An Efficient Lightweight Cryptography Hash Function for Big Data and IoT Applications

Zeyad A. Al-Odat  
Department of Communication and Computer Engineering  
Tafila Technical University  
Tafila, Jordan  
Email: zeyad.alodat@ttu.edu.jo

Eman M. Al-Qtiemat  
Department of Electrical and Computer Engineering  
North Dakota State University  
Fargo, ND, 58102  
Email: eman.alqtiemat@ndsu.edu

Samee U. Khan  
Department of Electrical and Computer Engineering  
Mississippi State University  
Starkville, MS  
Email: skhan@ece.msstate.edu

Abstract—This paper introduces a lightweight cryptography hash function for big data and IoT applications. The proposed design employs S-Box, linear transformation, and bit permutation functionalities. Conventional hash functions require memory and time to process big data and IoT applications. Therefore, a secure and quick lightweight cryptography protocol is needed. This design provides the security requirements of the conventional hash functions and the specifications of the lightweight cryptography protocols. The proposed design is tested and verified using three of the lightweight requirements: speed, memory and power consumption. The results show that the proposed design outperforms other lightweight protocols in terms of speed, memory and power consumption. Moreover, the security analysis shows that this design posses the general security requirements of the conventional hash protocols.

Keywords—Hash function, Lightweight Cryptography, Security, IoT, Big Data.

I. INTRODUCTION

Internet of Things (IoT) has emerged to include the majority of life applications and devices. With the increase of IoT devices, more data are processed [1]. Processing big data requires security and authenticity requirements that are achieved by cryptography protocols, e.g., encryption and decryption.

Conventional cryptography protocols require more time to process massive data that are generated by IoT devices. Particularly, encryption and decryption mechanisms that work on data byte by byte. These mechanisms are considered inefficient in case of massive data. Vice versa, simple cryptography primitives are easy to crack and don’t support the required security level [2]. Because conventional cryptography functions require memory and resources, a powerful lightweight cryptography primitive is required to support big data that are generated from IoT devices.

Secure Hash Algorithm is one part of cryptography protocols that require quick processing and authentic data transmission. A lightweight cryptography hash function is required in the case of IoT applications that involve intensive and sensitive data transactions through cloud computing. Especially, the IoT-based devices that are used in the majority of life necessities [3]. The conventional hash functions were improved over time to build powerful and secure hash protocols that are strong against security breaches and attacks. However, the lightweight cryptography protocols are less secure than conventional functions because of the limited design specifications [4].

In this work, we will focus on the design of a lightweight cryptography hash function. The proposed design provides a secure and fast cryptography hash scheme that posses the conventional hash functions security requirements and the lightweight specifications. Moreover, our design supports big data, IoT applications, and devices.

The rest of the paper is organized as follows: Section 2 provides preliminaries that are related to the proposed design; literature review is provided in Section 3; the proposed design is illustrated in Section 4; results and discussion in Section 5; Section 6 concludes the paper.

II. PRELIMINARIES

A. Merkle-Damgård Construction

Cryptography hash functions follow different construction models. Merkle-Damgård is the most popular construction model to build hash functions, particularly, SHA-1 and SHA-2 hash functions [5]. Figure 1 shows the general structure of Merkle-Damgård construction. An input message (M) is divided into number of equal size blocks (M₁, M₂, . . . , Mₙ). These blocks are processed sequentially using the compression function (F). The initial values (IV) is used to process the first message block. Then the intermediate value, after processing M₁, is used to process the second block calculation. After processing all blocks, the output of the last block calculation is the final output [6].

B. S-Box

Substitution box or as its known S-Box is one of the cryptography components that is used for substitution [7]. In our work, we employ 4 × 4 S-Boxes to select between input bits from an input message that is composed of IHV and a message block. The S-Box receives 4-bit input such that 2-bit from the message block and 2-bit from the IHV. The selection of the S-Box depends on the round constant (Rc) bit.
Bussi et al. proposed Neeva-hash, a lightweight cryptography hash function based on sponge mode of operation [13]. The proposed design relies on the sponge construction that is used to build the winning SHA-3 competition, Keccak. Neeva-hash maintains 224-bit output hash through two main phases, absorb and squeeze. In the absorbing phase, the input message is XORed with the initial state matrix and processed through five steps of the compression function. In the squeeze phase, the output hash is generated.

Following the sponge construction, Bogdanov et al. proposed Spongent [14]. The work employs the sponge construction model based on a wide present type permutation. The design comprises three phases, initialization, absorbing, and squeezing. In the initialization phase, an input message is padded and divided into a number of equal size blocks. In the absorbing phase, the message blocks are processed sequentially using the permutation compression function. In the squeezing phase, the output is interleaved and truncated to produce the output digest. Spongent supports 128, 160, 224, and 256 hash lengths. Mukundan et al. proposed Hash-One, a lightweight cryptography hash function based on sponge structure with two nonlinear feedback shift registers [15]. Like the aforementioned works, Hash-One involves absorb and squeeze phases. Hash One maintains 160-bit hash output that is generated in the squeezing phase after permutation.

Another sponge-based lightweight hash function is Quark [16]. Quark follows the general steps of the sponge structure model including initialization, sponge, and squeeze. Quark is built upon the stream cipher and KATAN block cipher. The hardware implementation of Quark showed improved results regarding memory management and speed.

To support wireless sensor networks, authors in [17] proposed a lightweight cryptography hash function for wireless image transmission in wireless sensor networks. The proposed design is based on the Enhanced Cuckoo Search (ECS) model. This enhanced model is used to build the compression function depending on the arbitrary Monte-Carlo method. The work is tested regarding several performance evaluation metrics including PSNR, MAE, Entropy, NPCR, and throughput.

IV. PROPOSED DESIGN

We propose a lightweight hash function that supports IoT sensitive devices. Figure 2 shows the general structure of the proposed design. An input message \( M \) with variable length is padded using 10’1 padding technique to make its size multiple of 512. Then, the padded message \( M^* \) is divided into equal size blocks \( B_1, B_2, \ldots, B_n \), each block size is 512-bit. The divided blocks are processed sequentially through compression function \( F \) in a bijective way, where a block is XORed with part of the IHV before and after the compression function processing. The IHV is 1024-bit and initialized according to the desired output hash size, i.e., if the desired output hash size is 256-bit, then the first two bytes of the IHV will be 0x0100 and the rest of bytes are set as zeros.

The proposed design comprises five phases; padding, parsing, setting the IHVs, compression function calculations, and
final hash generation phases. Each phase will be discussed in the subsequent text.

![Diagram of the proposed design](image)

**Fig. 2.** The proposed design

### A. Phase 1 - Message Padding

In the proposed design, we employed 10*1 padding technique. The general structure of this technique is depicted in Algorithm 1 below. An input message \((M)\) of variable length is set to be padded to make its size multiple of the block size \((B)\). As shown in the algorithm, the number of required padding bits is determined by calculating the remaining bits after dividing the length of the message over the block size. Then, the required number of bits \((P)\) is appended to the input message including two 1’s that represent the padding borders. \(M^*\) is generated, which represents the input message after padding.

**Algorithm 1: Padding Technique**

**Input:** Message \((M)\)

**Block Size \((B)\)**

**Output:** Padded Message \((M^*)\), s.t. \(\text{length}(M^*)\) is multiple of \(B\)

1. \(P = M \mod B\)
2. \(* = P - 2\)
3. \(M^* = M || 1 0^* || 1\)
4. Return \(M^*\)

### B. Phase 2 - Parsing the Padded Message

The padded message is divided into equal size blocks \((B_1, B_2, \ldots, B_n)\) of 512-bit. A block of size 512-bit is represented as eight 64-bit words, i.e., the message block \(B_i\) is represented as eight words \(B_i^0, B_i^1, B_i^2, \ldots, B_i^7\). Each message block is passed to the compression function calculation, as shown in Figure 2.

### C. Phase 3 - Assign the Initial Hash Values

In this phase, we followed the same procedure that was used by the JH-hash submission [18]. The IHV depends on the desired output hash length, where the size of the output hash is stored in the first two bytes of the IHV and the rest of the IHV is set to zero. The proposed design supports various hash lengths including 160, 224, 256, 384, and 512.

### D. Phase 4 - Compression Function Calculation

The message blocks are processed sequentially using the compression function \(F\), as shown in Figure 3. The compression function comprises five steps, bit grouping, S-box, linear transformation, permutation, and degrouping steps. In the grouping step, an input message block \(B_i\) and the IHV are grouped such that each S-Box receives 2-bit from each input (the message block and IHV). The S-Box step select which S-Box to use according to the round constant \((Rc)\). Then, the linear transformation layer shuffles the bits pass them to be permuted to the following step. This process continues for each block up to the end of the 18 rounds. The output after processing the current block is passed as IHV to the second block calculation. The proposed design process each block through 18 rounds, where the number of rounds is calculated according to (1).

\[ R = 6(d - 1), \]  

where \(R\) is the number of rounds and \(d\) is the dimension level. In our design the dimension level equal to 4.

![Diagram of the compression function](image)

**Fig. 3.** Compression function \(F\) in the proposed design
S-Boxes. The S-Box takes the input in such a way that it contains an equal number of bits from the IHV and the message block. The linear transformation layer \((L)\) is used to pass two words, consecutively by utilizing XOR operation to the received words. Then, \(P_3\) permutation is applied where the words are permuted and be adjacent to different words in the next round of block calculation. After the 18 rounds are processed, in the degrouping phase, all bits are returned to their starting position to generate the final hash. In our design, the output after processing the last block is 1024-bit. Therefore, the hash generation is performed by taking the least significant number of desired bits to produce the final hash.

The general structure of the internal round function is shown in Algorithm 2. The algorithm complies with Figure 4. It shows how the input data \((x_{0}, x_{1}, \ldots, x_{15})\) is permuted and passed through S-Box and Linear transformation layers.

**Algorithm 2: Round Compression Function**

1. for \(i = 0; \; i < 15; \; i = i + 1\) do
2.     if \(Rc_{i,j} = 0\) then
3.         \(v_i = S_0(x_i)\)
4.     else if \(Rc_{i,j} = 1\) then
5.         \(v_i = S_1(x_i)\)
6. for \(i = 0; \; i < 7; \; i = i + 1\) do
7.     \((w_{2i}, w_{2i+1}) = L(v_{2i}, v_{2i+1})\)
8. \((y_{0}, y_{1}, \ldots, y_{15}) = P_d(w_{1}, w_{2}, \ldots, w_{15})\)

Figure 4 shows one round calculation of the proposed design. The input to the S-Boxes represents 16 different blocks \((x_{0}, x_{1}, \ldots, x_{15})\), which are a mix from a message block and 512-bit of the IHV. The input is passed to the S-Box and the linear transformation layer, as mentioned before. Then the output of this round is passed to the second round calculations.

**E. Phase5-Generate the Final Hash**

The final hash is generated by taken the least significant number of bits of the final output, i.e., if the desired hash is 256-bit, then the least significant 256-bit of the output is considered as the final hash.

V. RESULTS AND DISCUSSION

The proposed design is implemented on Linux 19.10 platform with Intel i7-3400MHz and 16GB of RAM. The proposed design is compared with other designs regarding speed and power consumption.

**A. Speed**

To test the speed of the proposed design, we implemented our design using c++ programming language under Linux 19.10 LTS. For a fair comparison with other works, we also implemented the works that we compared our design with under the same conditions and specifications. The proposed design showed a significant improvement in terms of speed compared with other lightweight cryptography protocols hash works from the literature, as shown in Table II

**B. Hamming Distance**

Hamming distance is the number of bit differences between two different strings with the same length. In our test, we generated 1000 different messages. Each message is examined using 10 testing cases; each case represents the messages with 1-bit change. Then we measured the average Hamming distance of the ten cases of a message. Figure 5 shows the average Hamming distance of the 1000 random messages, where each point represents the average distance of the 10 testing cases.

**C. Resource Usage**

The proposed design is implemented using a personal computer. For a fair comparison, the proposed design and the other works were implemented in the same system conditions. Moreover, we shut down any unnecessary processes and made the CPU, eventually, run only the implementations. We measured the number of clock cycles that are needed to run the works and the amount of memory to process them, as long as the power consumption. According to [19], the power consumption is measured using (2).

\[ E = I \times V_{cc} \times \tau \times N, \]  

where \(E\) is the power consumption, \(I\) is the consumed current during \(T\) seconds, \(V_{cc}\) the voltage that the CPU operates on, and \(\tau\) is the clock period that is represented as \(\tau = \frac{1}{T}\).

The measured and calculated results showed that our proposal outperformed the other works in all terms, memory, clock cycles and power consumption, as shown in Table II.

**TABLE III
**

<table>
<thead>
<tr>
<th>File Size</th>
<th>Spongent-224</th>
<th>Neeva-hash</th>
<th>Proposed Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>10MB</td>
<td>1.95</td>
<td>1.08</td>
<td>0.58</td>
</tr>
<tr>
<td>50MB</td>
<td>7.24</td>
<td>6.35</td>
<td>2.94</td>
</tr>
<tr>
<td>100MB</td>
<td>24.21</td>
<td>14.5</td>
<td>8.24</td>
</tr>
<tr>
<td>300MB</td>
<td>54.21</td>
<td>36.45</td>
<td>16.77</td>
</tr>
<tr>
<td>500MB</td>
<td>102.3</td>
<td>66.52</td>
<td>27.65</td>
</tr>
<tr>
<td>1GB</td>
<td>194.45</td>
<td>140.54</td>
<td>77.24</td>
</tr>
</tbody>
</table>

**TABLE II
**

<table>
<thead>
<tr>
<th>File Size</th>
<th>Spongent-224</th>
<th>Neeva-hash</th>
<th>Proposed Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (MB)</td>
<td>360</td>
<td>188</td>
<td>78</td>
</tr>
<tr>
<td>Power</td>
<td>102.35</td>
<td>86.46</td>
<td>35.4</td>
</tr>
</tbody>
</table>
Fig. 4. One round of the compression function calculation of the proposed design

Fig. 5. Hamming distance of 256-bit output digest using the proposed design
D. Security Analysis

The proposed design is built in a construction scheme that is different from the designs in the literature. The majority of lightweight cryptography protocols are built using sponge construction. In our design, we utilize Merkle-Damgård construction with three different functionalities including S-Box, linear transformation, and bit permutation.

To examine the proposed design against possible attacks, we concluded the following points from the specifications of our proposal:

1) The S-Boxes will not produce identical output for the same input. This is guaranteed by the Rijndael S-box definition [20].
2) The proposed design achieved the lightweight cryptography requirements including speed, memory and power consumption. These requirements were examined using experiments from the previous section.
3) The proposed design produces 1024-bit as an output, which requires at least $2^{512}$ computation search to find a collision. Moreover, the internal S-Box shuffling and manipulation increase the number of random searches, which is much more than the birthday paradox attack. This property gives our proposal the upper hand in the case of preimage and second preimage attacks.

VI. CONCLUSION

This paper introduced a lightweight cryptography hash function that supports big data and IoT applications. The proposed work employs S-Box, linear transformation, and bit permutation paradigms. The results show significant improvements in terms of speed, memory and power consumption. In our design, we achieve the general requirements of the lightweight cryptography protocols while preserving the security specification of the conventional hash standards.

In the future, the proposed design will be tested on different hardware and platforms to support various IoT applications.

REFERENCES


