Secure Hash Algorithms and the Corresponding FPGA Optimization Techniques

ZEYAD A. AL-ODAT, North Dakota State University, USA
MAZHAR ALI and ASSAD ABBAS, COMSATS University Islamabad, Pakistan
SAMEE U. KHAN, Mississippi State University, MS

Cryptographic hash functions are widely used primitives with a purpose to ensure the integrity of data. Hash functions are also utilized in conjunction with digital signatures to provide authentication and non-repudiation services. The SHA has been developed over time by the National Institute of Standards and Technology for security, optimal performance, and robustness. The best-known hash standards are SHA-1, SHA-2, and SHA-3. Security is the most notable criterion for evaluating the hash functions. However, the hardware performance of an algorithm serves as a tiebreaker among the contestants when all other parameters (security, software performance, and flexibility) have equal strength. Field Programmable Gateway Array (FPGA) is a reconfigurable hardware that supports a variety of design options, making it the best choice for implementing the hash standards. In this survey, particular attention is devoted to the FPGA optimization techniques for the three hash standards. The study covers several types of optimization techniques and their contributions to the performance of FPGAs. Moreover, the article highlights the strengths and weaknesses of each of the optimization methods and their influence on performance. We are optimistic that the study will be a useful resource encompassing the efforts carried out on the SHAs and FPGA optimization techniques in a consolidated form.

CCS Concepts: • **Security and privacy** → **Cryptography**; • **Applied computing** → **Engineering**; • **Computer systems organization** → *Embedded systems*; • **Hardware** → *Hardware test*;

Additional Key Words and Phrases: Hardware, cryptography, optimization, SHA, FPGA

ACM Reference format:
Zeyad A. Al-Odat, Mazhar Ali, Assad Abbas, and Samee U. Khan. 2020. Secure Hash Algorithms and the Corresponding FPGA Optimization Techniques. *ACM Comput. Surv.* 53, 5, Article 97 (September 2020), 36 pages.
https://doi.org/10.1145/3311724

The work of Samee U. Khan was supported by (while serving at) the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: Z. A. Al-odat, Communication and Computer Engineering Department, Tafila Technical Universty, Tafila 66110, Jordan; email: zeyad.alodat@ndsu.edu; M. Ali, Department of Computer Science, COMSATS Institute of Information Technology, Abbottabad, Pakistan; email: mazhar@cuiatd.edu.pk; A. Abbas, Department of Computer Science, COMSATS University Islamabad, Islamabad Campus, Pakistan; email: assadabbas@comsats.edu.pk; S. U. Khan, Electrical and Computer Engineering Department, Mississippi State University, Starkville, MS 39762; email: skhan@ece.msstate.edu.

0360-0300/2020/09-ART97 $15.00
https://doi.org/10.1145/3311724

1 INTRODUCTION

In cryptography, integrity is provided with the hash functions. The hash functions compress the message to assist the process of the digital signatures. In the last decades of the 20th century, we witnessed Message Digest 4 (MD4), developed by Ronald Rivest for the provision of integrity [1]. The MD4 provided the basis for MD5, that is, a strengthened version of MD4 [2]. Both the MD4 and the MD5 worked to compute 128 bits digest. However, weakness in aforesaid algorithms resulted in the development of the Secure Hash Algorithm (SHA-0) by the National Institute of Standards and Technology (NIST) in 1993. SHA-0 was followed by SHA-1 in 1995 with the hash size of 160 bits. SHA-1 replaced MD5, which was exposed to collision attacks by then [3]. As with all of the security protocols, SHA-1 was also under constant scrutiny by the security experts. Wang et al. [4] claimed that collisions in SHA-1 can be found with complexity less than 2^{80} compression function calculations. Recently the claim of Wang et al. was validated by Marc et al. [5] by finding hash collisions for different files. Because SHA-1 was prone to the collision attack, theoretically (as shown by Wang et al.), development of SHA-2 started in 2001. SHA-2 has different flavors, such as SHA-256, SHA-384, and SHA-512 with hash values of 256, 384, and 512 bits, respectively [6]. Later on, SHA-224 was also introduced in 2004 to provide security strength of 3DES. The FIPS-180-3 defines the aforementioned four algorithms to be the part of SHA-2 standard [7].

SHA-1 and SHA-2 follow Merkle Damgard (MD) structure. The MD structure takes the message of pre-defined size and subsequently divides it into equal-size blocks. The final hash is computed using the dedicated compression function [8]. As well as SHA-1 and SHA-2 followed the same structure model, this created a fear among the experts that SHA-2 would also be exposed to the same kind of attacks. Therefore, NIST held a competition for the selection of a new hash standard (SHA-3). There were 64 submissions in the competition. Round 1 reduced the number of algorithms to 51. The second round of filtration selected 14 candidate algorithms. Finally, there were five algorithms (Grøstl, BLAKE, JH, Keccak, and Skein) in the decisive round. In October 2012, Keccak was announced as a new standard, becoming SHA-3 [9–11].

SHA-3 was different from the rest of the SHA family in the sense that it follows a sponge construction instead of Merkle Damgard. Sponge construction follows an absorbing squeezing structure where data are read in and processed in the absorbing phase followed by squeezing phase that gives the output. Besides the working model, SHA-3 also differs in the length of message size and hash length. More details for each of the aforementioned standards will be discussed in Section 2.

Hash functions have a variety of applications, including data integrity, verification, and authorization. Figure 1 shows the process of verifying message (M), which is sent over an insecure medium. The message hash is computed by a hash function in the source side and appended to the end of the message. At the destination side, the message hash is computed and compared with the appended hash value. If both hashes are equal, then the received message at the destination side is unaltered. The hash functions also assist in the digital signature that is part of authorization and validation. Moreover, hash functions are also used for the generation of random numbers and password protection [12]. To distinguish between different hash standards, a fair comparison needs to be applied. Therefore, hardware was one choice to implement and compare different hash standards. Field Programmable Gate Array (FPGA) is the best choice for hardware implementation of hash algorithms due to flexibility and adaptability [13]. Moreover, the FPGA is faster than the Central Processing Unit (CPU) and Graphic Processing Unit (GPU) [14]. FPGA is considered as an Application Specific Integrated Circuit (ASIC), which implements predefined function(s). ASIC is used when the speed matters most [15]. For instance, hardware security modules commonly use ASICs to accelerate the execution of cryptographic operations (like AES) [16]. Besides CPUs, GPUs, and ASICs, Hybrid Hardware System (HHS), which is a combination of different types of

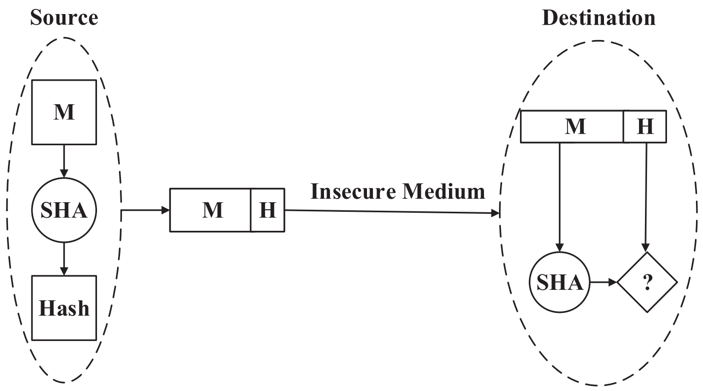


Fig. 1. Application of secure hash algorithm comparing the hash value of the message sent over insecure channel.

aforementioned hardware, can also be utilized. However, HHS tends to increase the price of the resulting modules. More details will be discussed in Section 3.

Unlike other works [17–20], this article not only discusses SHA families but also focuses on FPGA optimization techniques for implementing secure hash standards. In this survey, special attention is paid to the FPGA optimization techniques of three hash standards (SHA-1, SHA-2, SHA-3). The study covers several types of optimization techniques and highlights their contributions to the performance metrics of FPGAs. Moreover, this survey discusses the pros and cons of each of the optimization methods and investigates their effects on the performance. Furthermore, the article outlines various methods of optimization and organizes them according to their contribution to the SHA design.

However, previous surveys on the same area provide a specific explanation about the SHA standards and provide concepts of deploying SHA standards using the FPGA. The work presented in Reference [17] provides a survey and analysis of the hash functions as a combination between stream cipher and hash function. Their work focuses on the Linear Feedback Shift Register (LFSR) hash block. Jarvinen et al. conducted a comparative survey of high-performance cryptography algorithms [18]. This study analyzes the FPGA-based implementations of the most widely used cryptography implementations (AES, IDEA, DES, and SHA). However, the hash algorithm's analyses were minor and only includes MD5, SHA-1, and part of SHA-2. A theoretical survey on the secure hash algorithms is presented in Reference [19]. The work shows the different kind of hash algorithms and their applications. Chaves et al. presented a work that analyzes the implementations of the secure hash algorithms SHA-1, SHA-2, and SHA-3 on FPGA [20]. The work studied the possibilities to improve SHA-1 and SHA-2 hash functions by properly exploring the data dependencies and the hardware reuse. Moreover, the study compares between the FPGA implementations of SHA-3 finalists (Blake, JH, Skein, Grøstl, and Keccak). The study showed that the Keccak outperforms the other candidates in term of throughput and area. In this survey, a comprehensive study of FPGA-based implementation of the secure hash algorithms (SHA-1, SHA-2, and SHA-3) family is carried out. Besides, we discuss the design parameters and optimization techniques in more detail.

The rest of the article is organized as follows: Section 2 describes SHA standards and their functions. The hardware implementation of SHA and FPGA's performance metrics are discussed in Section 3. A literature review of different optimization techniques is presented in Section 4. Article discussion and analysis are carried out in Section 5. Section 6 concludes the article.

Table 1. The Secure Hash Algorithms SHA-1, SHA-2, and SHA-3 and Their Corresponding Parameters

SHA Family				
Algorithm	**Output**	**Block Size (bit)**	**Max Msg Size (bit)**	**Internal Round**
SHA-1	160	512	$2^{64} - 1$ [22]	80
SHA2				
224	224	512	$2^{64} - 1$ [6]	64
256	256	512	$2^{64} - 1$ [6]	64
384	384	1,024	$2^{128} - 1$ [6]	80
512	512	1,024	$2^{128} - 1$ [6]	80
512/224	224	1,024	$2^{128} - 1$ [6]	80
512/256	256	1,024	$2^{128} - 1$ [6]	80
SHA3				
224	224	1,152	unlimited	24
256	256	1,088	unlimited	24
384	384	832	unlimited	24
512	512	576	unlimited	24
SHAKE128	Arbitrary [11]	1,344	unlimited	24
SHAKE256	Arbitrary [11]	1,088	unlimited	24

2 SECURE HASH ALGORITHM FAMILIES

SHA takes a message with an arbitrary size, then through some calculations produces the message hash[1] [21]. The process is defined in Equation (1):

$$h = H(M), \tag{1}$$

where M is the input message, and h is the generated digest using the hash algorithm H.

Different parameters of SHA family are compared in Table 1. SHA-1 accepts messages of size less than 2^{64}, divides it into equal-size blocks of 512 bits each, processes it through 80 steps round computations and provides the final hash of 160 bits. SHA-2 follows the same structure as SHA-1, but it differs in the diversity of the output. SHA-2 has four different output options (number beside the hash represents the output length), with extra two options that take the truncated value of SHA-2/512. However, SHA-3 follows a different structure model but has the same options for the output hash as SHA-2 standard with two extra extensible output functions (SHAKE128 and SHAKE256) that can produce an output of any length. The number beside each standard reflects the output hash, but for SHAKE128 and SHAKE256 the number reflects the security level that each one of them supports against brute-force attack.

For any hash algorithm, the following properties must hold to consider it as secure:

(a) Preimage resistance: a property of easily getting the hash from a given message but difficult to extract the message back from a given hash.
(b) 2nd preimage resistance: means that it is difficult to find two messages that both generate the same hash.
(c) Collision resistance: the property of resisting the probability to generate the same hash for two different messages or more.

[1]Some references use the phrase "digest."

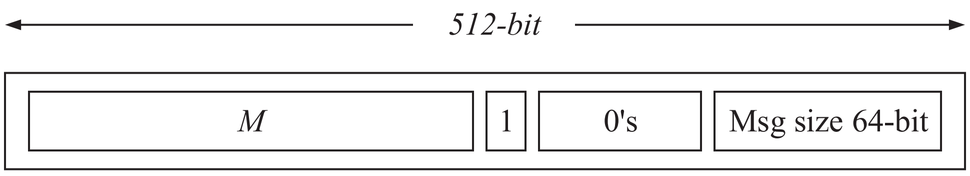


Fig. 2. Message padding mechanism.

Table 2. SHA-1 Round Functions and Constants

Round and Steps	**Round Function F *(B,C,D)***	**Round Constant (k_t)**
Round1 (0–19)	$(B \wedge C) \vee (\neg B \wedge D)$	0x5A827999
Round2 (20–39)	$(B \bigoplus C \bigoplus D)$	0x6ED9EBA1
Round3 (40–59)	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$	0x8F1BBCDC
Round4 (60–79)	$(B \bigoplus C \bigoplus D)$	0xCA62C1D6

2.1 SHA-1

SHA-1 was proposed after the MD5 hash algorithm. Despite the collision attack announced in 2017 by Stevens et al. [5], it is convenient to look into the details of SHA-1, because SHA-1 is still used by many entities and applications. SHA-1 follows Merkle Damgard (MD) structure. With 160 bits output hash, SHA-1 goes through several steps and compression operations before the output hash is produced [8].

SHA-1 takes a message of size $< 2^{64}$. The message is preprocessed by appending "1" and least number of "0*" until the message size is congruent to 448 mod 512. Then, the message size is added as a big-endian in the last 64 bits, as shown in Figure 2. Then, the padded message is divided into equal-size blocks, where they are processed sequentially using SHA-1 compression function. The compression function consists of 80 steps (0 to 79), divided into four consecutive rounds of 20 steps each. Every step t uses modular addition, left rotation, round function $\mathbf{F}_t$, and round constant $\mathbf{K}_t$, as shown in Table 2.

Each message block is partitioned into 16 consecutive words $(m_0, m_1 \ldots\ldots m_{15})$, then expanded to 80 words W_t using Equation (2):

$$W_t = \begin{cases} M_t & ,0 \le t \le 15 \\ (W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3} \oplus)_{\lll 1} & ,16 \le t, \le 79, \end{cases} \tag{2}$$

where $(\oplus)$ is logic XOR operation, and $(\lll)$ is the cyclic shift left operation.

SHA-1 has five working state variables (A, B, C, D, and E). The working state variables change after each step according to Equations (3), (4), (5), (6), and (7):

$$A_t = RL^5(A_{t-1}) \boxplus F_t(B_{t-1}, C_{t-1}, D_{t-1}) \boxplus E_{t-1} \boxplus W_t \boxplus K_t, \tag{3}$$
$$B_t = A_{t-1}, \tag{4}$$
$$C_t = RL^{30}(B_{t-1}), \tag{5}$$
$$D_t = C_{t-1}, \tag{6}$$
$$E_t = D_{t-1}, \tag{7}$$

where $(\boxplus)$ represents modular addition, and $(RL^n(X))$ is the left rotation of variable (X) by (n) bits.

The output hash is produced by the concatenation of the working state variables $(A \parallel B \parallel C \parallel D \parallel E)$ after processing the last block [7].

2.2 SHA-2

SHA-2 has four fixed output standards (224, 256, 384, 512) and two truncated versions (SHA-512/224, SHA-512/256). SHA-224 and SHA-256 work on 512-bit block size, with 16 words of 32 bits each. While SHA-384 and SHA-512 (and the truncated versions) work on a 1,024-bit block with 16 words of 64-bit word size. SHA-2 has eight working state variables (a, b, c, d, e, f, g, h), each of size equal to the word size of respective flavor [6].

Like SHA-1, SHA-2 performs padding process first by adding 1 and 0's to the end of the message followed by the message size, then divides it into 16 equal-size blocks according to the desired output hash, as depicted in Table 1. Afterward, it expands the message into 64 blocks using SHA-2 expansion equations (8), (9), and (10), for $t = 16$ to n:

$$\sigma_0 = ROTR^{r1}(W_{t-15}) \oplus ROTR^{r2}(W_{t-15}) \oplus SHR^{r3}(W_{t-15}), \tag{8}$$
$$\sigma_1 = ROTR^{q1}(W_{t-2}) \oplus ROTR^{q2}(W_{t-2}) \oplus SHR^{q3}(W_{t-2}), \tag{9}$$
$$W_t = W_{t-16} + \sigma_0 + W_{t-7} + \sigma_1, \qquad 16 \le t \le n, \tag{10}$$

where $ROTR^n(X)$ rotates word X to the right by n bits, and $SHR^n(X)$ shifts right word X by n bits. For SHA-224 and SHA-256 $n = 63$, $r1 = 7$, $r2 = 18$, $r3 = 3$, $q1 = 7$, $q2 = 19$, and $q3 = 10$, while $n = 79$, $r1 = 1$, $r2 = 8$, $r3 = 7$, $q1 = 19$, $q2 = 61$, and $q3 = 6$ for SHA-384 and SHA-512.

The eight working state variables (a, b, c, d, e, f, g, h) are initialized with fixed hexadecimal values defined by SHA-2 definition[2] [10]. Two common functions called Choose (Ch) and Majority (Maj), which are the SHA-2 manipulator Equations (11) and (12):

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z), \tag{11}$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z), \tag{12}$$

where symbols: $\oplus$, $\wedge$, and $\neg$: are the logical *XOR*, *AND*, and *NOT* operations, respectively.

To further strengthen SHA-2 hash function, two more functions are used: Sum$_0$($\sum_0$) and Sum$_1$($\sum_1$), as depicted in Equations (13) and (14):

$$\sum\nolimits_0(x) = ROTR^{r1}(x) \oplus ROTR^{r2}(x) \oplus ROTR^{r3}(x), \tag{13}$$
$$\sum\nolimits_1(x) = ROTR^{q1}(x) \oplus ROTR^{q2}(x) \oplus ROTR^{q3}(x), \tag{14}$$

where $r1 = 2$, $r2 = 13$, $r3 = 22$, $q1 = 6$, $q2 = 11$, and $q3 = 25$ for SHA-224 and SHA-256, while $r1 = 28$, $r2 = 34$, $r3 = 39$, $q1 = 14$, $q2 = 18$, and $q3 = 41$ for SHA-384 and SHA-512.

Algorithm 1 shows the full SHA-2 computations. The eight working state variables[3] W_t (a, b, c, d, e, f, g, h) change after each step, as can be seen in the Algorithm 1. The longest data path is the one corresponding to the a value, as it contains seven terms to be modified after each step. Moreover, round constant K_t appears at the same step.[4] The value of j corresponds to the number of iterations (rounds) performed by the algorithm. For SHA-224 and SHA-256, j=64; whereas for SHA-384 and SHA-512, j=80. After processing all n blocks of message M, the final message digest is obtained by concatenating all or parts of hash values $H_0^i, \ldots.., H_7^i$. The message digest for each version of SHA-2 algorithm is given by the concatenation symbol (||).

[2]Each flavor of SHA-2 has designated IHV_s different from each other.
[3]T1 and T2 are temporary variables used to calculate the value of a.
[4]Round constant K_t represents a fixed value, defined by SHA-2 definition. Each flavor of SHA-2 has its own K_t, as described in Reference [10].

ALGORITHM 1: SHA-2 Compression Function

```
   Input: Padded Message $M = \{M_0, M_1 \ldots, M_n\}$ Blocks
   Output: Output Hash (224 or 256 or 384 or 512)
1  Initialize_IHV :=
   $a = H_0^{(i-1)}$   $b = H_1^{(i-1)}$   $c = H_2^{(i-1)}$   $d = H_3^{(i-1)}$
   $e = H_4^{(i-1)}$   $f = H_5^{(i-1)}$   $g = H_6^{(i-1)}$   $h = H_7^{(i-1)}$
2  for $t \leftarrow 0$ to $n$ do
3      if $t < 16$ then
4          $W_t \leftarrow M_t$
5      else
6          $W_t \leftarrow \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$
7  for $t \leftarrow 0$ to $j - 1$ do
8      $T_1 = h + \sum_1(e) + Ch(e, f, g) + K_t + W_t$
9      $T_2 = \sum_0(a) + Maj(a, b, c)$
10     $h = g$
11     $g = f$
12     $f = e$
13     $e = d + T_1$
14     $d = c$
15     $c = b$
16     $b = a$
17     $a = T_1 + T_2$
18 for $t \leftarrow 0$ to $n$ do
19     $H_0^{(t)} = a + H_0^{(t-1)}$
20     $H_1^{(t)} = b + H_1^{(t-1)}$
21     $H_2^{(t)} = c + H_2^{(t-1)}$
22     $H_3^{(t)} = d + H_3^{(t-1)}$
23     $H_4^{(t)} = e + H_4^{(t-1)}$
24     $H_5^{(t)} = f + H_5^{(t-1)}$
25     $H_6^{(t)} = g + H_6^{(t-1)}$
26     $H_7^{(t)} = h + H_7^{(t-1)}$
27 return Hash
28 SHA-224 $\leftarrow H_0^{(n)} \| H_1^{(n)} \| H_2^{(n)} \| H_3^{(n)} \| H_4^{(n)} \| H_5^{(n)} \| H_6^{(n)}$
29 SHA-256,512 $\leftarrow H_0^{(n)} \| H_1^{(n)} \| H_2^{(n)} \| H_3^{(n)} \| H_4^{(n)} \| H_5^{(n)} \| H_6^{(n)} \| H_7^{(n)}$
30 SHA-384$\leftarrow H_0^{(n)} \| H_1^{(n)} \| H_2^{(n)} \| H_3^{(n)} \| H_4^{(n)} \| H_5^{(n)}$
```

2.3 SHA-3

SHA-3 was published in 2012, after a competition that was held by the NIST. Five candidates were selected for the final round (Keccak, Grøstl, BLAKE, JH, and Skien). Keccak won the competition as the next standard for SHA-3 [10]. Unlike the previous standards (SHA-1 and SHA-2), SHA-3 relies mainly on absorb and squeeze structure [23, 24], as shown in Figure 3. Sponge structure works on a state of $b = r + c$ bits, where r is the bit-rate and c is the capacity. This state is initialized with zeros. An input string is padded to make its size divisible by r. Then the padded string is divided into equal-size blocks (P_0, P_1, ..., P_i) each of size equal to r-bit. In the absorbing phase, each

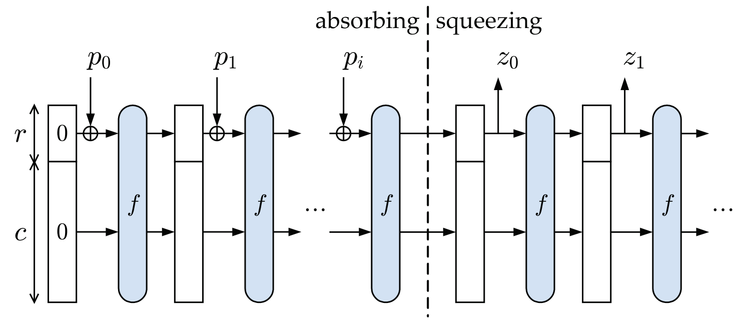


Fig. 3. Sponge construction [24].

block is XORed with the first r bits of the state b, manipulated with the permutation function f. After processing all blocks, the sponge operation toggles to the squeezing phase. In the squeezing phase, the least significant r bits of the state b are selected as output blocks (z_0, z_1, …). If the required output length is less than or equal to z_0 it will be taken from the least significant bits of z_0. Otherwise, the permutation function f is applied to the output of z_0 to produce z_1, where $z_0||z_1$ will be considered to produce the final hash (224, 256, 384, 512). This will continue until the required number of output bits is achieved [23]. Extra permutation operations are applied in case of arbitrary length (as in (SHAKE128 and SHAKE256).

SHA-3 Keccak supports two modes of operations. A fixed output hash mode support hashes (224, 256, 384, 512), and a variable-length hash mode (SHAKE128 and SHAKE256) produces an output of variable length according to the desired application.[5] In Keccak, b represents the permutation function's level (25, 50, 100, 200, 400, 800, 1,600). Yet, the most common permutation is (b = 1,600, l = 6), where b is computed using Equation (15):

$$b = 25 * 2^l, \qquad l = 0, 1, 2, 3, 4, 5, 6. \tag{15}$$

In the fixed mode operation, two groups are allowed. Group-1 ($r = 1,344, c = 256$) is used to compute hash values of length 224 and 256, while group-2 (r = 1,088, c = 512) is used to calculate the hashes of length 384 and 512 [25].

The fixed-size output hash is taken from the first step of the squeezing phase (z_0) by selecting the least significant bits (according to the desired hash output 224, 256, 384, and 512). When variable length hash is required, all bits of Z can be used according to the desired output. Moreover, the output can be taken from any Z_i [10].

The function of Keccak is depicted in Figure 4. The state $b = r + c$ is initialized with 0's. The length of b depends on the level of the permutation selected. As mentioned earlier, b = 1,600 is the most commonly used permutation. Each block is processed through several rounds that are determined by the l value, according to Equation (16):

$$Rounds = 12 + 2l, \qquad l = (0, 1, 2, 3, 4, 5, 6). \tag{16}$$

Keccak handles the state b as a 3D-Matrix ($A \times B \times C$), which is shown in Figure 5. Each of Keccak's rounds has distinct constant $RC[i]$ used inside permutation function. Figure 4 shows that each round consists of five steps denoted by Greek letters, $theta(\theta)$, $rho(\rho)$, $pi(\pi)$, $chi(\chi)$, and $iota(\iota)$. Each step manipulates the state matrix ($A \times B \times C$). The values of A and B are fixed to 5, while the value of C is represented by w according to Equation (17):

$$w = 2^l, \qquad l = (0, 1, 2, 3, 4, 5, 6). \tag{17}$$

[5]The numbers (128 and 256) reflect the security strength of each one of them.

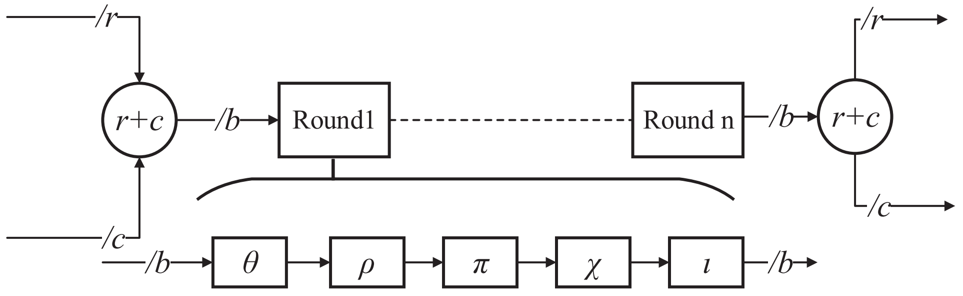


Fig. 4. Keccak round function, five steps are repeated for each round. The operation goes through theta (θ), rho (ρ), pi (π), chi (χ), and iota (ι) steps [25].

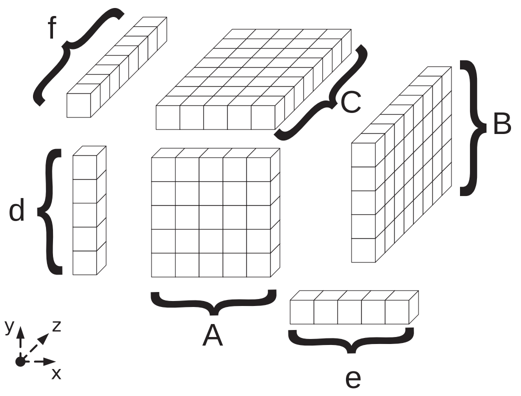


Fig. 5. Keccak state matrix ($A \times B \times C$), represented as 3D-Matrix. Each square represents one bit: (A) slice, (B) sheet, (C) plane, (d) column, (e) row, (f) lane. The individual figures listed here were taken from Reference [23].

The resultant b Matrix: $b = 5 \times 5 \times w$ bits. If l is chosen to be 6, then the matrix becomes $5 \times 5 \times 64 = 1{,}600$ bits, consequently number of rounds will be 24.

- *Theta* (θ) step: which operates on a 2D-Array (5×5), where each element contains w bits. A single 5×5 array can be seen as a slice, where the 1D-array of w bits is a lane, as shown in Figure 5. Theta step manipulates the state array according to Equation (18), where $C[x]$ and $D[x]$ represent lanes and $A[x, y]$ represents slice. Theta computes the parity of each column, then combines them with the XOR operator using Equation (18):

$$step(\theta) = \begin{cases} C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] & , x = 0,1,2,3,4, \\ D[x] = C[x-1] \oplus ROT(C[X+1],1) & , x = 0,1,2,3,4, \\ A[x,y] = A[x,y] \oplus D[x] & , x = 0,1,2,3,4. \end{cases} \tag{18}$$

- *Rho* (ρ) step, this step rotates one element (lane) of the state matrix $A[x, y]$ (which is 5×5) by i bits, as seen in Equation (19). The rotation offset value denoted by $r[x, y]$ is a constant value assigned according to Table 3.

$$step(\rho) = ROT(A[x,y], r[x,y]), \qquad (x,y) = 0,1,2,3,4. \tag{19}$$

- *Pi* (π) step: is a complement step to *rho*, as it takes the rotated lanes from *rho* step and puts them in different positions in the array matrix ($B[x, y]$) without modifying any value. It only

Table 3. The Constant Value r[x,y] in *Rho* (ρ) Step

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	25	39	3	10	13
$y = 1$	55	20	36	44	6
$y = 0$	28	27	0	1	62
$y = 4$	56	14	18	2	61
$y = 3$	21	8	41	54	15

Table 4. A Comparison between Different Hardware Choices

Hardware	Integration	Property	Advantage	Disadvantage
CPU	Desktop or laptop computers	Number of cores	Testing using software	Side channel attack, General purpose
GPU	Desktop, laptop, or accelerated supercomputer	Number of cores	Parallel computing and tested using software	Difficult programming
FPGA	Stand-alone part of ASIC	Application-specific	Easy to optimize, reconfigurable	Experience in HDL
ASIC	Stand-alone	Application-specific	Low price, reconfigurable	Pre-planning before design
HSS	Stand-alone or embedded	Combined design	Chose the advantage of the best hardware	Price and integration

permutes the matrix according to Equation (20):

$$step(\pi) \Rightarrow B[y, 2x + 3y] = ROT(A[x, y], r[x, y]), \tag{20}$$

where $x, y = 0, 1, 2, 3, 4$.

- *Chi* (χ) step: in this step, the B matrix, which was generated from the previous step, is manipulated according to Equation (20) and puts the result back in array matrix A according to Equation (21):

$$step(\chi) \Rightarrow A[x, y] = B[x, y] \oplus ((\bar{B}[x + 1, y]) \wedge B[x + 2, y]), \tag{21}$$

where $x, y = 0, 1, 2, 3, 4$, $\wedge$ bit-wise AND operation, and $\bar{B[]}$ is the bit-wise complement of B.

- *Iota* (ι) step: adds the round constant $RC[i]$ to the state matrix A at location $A[0, 0]$, where each round has a distinct 64-bit round constant. *Iota* step is represented by Equation (22):

$$step(\iota) \Rightarrow A[0, 0] = A[0, 0] \oplus RC[i], \tag{22}$$

where $RC[i]$ are 24 different 64-bit round constants, depicted from Reference [26].

3 HARDWARE IMPLEMENTATIONS OF THE SHA STANDARDS

Hash algorithms can be implemented in software and hardware. However, performance in hardware becomes an important criterion for breaking a tie between algorithms when all the security-related parameters are equally good. In the subsequent text, a discussion about different hardware that is used to implement SHA standards will be presented.

3.1 Choice of Hardware to Implement SHA

Different factors are taken into consideration to choose hardware for implementation. Table 4 provides a summary of different choices to implement cryptography protocols along with their advantages and disadvantages.

Central Processing Unit (CPU) is used in desktop and laptop computers. The main property of CPUs is the number of cores inside the processor. As CPUs are general purpose units, cryptography applications share the CPU resources with other applications. Graphics Processing Unit (GPU) is part of general-purpose units that can be found in desktop, laptop, or accelerated supercomputers. The GPUs run algorithms in parallel, thus giving more computation power than CPUs. Field Programmable Gate Arrays (FPGAs) are considered as Application Specific Integrated Circuit (ASIC), because it can be used to run single applications [27]. The FPGAs are used to implement a predefined function. Moreover, the FPGA is considered as reconfigurable hardware that is configured at the end-user by changing the layout of implementations. ASIC is manufactured to run exactly one application. Because ASIC is small in size, it is used when processing speed is important. ASICs can be reconfigured to run different applications. Moreover, ASIC's price is the lowest between the different hardware choices, because it comes in a small design package. Hybrid Hardware System (HHS) is the composition of different types of aforesaid hardware and are integrated to work together to form one powerful cryptographic system. The HHS gives better results in terms of performance but consequently increases the price [16, 28–30].

Different factors need to be considered when choosing any hardware: security, speed, and price. Security represents the resistance of a hardware against any kind of attack. CPUs and GPUs are vulnerable to side channel attacks, while the FPGAs are more secure, because they run for a specified process as ASIC. The second factor is the speed where it measures the time needed to finish the job. CPUs are capable of performing several operations and so cannot be too much optimized in one direction. The FPGAs are faster than CPUs and GPUs, because they are dedicated to a specific task. ASICs are the best in terms of speed, as they are flexible for design; however, the small area limits them from being used for bigger tasks. The third factor is the price. CPUs are cheap to be programmed and get the programs to run quickly. The GPUs are also quite easy to obtain, a bit more expensive to effectively program, and capable of efficient execution of programs. The FPGAs are more expensive and require the design of an algorithm using hardware description language (VHDL, Verilog), but once programmed they are easily reconfigured. The ASICs have a long design cycle, but once completed they can be manufactured easily and for a low price [14, 31]. The choice of FPGAs arises from the fact that they support different types of design methodologies, namely: software design methodology, hardware design methodology, and software with hardware design methodology. The software design methodology is supported using the Environment Development Kit (EDK) tool that comes with the FPGA hardware. The hardware design methodology is the ability of FPGA to support different levels of programming languages, e.g., VHDL and C++. Software with hardware methodology is the ability of FPGA to integrate software and hardware. Because FPGA is reconfigurable, it can be used for fast prototyping, which is reflected on the design cost [32]. According to the aforementioned factors, the FPGA is the best choice for implementing and optimizing SHAs.

3.2 FPGA Performance Metrics

The FPGAs can be as simple as a logic gate or complicated architecture as a microprocessor. The FPGA architecture is based on one of five components: transistors, logic gates, multiplexers, lookup tables (LUTs), and wide fan-in (AND-OR) structure. To explore more about FPGA, see Reference [33].

The FPGA implementation of the secure hash standards requires speed and memory, because it maintains many computations during hash generation. Therefore, optimizing FPGA designs in terms of speed and memory is crucial, because they affect the FPGA performance [34]. To measure the performance of FPGA, the following metrics need to be taken into consideration:

(1) Area: Reflects the total number of Configurable Logic Block (CLB) or Look Up Tables (LUTs) used for any design implementation. However, in some cases, the area comparison might be an unfair factor because of the differences in placements methodology of different FPGA manufacturers. Therefore, to compare area requirements, the same FPGA's manufacturer is recommended [34].
(2) Frequency: The operational clock frequency that a given FPGA can run.
(3) Throughput: Used to measure the speed of hardware implementation, which represents the number of bits processed in a given time. Throughput is defined by Equation (23):

$$Throughput = \frac{Block\_Size}{T * N_{clk}}, \tag{23}$$

where $Block\_Size$ is the total number of bits of a message block, T is the clock period, and N_{clk} is the number of clock cycles to hash a block.
(4) Throughput to Area Ratio: The comparison between performance metrics is recommended according to a fair factor. Taking the ratio of throughput to the area requirements can provide a fair comparison between different kinds of FPGAs [35].
(5) Power Consumption: Reflects the total amount of power consumed by the hardware when applying a specific design. Conventionally, this property is defined by the frequency level.

SHA hash standards were discussed in different works. Some of these works were dedicated to specific SHA standards [36–39], while the others compare more than one standard in their works [20]. In the next section, we categorize all optimization techniques of different hash standards on FPGA according to their influence on the optimization metrics. For each hash standard, the effects of optimizations will be addressed comprehensively.

4 OPTIMIZATION TECHNIQUES

The leading manufacturers of the FPGA market are Xilinx and Altera (now Intel). Nowadays, both of them control more than 90% of the FPGA market [40]. Therefore, the majority of hardware implementations of secure hash algorithms are deployed using Xilinx and Altera FPGAs. To optimize FPGA implementation, six optimization techniques are considered; (1) Carry Save Adder (CSA), (2) Pipelining, (3) Unrolling (unfolding), (4) FPGA resources, (5) Iterative method, and (6) Combined Optimization. The details about these methods will be discussed in the subsequent text.

The equations and structures of the SHA algorithms showed that all operations were based on mathematical and logical functions. Therefore, improving these operations will enhance the computational resources that are used during the hardware deployment. For instance, addition is the core math operation in SHA-1 and SHA-2, so enhancing the adder units will lead to a significant enhancement to the system [41]. This survey categorizes the FPGA optimization techniques of SHA algorithms according to the type of optimization. The optimization techniques of FPGAs are laid in one of the following categories:

(1) Carry Save Adder (CSA): is a small and fast digital adder, used in implementations to compute the sum operation of binary digits. CSA produces an output of size same as the size of input words [42]. CSA enhances the area and maintains throughput in some cases.
(2) Pipelining: is a data processing technique that processes a series of connected elements in parallel in a timed fashion [43]. Pipelining combines multiple steps into one step unless

they have data dependency between them. Pipelining can be used with all hash standards' optimization methods. It provides significant improvements in terms of throughput along with maintaining area requirements in some cases.

(3) Unrolling (Unfolding): is a loop transformation method that attempts to optimize the program execution speed. The speed optimization is accomplished by eliminating some loop control statements from programs [44].
(4) FPGA Resources: are predefined blocks that are built inside the FPGA. Moreover, FPGAs support the option of using extra or external resources. These resources include Block RAM (BRAM), Digital Signal Processing (DSP) unit, and Shift Register Look-up table (SRL) [45]. The purpose of BRAM is to store constants and move the combinational logic block burden to BRAM burden. The DSP unit is used to perform part of the logical and mathematical calculations, while the SRL is used to reduce time and area requirements.
(5) Iterative Method: is defined as mathematical operations that try to get the solution according to a series of improved approximations. Regarding FPGA optimization, the iterative method is deployed according to a specific performance metric and deploys a series of different optimization techniques [46].
(6) Mixed Optimization: where two or more optimizations are combined to form an extra optimized version.

All SHA hash standards are bounded by the dependencies between internal rounds. Through several types of optimizations, the most significant techniques were those that provided a good enhancement in the performance metrics: particularly, Throughput (Tp), Area (A), and Throughput/Area (Tp/A) ratio. In this section, we will discuss previous works and their contributions in this field.

4.1 FPGA Implementation of SHA-1

This section presents the various FPGA design implementations of SHA-1 hash standard. SHA-1 has a data dependency between internal steps, so any improvement to SHA-1 is relevant to this property. SHA-1 compression function has a critical path equation[6] that relies mainly on add operation. Carry Save Adder (CSA) is used in hardware optimization to enhance the performance of add operation. The CSA is small and fast, because it separates the addition and carry operations, thereby minimizing the delay time caused by the carry. Optimization through the CSA is frequently used in the implementation of SHA-1 and SHA-2 standards, as both of them rely mainly on addition inside critical path [20]. CSA is used along with Carry Look-ahead Adder (CLA) in some optimizations to pre-compute the critical path, which in turn reduces the addition operations and CSA units. The combination of CSA and CLA reduces the computation delay and saves more area resources. This scheme was adopted by Makkad et al., where CSA and CLA were used to optimize the implementations of unfolded, pre-computation, and four-stages pipeline [47]. CSA is used to compute the intermediate values between steps and reduce the computation delay. The proposed designs were tested and verified using Xilinx Virtex-6 (LX240T) FPGA, and Xilinx ISE 13.2 Design suit. The results showed that the four-stages pipeline design with CSA gave a good result in term of Throughput with 8,607.6 *Mbps*; while pre-computation, unfold, and CSA-basic architectures gave 2,596.2 *Mbps*, 1,927.5 *Mbps*, and 1,347.2 *Mbps*, respectively. Contrarily, the results showed that CSA-basic design gave the best figures in terms of area requirements, as depicted in Table 5. Another work that conducted a comparison between different optimization methods including CSA was presented in Reference [47]. The authors used CSA, loop unfolding (two operations),

[6]The critical path is the longest equation of the compression function as discussed in the SHA-1 and SHA-2 definitions.

Table 5. Comparison between Different FPGA Optimization Techniques to Implement SHA-1

Work	Opt-Tech[c]	FPGA[b]	Tp/Gbps	Area/slice	Freq/Mhz	Mbps/Slice
[47]	CSA	Virtex-6	1.3472	449	213.13	3.0004
	Unfold		1.9275	518	154.35	3.721
	4PPL		8.6076	1230	172.32	6.998
[48]	PPL	Virtex	2.5267	950	98.7	2.659
[52]	Basic with (4,8,10,20,40)[a] PPL	Virtex	(1.18, 2.35, 2.85, 5.73,-)	(2070, 3920, 4950, 9570, -)	(45.9, 45.9, 44.6, 44.8, -)	(0.57, 0.60, 0.57, 0.59, -)
		Virtex-E	(1.46, 2.93, 3.53, 6.76, 13.08)	(2140, 4030, 5060, 9430, 17690)	(57.0, 57.3, 55.2, 52.8, 51.1)	(0.68, 0.73, 0.69, 0.72, 0.74)
		Virtex-2	(1.82, 3.65, 4.12, 9.09, 17.92)	(2000, 3820, 4800, 9860, 18650)	(71.1, 71.2, 64.3, 71.0, 70.0)	(0.91, 0.95, 0.86, 0.92, 0.96)
		Virtex-4	(3.09, 6.19, 7.01, 14.18, 28.24)	(1990, 3930, 4750, 9000, 17560)	(120.7, 120.9, 109.6, 110.8, 110.3)	(1.55, 1.57, 1.48, 1.57, 1.61)
	Opt (4,8,10,20,40) PPL	Virtex	(3.05, 6.09, 7.46, 14.47,-)	(2490, 4750, 5940, 11060,-)	(59.3, 59.5, 58.3, 56.9, -)	(1.19, 1.28, 1.26, 1.31, -)
		Virtex-E	(3.79, 7.59, 9.23, 17.95, 38.35)	(2560, 4840, 6070, 11310, 20170)	(74.1, 74.1, 72.1, 70.1, 74.9)	(1.48, 1.57, 1.52, 1.59, 1.90)
		Virtex-2	(2.0, 3.97, 7.92, 15.87, 39.07)	(2410, 4590, 5770, 11860, 22000)	(92.4, 92.4, 83.5, 92.3, 95.5)	(0.83, 0.86, 1.37, 1.33, 1.77)
		Virtex-4	(8.05, 16.10, 18.21, 36.89, 88.37)	(2390, 4560, 5740, 10830, 20190)	(157.2, 157.2, 142.3, 144.1, 172.6)	(3.37, 3.53, 3.17, 3.41, 4.37)
[50]	Unfold	Virtex-2	5.9	2894	118	2.038
[51]	4x-Unfold	Virtex 2	0.983	2394	20.9	0.41
	4x-Unfold+4PPL		3.541	4258	41.5	0.83
[49]	(1, 4, 40)-PPL	Arria II	(1.746, 7.911, 80.306)	(402, 976, 6715)	(279.64, 316.76, 321.54)	(4.343, 8.11, 11.96)
		Virtex-5	(1.46, 10.740, 10.67)	(897, 1332, 6465)	(233.899, 430.024, 427.54)	(1.628, 8.06, 16.51)
	2x-Unfold+40PPL	Arria II	150.269	9799	308.17	15.335
		Virtex-5	223.618	11994	458.593	18.644
[53]	Parallel process	65nm FPGA	14.9	3986	236	3.74
[54]	Three module + SDK	Virtex-5	0.786	1351	124.502	0.58
[55]	HLS for AOCL	Stratix-5	3.033	-	-	-
[47]	Pre-Computation	Virtex-6	2.596	546	207.90	4.75
[56]	Basic	Virtex(4,5,6,7)	(4.0, 5.0, 5.1, 5.9)	(7737, 3899, 3787, 3743)	(77.9, 98.3, 99.8, 115.6)	(0.52, 1.28, 1.35, 1.58)
	Opt		(9.2, 11.9, 12.8, 14.3)	(9623, 4275, 4129, 4133)	(90, 116.3, 124.7, 139.6)	(0.96, 2.78, 3.10, 3.46)

[a]Each value in parentheses refers to its Pipelining level. The (Tp, Area, Freq, and Mbps/Slice) for each level represented in parentheses in the same order at the same raw.
[b]Virtex refers to Xilinx, while Arria refers to Altera.
[c]Basic: Implementation without optimization. CSA: Carry Safe Adder. xPPL: Pipelining to x level. Nx-Unfold: Unfolding method with N unrolled loops.
−Refers to a non-reported result.

pre-computation, and pipelining technique. The proposed work showed that four-stages pipeline exceeded the others by throughput but preceded them in terms of area requirements; while CSA optimization gave significant figures in terms of area, but the increased operational frequency was obvious, which in turn increases the power consumption.

The pipelining method gives an enhanced performance in terms of throughput, but influences the area requirements. Authors in Reference [48] proposed a targeted design approach, focusing on the increase of operating frequency (f-operation) and throughput. They focused on maintaining the area requirements without introducing a significant area penalty. The proposed methodology employed hardware reuse and pipelining techniques. The hardware reuse benefited from the predefined units without creating a new one for different operations. Moreover, the proposed design used multiplexing to control the flow of data and manage the units' usage. The proposed design curtailed the critical path of SHA-1 algorithm into two stages (addition and multiplexing). The authors tested their design on Xilinx Virtex FPGA. The results showed an improvement in terms of speed without significant effect on area requirements. The speed of optimization exceeded 2.5 *Gbps* throughput with an increased percentage by 37% over the basic structure and 950 slices of the area. Sometimes, the throughput requirements are much more important than area requirements, thus the designers focus only on optimizing the throughput. Pipelining can also be used jointly with other optimization methods to further improve the overall performance requirements of the system. Suhili et al. in Reference [49] proposed high speed and throughput design for SHA-1 using pipelining with unfolding technique. Five evaluation designs were tested using Xilinx Virtex-5 and Altera Arria-2 FPGAs. The designs used iterative, inner-round pipelining, four-stage pipelining, 40-stage pipelining, and 40-stage pipelining with two times unrolling ($40PPL + 2X$-Unroll). Pipelining is used to maintain the data dependency between internal steps of the compression function, while the unrolling expands the internal loops of function. Then, the combination will give the pipelining more space to manage the data dependencies. However, this kind of combination dramatically increases the area requirements. The proposed designs gave the best results for 40-stage pipelining combined with two times unrolling design, with a claimed throughput of 150.269 *Gbps* and 223.618 *Gbps* for Altera and Xilinx FPGAs, respectively.

One of the key enhancements to reduce the critical path of SHA-1 is the unrolling (unfolding) method. Loop Unrolling increases the number of instructions but increases the parallelism level. In Reference [50], the authors proposed an architecture to achieve higher parallelism and minimize the critical path. The proposed design employed two unfolded architectures: a pre-computation and hash core. The pre-computation architecture of SHA-1 used newly defined parameters that are pre-computed before other parameters. The hash-core architecture is used to compute the nth hash operation of the internal hash computation steps. Both designs were tested and verified using Xilinx Virtex-2 (xc2v1000) FPGA with 5.9 *Gbps* throughput and 2,894 of area slices. However, the work presented in Reference [51] tested unfolding architecture up to eight stages. Authors applied their work for SHA-1 and combined their work with the pipelining method. Their work was tested and verified using Xilinx Virtex-2 FPGA. The results showed a significant improvement in using unfolding along with pipelining better than using unfolding alone. The better results were achieved by using unfolding that reduced the number of required clock cycles to 12 for unpipelined version and 24 cycles for pipelined version. The results gave 3.541 *Gbps* throughput and 4,258 area slices using 4-stages unfolding with pipelining. While 893 Mbps throughput and 2,394 area slices were achieved without pipelining. As shown in Table 5, The enhancements are not guaranteed with the unrolling method unless they are used with a predefined criterion.

Since SHA-1 and SHA-2 follow the same construction model, both can be combined in one system, as presented by Michail et al. [52]. The proposed work was based on a pipelined design for area-throughput trade-offs for SHA-1 and SHA-256. The work compared between the optimized and non-optimized pipelined designs. The optimized version employed a loop unrolling technique, which unrolled two iterations with one mega step using pre-computation and CSA units. Both of the designs were implemented using four FPGAs: Virtex (xcv1000-6FG680), Virtex-E (xcv3200e-8FG1156), Virtex-II (xc2v6000-6FF1517), and Virtex-4 (xc4vlx100). The speed was used as an optimization goal for both base and optimized versions. The authors claimed that the best results for base architecture were those produced by Xilinx Virtex-4 FPGA, with 66.36 *Gbps* and 56.93 *Gbps* for SHA-1 (80PPL) and SHA-256 (64PPL), respectively. While the best throughput of the optimized versions were 88.37 *Gbps* with 40-stages PPL, and 69.27 *Gbps* with 32-stages PPL, on the same FPGA.

As well, the need for low-power designs is crucial in case of big data processing. Isobe et al. in Reference [53] implemented all processes of TLS/SSL into one FPGA. Low power consumption and high speed were gained by dividing processes into three phases. Phase one was a design of the RSA unit using parallel processing. Phase two included shared key cryptography and hash function unit for sending and receiving. While phase three employed a protocol processing block, cipher processing block, and data exchange block. The proposed work gave enhanced throughput figures without increasing the power consumption factor. Their work maintained the operating frequency of 65*nm* FPGA to the same levels of non-optimized version, as shown in Table 5.

Many designs were used to evaluate and validate SHA standards on Computer-Aided Design (CAD) tools [54, 55]. The work performed by Iyer et al. [54] introduced three modules (initial, round, and top) to model SHA-1 algorithm using FPGA. The evaluation and synthesis of SHA-1 algorithm were applied using the Xilinx Software Development Kit (SDK) toward Virtex-5 FPGA. The CAD tool was used to accelerate the calculation of SHA-1. Results gave a better speed than the regular processor and comparable figures with High-Level Synthesis (HLS), in terms of throughput. Janik et al. in Reference [55] used HLS tools for Altera OpenCL (AOCL) to accelerate the computations of SHA-1. The design was tested and verified using Quartux SDK toward Altera Stratix-5 FPGA. The results showed an enhancement in the system speed over CPU and GPU.

As mentioned before, SHA-1 and SHA-2 follow the same construction model, therefore, both of them can be combined on the same FPGA implementation. Michail et al. [56] proposed a high throughput and area-efficient multi-mode secure hash algorithm using the FPGA. Their work supported SHA-1 and SHA-2 (256, 512) output hashes. The system was able to produce hash according to the user selection of SHA-1 or SHA-2. The design was tested and verified using different FPGAs (Virtex-4, 5, 6, and 7). The proposed architecture employed pre-computation and pipelining to optimize the resulting design. The experiments were carried out for the base and optimized structures and proved the progression of the optimized architecture.

4.2 FPGA Implementations of SHA-2

The data dependency between internal steps of SHA-1 and SHA-2 plays a key role in performance improvements. SHA-2 follows the same construction model as SHA-1, so any improvement to SHA-1 is naturally applicable to SHA-2 [56]. The CSA is used in SHA-1 to speed up the calculation of the critical path. Similarly, the CSA was adopted by different works to optimize the FPGA implementation of SHA-2 [57–59]. Sun et al. proposed an architecture to optimize critical path calculation using the CSA [57]. The proposed design supports different SHA-2 flavors (256, 384, and 512) using the control-selection unit. The architecture was tested using *ModelSim*6.0*a* and targeted to Xilinx Virtex-2. The results showed improvements in terms of throughput and area comparing with SHA-1, as shown in Table 6.

To overcome the overhead of add operations, Mohamed et al. in Reference [58] used Carry Propagation Adder (CPA) with CSA. The longest critical data paths contain six and seven add operations, whereby the CPA is used to overcome the overhead of the modular addition computa-

Table 6. Comparison between Different FPGA Optimization Methods to Implement SHA-2

Work	Hash	Opt-Tech	FPGA	Tp/Gbps	Area(A)/slice	Freq/Mhz	Tp/A
[61]	256	Conrol-Unit	Virtex-5	1.58 Gbps	387	202.54	4.08
[57]	256	CSA	Virtex	0.291	2207	74	0.132
	384			0.250			0.113
	512			0.467			0.212
[52]	256	Basic with (2,4,8,16,32, 64) PPL	Virtex	(0.75, 1.50, 2.91, 5.70, -, -)	(1,330, 2,390, 4,580, 8,990, -, -)	(46.2, 46.9, 45.4, 44.5, -, -)	(0.56, 0.63, 0.64, 0.63, -, -)
			Virtex-E	(0.83, 1.66, 3.25, 6.35, 12.83, 29.85)	(1,360, 2,400, 4,790, 9,090, 17,810, 29,990)	(51.8, 51.8, 50.8, 49.6, 50.1, 58.3)	(0.61, 0.69, 0.68, 0.70, 0.72, 0.99)
			Virtex-2	(1.00, 2.00, 3.97, 7.92, 15.87, 39.07)	(1,440, 2,600, 5,290, 10,660, 18,980, 27,610)	(62.4, 62.4, 62.1, 61.9, 62.0, 76.3)	(0.69, 0.77, 0.75, 0.74, 0.84, 1.42)
			Virtex-4	(1.74, 3.48, 6.93, 13.82, 27.65, 56.93)	(1,580, 2,970, 6,290, 12,120, 23,520, 31,420)	(108.7, 108.8, 108.3, 108.0, 108.0, 111.2)	(1.10, 1.17, 1.10, 1.14, 1.18, 1.81)
		Opt with (2,4,8,16,32) PPL	Virtex	(1.83, 3.67, 6.95, 13.67, -)	(1,720, 3,100, 5,770, 11,900, -)	(57.3, 57.3, 57.3, 54.3, 53.4)	(1.06, 1.16, 1.20, 1.15, -)
			Virtex-E	(1.98, 3.97, 7.81, 14.98, 32.36)	(1,770, 3,160, 5,890, 12,160, 19,410)	(62.0, 62.0, 61.0, 58.5, 63.2)	(1.12, 1.25, 1.33, 1.23, 1.67)
			Virtex-2	(2.38, 4.77, 9.51, 18.97, 40.50)	(1,940, 3,500, 6,620, 13,780, 20,960)	(74.5, 74.5, 74.3, 74.1, 79.1)	(1.23, 1.36, 1.44, 1.38, 1.93)
			Virtex-4	(4.16, 8.33, 16.64, 33.20, 69.27)	(2,060, 3,960, 7,780, 15,260, 26,340)	(130.1, 130.1, 130.0, 129.7, 135.3)	(2.02, 2.10, 2.14, 2.18, 2.63)
[58]	256	CSA	Virtex-5	1.3596	1,203	170	1.13
[59]	256	CSA	Virtex-2	0.86772	1,187	110.10	0.731
		CSA+ Pre-Comp		0.909	1,274	115.46	0.713
[37]	512	5PPL	Virtex-2	6.989	7,012	54.6	2.659
			Virtex-E	9.126	7,151	63.4	1.276
			Virtex-6	9.126	7,151	71.3	1.276
			Virtex-7	11.674	7,219	91.2	1.617
[74]	256	DSP+BRAM	Stratix-3	1.621	795	205.8	2.03

(Continued.)

Table 6. Continued

Work	Hash	Opt-Tech	FPGA	Tp/Gbps	Area(A)/slice	Freq/Mhz	Tp/A
[60]	256	Basic+PPL	Virtex-2	1.009	1,373	133.06	0.735
		2x-unfold+PPL		0.997	2,032	73.975	0.491
		4x-unfold+PPL		0.909	2,898	40.833	0.314
	512	Basic+PPL		1.329	2,726	109.03	0.488
		2x-unfold+PPL		1.466	4,107	65.89	0.357
		4x-unfold+PPL		1.364	5,807	35.97	0.235
[56]	256 and 512	Basic	Virtex(4,5,6,7)	(4.0, 5.0, 5.1, 5.9)	(7,737, 3,899, 3,787, 3,743)	(77.9, 98.3, 99.8, 115.6)	(0.52, 1.28, 1.35, 1.58)
		Opt		(9.2, 11.9, 12.8, 14.3)	(9,623, 4,275, 4,129, 4,133)	(90, 116.3, 124.7, 139.6)	(0.96, 2.78, 3.10, 3.46)
[62]	256	Data Reuse	Virtex	0.649	431	35.5	1.5
			Virtex-4	0.915	422	50.06	2.17
			Virtex-5	1.18	139	64.45	8.49
[63]	All Flavors	Iterative	Virtex-6	1.063	766	132.91	1.38
		PPL		2.047	871	271.96	2.35
		Iterative		1.081	736	135.14	1.47
		PPL		2.040	905	271	2.25
		Iterative		1.496	1,651	116.9	0.91
		PPL		2.445	1,724	200.64	1.42
		Iterative		1.428	1,613	111.56	0.89
		PPL		2.348	1,811	192.57	1.31

*Please refer to the definitions in the bottom of Table 5.

tions. The optimization was performed on two levels, the Look-Up Table (LUT) level and the CSA level. The methodology was tested using Xilinx Virtex-5 FPGA (xc5vlx330t-2ff1738). The results showed that, in terms of delay, the optimization on LUT level gave better results than the CSA level. The same throughput of 1,359.6 *Mbps* was observed for both designs. While Algredo et al. in Reference [59] proposed two hardware architectures to compute the inner loop of SHA-2 using the CSA. The first architecture was based on re-arranging the data-flow of the critical path equations. The second architecture was based on two pre-computations instead of one. The proposed approaches relied mainly on the use of the CSA that was used for balancing data paths, and a state buffer. The systems were tested and verified using Xilinx Virtex-2 (XC2VP-7) FPGA. Both designs gave the same figures in all terms, as shown in Table 6.

To take advantage of the pipeline method, which is used to optimize the speed and throughput, authors in Reference [37] proposed a pipelined architecture to optimize SHA-2 in terms of throughput and area. To achieve the goal, a set of optimization techniques was applied, systematically, according to two levels. The first one is the algorithmic level, which includes loop unrolling and pre-computation. The second level involves circuit-level technique, which includes resource re-arrangement and use of special circuit resources such as CSA. Both levels were combined with thoroughly applied stage pipelining. The designs were tested and verified using Xilinx Virtex-(2, E, 6, and 7) FPGAs. The proposed techniques showed the best results with the 5-stage pipeline in terms of throughput and throughput/area.

Loop Unrolling (Unfolding) method is also adopted to optimize SHA-2 hash standard. Multiple rounds of the compression function are unrolled and processed in combinational logic components, which reduces the clock cycles required to compute hash function [36]. However, Loop Unrolling method gives no significant results in some cases, unless it is used with other optimization methods. For instance, unroll internal loops of compression function for multiple times will cause an extra area penalty. So the better use of loop unrolling is to combine it with other methods. Authors in Reference [60] proposed a design that combines unrolling optimization with pipelining. a new VLSI architecture was presented that combines a fast quasi-pipelined design with unrolling. The authors tested their architecture to find the best pipeline-unroll combination. The design was tested and verified using FPGA Xilinx Virtex-2 FPGA. The results showed that SHA-256 pipelined without unrolling gave the best throughput (1.009 Gbps), while the best result for SHA-512 was the unrolled two times with pipelining (1.466 Gbps).

According to the similarity of the internal construction of SHA-1 and SHA-2, any optimization or implementation toward SHA-1 will work with SHA-2 [56]. Some designs employ a hardware unit to optimize and validate SHA-2 hash function [61, 62]. In Reference [61], the authors proposed an efficient hardware implementation for SHA-256 and SHA-512 using Xilinx Virtex-5. The proposed work employs a control unit to manage the flow of data from the padding unit and passes the padded data to the hash computation unit. The process speeds up the hash production and yields better efficiency. However, a compact FPGA implementation for SHA-2 hash function was proposed in Reference [62]. The work designed a customized processor based on FPGA. The design relied on the data reuse to minimize memory access with the help of cache memory. The results showed an improvement in the critical path calculations and good figures in terms of throughput.

To support all flavors of SHA-2 standards along with optimizing them, Rote et al. in Reference [63] proposed a performance-enhanced architecture for SHA-2. The authors introduced a pipelined-round and fully iterative technique for SHA-2 standard (224, 256, 384, and 512). The design was implemented using Xilinx Virtex-6 FPGA. Results showed that the round pipeline architecture gives better results in terms of throughput and comparable figures in terms of area, concerning iterative architecture.

Power consumption is also considered when deploying SHA-2, by Thakur et al. in Reference [64]. They proposed a low power and simple implementation of SHA-2. Their work employed a control system to pass the message blocks without waiting for the synchronization signal. The proposed work decreases the operating frequency of the implementation, hence lowering the power-consumption factor.

SHA-2 can be enhanced when careful analyses are made. All aforesaid works showed that SHA-2's figures are versatile, depending on the kind of optimization. For instance, throughput gives good results in some cases and moderate in others.

4.3 FPGA Implementations of SHA-3 (Keccak)

Keccak or SHA-3 (officially) is the latest hash function announced by the NIST. Keccak competition took three rounds of selection to choose the winning competitor; particularly, the final round, which comprises five algorithms. The selection process relied mainly on the hardware deployment of the competitors. Because of that, we observed many FPGA implementations toward SHA-3. For instance, a comparison between the final five candidates was investigated by Gaj et al. [65]. The work compared design implementations of each candidate. The deployments were tested on both Altera and Xilinx FPGAs. The results showed that Keccak outperformed the other algorithms in both area and speed requirements. Moreover, an experimental HLS bench-marking for different SHA-3 candidates was proposed in Reference [66], where a new Xilinx Zynq board was used with high performance and flexibility. The authors claimed that the new board gave a more accurate comparison between the candidates. They are verified using Vivado studio development tool. The results showed that Keccak performed the best among the algorithms in terms of throughput and frequency.

The authors of Keccak claimed that it fully supports hardware implementations [67]. Among all implementations and optimizations of SHA-3,[7] we selected the most interesting works in SHA-3 (Keccak) area that provided the best performance of Keccak FPGA deployment. To follow the same event-listing fashion, we present the optimization methods in the same order they appeared in the previous two sections.

Keccak follows sponge structure model, which is different from SHA-1 and SHA-2, as discussed in Section 2. However, the CSA can also be used to optimize Keccak standard, jointly with other methods like unfolding, loop unrolling, and pipelining. Authors in Reference [68] proposed a compact FPGA implementation for Keccak function. The design focused on area optimization by merging rho, pi, and chi steps. The area reduction is accomplished by using the same processing units for the three steps, which accordingly reduces the number of units used for the FPGA deployments and consequently reduces the area requirements. The CSA was employed between the three merged steps to speed up the hash computation process. The proposed design was tested and verified using Xilinx Virtex-5 (xc5vlx330t) FPGA. Results showed that the area requirements have decreased to 240 slices with a significant throughput of 7.224 $Gbps$. In essence, the CSA reduces the cycles used to compute the final hash value, besides increasing the throughput of the overall design.

To achieve higher throughput, the pipeline method is used in Keccak FPGA designs. To give a good implication of this property, authors in Reference [69] proposed a pipelined architecture to implement Keccak hash function. The proposed design consists of four units (control unit, input/output buffer, padder unit, and Keccak round). The control unit is used for the synchronization of data flow between units. The input/output buffer is used to communicate with external modules. Padder unit is used for padding operation of the input. While the Keccak rounds involve the main data path for the Keccak computations and include the round's constants, the main optimization

[7]When SHA-3 is mentioned, it explicitly means Keccak.

of the proposed design was on the Keccak rounds by putting the pre-calculated round's constants in registers. Later, all constants are fed to *iota* step one per a round. The system was tested and verified using Xilinx Virtex-5 (XC5VFX70T) FPGA with a throughput of 12.68 Gbps. However, the area requirement was increased successively, as shown in Table 7.

To maintain the area requirements of the pipelined-optimized architecture. Authors in Reference [39] proposed an area-efficient design for the final five candidates of SHA-3 competitors (BLAKE, Grøstl, JH, Keccak, and Skein). The candidates were implemented using the pipelining method. All rounds of the proposed designs were kept unchanged, but a 1-stage pipeline is applied. The proposed architectures were tested and verified using Xilinx Virtex-5 and Virtex-6 FPGAs. The results showed that Keccak gave the best figures in terms of throughput/area. However, the throughput of the proposed design is significantly low, with 864 *Mbps* for Virtex-5 and 145 *Mbps* for Virtex-6, despite the area requirements being low for both FPGAs. However, to maintain the area and throughput requirements of FPGA designs, Michail et al. in Reference [70] proposed a pipelined architecture for SHA-3 Keccak hash function. Several pipeline stages were tested to determine the most appropriate number of pipeline stages. The design was tested and verified using Xilinx Virtex-4, Virtex-5, and Virtex-6 FPGAs. Experiments showed that the most appropriate number of pipeline stages was four, which produced the best results in terms of throughput, area, and throughput/area requirements. The best results of 37.63 *Gbps*, 4,117 *slices*, and 9.14 Tp/A were produced by the Virtex-6 FPGA.

Two stages of pipelined design for SHA-3 hash function were proposed in Reference [71]. The proposed work relies on four components: transformation round, registers, Version Selection initial XORing (VSX) module, zero state register, and control unit. Transformation round combines Keccak round functions ($theta$, rho, pi, chi, and $iota$), which in turn cuts the critical path to half. The VSX module responsible for selecting the appropriate Keccak state ([1,152, 448], [1,088, 832], [832, 768], [576, 1,024] and [1,024, 576]).[8] While zero state register is used to initialize the state with 0's. The control unit uses a finite state machine (FSM) to employ five states: S1 for version selection, (S2, S3) for computations, and (S4, S5) for final hash computation. The designs were tested and verified in terms of area, frequency, and throughput using Xilinx Virtex-5, Virtex-6, and Virtex-7 FPGAs. All results are converged around the same value in all terms, as shown in Table 7.

To build a system that supports multi-messages block, the authors in Reference [72] proposed a high-performance FPGA implementation of SHA-3 hash function. The proposed design used a pipelined multi-blocks message architecture that supports all SHA-3 flavors (224, 256, 384, and 512). The architecture employs a software scheduler that performs three functions. The first one processes the input message (pad and divide into blocks of 1,600 bits). The second function truncates the 512-bits output to the desired hash (224, 256, 284, and 512). The third function updates the state matrix in case of the multi-block message. The design was tested and verified on Xilinx Virtex-4 (XC4VLX200), Virtex-5 (XC5VLX330T), and Virtex-6 (XC6VLX760) FPGAs. The results showed an optimized throughput of two stages pipeline with the increased area used for the three selected FPGAs, as shown in Table 7.

The influence of the pipeline method on the unrolled architecture was also studied for Keccak. Suigar et al. in Reference [73] presented a low-cost and high-speed implementation of SHA-3 hash function. Comprehensive implementations of Keccak were employed using loop unrolling with and without pipelining. Seven architectures were tested and verified using Xilinx Spartan-3E FPGA. The architectures were the basic Keccak ($1x, 2x, 3x$)-unrolled architecture and ($1x, 2x, 3x$) unrolled architecture with (2,3,4)-stage pipelining. The unrolled architectures with pipelining

[8]See the Keccak definition in Section 3 for details.

Table 7. Comparison between Different FPGA Optimization Methods to Implement SHA-3 (Keccak)

Work	Opt-Tech	FPGA	Tp/Gbps	Area/slice	Freq/Mhz	Mbps / Slice
[69]	PPL	Xilinx-Virtex-5XC5VFX70T	12.68	4,793	317.11	2.71
[39]	PPL	Virtex-5	0.864	393	159	2.19
		Virtex-6	0.145	188	285	0.77
[71]	PPL	Virtex-5	18.7	1,702	389	10.98
		Virtex-6	19.1	1,649	397	11.6
		Virtex-7	20.8	1,618	434	12.9
[72]	2PPL	Virtex-4	12.912	5,494	269	2.350
		Virtex-5	16.896	2,652	352	6.371
		Virtex-6	18.768	2,296	391	8.174
[74]	DSP+BRAM	Stratix III	13.913	4,277	306.9	3.25
[78]	DSP+PPL	Spartan-6	9.00	4,865	-	1.85
[76]	SRL	Virtex-5	0.156	134	248	1.16
[73]	Basic	Spartan-3	4.65	4,443	102.6	1.05
	Unfold (x2, x3, x4)		(4.13, 3.42, 2.86)	(6,988, 8,665, 11,173)	(45.6, 25.2, 15.8)	(0.59, 0.39, 0.26)
	(x2-PPL2, x3-PPL3, x4-PPL4)		(8.23, 9.09, 10.11)	(6,409, 8,463, 10,226)	(90.8, 66.8, 55.8)	(1.28, 1.07, 0.99)
[70]	(1, 2, 3, 4)PPL	Virtex 4	(6.55, 12.91, 20.95, 27.07)	(2,365, 5,494, 8,647, 12,870)	(273, 269, 291, 282)	(2.77, 2.35, 2.42, 2.10)
		Virtex 5	(9.17, 16.90, 25.34, 34.27)	(1,581, 2,652, 3,197, 4,632)	(382, 352, 352, 357)	(5.80, 6.37, 7.93, 7.40)
		Virtex 6	(9.89, 18.77, 28.15, 37.63)	(1,115, 2,296, 3,965, 4,117)	(412, 391, 391, 392)	(8.87, 8.17, 7.10, 9.14)
[77]	DSP	Virtex-5	5.70	2,573	285	2.215
[68]	Merg (rho, pi , chi)	Virtex-5	7.224	240	301.02	30.1
[79]	DSP	Virtex-6	4.091	208	451.26	19.66
[81]	Merg 5-states	Virtex-5	17.132	1,291	377.86	13.27
[65]	Unfold,+PPL and circuit duplication (x1, x1-PPL2, x2-PPL2, x2PPL4)	Virtex-5	(7.18, 7.38, 7.13, 13.55)	(1,283, 1,774, 1,996, 3,428)	(-, -, -, -)	(5.60, 4.16, 3.57, 3.95)
		Virtex-6	(7.47, 8.11, -, 13.64)	(1,052,1,263, -, 2,550)	(-, -, -, -)	(7.10, 6.42, -, 5.35)
		Stratix-3	(8.03, 8.55, 13.09, 17.06)	(3,734, 4,484, 6,617, 8,934)	(-, -, -, -)	(2.15, 1.91, 1.98, 1.91)
		Stratix-4	(7.61, 8.96, 12.49, 17.33)	(3,723, 4,481, 6,580, 8,934)	(-, -, -, -)	(2.04, 2.00, 1.90, 1.94)
[85]	2-Parts	Virtex-5	11.50	1,388	278.39	8.48
		Virtex-6	15.76	1,167	394.01	13.83
		Virtex-7	16.58	1,418	414.54	11.97
[84]	Iterative and merge (ρ, π, χ)	Virtex-5	7.22	240	301.02	30.1

showed the best results in terms of throughput and area. The drawn implication after the incorporated experiments showed the defectiveness of unrolling in optimizing the design of Keccak.

The effect of the multi-level pipeline was explored by Jacinto et al. in Reference [75]. The proposed work synthesized Keccak using Xilinx Zynq-7000 FPGA. The High-Level Synthesis was deployed using Xilinx HLS Vivado studio integrated development environment. The HLS tool speeds up the process of analyzing and synthesizing the FPGA implementation of Keccak hash function. The simulation results showed that the pipelined version of Keccak produces a much better throughput than the non-pipelined version with 1.9 Gbps for pipelined and 0.00689 Gbps for the non-pipelined. However, the area requirement increases significantly for the pipelined version with 1,174 slices compared to 735 slices for the non-pipelined. Another work that compares the second-round candidates of the SHA-3 competition to measure the maximum clock frequency supported by each candidate is presented in Reference [66]. The authors claimed that the proposed design represents a testbed that can test the maximum frequency for any hardware implementation. The design supports two clock domains: one for hardware under test and the other for communication between system cores. The proposed work was tested and verified using Xilinx HLS Zynq and Vivado Studio IDE. Results showed that Keccak was among the best in terms of throughput while maintaining a lower frequency.

FPGA resources are used to make a fair comparison between the SHA algorithms. The FPGA resources include BlockRAM, DSP, and SRL. The authors in Reference [74] proposed an optimization technique for the FPGA implementation of 14 selected hash functions from round two of the selection competition of SHA-3. The optimization approach was concentrated on the use of the FPGA resources DSP and Block memory (BRAM) units. The proposed system was tested and evaluated for 256-bits output hash. The implementations were tested on Altera Stratix-3 FPGA. The authors divided the functions of the algorithms into categories according to the potential use of the FPGA resources. The testing results for the throughput showed an improvement concerning the throughput /combinational logic block (#CLB) ratio. However, some works relied on the use of SRL. In Reference [76], the authors designed an FPGA implementation of Keccak using SRL. The SRL register was used in the rho step. The results showed a reduced usage of the FPGA resources due to the reduction in LUTs usage. The proposed work outperformed other designs in terms of area requirement. The design was tested and verified on Xilinx Virtex-5 FPGA with 156 $Mbps$ throughput. Other architectures benefited from the use of the FPGA resources that are designated for the individual logic operations. Provelengios et al. in Reference [77] proposed an optimization technique using the DSP unit ($DSP48E$). The proposed work employed the DSP unit to compute the logic functions (AND, NOT, and XOR) that appear in $theta$ and chi steps. The design was tested and verified using Xilinx Virtex-5 FPGA. The result showed that using DSP unit was inefficient in low-complexity demand applications.

However, to test the efficiency of the DSP units with other optimization methods, Ayuzawa et al. in Reference [78] proposed an FPGA implementation for Keccak hash function using the DSP unit and pipelining. The authors employed an advanced DSP unit to optimize the overall throughput to area ratio. In their work, pipeline registers were inserted between steps of Keccak operations. All registers were used to support multi-message hashing. The proposed work was tested and verified using Xilinx Virtex-5 FPGA. Results showed a significant enhancement of 50% in terms of throughput to area ratio. In some cases, FPGA resources are used to reduce the power consumption of the FPGA designs. Aziz et al. [79] proposed a new approach in designing a low-power consumption for Keccak FPGA design. The work benefited from the full capability of $DSP48$ unit that is widely available in Xilinx FPGAs. The authors presented two designs: one for area distribution and the other for speed reservation. The DSP48 unit was employed for the calculation of the Keccak hash function. The proposed design was tested using Xilinx Virtex-6 FPGA. The low-frequency figures of 451 MHz were observed, which lower the amount of consumed power to 130 mW.

The flexibility of SHA-3 function influenced the researchers to widen their ideas and exploit the FPGA resources. The hardware convenience made the enhancement techniques of Keccak outperform SHA-1 and SHA-2 [80–84]. Chandran et al. in Reference [80] proposed a design using special logic gates. The work supported Keccak calculation in two ways: The first one was step-by-step algorithm, which benefited from the fact that Keccak runs the five steps 24 times. The second way was to design the algorithm using a multiplexer (MUX) that connects two blocks of Keccak function. The proposed designs were tested and verified using Xilinx Spartan-6 FPGA. The results showed reduced area requirements for the final implementations. Moreover, Rao et al. in Reference [81] proposed an architecture to enhance the implementation of SHA-3 on FPGA. The work was performed through two phases. In the first phase, the algorithm steps were logically combined to get a total of 25 equations. The second phase includes the hardware design of the algorithm. Manual fulfilling was applied to phase one, then forming the 1600-bit state. The resultant equations were fed to the proposed hardware architecture in phase two, which processed on LUT level. The architecture was tested and verified using Xilinx Virtex-5, Virtex-6 FPGAs. The authors got a throughput of 17.132 *Gbps* for Virtex-5 FPGA and 19.241 *Gbps* for Virtex-6 FPGA. The area requirement optimizations were also studied by kahari et al. [85]. A high-speed implementation of SHA-3 hash function in terms of area and frequency was proposed. Proposed work divided the Keccak hash calculation into two parts, namely, the sponge and round functions. The sponge part performs message initialization along with state matrix padding, while the squeeze phase performs the compression calculation and hash production. The proposed design was tested and verified using Xilinx Virtex-5, Virtex-6, and Virtex-7 FPGAs. Virtex-6 FPGA gave the best results in terms of area and efficiency, while Virtex-7 was the best in terms of throughput, as shown in Table 7.

As Keccak hash function supports variable length hashes (SHAKE128, SHAKE256). Previous works also investigated the deployments of Keccak toward variable-length output [83, 84]. A compact FPGA implementation is presented in Reference [83]. The authors designed a 1,024 hash output of SHA-3 using Xilinx Virtex FPGA. While Sravani et al. in Reference [84] supported variable length SHA-3 by combining *Rho*, *Pi*, and *Chi* steps into one step, as discussed before.

In general, optimization methods are either dedicated to a specific secure hash algorithm or focus on optimizing the architecture toward specific metric(s) (area, speed, throughput, power consumption). The resulting designs help to test any system in a predefined area of interest. Tables 5, 6, and 7 show the three SHA standards along with their FPGA devices that were used. The optimization methods that were used are listed in the tables. Moreover, the results of the area, throughput, frequency, and throughput/area ratio are depicted in the same tables. As a matter of observation, some designers use BlockRAM metric to calculate area requirements. Therefore, we used the method listed in Reference [86] that stated "each BlockRAM equal to 128 slices." Then the overall area will be calculated according to Equation (24):

$$Area = slices + (128 \times BlockRAMs). \tag{24}$$

Another important area we need to give insight to is error detection and correction schemes. Any design that relies on hardware may be affected by errors that arise during the computations. The error detection (fault detection) schemes for SHA hardware implementations were studied in different literature [41, 87–89]. A brief description of this area is discussed in the next section.

4.4 Error Detection and Correction

FPGA is used to implement hash algorithms, but the complexity of the SHA implementations increases the probability to cause faults in the hardware design. Any single error in any round causes an error for the final generated hash, because each round depends on the previous round (inheritance property) [41]. Therefore, any hardware design and implementation for hash functions needs to be reliable and authentic. In general, there are three methods to provide a digital design with a fault detection paradigm:

- Hardware Redundancy. Different hardware is used to perform design implementations, separately. Afterward, the final outputs are compared to check and detect errors. However, the process consumes more resources (duplication of all hardware) and consequently is an expensive scheme.
- Time Redundancy. A single hardware is used to perform the design at different times. Time redundancy method is less expensive, because it uses the same hardware but the task of checking final architecture is expensive in terms of time consumption.
- Information Redundancy. Uses additional information to check the integrity of data (i.e., parity bit). For instance, information can be appended to the end of data processed or injected into the system in a predefined time.

The faults that appear during computations are divided into two categories [41]: the first one is the permanent category, where the faults appear all the time during the calculation. The second category is the transient faults, which appear in some time slots and disappear for others. In general, faults affect the overall performance of the system.

The ability of the system to detect and recover from any error was taken into consideration by the authors in References [41, 87–90]. Bahram et al. in Reference [41] proposed a time redundancy technique to detect the faults in the design of SHA-512 algorithm. The final design is free from both permanent and transient faults. The authors in Reference [88] introduced a Totally Self Checking (TSC) design for SHA-256 hash function. The work concentrates on the faults that appear in a harmful environment, such as high computation situations or physical situations such as temperature increase. The proposed work relied on Concurrent Error Detection (CED) and used the TSC design. The results showed that the design can fully recover to 100% error-free system. The Proposed TSC SHA-256 scheme was tested and verified on Xilinx Virtex-5 (XC5VLX330) FPGA, with a throughput of 3.88 Gbps.

SHA-1 was also studied for fault detection. In Reference [87], a Totally Self-Checking (TSC) architecture for fault detection of SHA-1 and SHA-256 hash functions was proposed. The authors compared TSC design with hardware duplication method. The design can detect and recover 100% of odd erroneous and appropriately spread even erroneous. Results showed that the TSC architecture was better than the Duplicate With Checking (DWC) design in terms of area and efficiency.

Informational redundancy is used in Reference [90]. The authors used the technique of errors injecting to the system and checked the ability to detect errors and recover. The scheme injected errors in different stages of SHA-2 hash and investigated the error at the output stage. They were able to detect and recover from the injected error inside the hash computation process. Time and hardware redundancy is also investigated in Reference [89]. The authors introduced two designs for efficient fault detection of SHA-2. The first design relies on time redundancy block to detect any transient error in the internal calculation of the round operation. The second design used the hardware redundancy scheme with 100% of hardware overhead. The comparison between both designs was obvious, inasmuch as the first one is a time-consuming approach and the second one is expensive.

Regarding SHA-3 (Keccak), fault detection schemes were also discussed in different publications [91, 92]. Chandran et al. in Reference [91] proposed a performance analysis of modified SHA-3. The work employed an error-tolerant unit for SHA-3 calculation to provide a reliable SHA-3

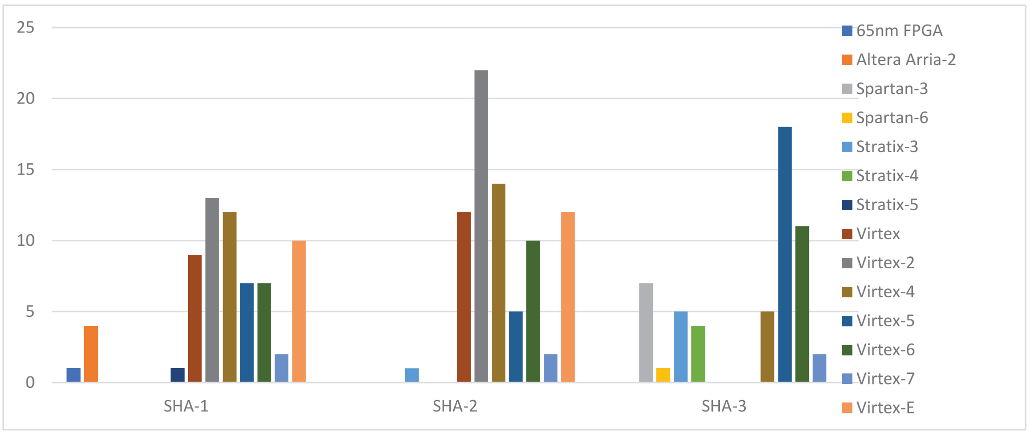


Fig. 6. Count of different FPGAs' optimizations that were used to implement SHA standards. The figure comprises all tables 5, 6, and 7.

architecture. Moreover, a multiplexer was used in the round function to calculate the output digest of the full Keccak standard. The results showed a reduced area and high throughput architecture. However, the design has a disadvantage of the time delay. The downside came from the operation of error tolerance that was performed in the same unit. The system was tested and verified using Xilinx Spartan FPGA. Error detection and correction scheme was explored in Reference [92] by Mestiri et al. The authors focused on predefined method and applied it to the hardware implementation of secure hash algorithms. The authors proposed a fault detection scheme for SHA-3 Keccak using scramble technique. The authors compared between the protected and unprotected version of Keccak. The results were approximately equal in case of frequency and throughput, but the area was increased with 63% for the protected Keccak concerning unprotected version.

In essence, to make the FPGA design of SHA complete, error detection and correction mechanisms need to be supported, because any single bit error will be reflected to the whole output. Different ways are available to optimize the configurable hardware implementations. FPGA is reconfigurable hardware that gives us total control of all resources [21]. The usage of Application Programming Interface (API) tools like FPGA is useful in the process of testing and evaluating the secure hash algorithms and ease comparison between them [13]. In the next section, a discussion of different FPGA designs of SHA will be carried out.

5 DISCUSSION

In this work, we have presented the hardware implementation and optimization of the popular cryptography hash algorithms SHA-1, SHA-2, and SHA-3. The majority of the FPGA implementations are established on both Xilinx and Altera manufacturers. In this study, Xilinx FPGAs are represented by Virtex, Virtex-2, Virtex-4, Virtex-5, Virtex-6, Virtex-7, Virtex-E, Spartan-3, and Spartan-6 FPGAs. While Altera FPGAs are represented by Altera Arria-2, Stratix-3, Stratix-4, and Stratix-5 FPGAs.

Figure 6 shows the count of the usage of different FPGAs to implement the SHA standards. The figure comprises all aforesaid Tables 5, 6, and 7 that were discussed in Section 4. The figure shows that some FPGAs were not used for some hash standard, i.e., Xilinx Virtex-2 FPGA was not used for any of SHA-3 designs. However, Xilinx Spartan-3 FPGA was only used for SHA-3 implementation. Moreover, some FPGAs were suitable for the three hash standards Xilinx Virtex-4, Virtex-5, and Virtex-6, as depicted in Figure 7.

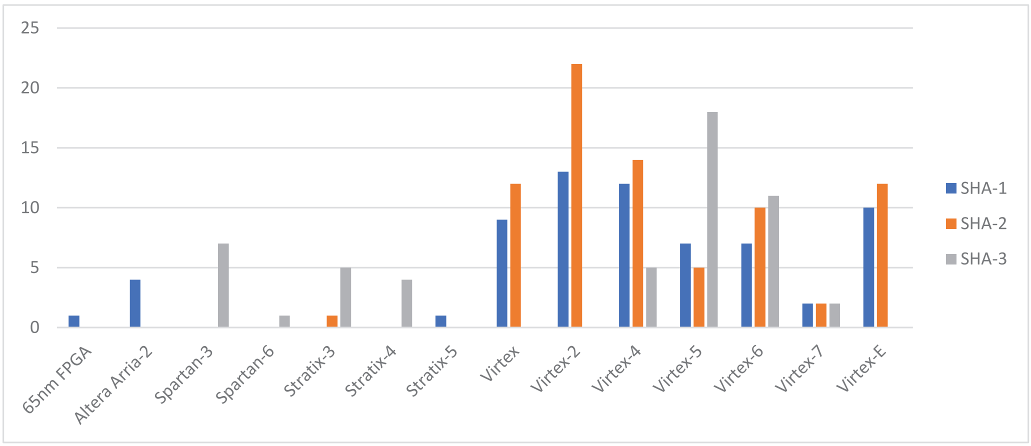


Fig. 7. Count of optimization regarding each type of FPGAs.

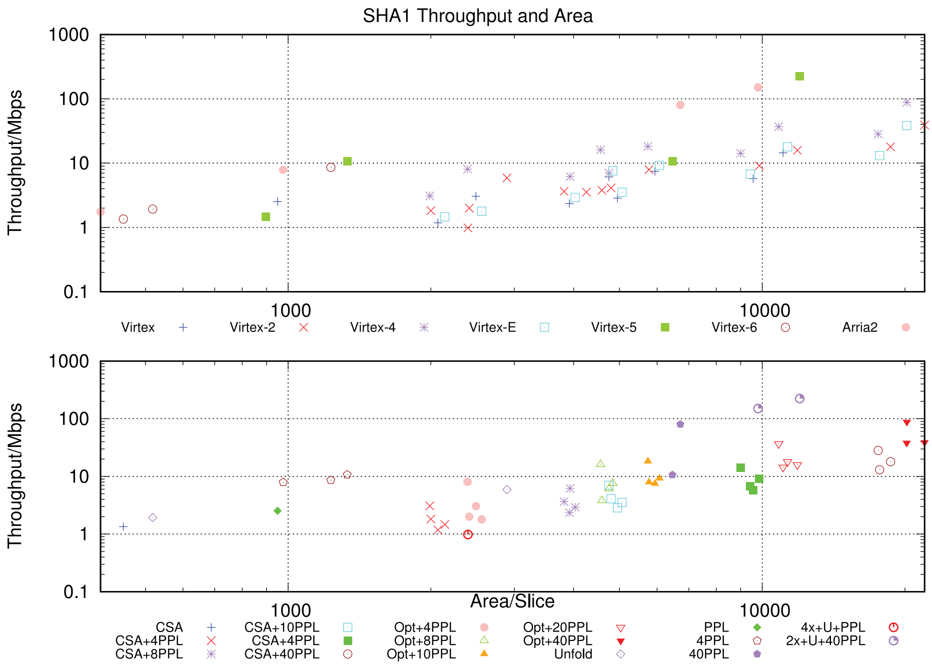


Fig. 8. Throughput (Gbps) and Area (slice) comparison of SHA-1 standard using the four main optimization methods.

Figures 8, 9, and 10 show different graphs for throughput and area with respect to the main optimization methods listed in Tables 5, 6, and 7. The figures depict the influence of optimization techniques on throughput and area requirements for the three hash standards. Each figure contains two graphs: the top graph represents the throughput and area comparison from the FPGAs perspective, while the bottom graph represents the throughput and area comparison from the optimization methods perspective. The x-axis of all figures represents the Area per slices, while the y-axis the throughput per Gbps. To help read the figures, pick any point from the top graph and

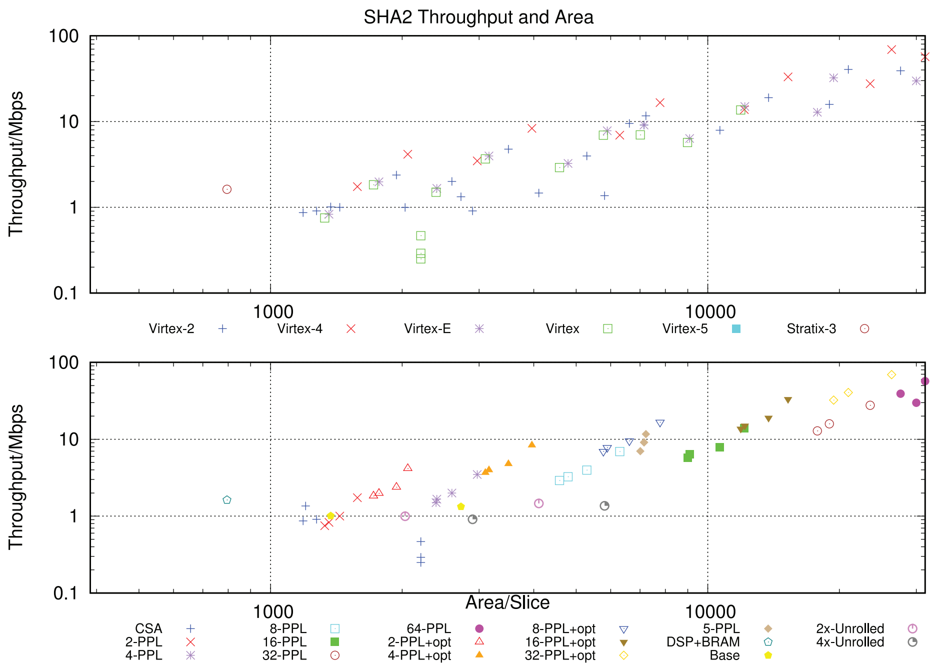


Fig. 9. Throughput (Gbps) and Area (slice) comparison of SHA-2 standard regarding the four main optimization methods.

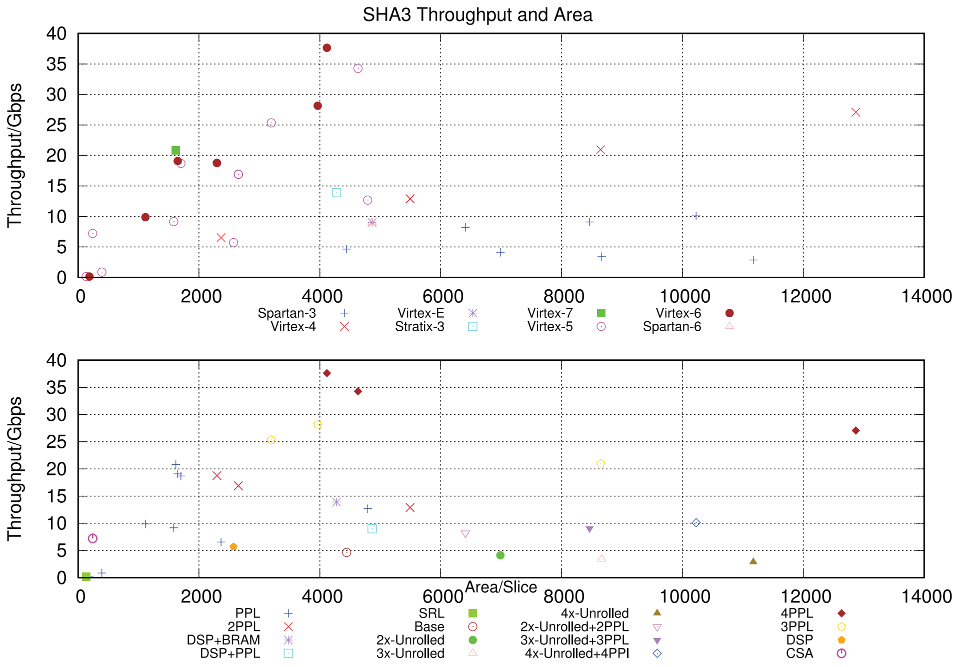


Fig. 10. Throughput (Gbps) comparison of SHA-3 standard regarding the four main optimization methods.

join it with the same point-position at the bottom graph. Then you will get the Throughput and Area requirements of the point that is implemented using the selected FPGA from the top graph and optimized with the corresponding optimization method from the bottom graph.

Figure 8 represents the throughput and area relationship regarding different FPGAs' implementations for SHA-1 hash standard. Even though the authors in Reference [52] claimed that FPGAs have an upper limit for each optimization method in which any pipeline optimization higher than eight will give unrealistic results. In spite of that, the designs of SHA-1 on Xilinx Virtex-5 and Altera Arria-2 proved that pipelining higher than eight stages can be used for optimization with a high throughput performance. Virtex-5 gives the best throughput among the others when used with two times unfold and 40-stage pipeline (2×-unfold+40PPL) optimization method. Altera Arria-2 comes after for the same design [49]. In the same figure, we can see how the CSA affects area requirements. Moreover, area slices will increase by the use of the CSA along with the pipelining (PPL) method. Loop-Unrolling shows a better throughput performance when used with pipelining, but the augmented area slices are significant.

Figure 9 shows that the majority of SHA-2 implementations and optimizations were carried out using Xilinx Virtex-2, Virtex-4, and Virtex-E FPGAs. The figure depicts Throughput and Area comparison of SHA-2 hash standard regarding different optimization methods. The best results for throughput is the optimized 32-stages pipelined architecture. However, the area requirements for 64-stages pipeline got more area slices than the others. In essence, the area requirements are increasing with the increase of the applied pipeline stage. Carry Save Adder (CSA) optimization shows more enhancements for the FPGA implementations of both SHA-1 and SHA-2 standards. The fact that SHA-1 and SHA-2 rely mainly on addition equations, CSA has a salient effect on their throughput. SHA-3 benefited from the CSA on designs that relied on the loop unfolding, where, after unrolling (unfolding) the addition process appeared a number of times and therefore necessitates the need for the CSA. However, Loop-Unfold (Unroll) method does not show any enhancement unless it is combined with other methods.

SHA-3 hash standard is compared regarding throughput and area for different optimization methods, as shown in Figure 10. The first insight is the majority of SHA-3 designs relied on the Xilinx Spartan-3, Virtex-4, Virtex-5, and Virtex-6 FPGAs. For throughput optimization, the best results were those of Xilinx Virtex-6 FPGA; while regarding area, Virtex-5 got better results than the others. The figure shows the upper hand of the pipelining method over the other optimization techniques in terms of throughput. But, the abundant increase of area requirements comes along with the increase in the number of pipeline stages. However, the FPGA resources give good results in terms of throughput and significant enhancements of area requirements. For instance, DSP+BRAM optimization method provides a moderate throughput value and good area figures but on different FPGA (Altera Stratix-3). The results show that some optimization methods are better to be implemented on specific hardware to achieve the required outcomes. Loop-Unfold optimization reflects the need to be combined with other optimization methods for a better performance.

In the case of other optimization methods, Figure 11 shows the optimization methods that are not using the main four components (CSA, PPL, Unfold, and FPGA-Resources). The bottom graph shows that some optimizations combine SHA-1 and SHA-2 standard, which are represented by the numbers 1 and 2. SHA-3 retains the upper hand over SHA-1 and SHA-2, as shown in the figure. The majority of the optimizations were implemented using Virtex-5, Virtex-6, and Stratix-3 FPGAs. SHA-1 gives a comparable result when optimized using one time unrolling with the hardware duplication.

Pipelining is used to optimize SHA-1 and SHA-2. A max of four-stages pipelined architecture was adopted because of the nonlinear nature of SHA-1 and SHA-2 in the early steps, as discussed earlier. Without any proved optimum pipeline studies, the best use of the pipeline architecture

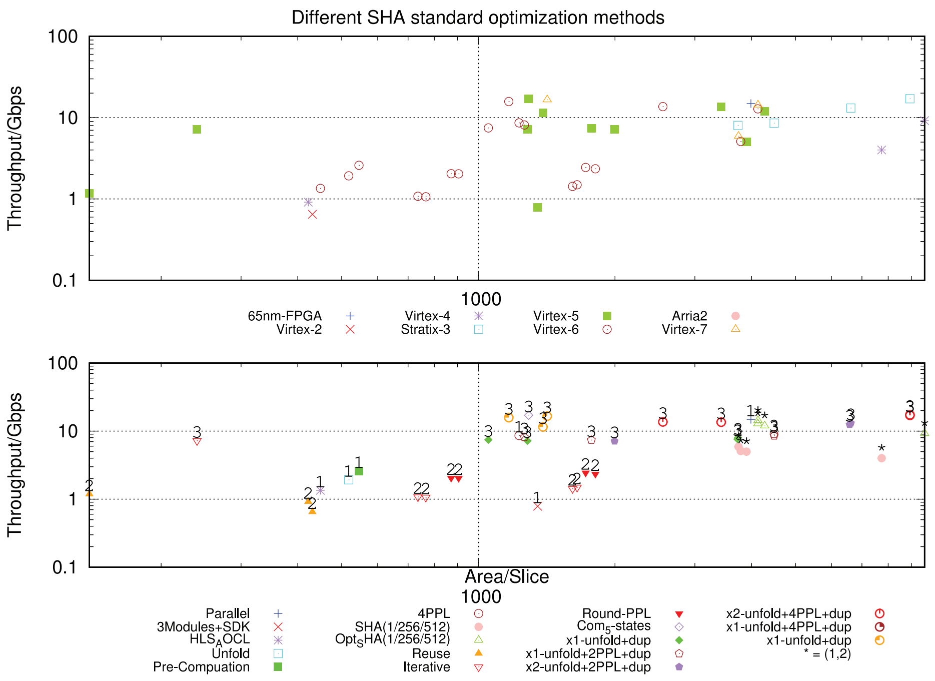


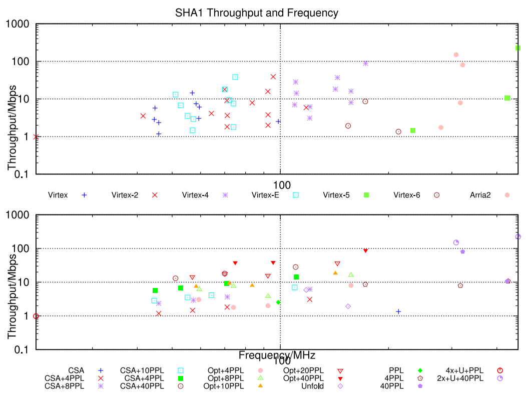
Fig. 11. Throughput (Gbps) and Area (slice) comparison of All SHA standards regarding different optimization methods.

is to increase the operating frequency, hence increasing the throughput but augmenting the area requirements. The authors of Keccak provided a prototype to support hardware implementation so Keccak outperforms the others in terms of Throughput and Area. However, the majority of Keccak implementations operate on a higher frequency than SHA-1 and SHA-2, which in turn reflected on the increase of power consumption.

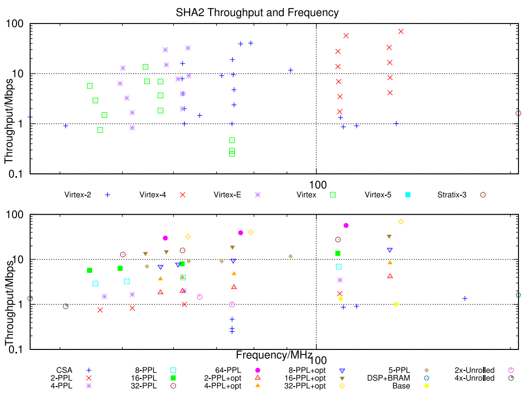
The hardware implementations of both SHA-1 and SHA-2 were not taken into consideration at the time of their design. SHA-1 and SHA-2 standards have non-linear equations in the early steps of calculations. The non-linearity of the equations opposed the hardware deployment methodologies, which affects the overall FPGA designs. The limited hardware support of Merkle-Damgard (MD) construction model diminishes the ability to exploit the FPGA hardware during the deployment of SHA-1 and SHA-2 standards. However, Keccak hash function, which won the selection competition of SHA-3 hash standard, fully supports hardware deployment. Moreover, the authors of Keccak provided a prototype for implementing their winning algorithm [93].

There is a trade-off between different FPGA optimization methods. Some designs affect one requirement positively but make a negative impact on others. Therefore, in the case of negative impact, the combination of two or more optimization methods is reflected in the overall performance of a design. For instance, the loop unrolling method decreases the number of clock cycles needed for calculations, but it makes a negative impact on the throughput, as shown in Figures 8, 9, and 10. Combining the loop unrolling with the pipelining method produces a good enhancement in terms of throughput with increased area requirements.

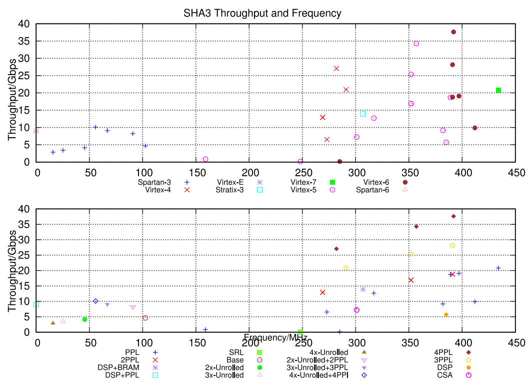
Figure 12 shows the relationships between the throughput and frequency for the three hash standards (SHA-1, SHA-2, and SHA-3). The x-axis represents the frequency and y-axis represents the throughput. The top part of each sub-figure represents the throughput with respect to the



(a) SHA-1



(b) SHA-2



(c) SHA-3

Fig. 12. The Throughput and Frequency relationships for the hash functions.

FPGA type, while the bottom part represents the throughput with respect to the optimization technique. The figure shows that the frequency and throughput have a proportional relationship, i.e., for a high throughput there is a higher frequency and vice versa.

The power consumption of an FPGA is determined by the sum of total static and dynamic power. The static power reflects the consumed power by the FPGA design and routing technology, and the dynamic power reflects the consumed power that is driven by the resource utilization of a design [94, 95]. The general power consumption equation is represented by Equation (25):

$$P = \sum_i C_i.V_i^2.f, \tag{25}$$

where, for a resource i, C_i is the capacitance, V_i is the voltage, and f is the frequency. According to Figure 12, the frequencies of SHA-1 hash standard are in the middle, and the frequencies of SHA-2 are to the left. This shows the lower frequencies that SHA-1 and SHA-2 run, which in turn reflected on the total power consumption, as depicted in Equation (25). However, SHA-3 frequencies are biased to the right, which infers that SHA-3 runs higher frequencies than SHA-1 and SHA-2, and consequently, reflected to the total consumed power.

6 CONCLUSIONS

This survey presented details about the Secure Hash Algorithms and their hardware implementations using the FPGAs. Moreover, a comparison of three hash standards (SHA-1, SHA-2, and SHA-3) was presented.

Optimization methods provide fair comparisons between various FPGA implementations of hash standards. CSA, pipelining, and Loop Unrolling are used to exploit the FPGA implementations of the secure hash Algorithms. Moreover, FPGA resources are used to mitigate the FPGA optimization techniques. CSA optimization technique is mostly used in optimizing SHA-1 and SHA-2 because of the non-linear nature of the early steps of them. It enhances the addition operation by manipulating fast carry save adder. The pipelining method showed good enhancements in terms of throughput and speed. However, the area requirement is increased significantly with the increase of the number of pipelining level. Loop unrolling (unfolding) method decreases the number of clock cycles needed for the computations, which effects on the throughput and area, i.e., it creates plenty of data that need to be processed simultaneously. Interestingly, the performance is decreased unless loop unrolling is combined with other optimization methods. For instance, Loop unrolling (unfolding) works better if it is combined with pipelining, as the combination reduces area and increases throughput requirements.

In this survey, we discussed the Secure Hash Algorithms (SHA-1, SHA-2, SHA-3) and their corresponding family members. The mean of any successful FPGA implementation is the one that puts all performance metrics into account and complies with all trade-offs. Because of the sequential nature and the non-linearity of the early steps of SHA-1 and SHA-2, it is difficult to get a better optimization using one method unless it is combined with other optimization methods. For instance, pipelining is combined with the loop unrolling method to exploit both techniques. To the extent of our study, SHA-3 is the most suitable hash standard to be implemented on the FPGA hardware. The implementation of SHA-3 on FPGA outperforms the preceding hash algorithms in terms of speed and area. However, in terms of power consumption, SHA3 consumes more power than the other standards, because it processes 1,600 bits in one clock cycle. Therefore, high frequency is needed and, consequently, power consumption increases significantly.

REFERENCES

[1] Burton S. Kaliski. 1991. The MD4 message digest algorithm. In *Advances in Cryptology—EUROCRYPT'90*, Ivan Bjerre Damgård (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 492–492.

[2] Ronald Rivest and S. Dusse. 1992. *The MD5 Message-digest Algorithm.* MIT Laboratory for Computer Science.

[3] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 19–35.

[4] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. 2005. Finding collisions in the full SHA-1. In *Proceedings of the International Cryptology Conference (CRYPTO'05)*, Vol. 3621. Springer, 17–36.

[5] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. In *Proceedings of the International Cryptology Conference (CRYPTO'17).* IACR, Santa-Barbara, United States, 570–596. DOI: http://dx.doi.org/10.1007/978-3-319-63688-7_19

[6] PUB FIPS. 2012. 180-2: Secure hash standard (SHS). *US Department of Commerce, National Institute of Standards and Technology (NIST)* (2012).

[7] FIB PUB. 2008. 180-3, Secure hash standard (SHS). *Federal Information Processing Standards Publication* (2008).

[8] Hans Delfs and Helmut Knebl. 2002. *Introduction to Cryptography.* Vol. 2. Springer.

[9] Shu-jen Chang, Ray Perlner, William E. Burr, Meltem Sönmez Turan, John M. Kelsey, Souradyuti Paul, and Lawrence E. Bassham. 2012. Third-round report of the SHA-3 cryptographic hash algorithm competition. *NIST Interag. Rep.* 7896 (2012).

[10] Quynh Dang. 2013. Changes in federal information processing standard (FIPS) 180-4, secure hash standard. *Cryptologia* 37, 1 (2013), 69–73.

[11] Morris J. Dworkin. 2015. *SHA-3 Standard: Permutation-based Hash and Extendable-output Functions.* Technical Report. National Institute of Standards and Technology.

[12] Krystian Matusiewicz. 2007. *Analysis of Modern Dedicated Cryptographic Hash Functions*. Citeseer.

[13] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Máire O'Neill, and William P. Marnane. 2010. A hardware wrapper for the SHA-3 hash algorithms. In *IET Irish Signals and Systems Conference (ISSC'10)*. 1–6.

[14] Christopher Cullinan, Christopher Wyant, Timothy Frattesi, and Xinming Huang. 2013. Computing performance benchmarks among CPU, GPU, and FPGA. Retrieved from www.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final.

[15] Veerpal Kaur and Aman Singh. 2013. Review of various algorithms used in hybrid cryptography. *International Journal of Computer Science and Network* 2, 6 (2013), 157–173.

[16] Miodrag Potkonjak. 2013. Hardware-based cryptography. (Feb. 19, 2013). US Patent 8,379,856.

[17] Sundararaman Rajagopalan, Rengarajan Amirtharajan, Har Narayan Upadhyay, and John Bosco Balaguru Rayappan. 2012. Survey and analysis of hardware cryptographic and steganographic systems on FPGA. *J. Appl. Sci.* 12, 3 (2012), 201.

[18] Kimmo Jarvinen, Matti Tommiska, and Jorma Skytta. 2005. Comparative survey of high-performance cryptographic algorithm implementations on FPGAs. *IEE Proceedings-Information Security* 152, 1 (2005), 3–12.

[19] K. Saravanan and A. Senthilkumar. 2013. Theoretical survey on secure hash functions and issues. *Int. J. Eng. Res. Technol.* 2, 10 (2013).

[20] Ricardo Chaves, Leonel Sousa, Nicolas Sklavos, Apostolos P. Fournaris, Georgina Kalogeridou, Paris Kitsos, and Farhana Sheikh. 2016. Secure hashing: SHA-1, SHA-2, and SHA-3. *Circ. Syst. Sec. Priv.* (2016), 81–107.

[21] Zhijie Shi, Chujiao Ma, Jordan Cote, and Bing Wang. 2012. Hardware implementation of hash functions. In *Introduction to Hardware Security and Trust*. Springer, 27–50.

[22] James H. Burrows. 1995. *Secure Hash Standard*. Technical Report. Department of Commerce Washington DC.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. The Keccak SHA-3 submission. *Submission to NIST (Round 3)* 6, 7 (2011), 16.

[24] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. Cryptographic sponges. Retrieved from http://sponge.noekeon.org.

[25] Christof Paar and Jan Pelzl. 2010. Hash functions. In *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, Berlin, Heidelberg, 293–317. DOI: http://dx.doi.org/10.1007/978-3-642-04101-3_11

[26] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. 2012. Keccak implementation overview. Retrieved from http://keccak. neokeon.org/Keccak-implementation-3.2.pdf.

[27] David Sheldon, Rakesh Kumar, Roman Lysecky, Frank Vahid, and Dean Tullsen. 2006. Application-specific customization of parameterized FPGA soft-core processors. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design*. IEEE, 261–268.

[28] Francisco Rodríguez-Henríquez, Nazar Abbas Saqib, Arturo Díaz Pérez, and Cetin Kaya Koc. 2007. *Cryptographic Algorithms on Reconfigurable Hardware*. Springer Science & Business Media.

[29] Svetlin A. Manavski et al. 2007. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Proceedings of the IEEE International Conference on Signal Processing and Communications*.

[30] Sikhar Patranabis and Debdeep Mukhopadhyay. 2018. *Fault Tolerant Architectures for Cryptography and Hardware Security*. Springer.

[31] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding performance differences of FPGAs and GPUs. In *Proceedings of the IEEE 26th International Symposium on Field-programmable Custom Computing Machines (FCCM'18)*. IEEE, 93–96.

[32] Imene Mhadhbi, Nejla Rejeb, Slim Ben Othman, Nabil Litayem, and Slim Ben Saoud. 2014. Design methodologies impact on the embedded system performances: Case of cryptographic algorithm. In *Proceedings of the World Symposium on Computer Applications & Research (WSCAR'14)*. IEEE, 1–6.

[33] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. 1993. Architecture of field-programmable gate arrays. *Proc. IEEE* 81, 7 (1993), 1013–1029.

[34] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. 2010. Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 264–278.

[35] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. 2011. Throughput vs. area trade-offs in high-speed architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 491–506.

[36] Robert P. McEvoy, Francis M. Crowe, Colin C. Murphy, and William P. Marnane. 2006. Optimisation of the SHA-2 family of hash functions on FPGAs. In *Proceedings of the IEEE Computer Society Symposium on Emerging VLSI Technologies and Architectures*. IEEE, 6.

[37] George S. Athanasiou, Harris E. Michail, George Theodoridis, and Costas E. Goutis. 2013. Optimising the SHA-512 cryptographic hash function on FPGAs. *IET Comput. Dig. Techniques* 8, 2 (2013), 70–82.

[38] H. E. Michail, Athanasios P. Kakarountas, George N. Selimis, and Costas E. Goutis. 2005. Optimizing SHA-1 hash function for high throughput with a partial unrolling study. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer, 591–600.

[39] Bernhard Jungk and Jurgen Apfelbeck. 2011. Area-efficient FPGA implementations of the SHA-3 finalists. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*. IEEE, 235–241.

[40] Paul Dillien. 2017. And the winner of best FPGA of 2016 is... *Principal, High Tech Marketing* (2017).

[41] Mohsen Bahramali, Jin Jiang, and Arash Reyhani-Masoleh. 2011. A fault detection scheme for the FPGA implementation of SHA-1 and SHA-512 round computations. *J. Electron. Test.* 27, 4 (2011), 517.

[42] Julia Drozd, Oleksandr Drozd, Valeria Nikul, and Julian Sulima. 2018. FPGA implementation of vertical addition with a bitwise pipeline of calculations. In *Proceedings of the IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT'18)*. IEEE, 239–242.

[43] Kentaro Katayama, Hidetoshi Matsumura, Hiroaki Kameyama, Shinichi Sazawa, and Yasuhiro Watanabe. 2017. An FPGA-accelerated high-throughput data optimization system for high-speed transfer via wide area network. In *Proceedings of the International Conference on Field Programmable Technology (ICFPT'17)*. IEEE, 211–214.

[44] Colin Yu Lin, Zhenghong Jiang, Cheng Fu, Hayden Kwok-Hay So, and Haigang Yang. 2017. FPGA high-level synthesis versus overlay: Comparisons on computation kernels. *ACM SIGARCH Comput. Archit. News* 44, 4 (2017), 92–97.

[45] Vaughn Betz. 2009. FPGA architecture for the challenge. Retrieved from http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html.

[46] Sergio Lucia, Denis Navarro, Oscar Lucia, Pablo Zometa, and Rolf Findeisen. 2018. Optimized FPGA implementation of model predictive control for embedded systems using high-level synthesis tool. *IEEE Trans. Industr. Inform.* 14, 1 (2018), 137–145.

[47] R. K. Makkad and A. K. Sahu. 2016. Novel design of fast and compact SHA-1 algorithm for security applications. In *Proceedings of the IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT'16)*. 921–925. DOI: http://dx.doi.org/10.1109/RTEICT.2016.7807963

[48] A. P. Kakarountas, G. Theodoridis, T. Laopoulos, and C. E. Goutis. 2005. High-speed FPGA implementation of the SHA-1 hash function. In *Proceedings of the IEEE Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*. 211–215. DOI: http://dx.doi.org/10.1109/IDAACS.2005.282972

[49] Shamsiah Suhaili, Takahiro Watanabe, and Norhuzaimin Julai. 2017. High speed and throughput evaluation of SHA-1 hash function design with pipelining and unfolding transformation techniques. *J. Telecomm. Electron. Comput. Eng.* 9, 3–10 (2017), 19–22.

[50] Eun-Hee Lee, Je-Hoon Lee, Il-Hwan Park, and Kyoung-Rok Cho. 2009. Implementation of high-speed SHA-1 architecture. *IEICE Electron. Expr.* 6, 16 (2009), 1174–1179.

[51] Yong Ki Lee, Herwin Chan, and Ingrid Verbauwhede. 2006. Throughput optimized SHA-1 architecture using unfolding transformation. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*. IEEE, 354–359.

[52] Harris E. Michail, George S. Athanasiou, Vasileios I. Kelefouras, George Theodoridis, Thanos Stouraitis, and Costas E. Goutis. 2016. Area-throughput trade-offs for SHA-1 and SHA-256 hash functions' pipelined designs. *J. Circ. Syst. Comput.* 25, 04 (2016), 1650032.

[53] T. Isobe, S. Tsutsumi, K. Seto, K. Aoshima, and K. Kariya. 2010. 10 Gbps implementation of TLS/SSL accelerator on FPGA. In *Proceedings of the IEEE 18th International Workshop on Quality of Service (IWQoS'10)*. 1–6. DOI: http://dx.doi.org/10.1109/IWQoS.2010.5542723

[54] Nalini C. Iyer and Sagarika Mandal. 2013. Implementation of secure hash algorithm-1 using FPGA. *Int. J. Inf. Comput. Technol.* 3 (2013), 757–764.

[55] Ian Janik and Mohammed A. S. Khalid. 2016. Synthesis and evaluation of SHA-1 algorithm using altera SDK for OpenCL. In *Proceedings of the IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS'16)*. IEEE, 1–4.

[56] Harris E. Michail, George S. Athanasiou, George Theodoridis, and Costas E. Goutis. 2014. On the development of high-throughput and area-efficient multi-mode cryptographic hash designs in FPGAs. *Integ. VLSI J.* 47, 4 (2014), 387–407.

[57] Wanzhong Sun, Hongpeng Guo, Huilei He, and Zibin Dai. 2007. Design and optimized implementation of the SHA-2 (256, 384, 512) hash algorithms. In *Proceedings of the 7th International Conference on ASIC (ASICON'07)*. IEEE, 858–861.

[58] Anane Mohamed and Anane Nadjia. 2015. SHA-2 hardware core for Virtex-5 FPGA. In *Proceedings of the 12th International Multi-Conference on Systems, Signals & Devices (SSD'15)*. IEEE, 1–5.

[59] Ignacio Algredo-Badillo, C. Feregrino-Uribe, René Cumplido, and Miguel Morales-Sandoval. 2013. FPGA-based implementation alternatives for the inner loop of the secure hash algorithm SHA-256. *Microproc. Microsyst.* 37, 6 (2013), 750–757.

[60] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane. 2006. Optimisation of the SHA-2 family of hash functions on FPGAs. In *Proceedings of the IEEE Computer Society Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*. DOI: http://dx.doi.org/10.1109/ISVLSI.2006.70

[61] Hassen Mestiri, Fatma Kahri, Belgacem Bouallegue, and Mohsen Machhout. 2014. Efficient FPGA hardware implementation of secure hash function SHA-2. *Int. J. Comput. Netw. Inf. Sec.* 7, 1 (2014), 9.

[62] Rommel García, Ignacio Algredo-Badillo, Miguel Morales-Sandoval, Claudia Feregrino-Uribe, and René Cumplido. 2014. A compact FPGA-based processor for the secure hash algorithm SHA-256. *Comput. Electric. Eng.* 40, 1 (2014), 194–202.

[63] Manoj D. Rote, N. Vijendran, and David Selvakumar. 2015. High performance SHA-2 core using the round pipelined technique. In *Proceedings of the IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT'15)*. IEEE, 1–6.

[64] Dipti Thakur and Utsav Malviya. 2018. Low power and simple implementation of secure hashing algorithm (SHA-2) using VHDL implemented on FPGA of SHA-224/256 core. *Int. J. Eng. Manag. Res.* 8, 1 (2018), 1–4.

[65] Kris Gaj, Ekawat Homsirikamol, Marcin Rogawski, Rabia Shahid, and Malik Umar Sharif. 2012. Comprehensive evaluation of high-speed and medium-speed implementations of five SHA-3 finalists using Xilinx and Altera FPGAs. *IACR Cryptology EPrint Archive* 2012 (2012), 368.

[66] Farnoud Farahmand, Ekawat Homsirikamol, and Kris Gaj. 2016. A Zynq-based testbed for the experimental benchmarking of algorithms competing in cryptographic contests. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'16)*. IEEE, 1–7.

[67] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 313–314.

[68] Alia Arshad, Arshad Aziz, et al. 2014. Compact implementation of SHA3-512 on FPGA. In *Proceedings of the Conference on Information Assurance and Cyber Security (CIACS'14)*. IEEE, 29–33.

[69] Hassen Mestiri, Fatma Kahri, Mouna Bedoui, Belgacem Bouallegue, and Mohsen Machhout. 2016. High throughput pipelined hardware implementation of the KECCAK hash function. In *Proceedings of the International Symposium on Signal, Image, Video and Communications (ISIVC'16)*. IEEE, 282–286.

[70] Harris E. Michail, Lenos Ioannou, and Artemios G. Voyiatzis. 2015. Pipelined SHA-3 implementations on FPGA: Architecture and performance analysis. In *Proceedings of the 2nd Workshop on Cryptography and Security in Computing Systems*. ACM, 13.

[71] George S. Athanasiou, George-Paris Makkas, and Georgios Theodoridis. 2014. High throughput pipelined FPGA implementation of the new SHA-3 cryptographic hash algorithm. In *Proceedings of the 6th International Symposium on Communications, Control and Signal Processing (ISCCSP'14)*. IEEE, 538–541.

[72] Lenos Ioannou, Harris E. Michail, and Artemios G. Voyiatzis. 2015. High performance pipelined FPGA implementation of the SHA-3 hash algorithm. In *Proceedings of the 4th Mediterranean Conference on Embedded Computing (MECO'15)*. IEEE, 68–71.

[73] Jarosław Sugier. 2014. Low cost FPGA devices in high speed implementations of Keccak-f hash algorithm. In *Proceedings of the 9th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX'14)*. Springer, 433–441.

[74] Rabia Shahid, Malik Umar Sharif, Marcin Rogawski, and Kris Gaj. 2011. Use of embedded FPGA resources in implementations of 14 round 2 sha-3 candidates. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'11)*. IEEE, 1–9.

[75] H. S. Jacinto, Luka Daoud, and Nader Rafla. 2017. High level synthesis using vivado HLS for optimizations of SHA-3. In *Proceedings of the IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS'17)*. IEEE, 563–566.

[76] Jori Winderickx, Joan Daemen, and Nele Mentens. 2016. Exploring the use of shift register lookup tables for Keccak implementations on Xilinx FPGAs. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, 1–4.

[77] George Provelengios, Paris Kitsos, Nicolas Sklavos, and Christos Koulamas. 2012. FPGA-based design approaches of Keccak hash function. In *Proceedings of the 15th Euromicro Conference on Digital System Design (DSD'12)*. IEEE, 648–653.

[78] Yusuke Ayuzawa, Naoki Fujieda, and Shuichi Ichikawa. 2014. Design trade-offs in SHA-3 multi-message hashing on FPGAs. In *Proceedings of the TENCON 2014-2014 IEEE Region 10 Conference*. IEEE, 1–5.

[79] Arshad Aziz et al. 2016. A low-power SHA-3 design using embedded digital signal processing slice on FPGA. *Comput. Electric. Eng.* 55 (2016), 138–152.

[80] Nithin R. Chandran and Ebin M. Manuel. 2016. Performance analysis of modified SHA-3. *Proced. Technol.* 24, Supplement C (2016), 904–910. DOI: http://dx.doi.org/10.1016/j.protcy.2016.05.168

[81] Muzaffar Rao, Thomas Newe, and Ian Grout. 2014. Efficient high speed implementation of secure hash algorithm-3 on Virtex-5 FPGA. In *Proceedings of the 17th Euromicro Conference on Digital System Design (DSD'14)*. IEEE, 643–646.

[82] Muzaffar Rao, Thomas Newe, Ian Grout, and Avijit Mathur. 2016. High speed implementation of a SHA-3 core on Virtex-5 and Virtex-6 FPGAs. *J. Circ. Syst. Comput.* 25, 07 (2016), 1650069.

[83] S. Bhargav and Drva Sharath Kumar. 2015. Compact implementation of SHA3-1024 on FPGA. *Int. J. Emerg. Eng. Res. Technol.* 3, 7 (2015), 79–86.

[84] M. M. Sravani and C. H. Pallavi. 2015. Design of compact implementation of SHA-3 (512) on FPGA. *Int. Res. J. Eng. Technol.* 2, 02 (2015), 41–46.

[85] Fatma Kahri, Hassen Mestiri, Belgacem Bouallegue, and Mohsen Machhout. 2016. High speed FPGA implementation of cryptographic KECCAK hash function crypto-processor. *J. Syst. Comput.* 25, 04 (2016), 1650026.

[86] Giacinto Paolo Saggese, Antonino Mazzeo, Nicola Mazzocca, and Antonio G. M. Strollo. 2003. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. Springer, 292–302.

[87] Harris E. Michail, George S. Athanasiou, George Theodoridis, Andreas Gregoriades, and Costas E. Goutis. 2016. Design and implementation of totally-self checking SHA-1 and SHA-256 hash functions' architectures. *Microproc. Microsyst.* 45 (2016), 227–240.

[88] Harris E. Michail, Apostolis Kotsiolis, Athanasios Kakarountas, George Athanasiou, and Costas Goutis. 2015. Hardware implementation of the totally self-checking SHA-256 hash core. In *Proceedings of the Eurocon International Conference on Computer as a Tool (EUROCON'15)*. IEEE, 1–5.

[89] Fatma Kahri, Hassen Mestiri, Belgacem Bouallegue, and Mohsen Machhout. 2017. An efficient fault detection scheme for the secure hash algorithm SHA-512. In *Proceedings of the International Conference on Green Energy Conversion Systems (GECS'17)*. IEEE, 1–5.

[90] Imtiaz Ahmad and A. Shoba Das. 2007. Analysis and detection of errors in implementation of SHA-512 algorithms on FPGAs. *Comput. J.* 50, 6 (2007), 728–738.

[91] Nithin R. Chandran and Ebin M. Manuel. 2016. Performance analysis of modified SHA-3. *Proced. Technol.* 24 (2016), 904–910.

[92] Hassen Mestiri, Fatma Kahri, Belgacem Bouallegue, Mehrez Marzougui, and Mohsen Machhout. 2017. Efficient countermeasure for reliable KECCAK architecture against fault attacks. In *Proceedings of the 2nd International Conference on Anti-Cyber Crimes (ICACC'17)*. IEEE, 55–59.

[93] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2014. The making of KECCAK. *Cryptologia* 38, 1 (2014), 26–60.

[94] Peggy Abusaidi, Matt Klein, and Brian Philofsky. 2008. Virtex-5 FPGA system power design considerations. *Xilinx WP285 (v1. 0)* (Feb. 14, 2008).

[95] Najeem Lawal, Fahad Lateef, and Muhammad Usman. 2015. Power consumption measurement & configuration time of FPGA. In *Proceedings of the Conference on Power Generation System and Renewable Energy Technologies (PGSRET'15)*. IEEE, 1–5.

Received January 2019; revised May 2020; accepted May 2020