Chapter 2
Hardware Implementation of Hash Functions

Zhijie Shi, Chujiao Ma, Jordan Cote, and Bing Wang

2.1 Introduction to Cryptographic Hash Functions

Hash algorithm is a type of cryptographic primitives. Hash algorithms take as input a message of arbitrary length, and produce a *hash* or *message digest* as output. This process can be denoted as:

$$h = H(M),$$

where M is the input message and h is the hash generated by the hash algorithm H. Normally, the size of the hash h is fixed by the algorithm. For a cryptographic hash function, the hash length should be large enough to prevent an attack from finding two or more messages that generate the same hash. Currently, the most commonly used hash algorithms are MD5 [1] and SHA-2 [2].

In general, the cryptographic hash algorithms should have the following properties:

1. *Preimage resistance*. Given a hash h, it should be difficult to find a message M such that $h = H(M)$. This is part of the one-way property. It is easy to compute the hash from a message, but it is extremely difficult to deduce a message from the hash.
2. *Second preimage (or 2nd-preimage) resistance*. Given a message M_1, it is difficult to find another message M_2 such that M_1 and M_2 generate the same hash. Any change in the input results in changes, often wild changes, in the hash.
3. *Collision resistance*. It should be difficult to find two messages M_1 and M_2 such that $M_1 \neq M_2$, but $H(M_1) = H(M_2)$.

A hash function that is both preimage resistant and 2nd-preimage resistant is called a one-way hash function. Note that preimage resistance does not imply 2nd-preimage

Z. Shi (✉) · C. Ma · J. Cote · B. Wang
Computer Science and Engineering Department, University of Connecticut, Storrs, CT, USA
e-mail: zshi@engr.uconn.edu; chujiao.ma@engr.uconn.edu; cote@engr.uconn.edu; bing@engr.uconn.edu

M. Tehranipoor and C. Wang (eds.), *Introduction to Hardware Security and Trust*,
DOI 10.1007/978-1-4419-8080-9_2, © Springer Science+Business Media, LLC 2012

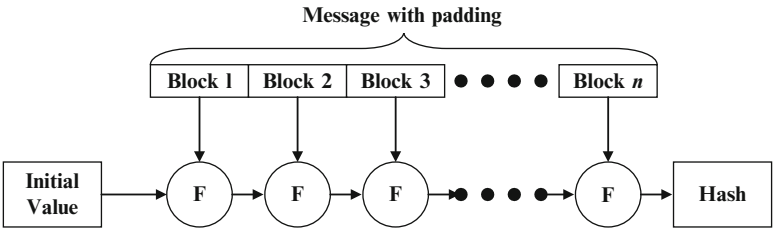


Fig. 2.1 The Merkle-Damgård model of hash functions

resistance, and vice versa. If a hash function is collision resistant (and thus also 2nd-preimage resistant), it is a collision resistant hash function (CRHF). Although collision resistance does not guarantee preimage resistance, most CRHFs in practice appear to be preimage resistant [3].

2.1.1 Construction of Hash Functions

Hash function can be constructed in several ways. The Merkle-Damgård model, illustrated in Fig. 2.1, has shown good properties over the years [4], and has been adopted in the design of many successful hash functions such as MD5 and SHA-2. In this model, the message is padded and divided into blocks of uniform length. The blocks are then processed sequentially with a compression function F. Starting with an initial hash, F repeatedly generates a new intermediate hash value from the previous one and a new message block. The output of the final compression function is the hash of the message. The Merkle-Damgård model makes it easy to manage large inputs and produce a fixed-length output. The security of this scheme relies on the security of the compression function F. It can be proven that if the compression F is collision resistant then the hash function constructed from the Merkle-Damgård model is also collision resistant [4].

MD5, SHA-2, and their variants are the most popular hash algorithms. They follow the Merkle-Damgård model and use logic operations such as AND, OR, and XOR in their compression functions. Recently, collision pairs have been found for MD5, SHA-0 and SHA-1 [5–8], making these algorithms vulnerable to attacks because they do not have the collision resistance property. Although no successful attacks have been reported for algorithms in the SHA-2 family, e.g., SHA-256 and SHA-512, these algorithms may be vulnerable to the same type of attacks because they are designed with similar principles: a Merkle-Damgård model with a compression function consisting of logic operations. Thus, it is imperative to study new compression functions and even new construction models for fast and secure hash functions. In fact, National Institute of Standards and Technology (NIST) held a public competition to select new hash algorithm, SHA-3, starting from 2007 [9].

SHA-3 is expected to be more secure than SHA-2. New hash design techniques are also adopted in many SHA-3 candidates. We will look at some of the candidates later in this chapter.

A hash function can also be constructed from a symmetric-key cipher, such as DES or AES. Several schemes have been proposed to turn a block cipher into a hash function. These scheme include Davies-Meyer, Matyas-Meyer-Oseas [10], and Miyaguchi-Preneel schemes [5, 6, 11, 12]. The Miyaguchi-Preneel scheme also has an iterative structure. It starts with an initial hash and updates the hash when a block of message is processed. The scheme can be expressed as $h' = E_{g(h)}(M_i) \oplus h \oplus M_i$, where M_i is message block i, h the hash before M_i is processed, h' the updated hash after M_i is processed, and $g()$ is a function that converts h to a key for the block cipher E.

Another notable algorithm is Whirlpool, a hash function constructed with the Miyaguchi-Preneel scheme [13]. It has been recommended by the New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project and adopted by International Organization for Standardization (ISO) as part of ISO 10118–3. The block cipher in Whirlpool is a dedicated cipher called W, which is similar to the AES algorithm despite significant differences. No successful attacks to Whirlpool have been found so far. However, the Whirlpool algorithm is much slower than other hash algorithms. For example, on Pentium III, Whirlpool's performance is 36.5 cycles/byte while MD5's is 3.7 cycles/byte and SHA-1's is 8.3 cycles/byte [14].

2.1.2 Application of Hash Functions

Hash algorithms can be used to verify data integrity. After receiving data, one can compute the hash of the received data, and compare it with the hash of the original data, which may be sent through secure channels or obtained from a trusted source. If they are the same, there is a high confidence level that the message was not modified during the transmission (because of the 2nd-preimage resistance). Nowadays, many software download sites provide the MD5 or SHA-2 hash of the software available for downloading.

Hash algorithms can also be used together with public-key algorithms to generate digital signatures. For example, Alice can sign a document by encrypting the hash of the message with her private key. The ciphertext can be used as Alice's signature. Anyone who wants to verify the signature can decrypt the ciphertext with Alice's public key, and compare the decrypted value with the hash generated from the message. This process is shown in Fig. 2.2. The left part of the figure generates the signature with Alice's private key, and the right part checks the signature with her public key.

Hash algorithms can be used for authentication as well. In this case, the hash value of the user's password, instead of the password itself, is transmitted and compared by the server. When computing the hash, the password may be concatenated with a random value generated by the server. Hence, the hashes are different every time, preventing an attacker sniffing on the network traffic

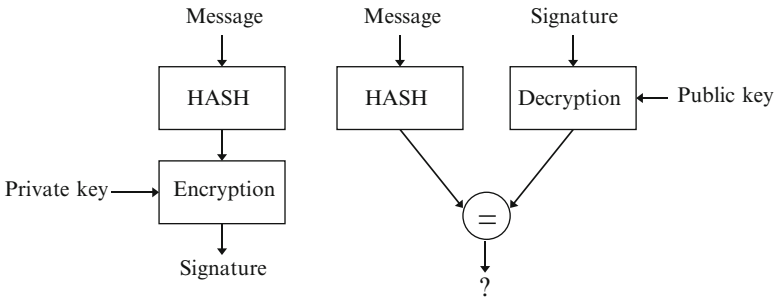


Fig. 2.2 Application of hash algorithm in digital signature

from reusing an old hash. For example, the Challenge-Handshake Authentication Protocol (CHAP) [15] uses this approach in the Point-to-Point Protocol (PPP) [16] for dial-up Internet connections.

Hash algorithms can also be used in Hash-based Message Authentication Code algorithms [17] have been used in many systems to provide both data integrity and message authentication, especially in resource constrained environments where public-key algorithms are too slow and energy consuming.

2.2 Hardware Implementation of Hash Functions

MD5 and SHA-2 (more specifically SHA-256) are commonly used hash functions today. Both of them adopt the Merkle-Damgård construction. As they have much in common, many optimization techniques can be applied to both algorithms.

2.2.1 MD5

MD5 is a common Merkle-Damgård-based hash function. A MD5 block has 512 bits, which can be divided into sixteen 32-bit words $(M_0, \ldots, M_{15})$. The internal hash state has 128 bits, which are stored as four 32-bit words and denoted by the 4-tuple (A, B, C, D), and is initially set to a predetermined initialization vector. The basic computation unit of MD5 is called a step. Each step alters the state variables (A, B, C, D), as shown in Equation 2.1, using a compression function F that takes B, C, and D as input and generates a 32-bit value as output. In the equation, $\lll$ indicates the left rotation operation and $W \lll S$ rotates bits in W to left by S. In total, MD5 has 64 steps. The first 16 steps are called round one, the next 16 round two, and so on. Rounds also have distinct compression functions, as specified in Equation 2.2, where i is the step number.

Table 2.1 A 3-stage MD5 pipeline

Steps	Computation of B
Step i	$B_i = B_{i-1} + (\text{AKM}_i + \text{F}(B_{i-1}, C, D)) \lll S_i$
Step $i+1$	$\text{AKM}_{i+1} = M_i + 1 + (\text{AK}_{i+1})$
Step $i+2$	$\text{AK}_{i+2} = (K_{i+2} + B_{i-1})$

$$\begin{aligned} A_{\text{new}} &= D \\ B_{\text{new}} &= B + (M_i + K_i + A + F(B, C, D)) \lll S_i \\ C_{\text{new}} &= B \\ D_{\text{new}} &= C \end{aligned} \tag{2.1}$$

$$F(B, C, D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & : 0 \leq i \leq 15 \\ (B \wedge C) \vee (C \wedge \neg D) & : 16 \leq i \leq 31 \\ B \oplus C \oplus D & : 32 \leq i \leq 47 \\ C \oplus (B \wedge \neg D) & : 48 \leq i \leq 63 \end{cases} \tag{2.2}$$

$$M_i = \begin{cases} M_i & : 0 \leq i \leq 15 \\ M_{5i+1(\text{mod } 16)} & : 16 \leq i \leq 31 \\ M_{3i+5(\text{mod } 16)} & : 32 \leq i \leq 47 \\ M_{7i \ (\text{mod} 16)} & : 48 \leq i \leq 63 \end{cases} . \tag{2.3}$$

As there are only 16 words in a message block, each word is used four times in the 64 steps. The message scheduler, Equation 2.3, determines which of the 16 message words M_i is used in a particular step i. For example, round one simply uses word M_i for step i, while round two uses $M_{5i+1(mod16)}$. Furthermore, $K_0, \ldots, K_{63}$ and $S_0, \ldots, S_{63}$ are predetermined constant values. After the completion of all 64 steps, the latest values (*A, B, C, D*) are added to the input values (*A, B, C, D*) of the block, using modulo 2^{32} addition. That sum is the updated hash value and serves as the input hash value for the next message block (512 bits) if there is one. The final hash is the concatenation of A, B, C, and D generated from the last block. The specification for MD5 was presented in [1].

2.2.1.1 Pipelining

The operations in each step of MD5 are not complicated. When implemented in hardware, the number of operations that can be done in parallel is limited by data dependency. The data dependency between the state variables of MD5 allows for interesting pipelining applications. In particular, since only B is updated in a step and other words are not, one may use data forwarding as described in [18]. Because some future values of A are known, the computation of B_{i+1} and B_{i+2} may start early. As shown in Table 2.1, the updated values of B can be computed in three stages. When B_i is computed in step i, AKM_{i+1} in step $i+1$ and AK_{i+2} in step $i+2$ can be computed in parallel for B_{i+1} and B_{i+2}.

Another approach to MD5 pipelining is to take advantage of the independency among rounds. For example, in a 4-stage pipeline with one stage dedicated to each

Table 2.2 FPGA implementations of MD5

Type	Block latency (ns)	Throughput (Mbps)
Iterative [21]	843	607
Pipelined (64 stage) [21]	706	725
Parallel (10x) [21]	875	5,857
Pipelined (3 stage) [18]	–	746
Iterative [19]	633	810
Pipelined (32 stage) [19]	764	21,428
Pipelined (32 stage), Loop unrolled [19]	511	32,035

round, the 16 steps in each round are computed iteratively within a stage. Therefore, the stages are much longer. One extension to this architecture is to unroll the loop for the 16 steps in each stage, requiring dedicated logic for each step rather than reusing logic for 16 steps in a pipeline stage. There are also performance gains as the results of the reduction in clock and register delay [19].

2.2.1.2 Other Optimizations

Carry save adders (CSA) are commonly used as an alternative to the full adder when there are multiple numbers to be added. A CSA is much smaller and faster than a regular adder because it does not require carry propagation. A regular adder is needed only at the end of a series of CSA additions. However, in FPGA implementations, the utility of using CSA is diminished since some platforms incorporate partial addition optimizations which allow ripple carry adders to perform just as well while using less area [20].

If the hash function is being implemented in FPGA, block RAM can be used to improve performance. Block RAM is a dedicated part of the chip which may be used to store constants. This takes the burden off the combinatorial logic blocks, freeing them for other purposes. This means that more CLBs are available to the routing algorithm, resulting in more direct routes and faster performance.

2.2.1.3 MD5 Performance

Table 2.2 lists some fast MD5 implementations. The FPGA implementations have reached extremely high throughput up to 32 Gbps. The reader should exercise caution while reviewing these numbers, as a high throughput does not imply a low latency. This is especially true for parallel block implementations. Processing blocks for a single message cannot be done in parallel since they are dependent on each other. Instead, a parallel block implementation is similar to using multiple processing units with independent input streams. Hence, the configuration is only useful when the input is parallel in nature.

Table 2.3 SHA-2 state variables, compression functions, and message scheduler

$A_{\text{new}} = T_1 + T_2$	$T_1 = H + \sum_1(E) + \text{Ch}(E, F, G) + K_t + W_t$
$B_{\text{new}} = A$	$T_2 = \sum_0(A) + \text{Maj}(A, B, C)$
$C_{\text{new}} = B$	
$D_{\text{new}} = C$	
$E_{\text{new}} = D + T_1$	$\text{Ch}(E, F, G) = (E \wedge F) \oplus (\neg E \wedge G)$
$F_{\text{new}} = E$	$\text{Maj}(A, B, C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$
$G_{\text{new}} = F$	$\sum_0(A) = (A \ggg s_1) \oplus (A \ggg s_2) \oplus (A \ggg s_3)$
$H_{\text{new}} = G$	$\sum_1(E) = (E \ggg s_4) \oplus (E \ggg s_5) \oplus (E \ggg s_6)$
$W_t = \begin{cases} M_t & 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$	
$\sigma_0(W_{t-15}) = (W_{t-15} \ggg s_7) \oplus (W_{t-15} \ggg s_8) \oplus (W_{t-15} \gg s_9)$	
$\sigma_1(W_{t-2}) = (W_{t-2} \ggg s_{10}) \oplus (W_{t-2} \ggg s_{11}) \oplus (W_{t-2} \gg s_{12})$	

Note :$\oplus$: bitwise exclusive OR, $+$: addition (modulo 2^{32} or 2^{64}), $\ggg$: rotate right, $\gg$: shift right

2.2.2 SHA-2

SHA-2 has several different modes, SHA-224, 256, 384, and 512, where the numbers denote the length of the hash. Shorthand "SHA-256" and the like refer to specific SHA-2 modes and not SHA-1, which has a 160 bit digest. SHA-2, like MD5, is a Merkle-Damgård-based hash and so the initial setup is similar. For SHA-224 and SHA-256, the message block size (512 bits) is the same as in MD5. However, the number of state variables is doubled. There are eight state variables in SHA-2. In SHA-384 and SHA-512, the word size is also doubled to 64 bits. Thus, the block size increases to 1,024 bits.

There are no rounds in SHA-2, although there are still 64 steps in SHA-256 and 80 steps for SHA-384/512. The message scheduler also involves more than simply shuffling the message chunks. Rotation and exclusive OR are used to combine words in earlier rounds. In addition, a set of new compression functions Ch, Maj, $\sum_0$, and $\sum_1$ replaces MD5's F function. Table 2.3 shows the operations in SHA-2, which are identical for all variants (although the word sizes and constants are different).

2.2.2.1 Iterative Bound Analysis

The optimization of an algorithm is limited by its data dependencies. A method called Iterative Bound Analysis (IBA) has been proposed to find the theoretical maximum performance limit [22]. This means that while advances in silicon process may increase the speed of implementations, the speed gained by refactoring an implementation from an architectural standpoint will have an inherent limit.

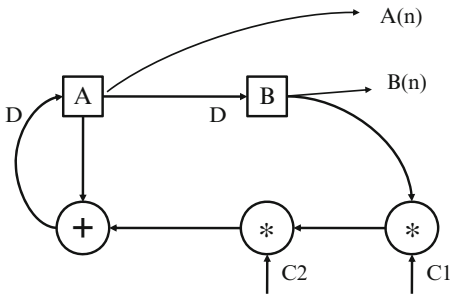


Fig. 2.3 A DFG example from [22]

To find the bound with IBA, the data flow graph (DFG) for the algorithm is first constructed. For example, Fig. 2.3 shows the DFG for the following equation:

$$A(n+1) = A(n) + B(n) * C1 * C2$$
$$B(n+1) = A(n)$$

In the equation, $C1$ and $C2$ are constants; A and B are variables updated in a loop. The DFG has two types of nodes. One type is the variables, such as A and B, and the other type is the operations that are needed for updating the variables, such as + and *. There are also two types of delays. Algorithmic delays, such as updating A and B, are denoted by D on edges that are essential for the algorithm and cannot be removed. Functional delays are from operation nodes, such as + and *, indicating the time needed for performing the operations. IBA assumes all the operations are atomic, i.e., functional operations cannot be split or merged.

A DFG may have loops, like the one shown in Fig. 2.3. A loop is defined as a path that begins and ends at the same node. It contains both algorithmic delays and functional delays. The sum of the functional delays on a loop is the running time of the loop. The performance bound for a loop is given by the loop's running time divided by the number of algorithmic delays in the loop. There may be multiple loops in the DFG. The iterative bound of the algorithm is the maximum of the performance bound for all loops in the DFG. It defines the best throughput that may be reached.

Two techniques are proposed in [22] to make the performance close to that of the iterative bound. One of them is called retiming, which moves the algorithmic delays (the D edges in DFG) through functional nodes. Retiming can reduce the critical path delay, which is the longest total functional delay between two adjacent algorithmic delays, by balancing the functional delays among algorithmic delays. The second technique in [22] is called unfolding, which is similar to loop unrolling. It performs multiple iterations of the loop in a single cycle. After unfolding, the DFG has more functional nodes, thus having more opportunities to reach the optimal throughput. Retiming can also be applied here to balance out the delays in an unfolded DFG.

Table 2.4 SHA-256 implementations

Type	Platform	Block latency (ns)	Throughput (Mbps)
Pipelined (4 stage), Unrolled [25]	FPGA	614	3,334
Operation reordering [20]	FPGA	433	1,184
Loop unrolled [22]	ASIC(0.13 μm)	86	5,975
Pipelined (3 stage) [26]	ASIC (0.13 μm)	∼70	>6,500

Loop unrolling is a commonly used technique in the implementation of hash algorithms. One implementation was able to increase the speed of SHA-512 by 30% with an area penalty of 48% [23]. In [24], SHA-512 achieved a speedup of 24.6% with an area penalty of 19.4 after 2x unrolling. Likewise, 4x unrolling yielded a 28.7% speed up with an area penalty of 75.6%, so returns are diminishing.

2.2.2.2 SHA-2 Performance

Table 2.4 summarizes the best performance of some SHA-2 implementations. The techniques mentioned in the previous section yield good results. However, SHA-2 is actually slower than MD5 because of the increased level of interdependency and complexity. These characteristics limit the amount of data forwarding pipelining and low-level parallel computation that can be applied. Note that we have some extremely low latency ASIC implementations for SHA-2. This is primarily due to high clock rate, e.g., approximately 1 GHz in [26, 27].

2.2.3 Optimized for Area

Sometimes a compact design is required for platforms with limited resources such as area or power. An example is RFID tags, which are very limited in both aspects. They typically have a maximum average current draw of 15 μA total [28]. As of 2009, RFID tags typically have 3,000 ∼ 4,000 gates available for cryptographic purposes [29].

The smallest implementation for SHA-256 was reported in [30], where the authors obtained a size of 8,588 gates on a 250 nm CMOS process. This implementation is capable of a throughput of 142 Mbps. The authors used a folding technique to optimize for area. This is basically the opposite of the unrolling process described earlier. In this case, a single iteration takes seven cycles to complete. While there are seven modulo addition operations in SHA-2, only one physical adder is synthesized and then reused in each step. Another technique shrinks the message scheduler, which takes up more space than the compression function. This reduction is primarily due to using only a single 32-bit register rather than sixteen 32-bit registers. These registers were used for timing purposes to obtain W_{t-2}, W_{t-7},

W_{t-15} and W_{t-16} by having the 16 registers in series so that the correct W values are always available. The area optimized version's single register holds temporary values that are written back into the memory block. The needed W values for each computation are stored in the memory rather than sequential registers.

Some applications do not require all the properties of a secure cryptographic hash function. For example, some applications may need only one-wayness, not requiring collision resistance. Therefore, some special purpose hash functions that have only the needed secure properties for particular applications can be implemented with smaller area. The downside of using a less known algorithm is that it has not been subject to as much scrutiny as popular algorithms have been. Therefore, the level of security provided by these hash functions is not well known. On the other hand, an obscure algorithm does have a unique (perhaps remote) advantage over popular ones in that someone may have indeed found an attack to a popular algorithm but has declined to publicize it.

2.3 SHA-3 Candidates

Due to security concerns of SHA-1 and recent advances in the cryptanalysis of hash algorithms, NIST held a public competition for a new hash algorithm standard, SHA-3, which is meant to replace SHA-2. Therefore, the new algorithm is expected to be more secure SHA-2. As of now, early 2011, the competition is at the final round, with five algorithms up for consideration. This section examines the constructions and properties of all five algorithms in the final round: Keccak based on sponge construction, BLAKE based on HAIFA construction, Grøstl based on the chop-Merkle-Damgård iterations, Skein based on tweakable block cipher Threefish and Unique Block Iteration (UBI), and JH based on iterative construction with generalized AES methodology [31].

2.3.1 Keccak

Keccak is a family of cryptographic hash functions based on sponge construction. Sponge construction, shown in Fig. 2.4, is a simple iterated construction with a variable-length input and arbitrary length output based on a fixed length transformation (or permutation) operating on a fixed number of bits. The fixed number of bits is the width of the permutation, or bit state b. The bit state is the sum of bit rate r and bit capacity c. The permutation in this case is called Keccak-f, the most important building block of this algorithm. There are seven Keccak-f permutations, indicated by Keccak-[b], where $b = 25 \times 2^{\ell}$ and ℓ ranges from 0 to 6 producing the values $b = 25, 50, 100, 200, 400, 800$, and 1,600. The default permutation is Keccak-f [1600] with $r = 1,024$ and $c = 576$ [32].

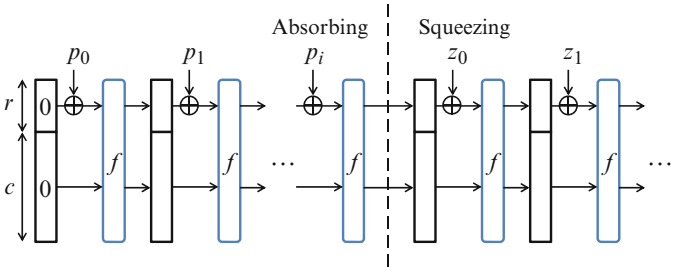


Fig. 2.4 Diagram of sponge construction

θ *step*

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \quad \forall x \; in \; 0 \ldots 4$$

$$D[x] = C[x-1] \oplus (C[x+1] \lll 1) \quad \forall x \; in \; 0 \ldots 4$$

$$A[x,y] = A[x,y] \oplus D[x] \quad \forall (x,y) \; in \; (0 \ldots 4, 0 \ldots 4)$$

ρ *and* π *step*

$$B[y, 2x+3y] = A[x,y] \lll r[x,y] \quad \forall (x,y) \; in \; (0 \ldots 4, 0 \ldots 4)$$

x *step*

$$A[x,y] = B[x,y] \oplus \big((\neg B[x+1,y]) \wedge B[x+2,y] \big) \quad \forall (x,y) \; in \; (0 \ldots 4, 0 \ldots 4)$$

t *step*

$$A[0,0] = A[0,0] \oplus RC$$

return A

Fig. 2.5 Pseudocode of Keccak-f

Before any permutation is performed, Keccak initializes state s and pads the message to make the string a multiple of r. The padded message, represented by P, is then divided into i blocks. For sponge construction, there are two phases. In the first phase (absorbing) the r-bit input message blocks are XOR-ed into the first r bits of the state, interleaved with applications of the function f (Keccak-f). This phase is finished when all message blocks are processed. In the second phase (squeezing) the first r bits of the state are returned as hash bits, interleaved with applications of the function f. This phase is finished when the desired length of hash is produced [33].

The basic block in Keccak is Keccak-f, an iterated permutation consisting of a sequence of almost identical rounds [4]. The number of rounds n_r depends on the permutation width b, and is computed as $n_r = 12 + 2l$, where $2^l = b/25$. This gives 24 rounds for Keccak-f[1600]. Each round consists of five steps illustrated in Fig. 2.5. In the figure, A denotes the complete permutation state array. $A[x, y]$ denotes a particular word in the 5×5 state array. $B[x, y]$, $C[x]$, and $D[x]$ are intermediate variables, and $C[x]$ and $D[x]$ are vectors. All the operations on the indices are done modulo 5. Rotation amounts $r[x, y]$ are constants. RC in the t step is the round constant. All the rounds of Keccak-f perform identical operations, except for using different round constant RC (consult [34] for the computation of RC).

Keccak-f is designed such that the dependence on CPU word length is only due to fixed rotations, leading to an efficient use of CPU resources on a wide range of processors. The default bit state for Keccak-f permutation (1,600) is chosen in favor of 64-bit architectures. When implemented on 32-bit processors, the rotation on 64-bit words is the only operation that cannot be simply divided into 32-bit operations. The algorithm's symmetry allows the exploitation of SIMD instructions and pipelining in processors, which results in compact code in software, or a compact coprocessor circuit suitable for constrained environments [34].

The full Keccak algorithm design has no known security vulnerabilities because the sponge construction has been successful against generic attacks. It is simple, allows variable length output, and is flexible (such as the tradeoff between bit rate and security).

2.3.2 BLAKE

Blake is based on the HAIFA iteration mode with a compression function that uses a modified version of the stream cipher ChaCha. While Merkle-Damgård only uses the previous hash value and the current message block in the compression function, HAIFA also uses a salt and a counter that indicates how many blocks have been processed as part of the input [35].

Depending on the message size and word length, different BLAKE hash functions can be used. BLAKE-224 and BLAKE-256 operate on 32-bit words, with block size of 512 bits and salt of 128 bits. BLAKE-384 and BLAKE-512 operate on 64-bit words with block size of 1024 bits, salt of 256 bits. The numbers in the algorithm name indicate the digest length.

All four variants use the same compression function and only differ by the initial value, the message padding, and the truncation of the output. The compression function of BLAKE adopts the wide-pipe design. A large inner state is initialized, injectively updated by message-dependent rounds, then compressed to return the next chain value. The actual compression method is based on modified version of ChaCha, which is a variant of Salsa20 family of stream ciphers. In the following, BLAKE-512 will be used as the example to explain the algorithm.

In BLAKE-512, the message is first padded and divided into blocks of 1,024 bits. The intermediate hash value is then initialized. The initial values used in BLAKE are same as those used in SHA-2 (e.g., BLAKE-512 uses the values from SHA-512, BLAKE-256 uses the values from SHA-224). Then, BLAKE iteratively updates the intermediate hash value using the compression function: $h_{i+1} \leftarrow \text{compress}(h_i, m_i, s, t_i)$, where m_i is the message block, s is the salt, and t_i is the counter indicating the number of bits that have been processed so far.

The compression function of BLAKE-512 has 16 rounds. The inner state of the compression function is represented as a 4×4 matrix of words. For each round, a nonlinear function G that operates on four words is applied to columns and diagonals of the state. This process is illustrated in Fig. 2.6. First, all four columns

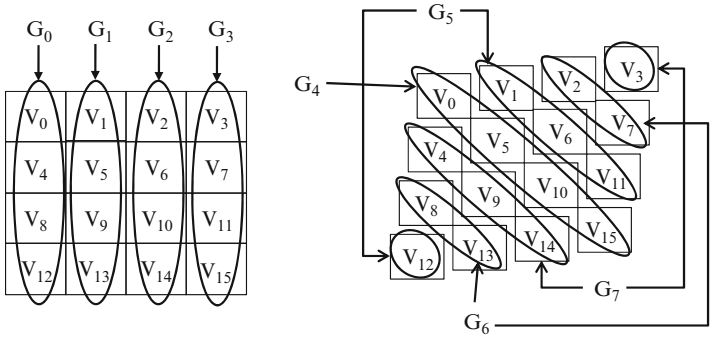


Fig. 2.6 Column step (*on the left*) and diagonal step (*on the right*) in BLAKE

Fig. 2.7 Pseudocode of G in BLAKE-512

$$
\begin{aligned}
a &\leftarrow a + b + \left(m_{\sigma_r(2i)} \oplus C_{\sigma_r(2i+1)}\right)\\
d &\leftarrow (d \oplus a) \ggg 32\\
c &\leftarrow c + d\\
b &\leftarrow (b \oplus c) \ggg 25\\
a &\leftarrow a + b + \left(m_{\sigma_r(2i+1)} \oplus C_{\sigma_r(2i)}\right)\\
d &\leftarrow (d \oplus a) \ggg 16\\
c &\leftarrow c + d\\
b &\leftarrow (b \oplus c) \ggg 11
\end{aligned}
$$

are updated independently with $G_0, \ldots,$ and G_3. This procedure is called a column step. Thereafter, the four disjoint diagonals are updated with $G_4, \ldots,$ and G_7. This procedure is called a diagonal step. Note that $G_0, \ldots,$ and G_3 can be computed in parallel because each of them updates a distinct column of the matrix. Similarly, $G_4, \ldots,$ and G_7 update distinct diagonals and thus can be parallelized as well.

The pseudocode of $G_i(a,\ b,\ c,\ d)$ is shown in Fig. 2.7, where σ_r represents a permutation of integers between 0 and 15. There are ten different permutations, which are reused when the round number is ten or greater. For example, round 10 uses σ_0 and round 11 uses σ_1.

After performing the round function, the new hash values are extracted from the state and salt:

$$
\begin{aligned}
h'0 &\leftarrow h0 \oplus s0 \oplus v0 \oplus v8\\
h'1 &\leftarrow h1 \oplus s1 \oplus v1 \oplus v9\\
&\ldots\\
h'7 &\leftarrow h7 \oplus s3 \oplus v7 \oplus v15
\end{aligned}
\quad .
$$

As the salt has only four words, each word is used twice. For example, $s0$ is used in $h'0$ and $h'4$.

Lastly, the algorithm truncates the final chaining value if needed. The truncation method slightly differs depending on the version of the algorithm [36].

There are several important design choices in BLAKE. The HAIFA iteration mode provides resistance to long-message second preimage attacks, and the local wide-pipe internal structure makes local collisions impossible. The compression algorithm is based on ChaCha, which has good performance and has been intensively analyzed. The properties of compression function also allow parallel processing. The G functions performed on the four columns and diagonals can be done in parallel, which allows convenient performance-area trade-off. The performance can be improved if more resources are available for performing multiple G functions in parallel.

2.3.3 Grøstl

Grøstl is based on the wide-pipe design and chop-Merkle-Damgård iterations. A wide-pipe design has an internal state that is larger than the final hash output. When all the message blocks are processed, the internal state is truncated to get the final hash output. This process is similar to chop-Merkle-Damgård, except Grøstl applies a final output transformation before truncating the result to the desired output length. The returned message digests can be any number of bits from 8 to 512, in 8-bit steps. The hash function processing message digest of size n is called Grøstl-n.

In Grøstl, the message is first padded to a multiple of l, the length of a message block. Then the padded message is split into l-bit message blocks m_1 to m_t, and each message block is processed sequentially. The compression function comes in two variants: for Grøstl that outputs up to 256 bits, l is defined to be 512; for larger outputs, l is 1024. An l-bit initial value is set as the initial hash value $h_0 = iv$, and then the message blocks are processed as follows:

$$h_i \leftarrow f(h_{i-1}, m_i) \quad \text{for } i = 1, \ldots, t$$

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h,$$

where f is the compression function and h_i is the updated state after processing message block i. The compression function consists of two permutations P and Q. The two permutations are identical except for the different round constants. Both permutations consist of the following four round transformations:

- AddRoundConstant
- SubBytes
- ShiftBytes (or ShiftBytesWide in permutations for 1,024-bit blocks)
- MixBytes

Both P and Q have the same number of rounds, which depends on the internal state size: ten rounds are recommended for $l = 512$ and fourteen rounds for $l = 1{,}024$. In each round, the round transformation is applied on a state represented by a matrix A of bytes. For the short variant, the matrix is 8 by 8, and for the large variant it is 8 by 16.

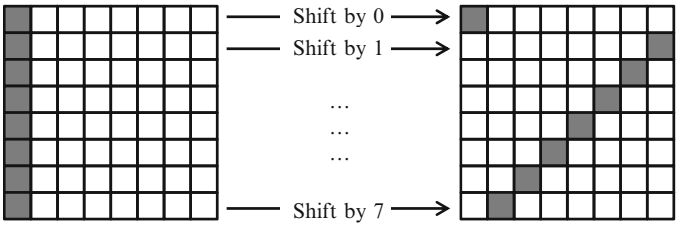
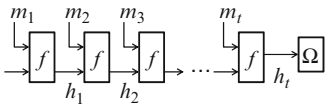


Fig. 2.8 ShiftBytes in Grøstl

Fig. 2.9 Diagram of Grøstl hashing process



The first step is AddRoundConstant, which adds (XOR) a round-dependent constant to the state matrix. The process in round i can be represented as $A \leftarrow A \oplus C_P[i]$ for permutation P and $A \leftarrow A \oplus C_Q[i]$ for permutation Q, where A is the state and $C_P[i]$ and $C_Q[i]$ are the round constants.

The second step is SubBytes, which substitutes each byte in the state matrix with its image taken from the S-Box. Grøstl uses the same S-Box as AES. If $a_{i,j}$ is the element in row i and column j of A, SubBytes transformation can be denoted as $a_{i,j} \leftarrow S(a_{i,j}), 0 \leq i < 8, 0 \leq j < v$.

The third step is ShiftBytes or ShiftBytesWide, which cyclically shift the bytes within a row to the left by the number of positions, wrapping as necessary. ShiftBytes is illustrated in Fig. 2.8. ShiftBytesWide uses the same method but each row has 16 bytes.

The last step of the transformation of the compression function is MixBytes, which multiplies each column of A by an 8×8 matrix of constants, defined as B. The transformation can be written as $A \leftarrow B \times A$.

When the last message block has been processed, the resulting h_t is then put through function Ω. The final hash is $H(m) = \Omega(h_t) = \text{trunc}_n(P(h_t) \oplus h_t)$. The function $\text{trunc}_n(x)$ discards all but the trailing n bits of x and $n < l$ [37]. The process is illustrated in Fig. 2.9.

Grøstl is very flexible. It can generate digest of many sizes, and seems secure against known attacks. The permutations are constructed with wide-pipe design, which makes all known, generic attacks on the hash function much more difficult. Therefore, it is possible to give strong statements about the resistance of Grøstl against large classes of cryptanalytic attacks. As Grøstl is based on AES, its countermeasures against side-channel attacks are well understood.

2.3.4 Skein

Skein is a family of hash functions based on tweakable block cipher Threefish and Unique Block Iteration (UBI) [38]. Tweakable block cipher allows Skein to hash configuration data along with the input text in every block and greatly improves Skein's flexibility. UBI is a chaining mode that combines an input chaining value with an arbitrary input size to a fixed output size.

Skein has three different internal state sizes, which are also the message block sizes: 256, 512, and 1,024 bits. Skein-512 is the default variant and consists of 72 rounds. Skein-1024 is an ultra-conservative and highly secure variant and consists of 80 rounds. On the other end, Skein-256 is a low-memory variant that can be implemented using about 100 bytes of RAM and consists of 72 rounds.

The UBI chaining mode combines an input chaining value with an arbitrary length input string and produces a fixed size output. Each UBI computation takes a message block and a 128-bit tweak value as input. The tweak value encodes how many bytes have been processed so far, whether this is the first and/or last block of the UBI computation, and what type of UBI computation it is. The types of UBI computations include key, configuration block, message, and output. A UBI call takes three parameters as input: the output of the previous UBI call (0 for the first call), the current content to be processed, and the type of content.

When used as a straightforward hash function, Skein consists of three UBI calls: a configuration call with the chaining value initialized to 0, then a message call that takes a message of up to $2^{96} - 1$ bytes long, and lastly an output call that truncates the chained value to produce the final hash.

- $IV = \text{UBI}\ (0, \mathit{Config}, \text{CFG})$
- $G = \text{UBI}\ (\mathit{IV}, M, \text{MSG})$
- $\mathit{Output} = \text{Truncate}\ (\text{UBI}(G, 0, \text{OUT}), o)$.

The UBI chaining mode is based on the tweakable block cipher Threefish. In Skein-512, the UBI has 72 rounds, each consisting of four simple nonlinear functions (MIX), followed by a permutation of the eight 64-bit words. The idea is that a larger number of simple rounds are more secure than fewer number of complex rounds. MIX, the basic block of Threefish, operates on two 64-bit words at a time. It uses only three mathematical operations: XOR, addition, and rotation on 64-bit words. Suppose the two words are A and B, the function of MIX can be expressed as $(A, B) = (A + B, (B <<< R) \oplus (A + B))$ [39], where R is a constant. The permutation on eight words that follows the MIX operation is the same for all rounds.

In Skein-512, a subkey is injected every four rounds, and the rotation constants R are repeated every eight rounds. The subkeys are generated from key words, tweak words, and counter value. For more details on subkey generation, please refer to Sect. 2.3.3 in the Skein submission document [38].

Skein has the option to include keys or to work in hash tree mode. To use Skein as a MAC function or any keyed hash function, a UBI call for the keys, with the

first input set to 0, is done before the configuration block. The output of the key UBI call is used as the first input of the configuration UBI call. In the optional hash tree mode, each leaf UBI call takes a message block as the input, and the output of every two UBI calls goes into a UBI call at the next level until only the root UBI remains.

Skein is designed to be simple, secure, and efficient. It is based only on three primitive operations: XOR, addition, and rotation of 64-bit words. The best attack made by the author on Threefish-512 is on 35 of the 72 rounds. Skein is quite efficient on a variety of platforms, especially on 64-bit processors. Skein-512, the default or primary variant, can be implemented in approximately 200 bytes of state. Skein-512 hashes data at 6.1 clock cycles per byte on a 64-bit CPU. This means that on a 3.1 GHz x64 Core 2 Duo CPU, Skein hashes data at 500 Mbps per core, almost twice as fast as SHA-512.

2.3.5 JH

JH is an iterative hash algorithm that produces hash values of 224, 256, 384, and 512 bits. The compression functions for all four versions are the same, and JH-512 will be used in the following discussion. For hardware implementation, the round functions of JH block cipher are identical and use techniques similar to the AES row rotations.

In JH-512, the message is first padded to a multiple of 512 bits, and then divided into blocks of four 128-bit words. For each iteration, the message block is put through the compression function F_8 to update the chaining value H_i of 1,024 bits: $H_i = F_8(H_{i-1},\ M_i)$. For the first iteration, $H_0 = F_8(H_{-1},\ M_0)$, where H_{-1} consists of two bytes representing the message digest size followed by 0s, and M_0 is set as 0. F_8 first compresses the 512-bit message block M_i with the first half of the previous chaining value H_{i-1}, feeds the result to function E_8 (to be described later), and then combines the output of E_8 with the message block. After all blocks have been processed, the n-bit hash value of the message is the last n bits of H [40].

The steps of compression function for each message block in JH-512, $H_i = F_8(H_{i-1}, M_i)$, is illustrated below, where A, and B denote two 1024-bit words:

1. $A_j = H_{i-1,j} \oplus M_i\ for 0 \le j \le 511$;
   $A_j = H_{i-1,j}\ for 512 \le j \le 1023$;
2. $B = E_8\ (A)$;
3. $H_{(i),j} = B_j\ for 0 \le j \le 511$;
   $H_{(i),j} = B_j \oplus M_i\ for 512 \le j \le 1023$;

The bijective function E_8 used in the compression function F_8 is based on d-dimensional generalized AES methodology (d is 8 in this case). The computation of $B = E_8\ (A)$ is given as follows:

1. Group the 1,024 bits in A into 2^8 4-bit elements to obtain Q_0;
2. For $r = 0$ to 34, $Q_{r+1} = R_8(Q_r, C_r^{(8)})$;

3. $Q_{36} = R_8^*(Q_{35}, C_{35}^{(8)})$;
4. De-grouping the 2^8 4-bit elements in Q_{36} to obtain B.

Each $C_r^{(8)}$ is a 2^8-bit constant. The computation of $Q = R_8(A, C_r^{(8)})$ consists of the following three steps and R_8^* only has step 1.

1. Consider A as 256 4-bit elements and replace each element with its image in an S-Box. There are two S-Boxes, and a bit in $C_r^{(8)}$ decides which S-Box to use for the corresponding 4-bit element. The results can be represented by 256 4-bit elements v_i $(0 \leq i \leq 255)$.
2. A linear transformation L is applied on each pair of 4-bit elements. The output of this step is $(w_{2i}, w_{2i+1}) = L(v_{2i}, v_{2i+1})$ for $0 \leq i \leq 127$. The linear transformation L implements a (4, 2, 3) Maximum Distance Separable code over GF (2^4). Each bit in (w_{2i}, w_{2i+1}) is an XOR of a set of bits from (v_{2i}, v_{2i+1}).
3. A permutation is done on the 256 4-bit elements generated in Step 2. $(Q_0, Q_1, \ldots, Q_{2^8-1}) = P_8(w_0, w_1, \ldots, w_{2^8-1})$.

For step 3, P_8 is a composition of three permutations: $P_8 = \phi_8 \circ P_8' \circ \pi_8$. All three functions are permutation on 2^8 elements. And the details of the three permutations are described as follows:

$$
\begin{aligned}
&B = \pi_8(A):\\
&B_{4i+0} = a_{4i+0} \ for\ i = 0\ to\ 2^6 - 1;\\
&B_{4i+1} = a_{4i+1} \ for\ i = 0\ to\ 2^6 - 1;\\
&B_{4i+2} = a_{4i+2} \ for\ i = 0\ to\ 2^6 - 1;\\
&B_{4i+3} = a_{4i+3} \ for\ i = 0\ to\ 2^6 - 1;\\
&B = P_8'(A):\\
&\quad b_i = a_{2i} \ for\ i = 0\ to 2^7 - 1;\\
&b_{i+2^7} = a_{2i+1} \ for\ i = 0\ to\ 2^7 - 1;\\
&B = \phi_8(A):\\
&\quad b_i = a_i \ for\ i = 0\ to\ 2^7 - 1;\\
&b_{2i+0} = a_{2i+1} \ for\ i = 2^6\ to\ 2^7 - 1;\\
&b_{2i+1} = a_{2i+0} \ for\ i = 2^6\ to\ 2^7 - 1;
\end{aligned}
$$

The compression function of JH algorithm is illustrated in Fig. 2.10. The message block is then XOR-ed with the first half of the previous hash value, the result of which is put through function E, which consist of a number of rounds that include S-Box, linear transformation L, and permutation P_d. The message is then XOR-ed with the second half of the output of E, which produces the updated hash value.

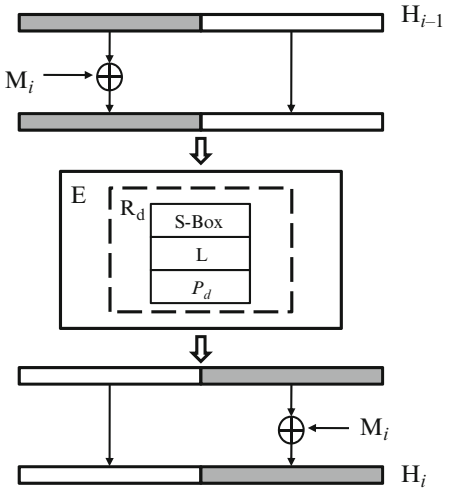


Fig. 2.10 Diagram of the compression function in JH

In JH-512, each message block has 64 bytes and passes through the 35.5 round compression function that involves 9216 4×4 bit S-Boxes. The large number of active S-Boxes ensures that JH is strong against differential attacks. As the key of the block cipher is constant, no extra variables are introduced into the middle of the compression function. Therefore, it is much easier to analyze the security with respect to differential attacks. As the block cipher outputs do not need to be truncated, the structure is also efficient.

2.3.6 Performance

Before we compare the performance of the candidate algorithms, we first give an overview of major operations in each algorithm. Table 2.5 summarizes the major operations in five algorithms in the final round of the SHA-3 competition. Most operations used in the five algorithms are simple. When implemented in hardware, fixed permutations and rotations with fixed amounts can be done with careful wire routing. S-Box can be implemented with logic operations. The more expensive operations include matrix multiplication in Grøstl and addition in BLAKE and Skein.

In addition to security, performance is one of the main criteria for evaluating the SHA-3 candidates. However, it is very challenging to compare all SHA-3 candidates due to various choices of technologies and multiple optimization goals. Most performance evaluation in literature is done on one to three algorithms at a time and cannot be adequately compared with each other due to difference in implementation and measurements.

Table 2.5 Major operations in five SHA-3 algorithms

Algorithm	Block size	No. of round	Major operations in a round
Keccak-1600	1,024 bits	24	$\oplus$, $\lll$, $\neg$, $\wedge$ on 64-bit words
BLAKE-512	1,024 bits	16	$+$, $\oplus$, $\ggg$ on 64-bit words
Grøstl-512	1,024 bits	16	$\oplus$, 8-bit S-Box, ShiftRow, matrix multiplication. Mainly 8-bit operations
Skein-512	512 bits	72	$+$, $\lll$, $\oplus$ operations on 64-bit words. 1 permutation of eight 64-bit words
JH-512	512 bits	35	$\oplus$, 4-bit S-Box, and a permutation of 256 4-bit elements

When implemented with hardware, fixed permutations, $\lll$, and $\ggg$ can be done with wire routing.

Table 2.6 Performance of five final SHA-3 candidates on FPGA [41]

Algorithm	Area (slices)	Max. frequency (MHz)	Throughput (Mbps)	Throughput/area (Mbps/slice)
SHA-2–256	656	125	985	0.966
SHA-2–512	1,213	110	1,264	0.713
Keccak-256	1,117	189	6,263	3.17
Keccak-512	1,117	189	8,518	4.32
BLAKE-32	1,118	118	1,169	0.707
BLAKE-64	1,718	91	1,299	0.449
Grøstl-256	2,391	101	3,242	1.257
Grøstl-512	4,845	123	3,619	0.799
Skein-512	1,786	84	1,945	0.706
JH	1,291	250	1,941	1.1

The FGPA implementations of the algorithms are compared in [41]. The results are summarized in Table 2.6. To fairly analyze the designs and variants, all designs were implemented in slice logic on a Virtex-5 FPGA. Message padding for all designs were included as part of the hardware, and the test was for message digest of size 224, 256, 384, and 512. The efficiency of the architecture is also compared in terms of throughput per unit area. According to the results in [41], Keccak has higher throughput while requiring smaller area.

Reference [42] compared the ASIC implementations of round-two candidates. The results are summarized in Table 2.7. The implementations are optimized for maximum peak throughput, with consideration for reasonable area. If the throughput can only be increased a few percent further at the cost of increase in area of several dozen percent, the slightly lower throughput with more area-efficient implementation is considered. The important implementation decisions in [42] for each algorithm are listed later.

- Keccak: One Kekkak-f round per cycle.
- BLAKE: Four G functions are implemented in parallel; two pipeline registers; additional cycle for chaining; carry-save adders.

Table 2.7 Comparison of five SHA-3 candidates with SHA-256 in [42][a]

Algorithm	Latency (cycles)	Area (GE)	Clock frequency (MHz)	Throughput (Gbps)
BLAKE-32	22	38,877	144.15	3.355
JH-256	39	51,212	259.54	3.407
Keccak-256	25	56,713	267.09	11.624
Grøstl-256	22	53,680	202.47	4.712
Skein-256	10	47,678	64.75	1.658
Skein-512	10	76,250	43.49	2.227
SHA-256	66	19,515	211.37	1.640

[a]Use the UMC 0.18 μm FSA0A_C standard-cell library

Table 2.8 The best performance of five SHA-3 candidates and SHA-256

Algorithm	Technology	Area	Frequency	Throughput (Gbps)
Keccak-1600[43]	ASIC (0.13 μm)	48 kGE	526 MHz	22
Keccak-1600[43]	FPGA Stratix III	4,684 ALUT	206 MHz	8.5
BLAKE-512 [44]	ASIC (90 nm)	79 kGE	532 MHz	18.8
BLAKE-512 [45]	FPGA Virtex 5	108 slices	358 MHz	0.3
Grøstl-256 [46]	ASIC (0.18 μm)	59 kGE		6.3
Grøstl-512 [47]	FPGA Virtex 5	19k slices	84 MHz	6.1
Skein-512 [39]	ASIC (32 nm)	61 kGE	1.13 GHz	58
Skein-512 [48]	FPGA Virtex 5			0.82
JH-256 [42]	ASIC(0.18 μm)	51 kGE	260 MHz	3.4
JH-512 [41]	FPGA Virtex 5	1.7k slices	144 MHz	1.9
SHA-256 [26]	ASIC (0.13 μm)		∼1 GHz	>6.5

- Grøstl: Shared P/Q permutation; S-Boxes and MixBytes separated; S-Boxes with one pipeline register.
- Skein: Eight Threefish rounds unrolled; generic adders.
- JH: 320 S-Boxes (one cycle per R8 round); combinational S-Boxes.
- SHA-2: No unrolling or quasi-pipelining; generic adders.

Table 2.8 shows the best performance of the five algorithms reported in literature, in terms of throughput. Skein has the highest throughput for ASIC, followed by Keccak. Keccak also has the highest throughput for FPGA implementations, followed by Grøstl. BLAKE and Skein have good performance in ASIC implementations, but poor performance on FPGA. The slow adders on FPGA may cause the performance degradation.

As the final SHA-3 algorithm has not been decided yet, the algorithm may still be tweaked and improved. For example, Keccak went through minor revision during each round of the competition. BLAKE went through minor revision during round 2 of the competition. The performance of BLAKE shown here is for the version before revision. There are no changes in the other three algorithms so far. On the other hand, as the competition is already in the final round, no major changes are expected from all candidates. Therefore, the impact of potential changes on the implementation should be small.

From Table 2.8, we can also see that four out of five SHA-3 candidates in the final rounds have an implementation that has higher throughput than SHA-256. Therefore, SHA-3 is expected to be more secure and, at the same time, have higher throughput than SHA-2.

References

1. Rivest R (1992) The MD5 message-digest algorithm. In: The Internet Engineering Task Force (ITEF) Internet Draft, no. RFC-1321, April 1992
2. National Institute of Standards and Technology (1994) Secure hash standard. In: Federal Information Processing Standards Publication 180–1, April 1994
3. Menezes A, Oorschot P, Vanstone S (1996) Handbook of Applied Cryptography, 1st edn. CRC Press, West Palm Beach, FL, USA
4. Damgard I (1990) A design principle for hash functions. In: Proceedings of Cryptology, Crypto '89, vol 435, pp 416–427
5. Wang X, Feng D, Lai X, Yu H (2004) Collisions for hash functions: MD4, MD5, HAVAL-128 and RIPEMD. http://eprint.iacr.org/2004/199.pdf. Accessed August 2004
6. Wang X, Yu H, Yin YL (2005) Efficient collision search attacks on SHA-0. In: Advances in Cryptology – CRYPTO'05, vol 3621, pp 1–16
7. Wang X, Yin YL, Yu H (2005) Finding collisions in the full SHA-1. In: Advances in Cryptology – CRYPTO'05, vol 3621, pp 17–36
8. Wang X, Hongbo Y (2005) How to break MD5 and other hash functions. In: Advances in Cryptology EUROCRYPT 2005, pp 19–35
9. National Institute of Standard and Technology (2007) Cryptographic hash algorithm competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html. Accessed November 2007
10. Matyas SM, Meyer CH, Oseas J (1985) Generating strong one-way functions with cryptographic algorithm. IBM Tech Disclosure Bull 27(10A): 5658–5659
11. Preneel B, Govaerts R, Vandewalle J (1989) Cryptographically secure hash functions: an overview. In: ESAT Internal Report, K. U. Leuven
12. Miyaguchi S, Iwata M, Ohta K (1989) New 128-bit hash function. In: Proceedings 4th International Joint Workshop on Computer Communications, pp 279–288
13. Barreto PSLM, Rijmen V (2000) The Whirpool hash function. http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html. Accessed November 2000
14. Nakajima J, Matsui M (2002) Performance analysis and parallel implementation of dedicated hash functions. In: Proceedings of EUROCRYPT 2002, Lecture Notes in Computer Science, vol 2332, pp 165–180
15. Lloyd B et al. (1992) PPP authentication protocols. In: The Internet Engineering Task Force (ITEF) Internet Draft, RFC-1334, October 1992
16. Simpson W (1994) The point-to-point protocol. In: The Internet Engineering Task Force (ITEF) Internet Draft, RFC-1661, July 1994
17. National Institute of Standards and Technology (2002) The keyed-hash message authentication code (HMAC). In: FIPS PUB, vol 198
18. Hoang AT, Yamazaki K, Oyanagi S (2008) Multi-stage pipelining MD5 implementations on FPGA with data forwarding. In: 16th International Symposium on Field-Programmable Custom Computing Machines, pp 271–272, April 2008
19. Wang Y, Zhao Q, Jiang L, Yi S (2010) Ultra high throughput implementations for MD5 hash algorithm on FPGA. In: High Performance Computing and Applications, pp 433–441
20. Chaves R, Kuzmanov G, Sousa L, Vassiliadis S (2006) Improving SHA-2 hardware implementations. In: Cryptographic Hardware and Embedded Systems-CHES 2006, pp 298–310

21. Jarvinen K, Tommiska M, Skytta J (2005) Hardware implementation analysis of the MD5 hash algorithm. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences, vol 9, p 298a
22. Lee YK, Chan H, Verbauwhede I (2007) Iteration bound analysis and throughput optimum architecture of SHA-256 (384,512) for hardware implementations. In: Proceedings of the 8th international conference on Information security applications, vol 256, pp 102–114
23. Lien R, Grembowski T, Gaj K (2004) A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512. In: Topics in Cryptologyâ CT-RSA 2004, pp 1995–1995
24. Crowe F, Daly A, Kerins T, Marnane W (2005) Single-chip FPGA implementation of a cryptographic co-processor. In: Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No.04EX921), pp 279–285
25. Athanasiou G, Gregoriades A, Panagiotou L, Goutis C, Michail H (2010) High throughput hardware/software co-design approach for SHA-256 hashing cryptographic module in IPSec/IPv6. Global J Comput Sci Technol 10(4): 54–59
26. Dadda L, Macchetti M, Owen J (2004) An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512). In: ACM Great Lakes Symposium on VLSI, pp 421–425
27. Dadda L, Macchetti M, Owen J (2004) The design of a high speed ASIC unit for the hash function SHA-256 (384, 512). In: Proceedings Design, Automation and Test in Europe Conference and Exhibition, vol 256, pp 70–75
28. Feldhofer M, Wolkerstorfer J (2007) Strong crypto for RFID tags – a comparison of low-power hardware implementations. In: 2007 IEEE International Symposium on Circuits and Systems, pp 1839–1842, May 2007
29. Peris-Lopez P, Hernandez-Castro J, Tapiador J, Ribagorda A (2009) Advances in ultra-lightweight cryptography for low-cost RFID tags: Gossamer protocol. Inform Security Appl 56–68
30. Kim M, Ryou J, Jun S (2009) Efficient hardware architecture of SHA-256 algorithm for trusted mobile computing. Architecture. Springer Verlag, Berlin, Heidelberg, New York, pp 240–252
31. Perlner R, Chang S, Kelsey J, Nandi M, Paul S, Regenscheid A (2009) Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition. September 2009
32. Bertoni G, Daemen J, Peeters M, Assche GV (2009) Keccak specifications Version 2. http://keccak.noekeon.org/Keccak-specifications-2.pdf. Accessed July 2011
33. Morawiecki P, Srebrny M (2010) A SAT-based Preimage Analysis of Reduced KECCAK Hash Functions. Santa Barbara, CA, 23–24 August 2010
34. Bertoni G, Daemen J, Peeters M, Assche GV (2010) Keccak sponge function family main document. http://keccak.noekeon.org/Keccak-main-2.1.pdf. Accessed June 2010
35. Biham E, Dunkelman O (2006) A framework for iterative hash functions: HAIFA. In: Second NIST Cryptographic Hash Workshop
36. Henzen L, Meier W, Raphael C-W, Phan, Aumasson J-P (2009) SHA3 Proposal BLAKE. 7 May 2009
37. Knudsen LR, Matusiewicz K, Mendel F, Rechberger C, Schlaffer M, Søren S, Gauravaram TP (2008) Grøstl – a SHA-3 Candidate
38. Lucks S, Schneier B, Whiting D, Bellare M, Kohno T, Callas J, Ferguson JWN (2008) The Skein Hash Function Family
39. Sheikh F, Mathew SK, Walker RKJ (2010) A Skein-512 hardware implementation. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/presentations/WALKER_skein-intel-hwd-slides.pdf. Accessed August 2010
40. Wu H (2009) The Hash Function JH. http://www3.ntu.edu.sg/home/wuhj/research/jh/. Accessed July 2011
41. Hanley N, Hamilton M, Lu L, Byrne A, O'Neill M, William P, Baldwin MB (2010) FPGA Implementations of the Round Two SHA-3 Candidates, August 2010
42. Feldhofer M, Kirschbaum M, Plos T, Schmidt J-M, Tillich ASS (2010) Uniform evaluation of hardware implementations of the round-two SHA-3 candidates. In: The Second SHA-3 Candidate Conference

43. Bertoni G, Daemen J, Peeters M, Assche GV (2010) The Keccak sponge function family: hardware performance. http://keccak.noekeon.org/hw_performance.html. Accessed November 2010
44. Henzen L, Aumasson J-P, Meier W, Phan R VLSI Characterization of the Cryptographic Hash Function BLAKE. http://www.131002.net/data/papers/HAMP10.pdf. Accessed July 2011
45. Beuchat J-L, Okamoto E, Yamazaki T (2010) Compact Implementations of BLAKE-32 and BLAKE-64 on FPGA
46. Grøstl – a SHA-3 candidate. http://www.groestl.info/implementations.html. Accessed July 2011
47. Baldwin B, Byrne A, Hamilton M et al. (2009) FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. http://eprint.iacr.org/2009/342.pdf. Accessed July 2011
48. Long M (2009) Implementing Skein Hash Function on Xilinx Virtex-5 FPGA. http://www.schneier.com/skein_fpga.pdf. Accessed February 2009

http://www.springer.com/978-1-4419-8079-3

Introduction to Hardware Security and Trust
Tehranipoor, M.; Wang, C. (Eds.)
2012, VIII, 427 p., Hardcover
ISBN: 978-1-4419-8079-3