A Flexible Gimli Hardware Implementation in FPGA and Its Application to RFID Authentication Protocols

SAFIULLAH KHAN1, WAI-KONG LEE2, (Member, IEEE), AND SEONG OUN HWANG2, (Senior Member, IEEE)

1Department of IT Convergence Engineering, Gachon University, Seongnam 13120, Republic of Korea
2Department of Computer Engineering, Gachon University, Seongnam 13120, Republic of Korea

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

This work was supported in part by the National Research Foundation of Korea (NRF) Grant by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant 2020R1A2B5B01002145, and in part by the Gachon University Research Fund of 2020 under Grant GCU-202004370001.

ABSTRACT Radio Frequency Identification (RFID) systems have bestowed numerous conveniences in a multitude of applications, but the underlying wireless communications architecture makes it vulnerable to several security threats. To mitigate these issues, various authentication protocols have been proposed. The literature accommodates comprehensive proposals and analysis of authentication protocols, but not many of them provide hardware implementations. In addition, there is diverse demand for hardware area and throughput (TP) requirements from RFID system components (tags, readers, database servers), which demand a flexible implementation strategy. This paper proposes a flexible implementation strategy for the lightweight authenticated encryption (AE) and hash function called Gimli, and applies it to a state-of-the-art authentication protocol. This allows the authentication protocol to be implemented efficiently, wherein the area and TP can be adjusted flexibly according to the RFID system requirements. This implementation strategy is generic; it can be used to implement any other AE and hash functions. This strategy can also be applied to other authentication protocols that heavily use AE and hash functions. To provide a detailed analysis, the hardware optimization techniques in each component of the RFID system for a state-of-the-art authentication protocol are analyzed. When implemented with the most area-optimized versions, we achieve TP of 740 Mbps and 420 Mbps for Gimli hash and Gimli AE, respectively, and for throughput-oriented implementation, the results are 3.08 Gbps and 1.43 Gbps, respectively. This shows that the proposed implementation strategies allow us to implement authentication protocols in a flexible manner to meet the differing requirements in TP and area for RFID applications.

INDEX TERMS Authenticated encryption, FPGA, Gimli, lightweight cryptography, RFID.

I. INTRODUCTION

The Internet of Things (IoT) is a network of devices linked via the Internet such that the devices can be monitored and managed remotely [1]. Common applications involve remote data collection and transmission, such as cyber physical systems, Radio Frequency Identification (RFID), smart cities, sensor networks, smart vehicles, etc. One of the most commonly used technologies in building IoT sensor nodes is RFID [2]. With the increasing acceptance of IoT technology by the public, the horizon for RFID utilization has gradually expanded.

RFID systems have been employed in a wide range of applications, ranging from object tracking to health care [3]. RFID systems consist of tags, readers, and a database server. The tag is an electronic device that can exchange data with the readers in the RFID system [4]. Generally, a tag has a small amount of storage and little computation capability [5]. RFID tags can be either passive or active. The RFID reader is the core component of the RFID system, having moderate storage and computation capabilities. The reader can activate the tags and can send and receive messages to and from the tags. In addition, it is responsible for availability of tag information at the application level [6]. The database server has a considerable storage capability and a high computation...
speed. The tag data are mainly stored and processed in the database server [5]. The overall architecture for an RFID system is shown in Figure 1.

![Diagram of RFID system architecture](image)

**FIGURE 1.** The radio frequency identification system architecture.

Since the tag and the reader communicate wirelessly in the RFID system, such communications are open to security threats between tag and reader [7], [8]. An attacker can eavesdrop on the communication channel to launch various attacks. To overcome such security threats, a mutual authentication protocol can be used to verify the source’s identity and authenticated encryption (AE) can be used to protect the confidentiality and data authentication.

RFID tags usually do not have a software-programmable processor, which requires cryptographic solutions to be realized only via hardware implementations. Furthermore, most of these devices rely on scanty power sources and may even need to use energy harvesting [9], power optimization techniques [10], and novel transmission technologies [11] to operate. Therefore, hardware-based techniques tend to provide a better cryptographic solution in constrained environments. Such solutions can balance several important parameters that include the hardware area, computation time, and power consumption [12]. Thus, for the IoT platform, the ideal case is to provide hardware-based security [13], but that requires a systematic application of optimization strategies for secure architectures. The hardware-based solutions for security can thus provide intrinsic security improvements over software solutions [14]. The FPGA has remained a premier choice for developing hardware systems. A study by Good and Benaissa [15] showed that it is no longer true that FPGAs are only used for prototyping. The use of FPGAs can bring additional advantages, such as reduced time to market and the ability to update the design conveniently. Additionally, Neil et al. pointed out [11] that evolving security protocols demand the use of devices that have reconfigurable capabilities.

The FPGAs are also the primary choice for designing RFID readers [16]. The major components in an FPGA reader involve baseband tasks and channelization processing that can be realized on an FPGA. In addition, providing security by employing authentication protocols can be integrated in the same chip. The authentication protocols can be implemented in FPGA using Verilog hardware description language, which are verified for correctness by simulating their test benches. The results obtained from the test bench is then compared with the already provided vectors for confirmation. All these utilities makes the FPGA an attractive choice for RFID applications.

The RFID authentication protocols can be based on either symmetric or asymmetric cryptographic schemes. Each of these schemes has its own advantages and drawbacks. For the symmetric protocols, the secret keys for tags should be stored inside the reader, or shared through secret communication. If the secret key gets compromised, the encrypted messages can be decrypted. This problem can be solved by adopting the asymmetric protocols, in which there is no need to share the secret key. However, the integration of asymmetric protocols with the RFID tags is a challenging task. Tight constraints with respect to lower power consumptions and small hardware footprint demand for highly optimized implementations in hardware. Naeem et al. [17] has presented an authentication protocol based on elliptic curve cryptography (ECC), while the information about hardware resource utilization is not provided. Symmetric cryptography is a practical solution for constrained devices due to its lightweight operations, and it is usually hardware friendly. Optimized implementations of such protocols by considering the constrained RFID systems can play important roles in the security for many smart applications [18].

This article presents a hardware implementation of Gimli AE as well as hash function [19] that share the same Gimli permutation hardware. The Gimli cipher was designed with the objective of being the single primitive that performs well on every platform. Therefore, that idea has compelled us to execute the implementations of Gimli for the FPGA platform. The proposed Gimli hardware implementations are then applied to a recent authentication protocol [32] to demonstrate its practical performance in RFID systems.

In a nutshell, for the RFID system, requirements for the reader (baseband tasks, channelization processing, AE, and hash implementation that can be integrated into an FPGA) and for the tags (cryptographic solutions can be realized only through hardware implementations) make FPGAs the finest choice for implementing the protocol in hardware. Since Gimli is the best cross-platform permutation as claimed by its authors and FPGAs are the premier choice in hardware, this
work proves its significance when implementing an authentication protocol using Gimli on an FPGA platform.

A. CONTRIBUTIONS

In RFID systems, the tags are constrained in area, which may require area-optimized implementation techniques. On the other hand, the reader needs to process massive amounts of communication with different tags, but it can also afford more hardware resources; a high throughput (TP) implementation strategy is suitable in this case. Due to this discrepancy in resources, optimized implementation techniques can be very different. This paper proposes a flexible strategy for efficiently implementing an authentication protocol in RFID applications, in which the contributions are summarized as follows:

1) The most computationally expensive operation in many authentication protocols is either AE or the hash function. Prior work that optimizes implementation of these cryptographic primitives only focused on one of the design metrics (a small hardware area, high TP or high throughput-to-area (T/A) ratio). This paper proposes a flexible FPGA implementation of a Gimli permutation that can be used for either AE or the hash function. The area consumption and TP can be varied flexibly by configuring the number of rounds executed in one clock cycle. The proposed area-optimized implementations of Gimli AE and the hash function can generate TP of 420 Mbps and 740 Mbps, respectively, while TP-optimized implementations can generate 3.08 Gbps and 1.43 Gbps for Gimli hash and Gimli AE, respectively.

2) The proposed flexible implementation strategy is applied to a state-of-the-art RFID authentication protocol [32] to provide a practical implication on the effectiveness of the proposed implementation strategy. The authentication protocols found in the literature provide a detailed theoretical analysis, but they usually do not provide detailed considerations of practical implementations. This is an important aspect when the authentication protocols are implemented on hardware-constrained platforms like RFID systems, where the different components involved require different implementation techniques. This article tries to fill the gap between theoretically analyzed authentication protocols and practical hardware implementations by applying the proposed implementation strategy and providing concrete hardware-resource utilization and computation times for FPGA platforms.

The rest of the paper is organized as follows. Section II discusses related work. The details of the Gimli AE scheme and the corresponding hardware implementations are presented in Section III. Implementation results and discussions are provided in Section IV. Section V provides details on application of the Gimli AE scheme to RFID authentication. Comparison of Gimli with other implementations are presented in Section VI. Finally, the paper concludes in Section VII.

II. RELATED WORK

Block ciphers or AE schemes are commonly used in protecting IoT data. Each of them has its own advantages and disadvantages. Block ciphers require little hardware area, but they can only protect data confidentiality. To allow more security features like authenticity and integrity, one needs to use a hash function, which requires a separate hardware module. On the other hand, AE schemes can offer confidentiality and message authenticity as well as integrity check. Compared to using a block cipher and hash function together, AE schemes tend to have less hardware area. This section discusses some of the well-known block ciphers and AE schemes.

Efficient implementation of block ciphers is a popular research topic. One notable example is the AES implementation presented by Wong et al. [20]. The S-box architecture in an AES cipher is realized through hard-coded LUT implementation, which is a pure combinatorial S-box. The architecture was implemented in Altera’s Cyclone IV E FPGA. The LUT-based design requires 2.14k LUTs for hardware implementation.

Marchand et al. [21] provided another interesting hardware implementation of block ciphers (Klein, Led, Lilliput, and Ktantan). Hardware implementations for these block ciphers follow two strategies: full-width and serial. The full-width implementations are aimed at increased computational efficiency, whereas the serialized implementations target area optimization. This work provides insight into the hardware area and computational efficiency of the well-known block ciphers. Depending on the implementation results, their work concludes that the best algorithm for one application is not necessarily the best for all applications.

Ascon is an AE scheme based on permutations. Khan et al. [22] discussed hardware implementation of Ascon employing different strategies (unrolled, round-based, and serialized). Implementing one permutation round in one clock cycle requires 2.06k LUTs in a Spartan-6 device. Similarly, Ascon, by implementing one permutation per clock cycle, consumes 1.64k LUTs [23]. A bit-sliced technique has been used for the S-box implementation, and implementations have been done in the Spartan-6. The bit-slice technique can save hardware resources, but sacrifices throughput and TP/A ratio.

Some area-optimized implementations of AE schemes are also found in the literature. Mancillas et al. [24] implemented five AE schemes from the second round of the NIST lightweight cryptography competition. The implementation was completed according to the hardware API for lightweight cryptography, and results were generated for the Xilinx Artix-7. Their goal was to achieve a reduction in hardware utilization below 2k LUTs, which is accomplished in their work.

Avik et al. [25] presented the hardware implementation of the Beetle Family lightweight AE scheme. In the current state of the art literature, Beetle achieved the smallest footprint in hardware implementations (600 LUTs) while maintaining 64-bit security. This is significantly smaller than known...
lightweight block ciphers, but only supports a 64-bit security level, which is only appropriate for lightweight applications with low security requirements. Another version of the cipher (to achieve high security) was designed that can provide 121-bit security.

From the review of the literature, we can infer that using an AE scheme instead of a block cipher and hash function can be more advantageous. Furthermore, the implementation of Gimli AE and hash with flexible configuration was not found in literature. The results section indicates higher TP and a better TP/A ratio are obtained with Gimli, making it ideal for applications like RFID authentication, as demonstrated in this paper.

III. GIMLI OVERVIEW AND HARDWARE ARCHITECTURES
A. HARDWARE ARCHITECTURES FOR GIMLI PERMUTATION

Gimli is a 384-bit lightweight permutation function for designing AE and hash functions [26]. In general, lightweight AE schemes like Gimli are a favorable choice for protecting data in resource-constrained systems. The permutation function in Gimli acts as the fundamental task for both AE and hashing, which is similar to other popular lightweight AE schemes (e.g. Ascon [27], Spook [28], and Skinny [29]). The Gimli permutation function comprises 24 rounds, where the main operations in each round are the logic functions, shift/rotate, or swap. The computation time for the permutation increases linearly as the number of rounds is increased.

The permutation function applies a sequence of rounds to the state, which is a 384-bit matrix as shown in Figure 2 (a). The state is represented as 32 words in a $3 \times 4$ matrix. Rows and columns are represented by $i$ and $j$, respectively, consisting of a sequence of 128 bits for each row and 96 bits for each column.

The permutation function is divided into three main operations that are either performed or omitted, depending upon the round number being executed. These operations include a non-linear layer, a linear layer, and addition with a constant.

The non-linear layer has three sub-operations: rotation of words in the first and second rows with a fixed constant, a non-linear $T$-function, and swapping of first and third rows. This constitutes the SP-box of the Gimli. This operation is mandatory, and is performed each iteration. Lines 4 to 10 of Algorithm 1 refer to the operations involved in the SP-layer.

The next operation is the linear layer that consists of two swap operations: the Small-swap and the Big-swap. The swap operations are performed after every four rounds, where the Small-swap starts from the first round and the Big-swap starts from the third round. Figure 2 (b) shows the swap operations on the state. Finally, for every round that is a multiple of 4, we XOR the constant with the state word, 00. All of these steps when put together give the Gimli permutation function, as shown in Algorithm 1.

Algorithm 1 The Gimli Permutation [19]

```
1: Input: $s = (s_{i,j}) \in \mathcal{W}^{3 \times 4}$
2: Output: $\text{GIMLI}(s) = (s_{i,j}) \in \mathcal{W}^{3 \times 4}$
3: for $r$ from 24 downto 1 inclusive do
4:   for $j$ from 0 to 3 inclusive do
5:     $x \leftarrow s_{0,j} \ll 24$ // non-linear layer
6:     $y \leftarrow s_{1,j} \ll 9$ // SP-box
7:     $z \leftarrow s_{2,j}$
8:     $s_{2,j} \leftarrow x \oplus (z \ll 1) \oplus ((y \land z) \ll 2)$
9:     $s_{1,j} \leftarrow y \oplus x \oplus ((x \land y) \ll 1)$
10:    $s_{0,j} \leftarrow z \oplus y \oplus ((x \land y) \ll 3)$
11:   end for
12: if $r \mod 4 = 0$ then
13:   // Small-Swap
14:   $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,1}, s_{0,0}, s_{0,3}, s_{0,2}$
15: else if $r \mod 4 = 2$ then
16:   // Big-Swap
17:   $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,2}, s_{0,3}, s_{0,0}, s_{0,1}$
18: end if
19: if $r \mod 4 = 0$ then
20:   $s_{0,0} = s_{0,0} \oplus 0x9e377900 \oplus r$ // Add constant
21: end if
22: end for
23: return $(s_{i,j})$
```

The design-space of the Gimli permutation function has been explored fully to analyze the trade-off between several important performance metrics, such as computation time, area, throughput, and energy and power utilization. The optimized Gimli hash and AE architecture is heavily dependent upon efficient implementations for Gimli permutation.

The proposed hardware architectures for the Gimli permutation aim to execute multiple rounds in one clock cycle, with the goal being to deliver best TP/A ratios. The technique most frequently employed is hardware re-utilization. This is possible because of the inherent structure of Gimli corresponding to its rounds.

![Figure 2. Gimli state and swap operations.](image)
For the hardware implementations, \( s \) rounds are performed in one clock cycle, where \( s = \{1, 4, 8, 12, 24\} \). Performing more rounds in one clock cycle (i.e., a larger \( s \)) reduces the overall execution time but requires a larger hardware area. Execution time can be estimated as \( cc = 24/s + 1 \). Referring to Algorithm 1, some steps are omitted in certain rounds, and hence, uniformity is achievable when the step is a multiple of four. Hardware utilization increases linearly when the number of rounds executed per clock cycle increases. Employing this technique makes the proposed architectures flexible. By selecting a different \( s \), the hardware architecture is scalable to accommodate different requirements in TP and hardware area for RFID implementation.

A detailed explanation of the hardware architecture of a single Gimli round is shown in Figure 3. This architecture can be scaled to \( n \) rounds where \( n = \{1, 2, \ldots, 24\} \). For demonstration purposes, Figure 3 shows the state when \( s = 1 \). The round function consists of three layers. The first is the SP-box, in which the three operations are executed in parallel. Out of these three operations, the first and third (rotation of words and swapping of words) do not require hardware, because they only involve memory read/write; this can be easily implemented through routing. The internal structure of the SP-box is shown in Figure 4, which only involves simple logic operations. It can be implemented through two levels of logic operations. After that comes the linear layer (comprising swap operations), which also involves only routing. The final part is addition with the constant, which varies according to the number of rounds being executed. To allow execution of multiple rounds in one clock cycle, this Gimli permutation architecture is duplicated multiple times. Hence, this consumes the major area of the entire Gimli architecture.

B. HARDWARE ARCHITECTURE FOR GIMLI HASH

Gimli hash is based on the sponge mode [30], [31], which is the simplest way of hashing by employing permutation functions. The output for Gimli hash is 256-bit, which is obtained at a 128-bit rate, and offers security at \( 2^{128} \) against a broad class of attacks. In general, the Gimli hash provides a fixed-length, 256-bit output which is obtained as a concatenation of two 128-bit blocks. The overall working of the Gimli hash is provided in Algorithm 2 while algorithms 3 and 4, respectively, provide details of the absorb and squeeze functions employed in the Gimli hash.

Algorithm 2 The Gimli Hash process [19]

1: Input: \( M \in \{0, 1\}^* \)
2: Output: \( \text{GIMLI-HASH}(m) = h \in \{0, 1\}^{256} \)
3: \( s \leftarrow 0 \)
4: \( m_1, \ldots, m_t \leftarrow \text{pad}(M) \)
5: for \( i \) from 1 to \( t \) do
6: if \( i = t \) then
7: \( s_{2,3} \leftarrow s_{2,3} \oplus 0x01000000 \)
8: end if
9: \( s \leftarrow \text{absorb}(s, m_i) \)
10: end for
11: \( h \leftarrow \text{squeeze}(s) \)
12: \( s \leftarrow \text{GIMLI}(s) \)
13: \( h \leftarrow h || \text{squeeze}(s) \)
14: return \( h \)

Algorithm 3 Absorb function [19]

1: Input: \( s = (s_{ij}) \in \mathcal{X}^{3 \times 4}, m \in \mathbb{F}_{256}^{16} \)
2: Output: \( \text{absorb}(s, m) = (s_{ij}) \in \mathcal{X}^{3 \times 4} \)
3: for \( i \) from 0 to 3 do
4: \( s_{0,i} \leftarrow s_{0,i} \oplus \text{touint}32(m_{4i}, \ldots, m_{4(i+1)}) \)
5: end for
6: \( s \leftarrow \text{GIMLI}(s) \)
7: return \( s \)

The hardware architecture of the Gimli hash is built upon the Gimli permutation function. At the beginning of the hash
Algorithm 4 Squeeze function [19]

1: Input: \( s = (s_{i,j}) \in W^{3 \times 4} \)
2: Output: \( \text{squeeze}(s) = h \in \mathbb{F}_{256}^{16} \)
3: \( h \leftarrow \text{tobytes}(s_{0,0}) || \text{tobytes}(s_{0,1}) || \text{tobytes}(s_{0,2}) || \text{tobytes}(s_{0,3}) \)
4: return \( h \)

computation, the state is loaded with all zeros, and the input message is then processed as 128-bit. The input is divided into 128-bit blocks, and one block is processed in one clock cycle in the proposed architecture. The serialized architecture of the hash is shown in Figure 5; however, the hardware architecture consists of a single permutation function block, and inputs to the block are controlled through the multiplexers, with control signal coming from a counter referring to the number of round being executed. One can vary the step, \( s \), in the permutation function to accommodate different area and clock-cycle requirements.

In the beginning, the Gimli state is set to zeros, and then, the message block is XOR’d to the top row of the state, as highlighted in Figure 6 (a). This is followed by the Gimli permutation, wherein each block of the input message is processed in the same way, except for the last block. The last block can be empty or partially filled. For a block of \( b \) bytes, XOR the block into the first \( b \) blocks of the state followed by XOR 1 into the next byte. One final XOR is required at byte 47 of the state. After that, another round of the Gimli permutation is applied. This ends the absorb process, which is also depicted in Figure 6 (a). After the input is fully processed, the squeeze step generates the hash output from the state. The first 128 bits of the state (the top row) constitute the most significant bits of the 256-bit hash. The Gimli permutation function is applied again, and the next 128 bits of the hash are extracted.

For a single message block, the hash function computation requires three permutations. After that, the increase in message blocks increases the number of permutations by 1. The hash computation requires more clock cycles in that case. The execution time can be computed as \( cc = \frac{24}{s} \times 3 + 2 \) for \( s = \{1, 4, 8, 12, 24\} \). The hardware area also increases as \( s \) is increased.

C. HARDWARE ARCHITECTURE FOR GIMLI AUTHENTICATED ENCRYPTION

Gimli AE is based on a conventional duplex mode of operation [31] for data encryption and decryption, which is
regarded as the most elementary way to encrypt something using permutation functions. The state is initialized with a 128-bit nonce (N) and a 256-bit key (K). The inputs for the cipher, i.e., associated data (AD) and plaintext (PT), are read in the same fashion as the Gimli hash (in the form of 128-bit blocks) and then XOR’d into the current state. The final block (partially full or empty) for both AD and PT are processed in the same way as the Gimli hash. The 256-bit key, endorsed by the NIST, can greatly reduce the concerns about different attack scenarios. Whenever a plaintext byte is XOR’d into the state byte, the new state byte is output as ciphertext. After the final block of plaintext, the first 128 bits of the state are output as an authentication tag. Algorithm 5 details Gimli AE.

Algorithm 5 The Gimli-AE Encryption process [19]

| Line | Description |
|------|-------------|
| 1    | **Input:** $M \in \{0, 1\}^s$, $A \in \{0, 1\}^s$, $N \in \mathbb{F}_2^{16}$, $K \in \mathbb{F}_2^{256}$ |
| 2    | **Output:** GIMLI-CIPHER-ENCRYPT($M, A, N, K$) = $C \in \{0, 1\}^s$, $T \in \mathbb{F}_2^{256}$ |
| 3    | **Initialization** |
| 4    | s ← 0 |
| 5    | for i from 0 to 3 do |
| 6    | $s_{0,i} \leftarrow \text{touint32}(N_{4i} \ldots | | N_{4i+3})$ |
| 7    | $s_{1,i} \leftarrow \text{touint32}(K_{4i} \ldots | | K_{4i+3})$ |
| 8    | $s_{2,i} \leftarrow \text{touint32}(K_{16+4i} \ldots | | K_{16+4i+3})$ |
| 9    | end for |
| 10   | s ← GIMLI(s) |
| 11   | **Processing AD** |
| 12   | $a_1, \ldots, a_t \leftarrow \text{pad}(A)$ |
| 13   | for i from 1 to s do |
| 14   | if $i == s$ then |
| 15   | $s_{2,3} \leftarrow s_{2,3} \oplus 0x1000000$ |
| 16   | end if |
| 17   | s ← absorb(s, $a_i$) |
| 18   | end for |
| 19   | **Processing Plaintext** |
| 20   | $m_1, \ldots, m_t \leftarrow \text{pad}(M)$ |
| 21   | for i from 1 to t do |
| 22   | $k_i \leftarrow \text{squeeze}(s)$ |
| 23   | $c_i \leftarrow k_i \oplus m_i$ |
| 24   | if $i == t$ then |
| 25   | $s_{2,3} \leftarrow s_{2,3} \oplus 0x1000000$ |
| 26   | end if |
| 27   | s ← absorb(s, $m_i$) |
| 28   | end for |
| 29   | $C \leftarrow c_1 | \ldots | c_t$ |
| 30   | $T \leftarrow \text{squeeze}(s)$ |
| 31   | **return** $C, T$ |

The proposed Gimli AE architecture is shown in Figure 7. The input to the permutation function is controlled by a counter throughout the execution. The counter records the number of rounds during the permutation function and makes the correct input for permutation. The state (before and after permutation computation) is stored in the registers. As soon as the input register is loaded with the values that are controlled through the counter, the permutation function starts its’ computation. The loading of input into registers and computation of the permutation function occur in a single clock cycle. As a result of the proposed architecture, the same hardware is utilized repeatedly to conserve hardware resources.

![FIGURE 7. The proposed architecture for Gimli authenticated encryption.](image)

The initial state is formed by the 128-bit input nonce and the 256-bit secret key, and this state is fed into the permutation function. The updated state is stored in the registers. AD processing is the next step; the 128-bit of AD is XOR’d with the top row of the updated state and fed back to the input registers.

As soon as the registers get loaded, computation of the permutation function starts. This continues till the last block of the AD is processed. For the last block (either empty or partially full), the processing is the same. The plaintext is the next input, and each 128-bit input is stored in the input register to get processed and is provided to the permutation hardware serially, with the final block processed in the same way. After plaintext block processing, 128-bit ciphertext is extracted from the top row of the updated state. After the final block of the plaintext is processed, the 128-bit state is the authentication tag. In this way, the whole encryption is performed using a single piece of permutation hardware.

The permutation function used for the AE computation can have varying values of $s$, which can help to achieve flexibility. Hardware area increases linearly as the number of rounds processed in one clock cycle increases with the reduction in clock cycle utilization. The clock cycles consumption can be estimated as $cc = 24/s \times 3 + 2$ for $s = \{1, 4, 8, 12, 24\}$.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

The proposed architectures were designed using Verilog HDL, and implementations were applied to the Spartan-6 device. All the proposed architectures were evaluated in terms of hardware area, computation time, TP and TP/A ratio. In this paper, TP is defined as the number of bits processed.
by a given hardware architecture in one second. It can be calculated as:

$$\text{TP} = \frac{\text{Frequency} \times \text{Bsize}}{\text{Latency}},$$

(1)

where Bsize is the number of bits processed. The TP/A ratio is another important metric for evaluating architectures that incorporate hardware area as well. For the proposed architectures, it can be measured as the ratio of TP to the LUTs consumed, and can be calculated as follows:

$$\frac{\text{TP}}{\text{A}} = \frac{\text{TP}}{\text{LUTs}}.$$  

(2)

Table 1 shows the implementation results for the permutation function. The possible number of rounds that can be executed in one clock cycle, i.e. \( s = \{1, 4, 8, 12, 24\} \), are listed in Table 1. To realize small-area architectures, it is advisable to execute one round in one clock cycle. In this case, the permutation function (Figure 3) is repeated 24 times and is completed in 24 clock cycles. The basic architecture for one round is utilized every clock cycle, reducing area consumption. However, in this case, latency would be higher, resulting in lower TP. The applications that are not strictly constrained by area can take advantage of a higher TP. As the number of rounds executed is increased per clock cycle, the critical path is increased, which results in decreased frequency. The clock cycles are inversely related to the number of rounds executed. The more rounds executed per clock cycle, the lower the latency.

| Scheme   | Area (LUTs) | Freq (MHz) | Time (ns) | Latency (cc) | TP (Gbps) | TP/A (Mbps/LUT) |
|----------|-------------|------------|-----------|--------------|-----------|-----------------|
| 1 round  | 907         | 549.71     | 1.81      | 25           | 2.814     | 3.10            |
| 4 rounds | 1890        | 343.07     | 2.91      | 7            | 6.273     | 3.31            |
| 8 rounds | 3351        | 192.81     | 5.18      | 4            | 6.169     | 1.84            |
| 12 rounds| 4863        | 138.85     | 7.20      | 3            | 5.924     | 1.22            |
| 24 rounds| 8665        | 72.38      | 13.81     | 2            | 4.632     | 0.52            |

The permutation function forms the basis for the design of the Gimli hash and AE architectures. The implementation results for the Gimli hash function are given in Table 2. The hardware area increases as the number of rounds executed per clock cycle increases. The critical path increases in this case, causing a reduction in the operating frequency. The hardware area for both the permutation and the hash are almost equal, because the permutation function is applied serially to compute the hash. The important point to note is that latency is increased while computing the hash. Since one hash function consists of three permutations (which are executed in series), as a result clock cycle utilization for the hash is approximately three times that of the permutation function.

The Gimli AE results are shown in Table 3. For computation of the encryption, the permutation hardware is utilized with inputs arranged using the multiplexers. As a result, there is a slight overhead in area utilized. The latency for encryption is also increased because a single encryption requires five permutations to be computed. The performance for encryption and decryption is similar, as both functions are based on the same permutation function, and the same operations are followed, which translates into similar hardware. The only expectation is that there is one additional step (to compare the generated tag with the received tag), which is a simple operation and can be implemented in software.

| Scheme | Area (LUTs) | Freq (MHz) | Time (ns) | Latency (cc) | TP (Gbps) | TP/A (Mbps/LUT) |
|--------|-------------|------------|-----------|--------------|-----------|-----------------|
| 1 round | 1715        | 408.59     | 3.447     | 122          | 0.42      | 0.24            |
| 4 rounds| 2229        | 342.14     | 2.92      | 32           | 1.36      | 0.61            |
| 8 rounds| 3599        | 182.09     | 5.49      | 17           | 1.37      | 0.38            |
| 12 rounds| 5193       | 134.39     | 7.41      | 12           | 1.43      | 0.27            |
| 24 rounds| 9555       | 70.43      | 14.19     | 7            | 1.28      | 0.13            |

TABLE 2. Hash function results.

| Scheme   | Area (LUTs) | Freq (MHz) | Time (ns) | Latency (cc) | TP (Gbps) | TP/A (Mbps/LUT) |
|----------|-------------|------------|-----------|--------------|-----------|-----------------|
| 1 round  | 1036        | 429.80     | 2.32      | 74           | 0.74      | 0.71            |
| 4 rounds | 1624        | 344.27     | 2.90      | 20           | 2.20      | 1.35            |
| 8 rounds | 3094        | 192.97     | 5.18      | 11           | 2.24      | 0.72            |
| 12 rounds| 4601        | 139.28     | 7.18      | 8            | 3.08      | 0.66            |
| 24 rounds| 9023        | 72.29      | 13.83     | 5            | 1.85      | 0.20            |

Power and energy efficiency are also very important for RFID applications. Hence, energy consumption \((e)\) and energy/bit \((e/b)\) are two important metrics that must be considered for efficient designs. Energy consumption can be calculated as

$$e = \frac{P \times \text{Latency}}{\text{Frequency}}.$$  

(3)

Similarly, the energy/bit \((e/b)\) reflects the cost of energy that is associated with one bit, and is calculated as

$$e/b = \frac{e}{\text{bits processed}}.$$  

(4)

The Xilinx XPower Analyzer was used to estimate the power for two operating frequencies: 13.56 MHz and 200.00 MHz. Table 4 shows the power and energy consumption for the Gimli hash and Gimli AE at 13.56 MHz and 200.00 MHz. The power consumption in this case is dependent on the switching activity and the critical path length. Higher switching paths constitute higher power consumption. Similarly, longer critical paths bring higher power consumption. The switching activities and critical path delays are different for both hash and AE. However, they converge at eight rounds for the hash function and at one round for AE to give the best results for power and energy consumption.

V. APPLICATION OF GIMLI TO RFID AUTHENTICATION

This section explains the application of Gimli cipher to a recently proposed RFID authentication protocol based on symmetric cryptography. Mansoor et al. [32] presented an
This paper evaluates the protocol by providing the hardware performance on an FPGA platform. Figure 8 and Figure 9 show the tag registration phase and tag authentication phase, respectively.

The protocol is based on three main components: (1) the database server, (2) readers, and (3) RFID tags. The network layout of the RFID system is divided into several RFID clusters; each cluster consists of a reader and many tags. The tags can move from one cluster to another. Every reader of a cluster authenticates the registered tags through the database server. Each reader and the database server share symmetric key $K_{rs}$. The authentication scheme consists of two main phases; (1) tag registration phase and (2) tags authentication phase.

The hardware architectures employed during each step of the tag registration and tag authentication phases are described subsequently. Figure 8 shows the steps in registration. This starts when the tag submits its identity, $ID_T$, to the database server, S.

The server generates a random number, $n_s$, and computes $K_{ts}$. After that, the tag’s identity, $AID$, is computed using encryption, and this encryption is done with the secret key of the server. The server stores this information and sends it to the RFID tag through a secure channel. Upon receiving this message, the tag stores this information in its memory.

This paper suggests employing Gimli cipher to design the hardware architectures required to perform the above-mentioned operations. The bit-lengths for the operands are selected to comply with the Gimli algorithm for hardware implementation. The random number is generated through a 128-bit Trivium PRNG [33] due to its reasonable throughput and small hardware resource consumption. Previously, Trivium was also used to generate random samples in crypto-processor [38], signifying that this is a reasonable choice of implementing PRNG. The Gimli hash generates 256-bit output, while Gimli AE generates 256-bit encryption output, which corresponds to the encryption input in this protocol.

### TABLE 5. Tag registration phase results.

| AE     | Module | Area (LUTs) | Frequency (MHz) | Latency (cc) | TP (Mbps) | TP/A (Mbps/LUT) |
|--------|--------|-------------|-----------------|--------------|-----------|-----------------|
| Step 1 | PRNG   | 0.171k      | 319.69          | 128          | 2319.69   | 1.869           |
|        | Hash   | 1.036k      | 429.80          | 99           | 555.70    | 0.536           |
|        | Encrypt| 1.715k      | 408.59          | 147          | 355.77    | 0.207           |

Figure 10 is the hardware architecture for tag registration. During registration, the tag sends its identity to the database server, which is only a communication operation, not computation. The computations for this phase only occur in the server. This includes 128-bit random number generation through the PRNG module, one hash operation, and the encryption operation achieved through the Gimli hash and encryption modules. The bit-lengths for each module are also shown in Figure 10. The Gimli hash and AE are realized in the same way as described in the previous sections. Table 5 shows the results for hardware area utilization, TP, and TP/A from the tag registration phase.
B. HARDWARE REQUIRED FOR TAG AUTHENTICATION

The hardware resources for the tag authentication phase are discussed in this section. This phase involves computations in the tag, the reader, and the database server. The authentication phase is divided into five steps, as shown in Figure 9.

1) STEP 1
In the first step of the authentication phase, the tag is responsible for all the computations. The tag generates a random number, $N_t$, then derives $N_x$ and computes the hash, $V_1$, as shown in Figure 9. The tag then initiates the authentication request by sending the computed credentials to the reader. Figure 11 shows the hardware architecture in the tag, along with the required bit-lengths. The operations involve the RPNG generator and a hash computation. The input to the hash function is an 896-bit number ($128 \times 7$). The computation of a hash with a larger input requires the same hardware; however, more clock cycles are required to process the input using the Gimli hash function. The time consumption for hash computation is 519.68 ns while the PRNG generates the random number in 800 ns. One 256-bit XOR operation is also involved, but it requires negligible hardware area compared to the hash function. The overall latency and hardware utilization for the PRNG and the hash are given in Table 6.

2) STEP 2
The reader is responsible for all the computations required in step 2. Upon receiving the request for authentication from the tag, the reader generates random number $N_r$ derives $N_y$,
TABLE 6. Tag authentication phase results.

| Step | Module | Frequency (MHz) | Latency (cc) | TP (Mbps) | TP/A (MbpS/LUT) |
|------|--------|-----------------|--------------|-----------|-----------------|
| 1    | PRNG   | 295.92          | 256          | 295.92    | 0.907           |
| 2    | Hash   | 429.80          | 224          | 245.67    | 0.237           |
| 3    | Hash   | 429.80          | 324          | 169.79    | 0.163           |
|      | Encrypt| 408.59          | 147          | 355.77    | 0.207           |
|      | Decrypt| 363.28          | 147          | 316.32    | 0.197           |
| 4    | Hash   | 429.80          | 174          | 316.17    | 0.305           |
| 5    | Hash   | 429.80          | 174          | 316.17    | 0.305           |

and computes the hash, $V_2$. The reader then sends a message to the database server for verification. Figure 12 shows the hardware architecture employed for generation of required credentials in this step. The hardware is similar to Step 1 except for a different input length for the hash function. The input bit-length for the hash computation is 1408-bit ($128 \times 11$), which requires different clock cycle computations but with similar hardware area utilization. The hash function can be computed in 751.68 ns while the PRNG requires 800 ns. Table 6 shows the latency information and hardware area for this step.

3) STEP 3

Step 3 involves the most compute-intensive operations for the authentication phase, which take place in the database server. Since the database server has sufficient computation and storage capabilities, it is not a critical issue to handle all these computations. Computations start with XOR operations and verification of hashes $V_1$ and $V_2$. Then, the database server verifies $AID_{r_3}$ through a decryption process. Upon successful verification, two hash functions, $V_3$ and $V_4$, are computed. The database server then updates $AID_{r_{(new)}}$ using the encryption process, and finally, computes $Z_T$ employing an XOR operation. Figure 13 translates the steps involved in Step 3 into the corresponding hardware. For verification of $V_1$ and $V_2$, the same hardware required in Step 1 and Step 2 is employed. For the computation of $V_3$ and $V_4$, again, the hash function required for that process is 640-bit ($128 \times 5$). In addition, this step requires computation of one encryption process and one decryption process, that are accomplished through the Gimli encryption and decryption modules. The input and output for both are 256-bit. This step also requires three 256-bit XOR operations. The hash functions for the verification process requires the same time as for the generation in Step 1 and 2, while the additional hash function can be computed in 403.68 ns. The encryption and decryption functions can be computed in 425.77 ns and 404.64 ns, respectively. Table 6 shows all the operations required for this step, along with the hardware area and time consumption.

4) STEP 4

The computations for Step 4 take place at the reader, when the database server sends message $M_{A_3}$ to the reader. Upon reception of this message, the reader computes the hash, and verifies its equality with the received $V_3$. If successful, the reader sends $M_{A_4}$ to the tag. If they are not equal, the reader terminates the session. Figure 14 (a) shows the hardware utilized in this step. Since this is a verification step, it only needs to compute the hash with a bit-length equal to the one utilized in Step 3 for $V_3$ generation. The input for the hash function is 640-bit ($128 \times 5$), which can be computed in 403.68 ns. Hence, only one hash function is employed in this step.

5) STEP 5

Upon receiving message $M_{A_4}$ from the reader, the tag verifies $V_4$ and updates $AID_{r_{(new)}}$ for the next authentication process.
These computations take place at the tag, and Figure 14 (b) shows the hardware modules involved. This step requires one hash function with 640-bit input \((128 \times 5)\) and one 256-bit XOR. The hash function requires 403.68 ns. Table 6 shows the hardware utilization and TP results for Step 5.

C. HARDWARE REQUIREMENTS FOR EACH COMPONENT

Each of the three components of the RFID system has its own hardware requirements, which are listed below.

1) TAG
- A PRNG that outputs a 256-bit random number
- A hash function with input of 896 bits \((128 \times 7)\)
- A hash function with input of 640 bits \((128 \times 5)\)
- Two 256-bit XOR operations

2) READER
- A PRNG that outputs a 256-bit random number
- A hash function with input of 1408 bits \((128 \times 11)\)
- A hash function with input of 640 bits \((128 \times 5)\)
- Two 256-bit XOR operations

3) DATABASE SERVER
- A PRNG that outputs a 128-bit random number
- Two encryption functions with input of 256 bits
- A decryption function with input of 256 bits
- A hash function with input of 256 bits \((128 \times 2)\)
- Two hash functions with input of 640 bits \((128 \times 5)\)
- A hash function with input of 1408 bits \((128 \times 11)\)
- A hash function with input of 896 bits \((128 \times 7)\)
- Three 256-bit XOR operations

The hardware requirements for each of the components in the RFID system are listed according to the hardware required for each step. As can be clearly seen, many of the hardware modules are repeated for each of the component. The system can be further optimized by reusing the same hardware for similar functions. For example, in the hardware requirements for a tag, instead of two separate hash functions, hardware for one hash function can handle generating the required results, since at no point in time are two hashes required simultaneously. The difference is that the hash function with a longer input requires more time to execute. The optimized hardware for each of the entities is provided in Table 7.

| TABLE 7. Hardware requirements for each component. |
|-----------------------------------------------|
| Entity | Hash | XOR | PRNG | Encryption | Decryption |
|--------|------|-----|------|------------|------------|
| Tag    | 1    | 1   | 1    | -          | -          |
| Reader | 1    | 1   | 1    | -          | -          |
| Server | 1    | 1   | 1    | 1          | 1          |

VI. COMPARISON

A comparison with some other implementations is presented in Table 8. From the results, it is evident that the proposed implementation for Gimli AE can outperform in terms of the operating frequency, TP, and TP/A metrics. Considering a deeper level of comparison with the schemes mentioned in Table 8, we can see that the lightweight AES implementation performed in [20] had higher TP and TP/A compared to our proposed design. However, the problem with this scheme is that it is not an AE scheme, and cannot share hardware while designing the hash function. Also it consumes more hardware resources compared to our proposed implementation.

The implementation of block ciphers is listed next [21]. All the block ciphers consume fewer hardware resources compared to the Gimli implementation. However, they lack being able to reuse same hardware for design of the hash function. They are good choices if they are only applied to encryption, but an application like the authentication protocols also requires a hash function. In that case, a separate hash function is required to complete the authentication, which imposes greater area overhead. Throughput is also an issue with block ciphers because a smaller area requires more latency to complete the operation, minimizing TP in the designs. Therefore, AE ciphers tend to be the best for such applications.

Ascon is a Round 2 candidate for the NIST standardization process, and is also a CAESAR finalist. It is an AE cipher, and the same permutation function can be used for the hash computation. Two Ascon implementations are discussed here. The first implementation [22] consumes 17% more area and generates 30% less TP and 40% less TP/A, compared to Gimli. Since these implementations are similar (both are based on one round per clock cycle), Gimli outperforms because of its inherent structure. A lighter version (bit-sliced S-box) was implemented [23]. Although that design can reduce some hardware resources, TP and TP/A are 70% less, compared to the Gimli implementation. Therefore, Gimli is considered superior for such applications.

Ascon is a Round 2 candidate for the NIST standardization process, and is also a CAESAR finalist. It is an AE cipher, and the same permutation function can be used for the hash computation. Two Ascon implementations are discussed here. The first implementation [22] consumes 17% more area and generates 30% less TP and 40% less TP/A, compared to Gimli. Since these implementations are similar (both are based on one round per clock cycle), Gimli outperforms because of its inherent structure. A lighter version (bit-sliced S-box) was implemented [23]. Although that design can reduce some hardware resources, TP and TP/A are 70% less, compared to the Gimli implementation. Therefore, Gimli is considered superior for such applications.

[24] discusses further implementations of AE. The authors have made hardware architectures for several AE schemes and provided the results. TP results provided in Table 8 show that Gimli can outperform all these implementations when TP is measured. Similarly, the Gimli cipher can generate the best

| TABLE 8. Comparison of implementation results. |
|-----------------------------------------------|
| Scheme      | Area (LUTs) | Frequency (MHz) | Throughput (Mbps) | TP/A (Mbps/LUT) |
|------------|-------------|----------------|-------------------|-----------------|
| AES [20]   | 2.15k       | 206.06         | 375               | 0.267           |
| Klein [21] | 0.789k      | 101            | 140.5             | 0.178           |
| Led [21]   | 0.76k       | 83.3           | 63.8              | 0.083           |
| Lileiput [21] | 0.44k    | 163.4          | 174.3             | 0.392           |
| Kianan [21] | 0.29k      | 178.7          | 14.4              | 0.049           |
| Ascon [22] | 2.08k       | 206.262        | 315.2             | 0.153           |
| Ascon [23] | 1.64k       | 146.1          | 114.0             | 0.070           |
| LOCUS_32 [24] | 1.84k   | 166.04         | 93.21             | 0.050           |
| ESTABE-ABS_32 [24] | 1.36k | 202.02         | 293.85            | 0.216           |
| ESTABE-Gift_32 [24] | 1.05k | 233.97         | 73.40             | 0.070           |
| COMET_32 [24] | 1.74k    | 196.85         | 427.20            | 0.246           |
| Oribellia-256_32 [24] | 1.43k | 246.24         | 230.06            | 0.161           |
| BeetleLight+ [5] | 0.61k | 381.59         | 187.9             | 3.050           |
| Gimli      | 1.71k       | 408.59         | 428.68            | 0.25            |
TABLE 9. Hardware requirements for each component.

| Architecture | Area (LUTs) | Latency (cL) | Frequency (MHz) | Time (µs) |
|--------------|-------------|--------------|-----------------|-----------|
| GP(2^{256})  | 8.9k        | 262          | 177.7           | 1480      |
| GP(2^{288})  | 20.6k       | -            | 226             | 6.01      |
| GP(2^{371})  | 17.8k       | 3792         | 269             | 13.87     |

TP/A ratio, compared to the AE schemes provided in [24]. In addition, the hardware area for the Gimli cipher is better than its counterparts discussed in [24].

Avik et al. [25] presented the hardware implementations for the Beetle lightweight AE scheme. The Beetle cipher requires the least hardware area and generates the highest TP and TP/A ratio. This efficiency is not free, however; it comes at the cost of sacrificing security. The Light+ version of the Beetle cipher can only provide 64-bit security, which cannot be sufficient in some cases. Hence we suggest employing the Gimli cipher for such applications.

Apart from the comparison of Gimli AE with the symmetric schemes discussed above, the evaluation of asymmetric schemes is also presented with respect to the hardware resources utilization and time consumption. Referring to the authentication protocol based on ECC presented by Naeem et al. [17], the main operations are point multiplication, point addition and hash function. Out of these main operations, the most hardware expensive and time consuming one is the point multiplication. To understand its hardware consumption and speed performance, we have summarized in Table 9, a few highly optimized implementations of point multiplication found in the literature. It can be clearly seen that the point multiplication consumes more hardware resources compared to the protocols based on symmetric cryptography (e.g., Gimli).

VII. CONCLUSION

To mitigate potential attacks on RFID systems, numerous authentication protocols have been presented, but most of them were only analyzed theoretically without considering hardware implementation. In addition, implementing authentication protocols in RFID systems needs to be flexible in order to accommodate various requirements in hardware area and TP. However, the majority of the prior work that optimized the time-consuming cryptographic primitives (hash functions and AE) were either area-optimized or TP-optimized. To close these gaps (i.e. the lack of flexibility and insufficient hardware implementation), this paper first proposed a flexible implementation strategy on Gimli permutation, wherein the hardware area and TP can be varied accordingly. The most area-optimized implementations can engender a TP of 740 Mbps and 420 Mbps for the Gimli hash and AE, respectively, which is suitable for RFID tags with a constrained area. In contrast, RFID readers can trade area to obtain higher TP, which can be achieved through our flexible implementation strategy. This strategy is applied to a state-of-the-art authentication protocol [32], and its effectiveness was demonstrated through experimental results.

The proposed flexible strategy can be further extended to implement other hash functions and AE schemes, as well as a different authentication protocol that uses these two cryptographic primitives. Note that authentication protocols that use both a hash function and an AE scheme in RFID tags or readers, can actually reuse the core hardware architecture to perform both AE and hash operations. For instance, the proposed Gimli permutation hardware architecture can be reused for both hash function and AE operations. In contrast to the conventional design, wherein the encryption (block cipher) and hash function are implemented as separate hardware modules, this approach can save significant hardware area. This implies that an authentication protocol that is designed based on both hash function and AE can be more advantageous than ones designed on block ciphers and hash functions; this is one of the promising research direction derived from this work. Another interesting direction is developing a more hardware-efficient architecture for RFID components in order to share hardware resources as much as possible. For instance, the possibility of sharing hardware between the PRNG and the hash function or AE can be explored. Advancement in these directions can greatly improve the practicality of using authentication protocols in real-world applications with optimal resource consumption. One promising application area for such efficient implementations is the healthcare system [34].

REFERENCES

[1] P. P. Ray, “A survey on Internet of Things architectures,” J. King Saud Univ.-Comput. Inf. Sci., vol. 30, no. 3, pp. 291–319, 2018.
[2] K. Fan, W. Jiang, H. Li, and Y. Yang, “Lightweight RFID protocol for medical privacy protection in IoT,” IEEE Trans. Ind. Informat., vol. 14, no. 4, pp. 1656–1665, Apr. 2018.
[3] B. Khoo, “RFID-from tracking to the Internet of Things: A review of developments,” in Proc. IEEE/ACM Int. Conf. Green Comput. Commun. Int. Conf. Cyber. Phys. Soc. Comput., Dec. 2010, pp. 533–538.
[4] C.-H. Hsu, S. Wang, D. Zhang, H.-C. Chu, and N. Lu, “Efficient identity authentication and encryption technique for high throughput RFID system,” Secur. Commun. Netw., vol. 9, no. 15, pp. 2581–2591, Oct. 2016.
[5] Y. Zhang and L. Bai, “Rapid structural condition assessment using radio frequency identification (RFID) based wireless strain sensor,” Autom. Constr., vol. 54, pp. 11–11, Jun. 2015.
[6] P.-M. G. Simon, E. F. Riggert, and S. E. Trivelpiece, “System and method for reading RFID tags across a portal,” U.S. Patent 9 519 811, Dec. 13, 2016.
[7] M. El Beqqual and M. Azziz, “Classification of major security attacks against RFID systems,” in Proc. Int. Conf. Wireless Telecomm., Embedded Intell. Syst. (WITS), Apr. 2017, pp. 1–6.
[8] U. Ayaz, T. A. Haq, S. Tainour, K. Mansoor, and S. Mahmood, “An enhanced biometric based RFID authentication scheme defending against illegitimate access,” in Proc. 14th Int. Conf. Emerg. Technol. (ICET), Nov. 2018, pp. 1–6.
[9] C. Alippi and C. Galperti, “An adaptive system for optimal solar energy harvesting in wireless sensor network nodes,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 55, no. 6, pp. 1742–1750, Jul. 2008.
[10] A. A. R. Haeri, M. G. Karkani, M. Sharifkhani, M. Kamarei, and A. Fotowat-Ahmad, “Analysis and design of power harvesting circuits for ultra-low power applications,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 64, no. 2, pp. 471–479, Feb. 2017.
[11] N. Smyth, M. McLoone, and J. V. McCanny, “WLAN security processor,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 53, no. 7, pp. 1506–1520, Jul. 2006.
[12] M. Knezevic, V. Nikov, and P. Rombouts, “Low-latency encryption—Is light-weight=light+wait?” in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Berlin, Germany: Springer, 2012, pp. 426–446.

[13] T. Xu, J. B. Wendt, and M. Potkonjak, “Security of IoT systems: Design challenges and opportunities,” in *Proc. IEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2014, pp. 417–423.

[14] C. J. McIvor, M. Mcloone, and J. V. Mccanny, “Hardware elliptic curve cryptographic processor over rnmGF(p),” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 9, pp. 1946–1957, Sep. 2006.

[15] T. Good and M. Benaisa, “Very small FPGA application-specific instruction processor for AES,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 7, pp. 1477–1486, Jul. 2006.

[16] N. Roy, A. Trivedi, and J. Wong, “Designing an FPGA-based RFID reader,” *Xcell J.*, vol. 2, no. 26, pp. 26–29, 2006.

[17] M. Naeem, S. A. Chaudhry, K. Mahmood, M. Karuppiah, and S. Kumari, “A scalable and secure RFID mutual authentication protocol using ECC for Internet of Things,” *Int. J. Commun. Syst.*, vol. 33, no. 13, p. e3906, Sep. 2020.

[18] D. Din, J. Y. L. Corre, D. Khvorotovich, L. Perrin, J. Großschädl, and A. Biryukov, “Triathlon of lightweight block ciphers for the Internet of Things,” *J. Cryptograph. Eng.*, vol. 9, no. 3, pp. 283–302, Sep. 2019.

[19] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier, “GIMLI: A cross-platform permutation,” in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.* Cham, Switzerland: Springer, 2017, pp. 299–320.

[20] M. M. Wong, M. L. D. Wong, C. Zhang, and I. Hajinaz, “Circuit and system design for optimal lightweight AES encryption on FPGA,” *IAENG Int. J. Comput. Sci.*, vol. 45, no. 1, pp. 52–62, 2018.

[21] C. Marchand, L. Bossuet, and K. Gaj, “Area-oriented comparison of lightweight block ciphers implemented in hardware for the activation mechanism in the anti-counterfeiting schemes,” *Int. J. Circuit Theory Appl.*, vol. 45, no. 2, pp. 274–291, Feb. 2017.

[22] S. Khan, W.-K. Lee, and S. O. Hwang, “Scalable and efficient hardware architectures for authenticated encryption in IoT applications,” *IEEE Internet Things J.*, vol. 8, no. 14, pp. 11260–11275, Jul. 2021.

[23] W. Diehl, F. Faramadh, A. Abdulgadir, J.-P. Kap, and K. Gaj, “Face-off between the CAESAR lightweight finalists: ACORN vs. Ascon,” in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 330–333.

[24] B. Ovilla-Martínez, C. Mancillas-López, A. F. Martínez-Herrera, and J. A. Bernal-Gutiérrez, “FPGA implementation of some second round NIST lightweight cryptography candidates,” *Electronics*, vol. 9, no. 11, p. 1940, Nov. 2020.

[25] A. Chakraborti, N. Datta, M. Nandi, and K. Yasuda, “Beetle family of lightweight and secure authenticated encryption ciphers,” *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, no. 2, pp. 218–241, May 2018.

[26] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier. (2020). Gimi: NIST LWC Second-Round Candidate Status Update. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round2/status-update-sep2020/gimli_update.pdf

[27] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. (2016). Ascon Vi2. Submission to the CAESAR Competition. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round2/spec-doc-md2/ascon-spec-round2.pdf

[28] D. Bellizia, F. Berti, O. Bronchard, G. Cissiers, S. Duval, C. Guo, G. Leander, G. Laurent, I. Levi, C. Monin, O. Pereira, T. Peters, F.-X. Standaert, B. Udvarhelyi, and F. Wiener, “Spoof: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher,” *IACR Trans. Symmetric Cryptol.*, vol. 2020, no. S1, pp. 295–349, Jun. 2020.

[29] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasaki, and S. M. Sim, “The SKINNY family of block ciphers and its low-latency variant MANTIS,” in *Proc. Annul. Int. Conf. Cryptol. Conf.* Berlin, Germany: Springer, 2016, pp. 123–153.

[30] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “On the indifferentiability of the sponge construction,” in *Proc. Annul. Int. Conf. Theory Appl. Cryptograph. Techn.* Berlin, Germany: Springer, 2008, pp. 181–197.

[31] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Duplexing the sponge: Single-pass authenticated encryption and other applications,” in *Proc. Int. Workshop Sel. Areas Cryptogr.* Berlin, Germany: Springer, 2011, pp. 320–337.

[32] K. Mansoor, A. Ghani, S. Chaudhry, S. Shamshirband, S. Ghayyur, and A. Mosavi, “Securing IoT-based RFID systems: A robust authentication protocol using symmetric cryptography,” *Sensors*, vol. 19, no. 21, p. 4752, Nov. 2019.

[33] C. De Cantiere and B. Preneel, “Trivium,” in *New Stream Cipher Designs*. Berlin, Germany: Springer, 2008, pp. 244–266.

[34] M. Haddara and A. Staaby, “RFID applications and adoptions in health-care: A review on patient safety,” *Procedia Comput. Sci.*, vol. 138, pp. 80–88, Jan. 2018.

[35] M. M. Islam, M. S. Hossain, M. K. Hasan, M. Shahjalal, and Y. M. Jang, “FPGA implementation of high-speed area-efficient processor for elliptic curve point multiplication over prime field,” *IEEE Access*, vol. 7, pp. 178811–178826, 2019.

[36] R. Salarifarid, S. Bayat-Sarmadi, and H. Mosanaei-Boorani, “A low-latency and low-complexity point-multiplication in ECC,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 9, pp. 2869–2877, Sep. 2018.

[37] M. Rashid, M. Imran, and A. Sajid, “An efficient elliptic-curve point multiplication architecture for high-speed cryptographic applications,” *Electronics*, vol. 9, no. 12, p. 2126, Dec. 2020.

[38] Y. Xing and S. Li, “An efficient implementation of the NewHope key exchange on FPGAs,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 3, pp. 866–878, Mar. 2020.

SAFIULLAH KHAN received the B.Sc. degree in electronic engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2013, and the M.Sc. degree in electrical engineering from COMSATS University Islamabad, Abbottabad campus, Pakistan, in 2017. He is currently pursuing the Ph.D. degree in computer engineering with Gachon University, Seongnam, South Korea. He worked with Research and Development Department, National Radio Corporation, Haripur, Pakistan, for two years. His research interests include efficient hardware implementations of cryptographic protocols, blockchain, and network security.

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

SEONG OUN HWANG (Senior Member, IEEE) received the B.S. degree in mathematics from Seoul National University, in 1993, the M.S. degree in information and communications engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, South Korea, in 2004. 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 worked 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*. * * *