{L}_8((x_0, y_0), ..., (x_7, y_7))$
return $((x'_0, y'_0), ..., (x'_7, y'_7))$

permutation. $\mathcal{L}_6$ is employed for SPARKLE384 while $\mathcal{L}_8$ is employed for SPARKLE512. The output from the *ARX-box* is fed to the Feistel round, which performs the following operations:

$$\begin{aligned}
&(t_x, t_y) \leftarrow (x_0 \oplus x_1 \oplus x_2, \ y_0 \oplus y_1 \oplus y_2) \\
&(t_x, t_y) \leftarrow (t_x \oplus (t_x \ll 16)) \lll 16, (t_y \oplus (t_y \ll 16)) \lll 16 \\
&(y_3, y_4, y_5) \leftarrow (y_3 \oplus y_0 \oplus t_x, y_4 \oplus y_1 \oplus t_x, y_5 \oplus y_2 \oplus t_x) \\
&(x_3, x_4, x_5) \leftarrow (x_3 \oplus x_0 \oplus t_y, x_4 \oplus x_1 \oplus t_y, x_5 \oplus x_2 \oplus t_y)
\end{aligned} \tag{6}$$

$$\begin{aligned}
&(t_x, t_y) \leftarrow (x_0 \oplus x_1 \oplus x_2 \oplus x_3, \ y_0 \oplus y_1 \oplus y_2 \oplus y_3) \\
&(t_x, t_y) \leftarrow (t_x \oplus (t_x \ll 16)) \lll 16, (t_y \oplus (t_y \ll 16)) \lll 16 \\
&(y_4, y_5, y_6, y_7) \leftarrow (y_4 \oplus y_0 \oplus t_x, y_5 \oplus y_1 \oplus t_x, y_6 \oplus y_2 \oplus t_x, \\
&\qquad y_7 \oplus y_3 \oplus t_x) \\
&(x_4, x_5, x_6, x_7) \leftarrow (x_4 \oplus x_0 \oplus t_y, x_5 \oplus x_1 \oplus t_y, x_6 \oplus x_2 \oplus t_y, \\
&\qquad x_7 \oplus x_3 \oplus t_y)
\end{aligned} \tag{7}$$

Branch permutation is just swapping left branches with right branches:

$$\begin{aligned}
&(x_0, x_1, x_2, x_3, x_4, x_5) \leftarrow (x_4, x_5, x_3, x_0, x_1, x_2) \\
&(y_0, y_1, y_2, y_3, y_4, y_5) \leftarrow (y_4, y_5, y_3, y_0, y_1, y_2)
\end{aligned} \tag{8}$$

ESCH256 and ESCH384 are the two instances for the hash, as shown in Fig. 4. The lengths of the hash output for ESCH256 and ESCH384 are 256-bit and 384-bit, respectively, with primary member ESCH256. A sponge construction is employed where a slim version is employed during absorption and squeezing while a big version is employed during these phases. Rate r is fixed at 128-bit, which means the message should be padded with 1 followed by the minimum number of 0's to make it a multiple of 128. Different numbers of permutations are required for both hashes: SPARKLE384$_7$ and SPARKLE384$_{11}$ for ESCH256; SPARKLE512$_8$ and SPARKLE512$_{12}$ for ESCH384. The message block is first transformed through $\mathcal{M}_{h_b}$, where $h_b = n_b/2$ in which n_b is the number of branches. To generate the output, 128-bit blocks are extracted from the state, and permutation is performed again before the final block is extracted. For the input message where the length is a multiple of r, no padding is performed. A constant is XORed to the inner part of the state, which is different depending on whether the message is padded or not.

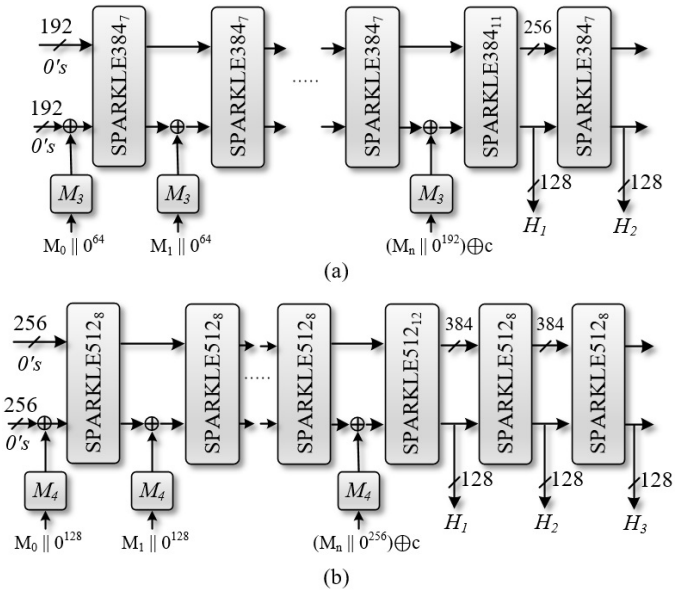


Fig. 4. (a) ESCH256 and (b) ESCH384.

4) XOODYAK*:* XOODYAK is a symmetric key cryptographic object that can be employed for hashing, pseudo-random bit generation, authentication, encryption, and authenticated encryption. The underlying permutation in XOODYAK is XOODOO, specified by the number of rounds, n_r. XOODOO was inspired by Gimli, [27] and the round function is much closer to Keccak. The permutation is an iterated structure applied to round functions in the 384-bit state, which is divided into three planes, each a 128-bit state. Sheets are arrays of three lanes on top of each other. The XOODOO state, sheet, and plane are illustrated in Fig. 5.

The permutation for the hash computation consists of 12 round functions, and each round function in turn has five steps. Algorithm 5 shows the steps involved in one permutation round. For the first step (the mixing layer), the planes (A_i) are added, and the result is given a cyclic shift, which is then

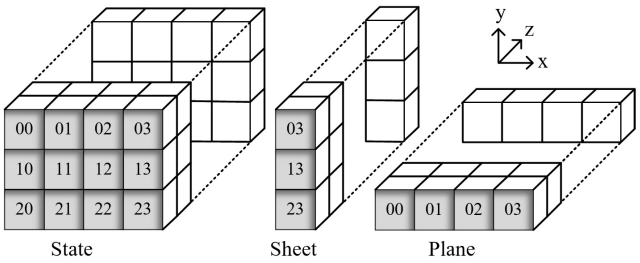


Fig. 5. XOODOO state, sheet, and plane

added to each plane. The next step is plane shifting for which two planes are given as a cyclic shift with a certain constant specified in the algorithm. This is followed by addition of the constant. The next-to-last step is the non-linear layer. This step takes the bit-wise complement of each plane and computes the bit-wise product (AND) with the un-complemented plane values. The updated values are then XORed with the original plane values. The non-linear layer operates in three-bit units, so it can be considered a parallel application of three-bit S-boxes on 128 columns. Another plane shifting with different constant values constitutes the final step of the round function. One of the limitations mentioned by the authors is that XOODYAK is inherently serial at the construction level.

Algorithm 5 XOODOO.

Require: A'_0, A'_1, A'_2 (Updated 128-bit sheets)
Ensure: A_0, A_1, A_2 (Input 128-bit sheets)
Mixing Layer 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 $\in \{0,1,2\}$
Plane Shifting ρ_{west}
$A_1 \leftarrow A_1 \lll (1,0)$
$A_2 \leftarrow A_2 \lll (0,11)$
Addition of Constant i
$A_0 \leftarrow A_0 + C_i$
Non-linear Layer $(\mathcal{X})$
$B_0 \leftarrow \overline{A_1} \cdot A_2$
$B_1 \leftarrow \overline{A_2} \cdot A_0$
$B_2 \leftarrow \overline{A_0} \cdot A_1$
$A_y \leftarrow A_y + B_y$ for y $\in \{0,1,2\}$
Plane Shifting ρ_{east}
$A_1 \leftarrow A_1 \lll (0,1)$
$A_2 \leftarrow A_2 \lll (2,8)$

III. HASH FUNCTION IMPLEMENTATIONS AND OPTIMIZATIONS

A. PHOTON-Beetle

The implementations of PHOTON-Beetle in the literature follow a single round-based architecture where one permutation is executed in one clock cycle. However, some IoT applications use sensor nodes that demand a highly area-optimized implementation. Therefore, this work explores a novel strategy to implement a serialized version of the MixColumnSerial operation by re-using the multiplier design, that is area-optimized. A round-based architecture with a novel S-box based on K-map simplification is also presented in this work.

To illustrate the implementation of PHOTON-Beetle, a single permutation round is presented first from the hardware point of view. From Algorithm 1, the first step during permutation is AddConstant, which can be accomplished in hardware as an XOR operation of the state matrix with constants defined by the algorithm. RC is the round constant while IC is the internal constant. The constants are added to the first column of the state matrix. The next step is SubCells, simply a substitution for each of the elements in the state matrix. This can be achieved by either storing the corresponding values in block RAM (BRAM) or computing the values on the fly. For faster execution and to reduce memory accesses, this work proposes to design the S-Box at the gate level instead of storing the values. To design the S-box, the K-map technique is employed, which can provide an optimized version of the S-box with respect to the number of gate levels in order to reduce the critical path delay. Fig. 6 shows the architecture of the S-box. Each of the four-bit elements of the state is fed to the circuit, and the corresponding output for the S-box is generated. Parallel operations of such S-boxes compute this step. This is followed by the Shiftrows operation, which is a very simple operation in the hardware, equivalent to rearranging the wires with no actual hardware involved. In order to obtain the hash, the most expensive operation for this algorithm is MixColumnSerial which involves matrix multiplication. For efficient hardware implementations of PHOTON-Beetle-Hash, this work proposes two techniques: parallelized and serialized. The terms parallelized and serialized for the hash function originate from implementation of the parallel and serial versions of the matrix multiplication, because it is the most expensive operation in the computation of the permutation.

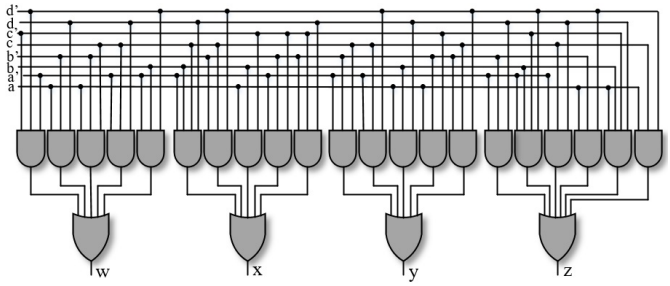


Fig. 6. The S-box for PHOTON-Beetle.

The matrix multiplication operation can be depicted using Fig. 7. Prior works from the literature followed the parallelized technique, where the final result of matrix multiplication is computed in one clock cycle. This can be achieved because all the elements in the state matrix can be accessed independently, providing a chance to design an architecture that is able to compute the resultant matrix in one clock cycle. The multipliers $x_{01}...x_{64}$ take one row from matrix M and one column from matrix X and generate one element of the resultant matrix. All these multipliers operate in parallel. For the proposed novel serialized technique, the MixColumnSerial operation is computed in series. This technique takes the first

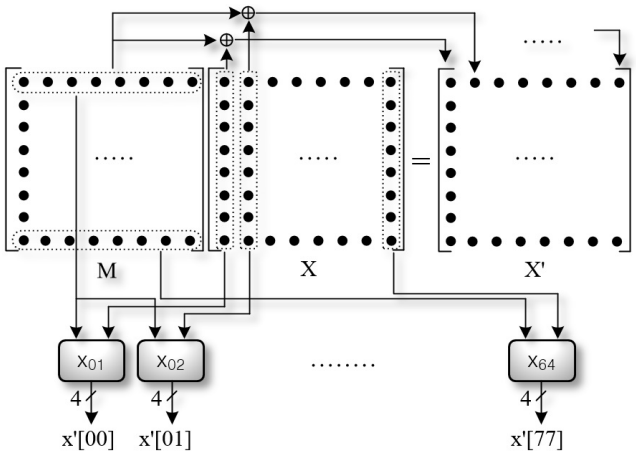


Fig. 7. The underlying matrix multiplication.

row of matrix M and multiplies it with all the columns of matrix X in one clock cycle. In this way, multiplication can be computed in eight clock cycles. The multipliers $x_{01}...x_{64}$ are divided into groups, each having eight multipliers. The same hardware is utilized, thus reducing the area consumption at the cost of clock cycle utilization, but the critical path remains the same. For the PHOTON-Beetle-Hash computation, MixColumnSerial consumes around 70% of the total hardware area. Comparing both techniques, by employing the serialized technique, an area reduction of almost 40% can be achieved.

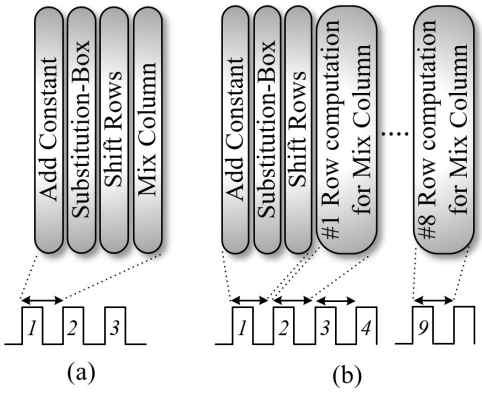


Fig. 8. Clock cycle instances for PHOTON-Beetle.

A detailed explanation with respect to clock cycles and hardware re-utilization is presented in Fig. 8. Fig. 8 (a) shows one permutation round executed in one clock cycle under the parallelized architecture. Because the permutation is repeated 12 times, 12 clock cycles are required to compute the whole permutation function. The number of clock cycles for the parallelized architecture can be estimated as $12 \times p + 2$, where p represents the number of permutation functions employed, which in turn is dependent upon the length of the input message. Each permutation function requires 12 clock cycles, where two clock cycles are required to extract the hash. The same hardware is utilized as the output is fed again into the same hardware to compute the next permutation round. Fig. 8 (b) illustrates the operations performed during each clock cycle under the serialized architecture. One permutation round consumes 10 clock cycles (one clock cycle is overhead from arranging the values). Twelve rounds for permutation thus require 120 clock cycles, and clock cycles for hash computation employing the serialized architecture can be estimated as $120 \times p + 2$. Instead of using the whole parallel multiplier, this architecture reuses only eight multipliers, which leads to area reduction.

TABLE II
IMPLEMENTATION RESULTS FOR PHOTON-BEETLE HASH.

Technique	Area (LUTs)	Freq. (MHz)	Latency (cc)	Time (μs)	TP (Mbps)	TP/A ratio (Mbps/LUT)
Virtex-7						
Parallelized	998	254.45	38	0.149	1073.82	1.075
Serialized	602	241.95	362	1.496	106.95	0.177

The performance of parallelized and serialized hash functions is given in Table II. As can be seen, hardware consumption in terms of LUTs can be reduced by employing the serialized version, but at the cost of more clock cycles. The serialized version achieved almost a 40% reduction in hardware area because the hardware is re-utilized. The operating frequency for both the designs is the same because the critical path for both architectures is identical. Processing more multiplications add circuitry that runs in parallel, thus increasing the hardware area. For a single permutation round, the serialized version takes nine clock cycles, making clock cycle consumption $10\times$ more than the parallelized version. This directly affects computation time, which increases by $10\times$ that of the parallelized version. TP and TP/A ratios follow the same trend.

B. Ascon

Unlike the execution of one permutation round per clock cycle, the design space for Ascon has been fully explored based on several possible number of permutations in each clock cycle. We observed a trade-off between various important performance parameters, which is missing from the prior works.

A round-based implementation of Ascon is presented where different numbers of permutation rounds are executed in one clock cycle. The architecture for a single permutation round is given in Fig. 9. Each permutation round is divided into three steps, which are discussed here with respect to the hardware implementations. The addition of the round constant is a simple operation as it takes the round constant and XORs it to the content of word x_2. A single XOR operation is employed. The next step is S-box implementation, which consumes the majority of the area for the permutation round. Although realization of the S-box through BRAM is possible, this work focuses on implementation of the S-box using combinational logic. The combinational logic implementation based on a bit-slice technique has the advantage of being fast and secure simultaneously, because side-channel attacks can compromise the values stored in BRAM. 64 of such S-boxes operate in parallel for each round to update the whole state. The final layer (a linear diffusion layer) is shifting, and the XOR operation is applied to the words, which is again

simple to achieve in hardware. Shifting requires no hardware because it can be achieved through correct bit selection. This constitutes the Ascon round core, when one permutation round is implemented in one clock cycle. The internal state between permutation operations is stored in five 64-bit registers. The number of permutation rounds implemented in one clock cycle can be increased as well. This work presents z permutation rounds to be implemented in one clock cycle. Because the algorithm specifies the number of permutation rounds to be 12, the possible number of permutations to be fit into one clock cycle must be a factor of 12. A round core (employing z permutation rounds) is employed, and the updated intermediate state is again fed into the same hardware, thus re-utilizing the hardware. The round core consumes the majority of area, but a small amount is consumed by the control circuit that is responsible for managing the inputs and outputs for the round core.

The proposed strategy implements several permutation rounds in one clock cycle; therefore, for Ascon hash hardware implementations, z permutations are computed in one clock cycle, where $z = \{1, 2, 4, 6\}$. Squeezing in more permutations per clock cycle can reduce clock cycle consumption but at the cost of more hardware area. The number of clock cycles utilized can be computed as $cc = 96/z + 4$.

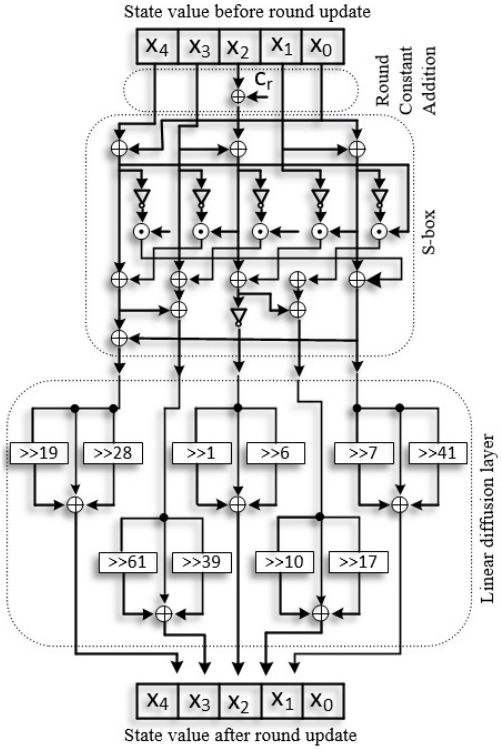


Fig. 9. A single permutation round for Ascon.

Table III provides performance results for the Ascon hash on the FPGA platform. As the number of permutation rounds executed in one clock cycle increases, area consumption increases linearly, as depicted by the growing number of LUTs. When more permutations are packed in per clock cycle, the critical path increases, thus limiting the operating frequency. However, the number of clock cycles required is reduced because more permutations are forced to execute in one clock cycle. The overall time to compute the hash is an important parameter, which in turn is dependent upon both operating frequency and clock cycles. More clock cycles mean more time consumed, i.e., a direct relation, while time is inversely related to frequency. The overall time period decreases till four permutation rounds are executed in a clock cycle, while for six rounds in one clock cycle, the time period increases. This is because the critical path becomes long enough to negate the reduced clock cycle effect. Two important parameters are TP and TP/A because they incorporate every parameter and provide a picture for the overall architecture performance. By increasing the number of permutations in one clock cycle, TP increases. The maximum TP can be achieved for $z = 4$ but further increasing the permutations results in degradation of TP. This is because the critical path increases, limiting the operating frequency, and thus TP. Consequently, the overall computation time increases despite the decrease in latency. TP/A incorporates area as well, and is the ratio of TP to LUTs consumed. TP/A decreases linearly as the area increases with more permutation rounds per clock cycle.

TABLE III
IMPLEMENTATION RESULTS FOR ASCON HASH.

Technique	Area (LUTs)	Freq. (MHz)	Latency (cc)	Time (μs)	TP (Mbps)	TP/A ratio (Mbps/LUT)
Virtex-7						
1-round	773	272.03	100	0.367	697.54	0.993
2-rounds	1290	224.56	52	0.231	1108.22	0.869
4-rounds	2550	162.68	28	0.172	1488.37	0.581
6-rounds	3855	108.45	20	0.184	1391.30	0.373

C. SPARKLE

The first FPGA implementations of the SPARKLE hash function based on a possible configuration of the ARX-box are provided in this work. Similar to the previous hash function architectures that we propose, we explored the possibilities of serializing the ARX-box in SPARKLE to achieve an area-efficient architecture. Optimized hardware implementations of ESCH256 and ESCH384 are presented in this section. For both ESCH256 and ESCH384, the input message is not directly fed into the permutation rounds. The pre-processing is first performed, on the input message. Module M, responsible for pre-processing, consumes four clock cycles because the full message length can be simultaneously processed in parallel irrespective of the number of blocks in messages. Once the message is transformed, the hash computations can be performed. For the computation of ESCH256 and ESCH384, three optimization techniques have been employed.

1) Parallelized ARX-box: In the first version, one permutation round is executed in one clock cycle. The ARX-box inside each permutation round is processed in a fully parallel fashion. The output from the ARX-box is then transferred to the diffusion layer. Six and eight ARX-boxes are employed for ESCH256 and ESCH384, respectively, in parallel for each round. During the computation of the hash, one such round of permutations is adopted, which is then re-used to compute the entire hash. The transformed input message

(192-bit for ESCH256 and 256-bit for ESCH384) is the input for the permutation round. The permutation round is re-used for the next seven clock cycles to generate the output for SPARKLE384_7, and eight clock cycles are needed for SPARKLE512_8. Once complete, the next transformed input message block is XORed to the internal state. For the last message block, SPARKLE384_{11} is employed for ESCH256, and SPARKLE512_{12} is employed for ESCH384 and the first 128-bit hash are extracted at the end of the permutation. This requires 11 and 12 clock cycles; after that SPARKLE384_7 and SPARKLE512_8 are employed again, and the final 128-bit hash is extracted. SPARKLE512_8 is employed three times because the length of the output hash is 384-bit. The hash outputs are concatenated to achieve the final hash value. The clock cycle consumption in this case is equivalent to the number of permutation rounds plus the initial message processing overhead. Fig. 10 (a) shows a single permutation round based on the parallelized approach.

2) Serialized ARX-box: For the second version of hash computation, a serialized approach was adopted. Since all the ARX-boxes operating in parallel consume a larger hardware area, it is beneficial to employ a serialized approach to reduce area consumption. The basic component for this approach consists of one single ARX-box and one diffusion layer. The message pre-processing is the same, and once the pre-processing is completed, the input is provided to a single ARX-box. The output is fed into the same hardware again to compute the required ARX-box output. To compute the ARX-box output during each round, this architecture requires six and eight clock cycles for ESCH256 and ESCH384, respectively. One clock cycle is required to compute the diffusion layer. Computing the hash by employing a serialized architecture has certain advantages. Only a single ARX-box is required. The length of the registers that save the intermediate state is also reduced, and as a result, hardware area can be minimized. Fig. 10 (b) shows a single permutation round based on the serialized approach.

3) Hybrid approach: Carrying this work forward, a hybrid approach is proposed, where there is the possibility of processing a certain number of ARX-boxes in one clock cycle. This version of the hash implementation refers to such a technique. From the performance point of view, this approach is balanced between the two techniques discussed above, where hardware consumption is slightly increased to reduce the clock cycles. The hash computation is the same. For one permutation round, three clock cycles are required for the ARX-boxes and one clock cycle for the diffusion layer with ESCH256. ESCH384 requires one more clock cycle for the permutation round. Fig. 10 (c) shows a single permutation round based on the hybrid approach.

Table IV provides the implementation results for the three versions of ESCH256 and ESCH384 on the FPGA platform. Area consumption is linearly related to the number of ARX-boxes employed in one clock cycle. The first version consumes the most hardware area, where one round of permutations is computed for all the parallelized ARX-boxes. However, this architecture consumes the fewest clock cycles because the maximum number of operations are parallelized. Since the critical path elongates, it engenders a limit on the operating frequency in this case. In the second version, where only one ARX-box is employed with a serialized structure, the critical path is smaller, and the maximum operating frequency is achieved. Since the highest parallelized version consumes the fewest clock cycles, which in turn affects the overall computation time, which is hence the least for this architecture. The longest time is taken by the serialized version because maximum clock cycles are consumed for this architecture. TP and TP/A show the same trend, with the most parallel version generating the highest TP and TP/A.

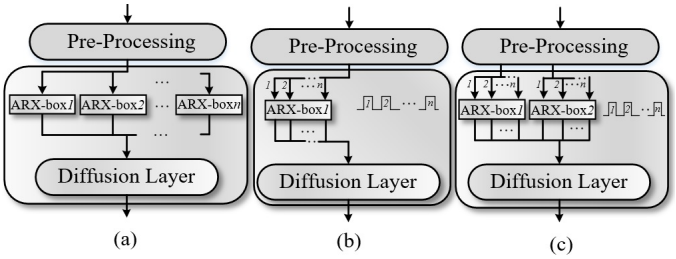


Fig. 10. Single permutation round in SPARKLE for (a) the parallelized ARX-box, (b) the serialized ARX-box, and (c) the hybrid approach.

TABLE IV
IMPLEMENTATION RESULTS FOR SPARKLE HASH.

Technique	Area (LUTs)	Freq. (MHz)	Latency (cc)	Time (μs)	TP (Mbps)	TP/A (Mbps/LUT)
ESCH256 (Virtex-7)						
Parallelized	2015	149.63	31	0.207	1236.71	0.613
Serialized	1530	186.70	206	1.103	232.09	0.151
Hybrid	1656	184.63	132	0.714	358.54	0.216
ESCH384 (Virtex-7)						
Parallelized	2665	144.42	43	0.297	861.95	0.323
Serialized	1762	188.60	367	1.945	131.61	0.074
Hybrid	1931	184.19	223	1.210	211.57	0.109

D. XOODYAK

Previous implementations of XOODYAK in the literature are limited to executing one round per clock cycle. In this paper, the XOODYAK architecture was designed to execute multiple permutations per clock cycle.

The mode of operation on top of the XOODOO permutation is called a cyclist. For the hash computation, the cyclist is initialized with no key. In unkeyed mode, the cyclist is equivalent to the sponge construction, where an input message of arbitrary length is absorbed, and a specific length of output is then squeezed. For the hash computation in XOODYAK, the rate, r, is 128-bit while the capacity, c, is 256-bit. The message is parsed into 128-bit groups, and then permutation is applied. The state represents the digest of all the states absorbed this far, and the digest is referred to as *history*. The 256-bit of the hash is also extracted the same way, in two steps. A cyclist uses up to two bytes for domain separation.

For the hardware implementation of XOODOO, a single hash round is computed first. Since the basic hardware architecture is simple, i.e., one permutation round for XOODOO with very little space for optimizations, round-based implementations are

therefore the best choice for hash computation. The basic architecture for a single permutation round implemented on the hardware is shown in Fig. 11.

The hardware architecture for XOODOO consists of five steps. The first step (the mixing layer) is a simple XOR operation and shifting. These operations can be realized by employing XOR gates. The shifting operation does not consume any hardware because it can be achieved through the right selection of bits. The second and third steps can be parallelized because the inputs are not dependent upon each other, and hence, parallelization can be achieved. Again, plane shifting does not involve any hardware consumption. The constant addition can be realized through XOR operations. The operations involved in a non-linear layer require inversion plus AND operations, which are not expensive in hardware. The final step is plane shifting, which does not require any actual hardware. The final two steps are computed in series as they wait for the updated plane values. This constitutes one permutation round. Several permutation rounds are grouped together to compute the permutation.

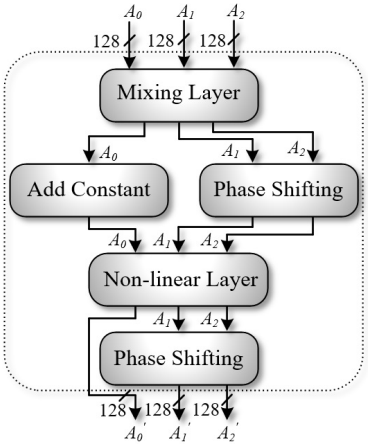


Fig. 11. A single permutation round for XOODOO.

For the hash, we implemented a round-based approach. We chose one, two, and three rounds of permutations to be executed in one clock cycle. As the number of rounds in one clock cycle increases, clock cycle utilization decreases. The clock cycles for each technique can be estimated as $(36/n)+2$, where n is the number of rounds executed in one clock cycle.

The implementation results are shown in Table V. The results show that with the increase in the number of rounds executed per clock cycle, hardware consumption also increases linearly. On the other hand, doing so increases the critical path length, therefore decreasing the operating frequency. Clock cycle consumption decreases when more permutation rounds are packed into each clock cycle. TP also increases with the increasing number of rounds because of the reduced time needed for computation. On the other hand, TP/A decreases as the area increases, with more rounds executed per clock cycle.

TABLE V
IMPLEMENTATION RESULTS FOR THE XOODYAK HASH.

Technique	Area (LUTs)	Freq. (MHz)	Latency (cc)	Time (μs)	TP (Mbps)	TP/A (Mbps/LUT)
Virtex-7						
1-round	678	325.41	38	0.116	2206.89	3.255
2-rounds	1192	241.83	20	0.082	3121.95	2.619
3-rounds	1701	180.44	14	0.077	3327.67	1.954

IV. ANALYSIS AND COMPARISON

A. Analysis of the proposed architectures

This section first provides an analysis of the hash functions implemented in the previous sections. The insights into the architectures that make them stand apart from each other are discussed. Furthermore, a comparison of the proposed architectures with state-of-the-art implementations is presented.

1) Internal architecture: PHOTON-Beetle is based on AES-like architecture [28]. For PHOTON-Beetle, the diffusion matrix is very lightweight, and shows good potential for a highly parallelized implementation. Each element of the matrix multiplication can be computed separately, independent of the others, providing a high level of parallelization. In addition, for the PHOTON-Beetle-Hash, the first 128-bit input message block followed by the 32-bit blocks, requires more permutation calls than the other corresponding hash functions, and as a result, generates lower TP. PHOTON is also part of the ISO-IEC:29192-5 standard, which deals specifically with lightweight cryptography.

For Ascon, unlike the other algorithms, the first input for the permutation round is defined by a constant initialization vector that specifies the algorithm parameters. After that, 12 rounds of permutation are applied to it. From the implementation point of view, this permutation call can be saved by saving the values of the permutation in BRAM, and then fetching them before the permutation starts. However, we do not save the values in BRAM, and compute them before processing the input message. The reason is that the proposed architecture for Ascon re-utilizes the hardware core and does not have any area overhead when processing IV. There is only an overhead of a few clock cycles, depending on the Ascon implementation variant. Therefore, our strategy refrains from saving data to BRAM and then fetching it again. The authors in [21] demonstrated a serialized version of the Ascon S-box making it ideal for area-constrained applications.

SPARKLE processes input messages that are not directly fed into the permutation rounds, but are expanded through the application of a linear function. The introduction of the linear layer to pre-process the message blocks exerts overhead on area consumption. As a result, SPARKLE has the highest area consumption, compared to the other hash functions. Comparing single-round implementations of all the hash functions, SPARKLE used the most LUTs.

XOODYAK has a structure similar to Gimli and follows a sponge-based architecture. The main property of XOODYAK is that it characterizes the simplest round function where it only employs XOR, AND, and inversion, which can easily be accomplished through logic gates. Each of the five steps in the

round function has to wait for the output from the previous step, pushing it more towards a serialized architecture. This is also admitted by the authors, in that XOODYAK permutations are inherently serial at the construction level.

2) TP and TP/A ratio: The important performance metrics are TP and TP/A ratio because they incorporate all the performance metrics together. TP can be realized as the number of bits that any hardware architecture can process in a specific time. TP was calculated in mega-bits per second during the evaluation process. TP/A also incorporates hardware area consumption, and is the ratio of TP to LUTs utilized by any architecture. TP and TP/A for the hash functions based on one round of permutation per clock cycle are shown in Fig. 12. Because TP is the number of bits processed in a certain time, it is therefore dependent upon the size of the input chunk processed in one clock cycle. Both XOODYAK and SPARKLE have a 128-bit block size, but XOODYAK has a simpler architecture, so it provided the highest TP. Ascon achieved the lowest TP, because it lacks parallelization in a single round and processes the input in 64-bit chunks. TP/A ratio was also the highest for XOODYAK because of the simpler architecture that consumes less area. SPARKLE, on the other hand, had the lowest TP/A ratio because it incurs overhead in the hardware by processing the input messages in a transformed state. Ascon and PHOTON-Beetle are similar in TP/A ratio.

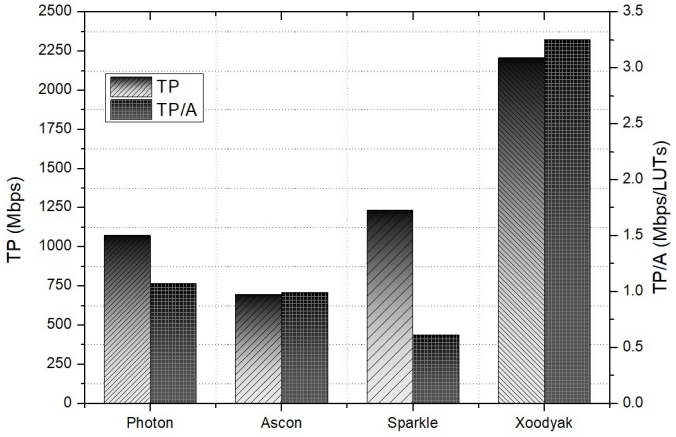


Fig. 12. TP and TP/A ratio for the hash functions.

B. Comparison

Hash functions that are candidates in the ongoing standardization process have few FPGA implementations in the literature. Table VI provides the comparison of the proposed architectures with other implementation strategies. PHOTON-Beetle was implemented in [11] and [12], where the authors performed bench-marking for the hash functions during Round-2 of the NIST competition. Both techniques apply a single permutation function round in one clock cycle. Our proposed parallel architecture for the same platform also applies a single permutation round in one clock cycle. But the novel optimized S-box, along with the matrix multiplication architecture, reduces the area consumption to half. Similarly, the serial architecture helps with a further area reduction of 40%. The serial implementation of PHOTON-Beetle has lower TP compared to the implementations presented in [11] and [12]. This is because the serial implementations are designed to achieve the smallest hardware area, trading off the TP performance. The area reduction of almost $3\times$ can be achieved at the cost of TP reduction to almost 42% compared to [11] and [12]. The overall TP/A ratio for our serial implementation is still higher, demonstrating that the area reduction is significant. This implementation strategy can be highly beneficial to applications that are area constrained.

TABLE VI
COMPARISON WITH THE STATE-OF-THE-ART IMPLEMENTATIONS.

Technique	Platform	Area (LUTs)	Freq. (MHz)	TP (Mbps)	TP/A ratio (Mbps/LUT)
PHOTON-Beetle					
[11]	Artix-7	2065	178	227.8	0.110
[12]	Artix-7	2100	-	230	0.109
Parallel	Artix-7	1021	210.79	**888.88**	**0.870**
Serial	Artix-7	**609**	215.19	95.12	0.156
Ascon					
[11]	Artix-7	1723	219	987	0.572
[12]	Artix-7	1700	-	980	0.576
[19]	Artix-7	2181	242	1032	0.473
[20]	Kintex-7	**629**	429	-	-
1-round	Artix-7	770	260.75	668.40	**0.868**
6-rounds	Artix-7	3728	64.07	792.56	0.208
1-round	Kintex-7	702	268.24	646.46	0.846
6-rounds	Kintex-7	3728	100.36	**1354.49**	0.356
SPARKLE					
1-round	Virtex-7	2015	149.62	**1236.71**	**0.613**
1 ARX-box	Virtex-7	**1530**	186.70	232.09	0.151
XOODYAK					
[11]	Artix-7	2040	170	920	0.450
[12]	Artix-7	2100	-	1180	0.561
1-round	Artix-7	**750**	239.92	1620.25	**2.160**
3-rounds	Artix-7	1728	146.79	**2694.73**	1.559

Ascon implementations in [11] and [12] provided round-based implementations. The proposed bit-sliced S-box architecture needs up to $2.2\times$ fewer LUTs for the same platform. Similarly, our proposed architecture generates a 35% higher TP/A ratio (1-round). The authors in [19] introduced round-based implementations for Ascon. In addition, there are multiplexers that select the appropriate value for the data during each iteration. Compared to our proposed architecture, we use almost $2.8\times$ fewer LUTs. The implementations in [19] engendered higher TP at the cost of more area, and therefore, we achieved almost double the TP/A ratio. Ascon with a modified version of the S-box was presented in [20], which was implemented on the Kintex platform. There is no direct comparison possible in this case, because it implements a modified version of the S-box. This work also presents several trade-off strategies during implementations. For Ascon implementations (1-round), the TP is lower compared to [11], [12], and [19]. However, the area reduction of $2.2-2.8\times$ can be achieved by TP reduction of 68%. The TP/A ratio of 1-round Ascon is still higher compared to [11], [12], and [19], demonstrating that the area reduction is notable.

The first SPARKLE implementations for the FPGA platform have been presented in this paper, and therefore, there is no candidate for comparison. We proposed XOODYAK implemen-

tations where a certain number of rounds are executed in a clock cycle. XOODYAK has a serialized architecture, but our proposed architecture achieves the best reduced critical path by executing some components in parallel. This directly affects the operating frequency. The operating time is reduced as a result, and a higher TP can be achieved. The achieved TP is almost double that from state-of-the-art implementations. For XOODYAK implementations, as the number of permutations executed in each clock cycle increases, the TP is also increased. However, the TP escalation comes at the cost of more area consumption. The area has been traded-off for better results in TP for the XOODYAK implementations. SPARKLE implementations exhibit similar performance when the underlying ARX-box is assessed by serial, parallel and hybrid approaches.

V. APPLICATIONS

Hash functions are an integral part of the blockchain. Blockchain is based on the concept of consensus algorithms, i.e., proof-of-work [29]. Proof-of-work refers to the fact that one must be able to prove a certain amount of computational resources was put into solving a problem. By employing a cryptographic hash function and then requesting an input that yields a hash result with a specific pattern is a popular choice for proof-of-work. The more confined the hash output is, the harder it is to solve the problem, thus demanding more work. In the current era, merging IoT technology with blockchain is a potential research direction. In order to use the computationally expensive blockchain technology for the IoT environment, the consensus algorithm must be modified. Using lightweight hash functions can reduce the computation overhead for consensus algorithms, making blockchain compatible with IoT environments. In addition, modified versions of the consensus algorithms that are based on lightweight hash functions have been presented [30].

VI. CONCLUSION AND FUTURE WORK

To achieve data integrity in an IoT environment, hash functions are of prime importance. This work presented implementations and optimizations for lightweight hash functions that are finalists in the NIST standardization competition (PHOTON-Beetle, Ascon, SPARKLE, and XOODYAK). Implementation results showed that PHOTON-Beetle can save 50% of the hardware area, whereas the serialized version of the algorithm can further reduce area consumption by 40%. For the Ascon implementations, an increase in TP and TP/A ratio was achieved. The simplified implementations for XOODYAK achieved almost double the TP compared to state-of-the-art implementations. We also provided the first FPGA implementations for the SPARKLE hash function. In addition, analysis based on internal architectures was conducted on the hash functions.

The proposed work can contribute to protecting IoT communications through an integrity check that can be helpful in many IoT applications. Apart from that, the hardware implementations are also important for reviewers in choosing the final candidate for the hash competition.

Carrying this work forward, the lightweight hash functions and their corresponding hardware implementations can be used to design lightweight hash-based signatures for authenticating the IoT devices. Efficient hash-based message authentication codes can also be developed to verify the legitimacy of data sent through the IoT network [31]. Both the lighter version of signatures and message authentication codes are essential in securing the IoT applications.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] M. M. Sravani and S. A. Durai, "On efficiency enhancement of SHA-3 for FPGA-based multimodal biometric authentication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 4, pp. 488–501, 2022.

[5] M. J. Dworkin *et al.*, "SHA-3 standard: Permutation-based hash and extendable-output functions," 2015.

[6] 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.

[7] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1. 2," *Submission to the CAESAR Competition*, vol. 5, no. 6, p. 7, 2016.

[8] J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer, "Xoodyak, a lightweight cryptographic scheme," 2020.

[9] C. Beierle, A. Biryukov, L. C. dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, Q. Wang, and A. Biryukov, "Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family," *NIST round*, vol. 2, 2019.

[10] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda, "PHOTON-beetle authenticated encryption and hash family," *NIST Lightweight Compet. Round*, vol. 1, p. 115, 2019.

[11] K. Mohajerani, R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "FPGA benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process: methodology, metrics, tools, and results," *Cryptology ePrint Archive*, 2020.

[12] K. Mohajerani, R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. K. Gaj, "Hardware benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 164–169.

[13] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of things (IoT) communication protocols," in *2017 8th International conference on information technology (ICIT)*. IEEE, 2017, pp. 685–690.

[14] B.-L. Tan, K.-M. Mok, J.-J. Chang, W.-K. Lee, and S. O. Hwang, "Risc32-lp: Low-power fpga-based IoT sensor nodes with energy reduction program analyzer," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4214–4228, 2021.

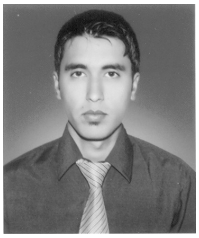
[15] S. Hadayeghparast, S. Bayat-Sarmadi, and S. Ebrahimi, "High-speed post-quantum cryptoprocessor based on RISC-V architecture for IoT," *IEEE Internet of Things Journal*, 2022.

[16] 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.

[17] A. Abduvaliev, S. Lee, and Y.-K. Lee, "Simple hash based message authentication scheme for wireless sensor networks," in *2009 9th International Symposium on Communications and Information Technology*. IEEE, 2009, pp. 982–986.

[18] S. Abed, R. Jaffal, B. J. Mohd, and M. Al-Shayeji, "An analysis and evaluation of lightweight hash functions for blockchain-based IoT devices," *Cluster Computing*, vol. 24, no. 4, pp. 3065–3084, 2021.

[19] B. Rezvani, F. Coleman, S. Sachin, and W. Diehl, "Hardware implementations of NIST lightweight cryptographic candidates: A first look," *Cryptology ePrint Archive*, 2019.
[20] K. Mandal, D. Saha, S. Sarkar, and Y. Todo, "Sycon: a new milestone in designing ASCON-like permutations," *Journal of Cryptographic Engineering*, pp. 1–23, 2021.
[21] S. Khan, W.-K. Lee, and S. O. Hwang, "Scalable and efficient hardware architectures for authenticated encryption in IoT applications," *IEEE Internet of Things Journal*, vol. 8, no. 14, pp. 11 260–11 275, 2021.
[22] "Lightweight cryptography," NIST, 2018. [Online]. Available: https://csrc.nist.gov/Projects/Lightweight-Cryptography
[23] J. Guo, T. Peyrin, and A. Poschmann, "The PHOTON family of lightweight hash functions," in *Annual cryptology conference*. Springer, 2011, pp. 222–239.
[24] E. Miles and E. Viola, "Substitution-permutation networks, pseudorandom functions, and natural proofs," *Journal of the ACM (JACM)*, vol. 62, no. 6, pp. 1–29, 2015.
[25] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1. 2: Lightweight authenticated encryption and hashing," *Journal of Cryptology*, vol. 34, no. 3, pp. 1–42, 2021.
[26] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge functions," in *ECRYPT hash workshop*, vol. 2007, no. 9. Citeseer, 2007.
[27] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo *et al.*, "Gimli: a cross-platform permutation," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 299–320.
[28] J.-S. Ng, J. Chen, K.-S. Chong, J. S. Chang, and B.-H. Gwee, "A highly secure FPGA-based dual-hiding asynchronous-logic AES accelerator against side-channel attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2022.
[29] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.
[30] S. Khan, W.-K. Lee, and S. O. Hwang, "Aechain: a lightweight blockchain for IoT applications," *IEEE Consumer Electronics Magazine*, vol. 11, no. 2, pp. 64–76, 2021.
[31] G. Saldamli, L. Ertaul, and A. Shankaralingappa, "Analysis of lightweight message authentication codes for IoT environments," in *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 2019, pp. 235–240.



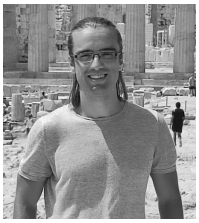
Safiullah Khan (Student Member, IEEE) received his B.Sc. in electronic engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2013, and an M.Sc. in electrical engineering from COMSATS University Islamabad, Abbottabad campus, Pakistan, in 2017. He is currently pursuing his Ph.D. in computer engineering from Gachon University, Seongnam, South Korea.

He worked in R&D department of National Radio and Telecommunication Corporation, Haripur, Pakistan, for two years. His research interests include efficient hardware implementations of cryptographic protocols, blockchain, and network security. He is the Chair of IEEE Student Branch at Gachon University, South Korea.

Wai-Kong Lee (Member, IEEE) received a B.Eng. and an M.Sc. from Multimedia University in 2006 and 2009, respectively. He received a 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. Prior to entering academia, he worked for several multi-national companies, including Agilent Technologies (Malaysia) as an R&D engineer. His research interests are cryptography, numerical algorithms, GPU computing, the Internet of Things, and energy harvesting. He is currently a postdoctoral researcher at Gachon University, South Korea.

Angshuman Karmakar received a B.E. in computer science and engineering from Jadavpur University, Kolkata, an M.Tech. in computer science and engineering from the Indian Institute of Technology, Kharagpur, and a Ph.D. from Katholieke Universiteit Leuven (KU Leuven), Belgium, for his dissertation titled "Design and Implementation Aspects of Post-Quantum Cryptography." He is one of the primary designers of the post-quantum Saber KEM scheme which is one of the finalists in the NIST's post-quantum standardization procedure. He received FWO Post-Doctoral Fellowship with the COSIC Research Group, KU Leuven. He is currently working as an assistant professor at department of Computer science and engineering at the Indian Institute of Technology, Kanpur, India. His research interests span different aspects of lattice-based post-quantum cryptography and computation on encrypted data.



Jose Maria Bermudo Mera received a B.Eng. and an M.Sc. in telecommunications engineering from the Technical University of Madrid, Spain, and a Ph.D. in electrical engineering from the Katholieke Universiteit Leuven, Belgium, for his thesis "Implementation Aspects of Lattice-Based Cryptography." He is currently cryptography hardware design engineer at PQShield Ltd. He is also a Team Member of the post-quantum Saber key-encapsulation mechanism scheme, which is one of the finalists in the NIST's post-quantum standardization procedure. His research interests include implementation of cryptography on software and hardware platforms and physical security.

Abdul Majeed received a B.S. in Information Technology from the UIIT, PMAS-UAAR, Rawalpindi, Pakistan, in 2013, an M.S. in Information Security from the COMSATS University, Islamabad, Pakistan, in 2016, and a Ph.D. in Computer Information Systems and Networks from the Korea Aerospace University, in 2021. He worked as a Security Analyst with Trillium Information Security Systems (TISS), Rawalpindi, Pakistan, from 2015 to 2016. He is currently an Assistant Professor with the Department of Computer Engineering, Gachon University, South Korea. His research interests include privacy-preserving data publishing, statistical disclosure control, privacy-aware analytics, and machine learning.

Seong Oun Hwang (Senior Member, IEEE) received a B.S. in mathematics from Seoul National University in 1993, an M.S. in information and communications engineering from the Pohang University of Science and Technology in 1998, and a Ph.D. in computer science from the Korea Advanced Institute of Science and Technology in 2004, South Korea.

He worked as a Software Engineer with LG-CNS Systems, Inc., from 1994 to 1996. He worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He was as a Professor with 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 an Editor of *ETRI Journal*.