Design for Scientific RENAISSANCE

DOUBLE-A – A NEW CRYPTOGRAPHIC HASH FUNCTION (ITS DESIGN)

Abdullah Issa[1, a], Mohammad A. Al-Ahmad[2, a], Abdullah Al-Saleh[3, a]

[1]Computer Science Department. College of Basic Education, Public Authority for Applied Education and Training, Kuwait City, Kuwait

[a]a.issa795@gmail.com, [b]malahmads@yahoo.com, [c]abdullah.n.sy@gmail.com

ISSN: 2231-8852

ABSTRACT

This paper examines the outline decisions of the sponge constructed cryptographic hash function Double-A. Firstly, the relative favorable circumstances of why a stream mode cipher is utilized instead of a block mode are given. Furthermore, a portrayal of what a sponge function is, the way it is designed and what are its fundamental components. At long last, after a brief review of the Salsa20 stream cipher and its structure, the decisions of the states width, rounds and operations in the pseudorandom function *f* are talked about in subtle element to show how and why they are utilized as a part of the stage of Double-A.

Keywords: Sponge function, Salsa20, absorb, permutation, squeeze, Double-A

1. Introduction

Cryptographic hash functions are considered for all intents and purposes difficult to reverse, that is, to reproduce the information from its hash esteem alone. These one-way hash functions have been called "the workhorses of modern cryptography" (Schneier Bruce , 2014). At any rate, it must have the accompanying properties: Pre-image resistance, Second pre-image resistance and Collision resistance. So as an aftereffect of an extensive number of attacks on hash functions, for example, MD5 and SHA-1 of the supposed MD4 family, and general attacks on the regular construction method, there is an expanding requirement for considering elective development strategies and standards for future hash functions. In this paper, another hash function is created which utilizes a permutated sponge structure and the operations of the Salsa20 stream figure. A sponge construction can be seen as a function which takes an arbitrary estimated input and registers an output d of any length required by the user, in any case, in this paper the Double-A-512 of altered summary (n=512 bits) will be talked about and clarified with the goal that it'll be conceivable to demonstrate its configuration decisions and the operations utilized as a part of the change of Double-A. Moreover, the test vectors after its implementation, properties for its state width of b=1600 bits, capacity of 2n (c= 1024 bits) and as b= r+c, the bitrate is (r= 576 bits). The capacity "c" was the initial move towards isolating the hash digest length from the security level of hash functions. It is the measure of data in the state that is shielded from changes when new info is consumed so it is clear that as the capacity diminishes, the vulnerability of the sponge function increments. In a Merkle-Damgard hash work, an inward collision is characterized as a collision in the output and in light of the fact that the capacity is equivalent to its digest (c=n) so Guido Bertoni's analysis demonstrates that for the output length n, most of the current changes will give great imperviousness to sponge functions when the length of the capacity is huge (Guido Bertoni, et al., 2011).

Presently, the entire work of a cryptographic hash function essentially lies in its encryption calculation and two primitives are expected to fabricate a solid encryption calculation (Yaser Jararweh, et al., 2012). They are *confusion* and *diffusion*. As *Claude Shannon*'s expressed in his hypothesis that, confusion is essentially the operation in which the relationship between the message and its digest will be kept obscure and diffusion is the operation of spreading the impact of every message bit with a specific end goal to shroud its measurable property. So to condense it, the confusion operation keeps up the one way property while the diffusion helps in fortifying the collision resistance. The 3 segments utilized here as a part of request to complete these two primitives are: (1) *Permutation*: It is a procedure of swapping the information with one another with the end goal of taking care of the diffusion operation. Contingent upon the calculation itself, the measure of information to be swapped is resolved. Information could be swapped by swapping bits at littler scales, and might likewise swap numerous words at bigger scales. (2) *Logical functions*: This procedure is performed by utilizing logical gates. These incorporate AND, XOR, OR and NOT for the purpose of confusion. The most used logical gates as a part of cryptography papers is the XOR since its principle function is adjusting, as it's additionally outlandish for an attacker to figure the information to a XOR with observing just at its output. (3) *Modular arithmetic function*: This procedure is likely used for diffusion through the spread of convey and era. The generally utilized arithmetic operations are the Modular Multiplication and Addition. In spite of the fact that just the measured addition operation is used as a part of the operations here.

Double-A used in Salsa20’s encryption algorithm is a newly constructed sponge function. Table 1, shows all the sponge constructed hash functions (using a transformation or a permutation) and whether constructed with the help of previous block/stream ciphers.

Table 1: Sponge functions

Sponge Function	Year	Structure	Symmetric Encryption Cipher Used	Ciphers that Helped
GLUON (Thierry P. Berger, et al., 2011)	2012	T-Sponge	Stream	X-FCSR-v2 and F-FCSR-H-v3
PHOTON(Jian Guo, et al., 2011)	2011	P-Sponge	Block	AES , PRESENT, LED
QUARK (Jean-Philippe Aumasson, et al., 2012)	2010	P-Sponge	Block and Stream	KATAN / Grain
SipHash (Jean-Philippe Aumasson and Daniel J. Bernstein, 2012)	2012	JH-style T-Sponge	-	BLAKE and Skein
SPN-Hash (Choy, J., et al., 2012)	2012	JH-style P-Sponge	Block	AES , LED and PHOTON
SPONGENT (Bogdanov, A., et al., 2011)	2011	P-Sponge	Block	PRESENT
Spritz (Ronald L. Rivest, et al., 2014)	2014	Sponge	Stream	RC4
Keccak (G. Bertoni, et al., 2011)	2008	P-Sponge	Block	Noekeon and Rijndael
LHash (Wenling Wu, et al., 2013)	2013	Feistel-PG	-	Extended sponge function

Block Ciphers VS. Stream Ciphers

Typically a cipher takes a plain-text as input and produces a ciphertext as output. In cryptography, stream ciphers and block ciphers are two encryption/decryption algorithms that belong to the family of symmetric key ciphers, though there are some key differences.

Block ciphers encrypt fixed length blocks of bits, while stream ciphers combine plain-text bits with a pseudorandom cipher bits stream. Block ciphers use the same transformation, while stream ciphers use varying transformations based on the state of the engine. Stream ciphers usually execute faster than block ciphers. In terms of hardware complexity, stream ciphers are relatively less complex. They are the typical preference over block ciphers when the plain-text is available in varying quantities, because block ciphers cannot operate directly on blocks shorter than the block size. The disadvantage of block ciphers is that the key words consume valuable communication resources. It means that a 64-byte block cipher with a 32-byte key needs to repeatedly sweep through 96 bytes of memory for its 64 bytes of output; on the other hand, stream ciphers repeatedly sweeps through just 64 bytes of for its 64 bytes of output. (A. Biryukov and D. Wagner, 1999; Audia S.Abd Al-Rasedy and Ameer A.J Al-Swidi, 2010) From the differences, both look secure enough but it's more preferable to choose the stream encryption algorithms because they're relatively less complex to implement in terms of hardware.

2. Design Choices

2.1 Sponge function

The sponge function is a simple iterated construction for building a function F with variable length input and arbitrary length output based on a fixed-length permutation *f* operating on a fixed number b bits. Here b is called the state width. It operates on a state of b= r + c bits. Firstly, the input string (message) will be padded with a specific rule and divided into equal sized blocks of bitrate bits. After that, the Initial state is initialized to zero. The sponge function then processes the message into two phases as elaborated in Figure 1. The Absorbing phase (the first phase), the bitrate message blocks are XOR'ed into the state, interleaved with other applications of the internal permutation. After all the message blocks have been processed, the sponge function will move to the second phase (Squeezing phase). Here, the first r bits of the state are returned as part of the output, interleaved with applications of the internal permutation. The squeezing phase is completed after the desired length of the output digest has been produced (Guido Bertoni, et al., 2011). In case the output length is not a multiple of the bitrate bits, it will therefore be truncated (shortened by cutting it off at either the leftmost or rightmost bits of the digest, depending on its size). One of the goals of using this sponge hash function is having security against generic attacks and to make the use of the permutation more simple, flexible, and functional. A Sponge function is always built from three components:

1- The Initial state S (which is initially 0), containing b= r + c bits.
2- A function f of fixed length that permutes the state memory
3- A padding function P which appends enough bits to the message so that the length of the padded input is a whole multiple of the bitrate r.

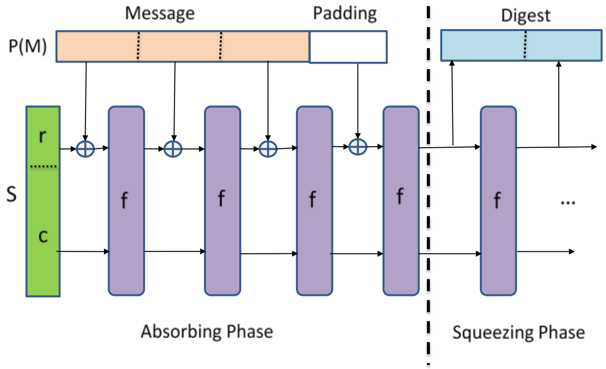


Figure 1. Sponge construction

2.2 Double-A

The proposed output length (digest) is 512 bits. A state of 1600 bits is chosen because a large internal state can prove its security against internal collisions, which are: “variable inputs that are lead to a similar internal state and eventually to a similar digest”. In order to determine the security claim of this hash function with respect to an ideal sponge function (random sponge), the value of the capacity used should be high enough to offer a good amount of resistance against collisions. In a sponge function, it is limited by the resistance against inner collisions, in which the expected complexity is of the order $2^{c/2}$ and “capacity” should be large so that when generating inner collisions, it will not be possible to even become feasible in the time that the hash function is used. Remember that for the sponge construction there are no generic attacks with the expected amount of workload of order below $2^{c/2}$ (Guido Bertoni, et al., 2010; Biham and A. Shamir, 1991), therefore one is able to conclude that a lower bound for the expected complexity for generating a *collision* is min ($2^{n/2}$, $2^{c/2}$) and for generating a *second pre-image* is min (2^{n}, $2^{c/2}$). Hence, if c>2n, randomization increases the strength against signature forgery due to generic attacks against the hash function from $2^{n/2}$ to 2^{n}. If the capacity is between n and 2n, the increase is from $2^{n/2}$ to $2^{c/2}$. If c<n, randomized hashing does not significantly increase the security level.

So choosing c = 2n for DOUBLE-A is particularly needed by the requirement that (second) pre-image resistance should be at least 2n. Therefore, a capacity of 1024 bits has been chosen for the choice that the expected workload of an attack should be 2^{512} calls to the underlying permutation. Note that requiring a resistance of 2^{512} is quite strong. Note also that padded message is never XOR’ed into the capacity portion of the initial state nor are any bits of the capacity are ever directly output. In hash functions, resistance to collisions or pre-image attacks depends upon c. It will offer the same resistance as would a random oracle truncated to the hash function’s output length against collisions and pre-image attacks. Thus, r, the number of message bits processed per block permutation, depends upon the output hash size which will be r = b – 2n(c) = 576 bits.

A. State Memory (Initial State) [S]

Also called the “root state”, the b bits (r+c) are initialized to zero. The root state has a fixed value and shall never be considered as an input. It’s distributed to a 5x5 matrix of bytes. Each block represents 8 bytes.

B. Padding Rule

The padding rule is needed because hash functions are defined to work on an arbitrary integer number of blocks, in this case, 576 bits. The minimal quantity of data that can be processed by a hash function is a single block. So, if the message size is not an integer multiple of the block size r, one has to pad it to the right size. This rule could be applied to messages of variable size. The padded input thus will be broken into r-bit blocks. This almost prevents a hash function from being vulnerable or open to attacks, such as length extension.

A simple padding scheme is a single digit ('1') bit is added at the end of the message and then as many ('0') bits as required are added after it. The ('0') bits added depends on the block to which the message block needs to be completed. In bit terms this is "100 --- 000". To harden the collision resistance property even further, the length of the message is added in an extra block and this is called “length padding”. One also needs to make the “message + padding” have one and one only interpretation, otherwise it would be simple to create collisions. Therefore adding the length padding gives a unique way to interpret the message + padding couple. The padded messages equation is:

$$P(M) = M||P + 1 + 0^* + m_x \quad (1)$$

Where M is the message, P is the pre-determined bit string, “0* = 0000…” and m_x is the length of the message.

C. Function f

This round function that uses the ARX (Addition – Rotation – XOR) operations was inspired from the stream cipher Salsa20. A brief review of Salsa20 and its encryption function is discussed and an explanation on why it was chosen for the permutation.

2.3 SALSA 20

The cryptographic stream cipher Salsa20 was designed by *Daniel J. Bernstein* and introduced in March 2005 to show that one is able to use a strong hash function algorithm to encrypt data (Daniel J. Bernstein, 2005). He generally went for a long simple operation, rather than a shorter complexes one because they are able to circuits, so therefore can reach the security level as other operations.

The Salsa20 encryption function is a long chain of three simple operations on 32-bit words:

- Addition, is the sum of [$y \boxplus z \bmod 2^{32}$] of two 32-bit words y, z;
- Exclusive-or, is the XORing [$y \oplus z$] of two *32-bit words y, z; and*
- Rotation, is the rotating [y<<k] of a 32-bit word y by k bits to the left, where k is constant.

The symbols used, $\boxplus$ is addition modulo 2^{32}, <<< is the left-rotate operation and $\oplus$ is XOR. Salsa20 organizes the words as follows. First, the 64 bytes, 16 bytes of constants the 8-byte nonce (unique key), 32-byte key and the 8-byte block counter (initially 0). Its function is called a double-round function, and it consists of a column-round function followed by a row-round function. The double-round of Salsa20 is repeated 10 times. At the end, it sums up the 16 resulting words to the 16 original words in order to produce the 16 word (512 bits) output Z (Tsukasa Ishiguro, 2015):

$$Z = K + K^{20} \qquad (2)$$

This function acts on the 4 × 4 matrix of 32-bit words written as shown in Figure 2.

$$K = \left\{ \begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix} \right\}$$

=

Constant	Key	Key	Key
Key	Constant	Nonce	Nonce
Counter	Counter	Constant	Key
Key	Key	Key	Constant

Figure 2. Salsa20's matrix distribution

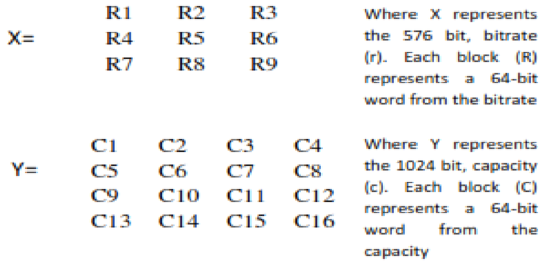
Operations of Salsa 20

$K[4] \oplus = (k[0] + k[12]) \lll 7; \quad K[9] \oplus= (k[5] + k[1]) \lll 7;$
$K[14] \oplus= (k[10] + k[6]) \lll 7; \quad K[3] \oplus= (k[15] + k[11]) \lll 7;$
$K[8] \oplus= (k[4] + k[0]) \ll 9; \quad K[13] \oplus= (k[9] + k[5]) \ll 9;$
$K[2] \oplus= (k[14] + k[10]) \ll 9; \quad K[7] \oplus= (k[3] + k[15]) \ll 9;$
$K[12] \oplus= (k[8] + k[4]) \ll 13; K[1] \oplus= (k[13] + k[9]) \ll 13;$
$K[6] \oplus= (k[2] + k[14]) \ll 13; \quad K[11] \oplus= (k[7] + k[3]) \ll 13;$
$K[0] \oplus= (k[12] + k[8]) \ll 18; K[5] \oplus= (k[1] + k[13]) \ll 18;$
$K[10] \oplus= (k[6] + k[2]) \ll 18; K[15] \oplus= (k[11] + k[7]) \ll 18;$
$K[1] \oplus= (k[0] + k[3]) \ll 7; \quad K[6] \oplus= (k[5] + k[4]) \ll 7;$
$K[11] \oplus= (k[10] + k[9]) \ll 7; \quad K[12] \oplus= (k[15] + k[14]) \ll 7;$
$K[2] \oplus= (k[1] + k[0]) \ll 9; \quad K[7] \oplus= (k[6] + k[5]) \ll 9;$
$K[8] \oplus= (k[11] + k[10]) \ll 9; K[13] \oplus= (k[12] + k[15]) \ll 9;$
$K[3] \oplus= (k[2] + k[1]) \ll 13; \quad K[4] \oplus= (k[7] + k[6]) \ll 13;$
$K[9] \oplus= (k[8] + k[11]) \ll 13; K[14] \oplus= (k[13] + k[12]) \ll 13$
$K[0] \oplus= (k[3] + k[2]) \ll 18; \quad K[5] \oplus= (k[4] + k[7]) \ll 18;$
$K[10] \oplus= (k[9] + k[8]) \ll 18; K[15] \oplus= (k[14] + k[13]) \ll 18;$

The following operations represent a double-round function of Salsa20 on matrix K, which is a column-round function followed by a row-round function. They are modified by "XORing a rotated sum" starting with the words under the diagonal (constant) in the column-round and on the right of it in the row-round. $[k \oplus= y]$ is an abbreviation for $[k = k \oplus y]$. This function consists of 2 rounds, so it will be repeated 10 times and then the output $Z = K + K^{10}$; where 10 is the number of double-rounds (G. Bertoni, et al., 2008).

2.3.1 Modifying Salsa20's Round Function to Fit the State

Double-A's permutation applies the same double-round function but using the ARX algorithm on 64-bit words instead of 32-bit words. As it's a sponge function, there will not be a key neither the nonce nor a counter. The bitrate and capacity will take their place in the matrix instead. However, remember the capacity will enter the permutation but will not be affected in any way by the operations so they will only be performed on the bitrate r. And instead of 16 (32-bit words) in a 4x4 matrix which is equal to 512 bits state, two matrixes are introduced. One for the capacity (1024 bits) of 16 (64-bit words) in a 4x4 matrix and the other for the bitrate (576 bits) of 9 (64-bit words) in a 3x3 matrix, both equal to the bit state of b=1600 bits. The states matrix will look like this:



$$X = \begin{array}{ccc} R1 & R2 & R3 \\ R4 & R5 & R6 \\ R7 & R8 & R9 \end{array}$$

Where X represents the 576 bit, bitrate (r). Each block (R) represents a 64-bit word from the bitrate

$$Y = \begin{array}{cccc} C1 & C2 & C3 & C4 \\ C5 & C6 & C7 & C8 \\ C9 & C10 & C11 & C12 \\ C13 & C14 & C15 & C16 \end{array}$$

Where Y represents the 1024 bit, capacity (c). Each block (C) represents a 64-bit word from the capacity

2.3.3 Operations, Structure and Rounds

This section will explain the operations (Addition – Rotations – XOR), the block size, the state's matrix and the number of rounds in our model. Their arguments and counterarguments are mainly discussed (Y.Nir, A. Langley, 2015).

A. Addition

It is discussed in the introduction of the paper that why arithmetic functions are important for the diffusion and the most two popular operations are the addition and multiplication. An advantage in using the modular addition operation is that they produce less complicated output bits which are functions to the input bits so the mixing is done in a few simple operations. Therefore, simple integer series operations are always fast while the modular multiplication is not consistently fast and what really matters is not integers multiplication speed, but rather the constant-time integer multiplications speed, which is mostly much slower.

B. Rotations

Rotations are about one third of the operations used in the encryption function. The basic argument why they chose rotations rather than shifts is that comparing one XOR operator of rotated quantities gives the same amount of diffusion as two XOR operators of shifted quantities and while, on the other computers, rotation operation saves time. The chosen rotation distances [7, 9, 13, and 18] in Salsa20 are doing a fine of distributing many low-weight changes across the bit positions in a few numbers of rounds. But as the matrix is 3x3, only the first 3 rotational distances [7, 9, and 13] are used. The specific choice of distance does not really matter though as there are some software that only accepts the same sequence of distances.

C. XOR

You might criticize that encryption by the logical function XOR is very simple but it is usually a one-time pad that it achieves the amount of secrecy and the level of integrity, so using XOR is perfectly fine.

D. Block Size

So instead of the 512-bit 4x4 array (using 32-bit words), a larger block size of 576-bit 3x3 array (using 64-bit words) is used. It is easily produced because of similar structure. A larger block sized state is used here which seems to provide almost the same amount of mixing as the first Salsa20's cipher rounds. Therefore, this helps in saving time. The counter argument is that a large block size may also lose time too. Computers are made for computations that do not have too much data.

E. State's matrix

The 9 word bitrate (576-bits) as a 3x3 array is viewed. In the first round, each column is modifying their words by performing 9 serial operations (ARX-ARX-ARX) and during the second round each row is performing the same number of serial operations to modify their words. So in 2 rounds, 576 bits are modified using only 18 serial operations, while Salsa20 modifies 512-bits using 24 serial operations in the same number of rounds. Therefore, this allows much faster diffusion of changes among the words in less time.

F. Rounds

The same number of 20 rounds as Salsa20 is chosen. It can be imagined that using a few rounds for increasing the speed. But this type of variability will complicate hardware implementations and will seem to tempt users to reduce the number of rounds as much as possible although the hash function relies more importantly on what is inside those rounds. It is always possible to get a higher speed without lowering the confidence but the basic counter argument here is that these numbers of rounds will do their job, which is diffusing the input as many times it can.

2.4 (Double-A) permutation

After recalling the Salsa20 stream cipher, explaining the operations that are used and how some changes were modified to fit the permutation of Double-A, an example for what happens inside f is given. Assume the X matrix (bitrate matrix) has the following values:

R1	R2	R3
R4	R5	R6
R7	R8	R9

→

x6	x7	x8
x11	x12	x13
x16	x17	x18

And the encryption function of three operations on 64-bit words is:

- 64-bit addition, is the sum $[y \boxplus z \bmod 2^{64}]$ of two 64-bit words y, z;
- 64-bit exclusive-or, producing the XOR $[y \oplus z]$ of two 64-bit words y, z; and
- 64-bit rotation, producing the rotation $[y \lll k]$ of a 64-bit word y by b= [7, 9, 13] bits to the left, where k is constant.

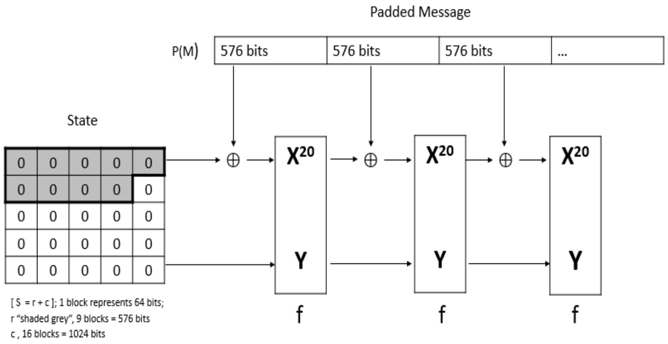
The operations for a double-round permutation (Column-round followed by a row-round) will be:

$x11 = x11 \oplus ((x6 + x16) \ll 7)$ $x17 = x17 \oplus ((x12 + x7) \ll 7)$
$x8 = x8 \oplus ((x18 + x13) \ll 7)$ $x16 = x16 \oplus ((x11 + x6) \ll 9)$ $x7 = x7 \oplus ((x17 + x12) \ll 9)$
$x13 = x13 \oplus ((x8 + x18) \ll 9)$
$x6 = x6 \oplus ((x16 + x11) \ll 13)$ $x12 = x12 \oplus ((x7 + x17) \ll 13)$
$x18 = x18 \oplus ((x13 + x8) \ll 13)$ $x7 = x7 \oplus ((x6 + x8) \ll 7)$
$x13 = x13 \oplus ((x12 + x11) \ll 7)$ $x16 = x16 \oplus ((x18 + x17) \ll 7)$
$x8 = x8 \oplus ((x7 + x6) \ll 9)$ $x11 = x11 \oplus ((x13+ x12) \ll 9)$
$x17 = x17 \oplus ((x16 + x18) \ll 9)$ $x6 = x6 \oplus ((x8 + x7) \ll 13)$
$x12 = x12 \oplus ((x11+x13) \ll 13)$ $x18 = x18 \oplus ((x17+x16) \ll 13)$

These operations will be repeated 10 times in the permutation f for the 3x3 X (bitrate) matrix while the 4x4 Y (capacity) matrixes will not be affected.

A. Absorbing Phase

In this phase, the r-bit message are XORed into the first r bits of the state, interleaved with the permutation of the function *f* as demonstrated in Figure 3.



[S = r + c]; 1 block represents 64 bits;
r "shaded grey", 9 blocks = 576 bits
c , 16 blocks = 1024 bits

Figure 3. Absorbing phase

B. Squeezing Phase

In this phase, the first bitrate r bits of the state are returned as the digest blocks, interleaved with the permutation of the function f. Although after the final block permutation, the leading n bits of the state are the desired hash and because r is greater than n, there is actually never a need for additional block permutations in the squeezing phase. But as r = 576 bits so the first r bits are returned to the output block and truncated to be 512 bits which is the digest as illustrated in Figure 4.

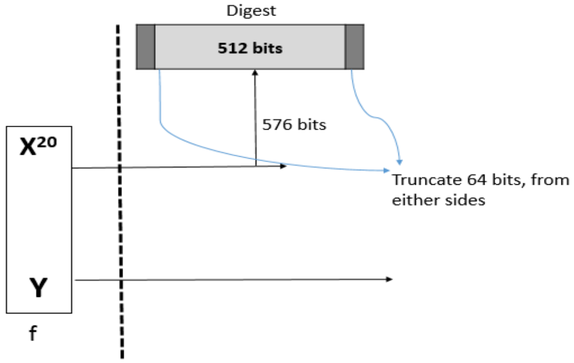


Figure 4. Squeezing Phase

3. Pseudo-code Description and Test Vectors

Double-A-f (X)

```
{Double-Round (X)
        For I = 0 to 9 do
        Column-Round
                X ← Addition
                X ← Constant rotation
                X ← XOR the rotated sum
        Row-Round
                X ← Addition
                X ← Constant rotation
                X ← XOR the rotated sum
  Return X}
                                          Double-A[r, c] (M)      *m_x = Length
                                                                   of the message
{// Initialization and Padding//                                   (M)
        S [r, c] = 0,
        P(M) = M || 0x01 || 0x00 || … || 0xm_x

//Absorbing Phase//                                       *P_i = padded
        Forall block P_i in P(M)                          message P
                S(r)←S(r) XOR P_i
                S(r)= Double-Round (S(r))                 distributed into
                                                          blocks of r-bits*
//Squeezing Phase//
                Z = Z || S(r)
                Output truncated (Z)                      *Z = Digest*
Return Z}
```

Table 2. Test vectors for DOUBLE-A-512

DOUBLE-A (“ ”)
DB8ADF56E71612BC2BF88FA71AD71300B10A1704232D0CD12647F5D55F AA08A01E6527E6BA749B16DB8ADF56E71612BC4B41ECED86930A12FC4 CF1820BD53266
DOUBLE-A ("The five boxing wizards jump quickly.")
96DA45779F8CBA4B0D5147A0610AA6814F4731F5929AA0163B6017EEB1B AAD77FEACD777A24B1F2D796B15965DC5216B0D5147A0610AA68DD688 9BA8BD8319AA
DOUBLE-A ("The five boxing wizards jump quickly")
F226D22B6918B3B73FC37A7627D60295C3E0F5A42E4046005EFC7F49675B 80613E0F3345C8EB5B47C8C4D7BCBE10EF8D3FC37A7627D60295F681FA2 12A2738A0

4. Conclusion

This paper analyzed and examined the outline decisions of the sponge constructed cryptographic hash function Double-A. From the analysis it can conclude that stream mode cipher is utilized instead of a block mode. After a brief review of the Salsa20 stream cipher and its structure, the decisions of the states width, rounds and operations in the pseudorandom function *f* are talked about in subtle element to show how and why they are utilized as a part of the stage of Double-A.

REFERENCES

Schneier, Bruce. "Cryptanalysis of MD5 and SHA: Time for a New Standard". Computerworld, 2014.

Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions. 2011

Yaser Jararweh ,Hala Tawalbeh , Lo'ai Tawalbeh and Abidalrahman Moh'd. FPGA Performance Evaluation of SHA-3 Candidate Algorithm, 2012

Thierry P. Berger, Joffrey D'Hayer, Kevin Marquet, Marine Minier. The GLUON family: a lightweight Hash function family based on FCSRs, 2011

Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON Family of Lightweight Hash Functions, 2011

Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Maria Naya-Plasencia. Quark: a lightweight hash, 2012

Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: a fast short-input PRF, 2012

Choy, J., Yap, H., Khoo, K., Guo, J., Peyrin, T., Poschmann, A., & Tan, C. H. (2012). SPN-hash: improving the provable resistance against differential collision attacks. In Progress in Cryptology-AFRICACRYPT 2012 (pp. 270-286). Springer Berlin Heidelberg.

Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varıcı, K., & Verbauwhede, I. (2011). SPONGENT: A lightweight hash function. In Cryptographic Hardware and Embedded Systems–CHES 2011 (pp. 312-325). Springer Berlin Heidelberg.pdf at kuleuven.be

Ronald L. Rivest, Jacob C. N. Schuldt. Spritz- a spongy RC4-like stream cipher and hash function, 2014

G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. The Keccak reference, round 3 submission *to NIST SHA-3*, 2011

Wenling Wu, Shuang Wu, Lei Zhang, Jian Zou, and Le Dong. LHash: A Lightweight Hash Function, 2013

Audia S.Abd Al-Rasedy and Ameer A.J Al-Swidi. An advantages and Disadvantages of Block and Stream Cipher, 2010

Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011.

Guido Bertoni, Joan Daemen, Michael Peeters and Gilles Van Assche. Keccak sponge function family main document, 2010

Daniel J. Bernstein. Salsa20 design, 2005

Tsukasa Ishiguro. Modified version of “Latin Dances Revisited: New Analytic Results of Salsa20 and ChaCha”, 2015

Daniel J. Bernstein. Salsa20 security, 2005

Y.Nir, A. Langley, Internet Research Task Force (IRTF) from Google, Inc. ChaCha20 and Poly1305 for IETF Protocol, 2015

G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the indifferentiability of the Sponge construction. In N. P. Smart, editor, EURO- CRYPT, volume 4965 of Lecture Notes in Computer Science, pages 181– 197. Springer, 2008.

Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. J. Cryptology, 4(1):3–72, 1991.

A. Biryukov and D. Wagner. Slide attacks. In L. R. Knudsen, editor, FSE, volume 1636 of Lecture Notes in Computer Science, pages 245–259. Springer, 1999.