Automobile Security: Using Lightweight Cryptographic Hash Functions

Authors Name/s per 1st Affiliation (*Author*)
line 1 (of *Affiliation*): dept. name of organization
line 2-name of organization, acronyms acceptable
line 3-City, Country
line 4-e-mail address if desired

Authors Name/s per 2nd Affiliation (*Author*)
line 1 (of *Affiliation*): dept. name of organization
line 2-name of organization, acronyms acceptable
line 3-City, Country
line 4-e-mail address if desired

***Abstract*—A new branch in cryptography, the lightweight cryptography is faster, lighter and energy efficient as compared to the classical cryptography and has been designed to enhance the security level in pervasive computing applications. It has been designed to reduce the power consumption, key size, area, cycle rate and throughput rate in smart but resource constrained devices. Nowadays, everything around is getting automated to ease human lives. But besides this increasing use of technology, the threat to the security of the concerns are also getting increased. Likewise, automated automobile systems are also vulnerable to threat attacks as conventional cryptographic techniques cannot provide security to such real-time systems. In this paper, a general study of some lightweight cryptographic hash functions has been done to provide security to the automobile systems.**

Keywords—Lightweight cryptography, Automobile Security, Lightweight Hash Functions, SHA-1, ShA-2, SHA-3, Photon, Lesamnta-LW, SipHash, Spongent, JH Hash.

I. Introduction

Lightweight Cryptography is an inclusive term to cope with the growing demand of security features in pervasive computing. To implement the conventional cryptography in such devices is unsuitable because of the mathematical complexity associated with cryptographic primitives. The two important primitives of lightweight symmetric cryptography are lightweight block cipher and lightweight hash algorithm. Hash function takes messages of random input sizes and produces output messages with a fixed size. A cryptographic hash function H with an n-bit output is expected to possess collision resistance, preimage resistance and second preimage resistance as the main security properties. In the employment of lightweight hash functions, higher security levels mean an increased level of collision resistance as well as preimage and second preimage resistance in comparison with the common SHA standard. Out of various hash construction techniques like Merkle-Damgard Construction, Wide Pipe Construction, HAIFA Construction, and Sponge Construction, majority of lightweight hash functions are designed and implemented by Sponge Construction Method (SPGM). The only major issue in lightweight hash comparison is the nature of their designs, which is very sensitive to small variations in comparison metrics.

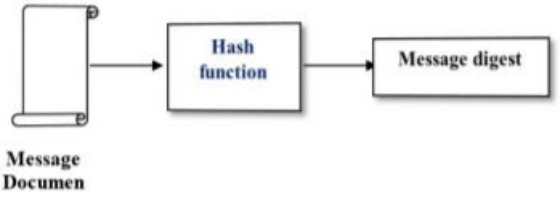

Fig. 1. Hash Function [1]

II. Different Lightweight Cryptographic Hash Functions

A. SHA-1

Secure Hash Algorithm 1, abbreviated as SHA-1, is designed by the United States National Security Agency and is a U.S. Federal Information Processing Standard. It is a cryptographic hash function which allows an input of any size and produces a fixed size output to it, known as the hash value or the message digest of 160-bit or 20-bytes, which when rendered as a hexadecimal number is 40 digits long. The message digest produced by SHA-1 is similar to the principles used in the design of message digest algorithm for MD2, MD4 and MD5, by Ronald L. Rivest of MIT, but this generates a larger hash value.SHA-1 is not considered secured against well-funded opponents since 2005, and many organizations have recommended it's replacement by SHA-2 or SHA-3 later. In February 2017, CWI Amsterdam and Google

announced that they performed a collision attack against SHA-1, publishing two dissimilar PDF files, but they had produced the same hash. The attacks possible in SHA-1 are as follows

- The SHAppening
- SHAttered - first public collision
- Birthday-near-collision Attack - first practical chosen-prefix attack.

B. SHA-2

SHA-2 or Secure Hash Algorithm 2 is a cryptographic hash function that has been designed by the United States National Security Agency (NSA). SHA-2 is designed on the Merkle-Damgard structure, from a one-way compression function itself built using the Davies-Meyer structure from specialized and classified block cipher [7]. It includes notable changes as compared to SHA-1. SHA-2 includes a total of six hash functions, with hash values as 224, 256, 384 or 512 bits. SHA-256 and SHA-512 are the hash functions which are computed with 32-bit and 64-bit words, respectively. At present, the best public attacks break preimage resistance for 52 out of 64 rounds of SHA-256 or 57 out of 80 rounds of SHA-512, and collision resistance for 46 out of 64 rounds of SHA-256. Possible attacks are:

- Collision Attack
- Pre-image Attack
- Birthday Attack
- Brute-force Attack
- Rainbow Table
- Side-channel Attack
- Length Extension Attack

C. SHA-3

SHA-3 was released by NIST on August 5, 2015 and is the newest member in the family of Secure Hash Algorithm standards. SHA-3 does not have MD-5 like structure as in SHA-1 and SHA-2. SHA-3 is a part of the larger cryptographic primitive family Keccak. SHA-3 can be directly substituted for SHA-2 in current applications if required, and is used to significantly improve the robustness of NIST's overall hash algorithm toolkit. Possible Attacks in SHA-3 is less as compared to other SHA functions, and it usually suffers from quantum attacks.

D. Photon

Photon is a lightweight hash function designed by Guo, Peyrin, and Poschmann, and is widely used in RFID applications. Being the most compact hash function, in comparison to other lightweight hash functions, Photon is reaching areas near to the theoretical optimum. It is also known for its speed among its competitors. Its output size is $64 \leq n \leq 256$, and the input and output bit rates are r and r`, respectively. Photon is a P-Sponge and is based on an AES-like permutation. The smallest security parameter (PHOTON-80/20/16), has the bit rate during absorption phase as 20 compared and during the squeezing phase as 16.

The sponge-like domain extension algorithm helps in trusting the security of the Photon hash functions as long as the internal permutation does not present any structural flaw of any kind. The permutation includes 12 iterations performed over a square of nibbles of 4-bits. Attacks possible on Photon are as follows [4]:

- Differential/Linear Cryptanalysis
- Rebound and Super-box attacks
- Cube testers and Algebraic Attacks

E. Spongent

The lightweight hash function named Spongent has been designed by Bogdanov. It is based on the sponge construction and is detected with Present-type permutation. Hence, it can be said that Spongent is a P-Sponge in which the permutation is a modified version of the block cipher present. Spongent has a total of 13 variants for each collision/(second) preimage resistance level and implementation constraint.

The initial value is b-bit 0 in Spongent and the hash size n in all its variants is equal to either capacity c or 2c. The number of rounds of the present-like permutation is 45 for Spongent-88 and is 140 for Spongent-256. There is no attack on Spongent other than the linear distinguishers for reduced-round versions. Attacks possible are as follows [26]:

- Differential Cryptanalysis
- Collision Attack – Rebound Attack
- Preimage resistance – Pre-computation and Man-in-the-middle
- Linear attacks

F. Lesamnta-LW

Merkle-Damgård designed the lightweight hash function Lesamnta-LW. It is a 256-bit hash function. Lesamnta-LW mainly uses the methods used in the AES-based block cipher. It's main focus is on a compact and fast hashing for lightweight application with a total of 21202120 security level. The primary target lies for 8-bit CPUs for short message hashing. Lesamnta-LW's design is based on the Merkle-Damgård as domain extension and employs AES as the compression function. Lesamnta-LW hardware weighs 8.24 KG on 90 nm technology. It offers 50 bytes of RAM and deals with short messages on 8-bit CPUs. Possible attacks on Lesamnta-LW are as follows [15]:

- Differential and Linear Attacks

- Higher order differential and Interpolation Attack
- Impossible Differential Attack
- Related-Key Attacks
- Collision Attacks using Message Modification
- Attacks on the Lesamnta-LW Compression Function Using Self-Duality

G. SipHash-2-4

SipHash belongs from the family of pseudorandom functions improved for short inputs. Marked applications are network traffic authentication and hash-table lookups which are safeguarded against hash-flooding denial-of-service attacks. SipHash is much simpler as compared to MACs based on universal hashing. It is also faster on short inputs. SipHash has well-defined security goals and competitive performance when compared with dedicated designs for hash-table lookup. Like, when it processes a 16-byte input with an unused key in 140 cycles on an AMD FX-8150 processor, the result is much faster than the MACs. Possible Attacks on this hash function are as follows [17]:

- Rotational Attack
- Cube Attack
- Rebound Attack
- XOR-linearized Characteristics
- Truncated Differentials
- Internal Collisions
- Key Recovery
- State Recovery

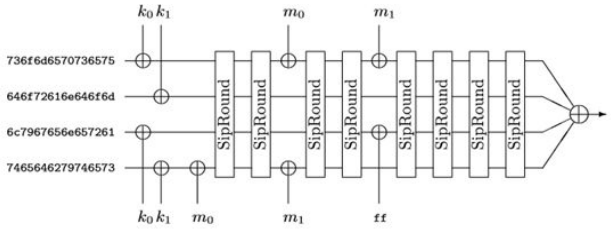

Fig. 2. SipHash-2-4 Processing a 15-byte message [17]

H. JH-Hash

JH-Hash hash function has been designed by Hongjun Wu et al. It consists of four versions namely JH-224, JH-256, JH-384, and JH-512, based on the same compression function which is the reason why it is easier to combine them in one hardware implementation. In software using the SSE2 instruction set, the bit-slice implementation of JH hash function is quite systematic. The standard variant of JH-Hash has an internal state H which consists of 256 4-bit elements. Its hardware implementation is simple and efficient due to the simple S-boxes and linear transformation. Possible attacks are as follows [20]:

- Collision Attacks
- Second-preimage and Preimage Differential Attack
- Truncated Differential Collision and Second-Preimage Attack
- Algebraic Attack
- Rebound Attack

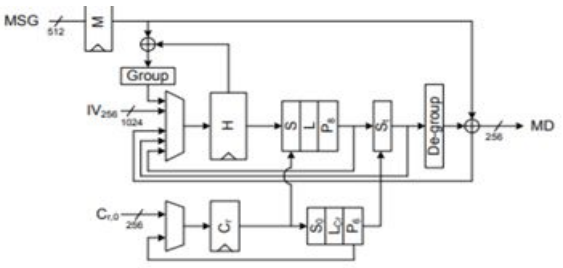

Fig. 3. Implementation of JH-256

III. Areas of Comparison Between Different Lightweight Cryptographic Hash Functions

Comparison of different lightweight hash function is quite challenging as besides being dependent on technology completely, it also demands consideration of different characteristics and metrics which are as follows:

- Area: Measured in GE.
- Cycles: The number of clock cycles used to compute and read out the ciphertexts.
- Time: The ratio of the number of cycles to the operating frequency in seconds.
- Throughput: The rate at which new output is produced with respect to time.
- Power: The estimated power consumption on the gate level by using the Power Compiler.
- Efficiency: Hardware efficiency is measured by dividing the throughput to area ratio.

IV. Methodology of working on an ARM based AURIX processor

The lightweight hash functions were at first implemented on a LINUX system, to understand their working mechanism and also to check with the outputs they generate for different blocks of inputs. After that the hash functions were run on the HighTec IDE, which is a C compiler for TriCore, PowerPC and ARM. Codes were modified according to the platform. After the build was complete for all the hash functions, and

the codes were running error free, a connection was established between the PC and the 2nd generation AURIX TC3xx starter kit, a 32-bit high performance real-time microcontroller with powerful on-chip peripherals. Then the codes for the hash functions were burnt on the hardware, and were debug subsequently, on the htc-workspace. Errors which occurred were looked after and were removed, for successful execution of the codes. “LEDON(0);” and “LEDOFF(0);” tags were included in the code, keeping the main execution part in between to measure the exact run time for the codes. After the codes executed successfully, the PC and the board were also connected to an Oscilloscope, so as to measure the runtime for the hash functions. Codes were burnt and run on the board, and the time taken for it to run successfully was noted from the oscilloscope. Then, to verify if the run time recorded from the oscilloscope is correct or not, the Assembly Codes for the hash functions were studied from the Mixed Mode feature in the workspace. After this the timing for the hash functions were analysed by making a thorough study of the assembly codes, and counting the time taken by looking into the number of lines executed for assembly codes, for every corresponding lines of the code in C language for the hash functions, and an estimate of total time was calculated.

V. Assembly code analysis

1. SHA-1

SHA1_init():19
SHA1_update():1000*(28+(6+88+73+75+73+70+56+79+61+71+57+81+79+75+78+78+70+65+70+67+56+20)+13-63(6+88+73+75+73+70+56+79+61+71+57+81+79+75+78+78+70+65+70+67+56+20))
SHA1_final():3+4+12*8+7+33+3*(28+(6+88+73+75+73+70+56+79+61+71+57+81+79+75+78+78+70+65+70+67+56+20))

Total:19+1000*(28+(6+88+73+75+73+70+56+79+61+71+57+81+79+75+78+78+70+65+70+67+56+20)+13-63(6+88+73+75+73+70+56+79+61+71+57+81+79+75+78+78+70+65+70+67+56+20))+3+4+12*8+7+33+3*(28+(6+88+73+75+73+70+56+79+61+71+57+81+79+75+78+78+70+65+70+67+56+20))

2. SHA-3

Calc_sha_256() - 15
    Init_buf_state() - 3
    Calc_chunk() - 52+6*7+10
Calc_sha_256() - 8*8+4*(4+16*(119)+8*8+8*14)

Total: 15+3+52+6*7+10+8*8+4*(4+16*119+8*8+8*14)

3.SHA-256

Sha3_hashbuffer() - 10
    Sha3_init() - 15
Sha3_hashbuffer() - 5
    Sha3_setflags() - 18
Sha3_hashbuffer() - 4
    Sha3Update() - 143
Sha3_hashbuffer() - 2
    Sha3finalize() - 54
Sha3_hashbuffer() - 6

Total : 10+15+5+18+4+143+2+54+6

4. PHOTON

TestVector(): 14
    hash(): 6
        init(): 6+5*1+5*9+3+7
            Wordxorbyte():29
        CompressionFunction():4
            Wordxorbyte():29
            Permutation(): 9
                AddKey(): 5*14+1
                PrintState(): 1
                SubCell(): 5*(5*15))
                PrintState():1
                ShiftRow():(4+5*(4+5*3+5+5*15))
                PrintState():1
                MixColumn():(9+5*4+5*14)
                    FieldMult():2+4*11
                MixColumn():17
                SCShRMCS():0
                PrintState():1
    hash(): 2+(ceil(20/8)+1)*11+1*14+9
        CompressionFunction():same as above
    hash(): 3
        Squeeze(): 8
            WordToByte(): 26
                Writebyte(): 36
            WordToByte(): 3
            Permutation(): Same as above
        Squeeze(): 2

Total:14+6+6+5*1+5*9+3+7+29+(4+29+(9+(5*14+1)+1+(5*(5*15))+1+4+5*(4+5*3+5+5*15)+1+(9+5*4+5*14)+(2+4*11)+17+1))+2+(ceil(20/8)+1)*11+1*14+9+(4+29+(9+(5*14+1)+1+(5*(5*15))+1+4+5*(4+5*3+5+5*15)+1+(9+5*4+5*14)+(2+4*11)+17+1))+3+8+26+36+3+(9+5*14+1)+1+(5*(5*15))+1+4+5*(4+5*3+5+5*15)+1+(9+5*4+5*14)+(2+4*11)+17+1)+2

5. SPONGENT

Spongent(): 22
    Permute(): 15

Nextvalueforlfsr(): 10
Permute(): 3+34*14+1
PLayer(): 7+(272-1)*21+9+10
Permute(): 3
Spongent(): 16
Permute(): Same as above
Spongent(): 8
Permute(): Same as above

Total:22+(15+10+3+34*14+1+7+(272-1)*21+9+10)+3+16+(15+10+3+34*14+1+7+(272-1)*21+9+10)+8+(15+10+3+34*14+1+7+(272-1)*21+9+10)

6. SIP HASH

test_vectors(): 1+16*9+5+64*8
Siphash:(85+((13/8)*(34+2*48))+62+6+2*42+10+2*38+17+2)

Total:1+16*9+5+64*8+(85+((13/8)*(34+2*48))+62+6+2*42+10+2*38+17+2)

7. LESAMNTA-LW

showTestVector(): 23+(32*7)+4
Hash(): 1
Init(): 22
Update(): 12
setMessage(): 18
Update(): 4
CompressionFunction(): 30
blockcipher(): 6
KeySchedule**()**:
4+(64*17)
functionQ(): 2
toUint8(): 1
functionQ(): 13
Mixcolumn(): 16
functionQ(): 2
toUint32(): 28
KeySchedule(): 7
blockcipher(): 9
messageMixing(): 6
functionG(): 3
functionQ():(2+1
+13+16+2+28)*2
functionR(): 1
toUint8(): 2*(1)
toUint32(): 2*28
messageMixing(): 8
blockcipher(): 8
CompressionFunction(): 5
Update(): 16
setRemainingMessage(): 8+(32s*15)
Hash(): 2
Final(): 4
paddingMessage(): 8
compressionFunction(): same as above
Final(): 12
compressionFunction(): same as above
Final(): 4
toBitSequence256(): 13

Total:23+(32*7)+4+22+12+18+4+(30+6+4+(64*17)+2+1+13+16+2+28+7+9+6+3+(2+1+13+16+2+28)*2+1+2*1+2*28+8+8+5)+16+8+(32*15)+2+4+8+(30+6+4+(64*17)+2+1+13+16+2+28+7+9+6+3+(2+1+13+16+2+28)*2+1+2*1+2*28+8+8+5)+12+(30+6+4+(64*17)+2+1+13+16+2+28+7+9+6+3+(2+1+13+16+2+28)*2+1+2*1+2*28+8+8+5)+4+13

VI. RESULTS

HASH FUNCTIONS	TIME	No. of Lines of Assembly Code
PHOTON	14.82606 ms	13688
SPONGENT	200.58801 ms	18715
LESAMNTA-LW	988.805 µs	5111
SHA-1	76.87200 ms	15369
SHA-256	47.01 µs	254
SHA-3	1.28504 ms	8522
JH HASH	2.29 µs	200
SIP HASH	432.527204 µs	1264

Fig. 4. Runtime of differing Lightweight Hash Function

Thus, as is evident from the tale above, there is an increase in the runtime of the lightweight hash functions with an increasing number of the lines of the assembly codes.

Hence, based on the experiment conducted we can conclude that JH-HASH is the fastest algorithm, with the least number of the lines of assembly code, followed by SHA-256, being the second best, then SIP-HASH lightweight hash function, followed by LESAMNTA-LW, SHA-3, PHOTON, SHA-1 and PHOTON in an increasing order to their run-time.

Our purpose to find the best lightweight hash functions, to be applied on real-time systems, with the purpose of providing a better security so that the chances of the systems being

intruded by foreign organizations or systems is reduced, has been achieved. These lightweight hash functions are much more reliable to be applied on modern IoT-based automobile systems than the traditional Encryption or hashing algorithms. The study made by conducting the above experiment can be visualized in an easier and better way by looking into the graph below, which shows the hashing algorithms in an increasing order of their runtime, on the hardware.

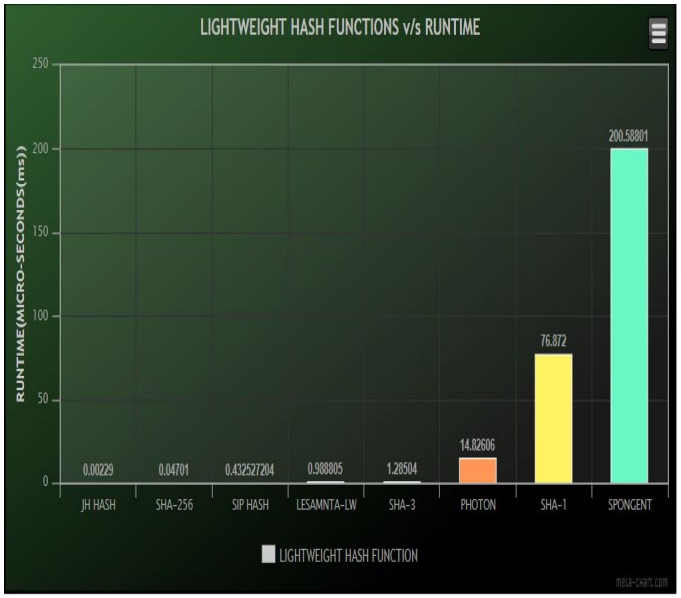

Fig. 5. Graph showing the Lightweight Hash Functions with an increasing runtime. A Lightweight Hash Functions v/s Runtime Plot

Conclusion

This paper provides a comparison between different lightweight cryptographic hash functions which can be used to provide reliable and not easily intrudeable security to the modern IoT-based, real-time automobile systems, where the classical cryptographic mechanisms may fail to work as equally efficiently.

References

[1] Baraa Tareq Hammad, Norziana Jamil, Mohd Ezanee Rusli and Muhammad Reza Z`aba, "A survey of Lightweight Cryptographic Hash Function," in International Journal of Scientific & Engineering Research Volume 8, Issue 7, July-2017

[2] "Comparison of Different Lightweight(LW) Crypto-Hash Functions for IOT," in Comparison of Different Lightweight(LW) Crypto-Hash Functions for IOT: [Essay Example], 2966 words GradesFixer, 2019

[3] Xu Guo and Patrick Schaumont, "The Technology Dependence of Lightweight Hash Implementation Cost," in Center for Embedded Systems for Critical Applications (CESCA) Bradley Department of Electrical and Computer Engineering Virginia Tech, Blacksburg, VA 24061, USA

[4] Jian Guo1, Thomas Peyrin, and Axel Poschmann," in The PHOTON Family of Lightweight Hash Functions"

[5] https://www.cryptolux.org/index.php/Lightweight_Hash_Functions#DM-PRESENT

[6] https://en.wikipedia.org/wiki/SHA-1

[7] https://en.wikipedia.org/wiki/SHA-2

[8] https://en.wikipedia.org/wiki/SHA-3

[9] Stefan Mangard,"Cryptographic Hardware and Embedded Systems-CHES 2010" in 12th International Workshop, Santa Barbara, USA, August-2010

[10] Hirotaka Yoshida,"On the Design and Use of Lightweight Cryptography for Cyber-Physical Systems", AIST, Japan, Kolkata, India (16 November 2018)

[11] William J. Buchanan, Shancang Li & Rameez Asif,"Lightweight Cryptography Methods" in Journal of Cyber Security Technology, ISSN: 2374-2917, March-2018

[12] Honorio Martin, Pedro Peris Lopez, Enrique San Millan, and Juan E. Tapiador," A lightweight implementation of the Tav-128 hash function"in IEICE Electronics Express, Vol.14, No.11, 1–9

[13] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Mar´ıa Naya-Plasencia," Quark: a lightweight hash",April-2012

[14] Andrey Bogdanov, Miroslav Kne˘zevi´c, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede," spongent: A Lightweight Hash Function"

[15] Shoichi HIROSE, Kota IDEGUCHI, Hidenori KUWAKADO, Members, Toru OWADA, Bart PRENEEL, Nonmembers, and Hirotaka YOSHIDA," An AES Based 256-bit Hash Function for Lightweight Applications: Lesamnta-LW",in Special Section on Cryptography and Information Security, IEICE TRANS. FUNDAMENTALS, VOL.E95–A, NO.1 JANUARY 2012

[16] Yuhei Watanabe, Hideki Yamamoto, Hirotaka Yoshida," A Study on the Applicability of the Lesamnta-LW Lightweight Hash Function to TPMS"

[17] Jean-Philippe Aumasson and Daniel J. Bernstein," SipHash: a fast short-input PRF"

[18] St´ephane Badel, Nilay Da˘gtekin, Jorge Nakahara Jr, Khaled Ouafi, Nicolas Reff´ e, Pouyan Sepehrdad, Petr Su˘sil1, Serge Vaudenay," ARMADILLO: a Multi-Purpose Cryptographic Primitive Dedicated to Hardware"

[19] Jiali Choy, Huihui Yap, Khoongming Khoo, Jian Guo, Thomas Peyrin, Axel Poschmann, and Chik How Tan," SPN-Hash: Improving the Provable Resistance Against Differential Collision Attacks"

[20] https://www3.ntu.edu.sg/home/wuhj/research/jh/

[21] Wenling Wu, Shuang Wu, Lei Zhang, Jian Zou, and Le Dong," LHash: A Lightweight Hash Function (Full Version)"

[22] Elif Bilge Kavun and Tolga Yalcin," A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications"

[23] https://keccak.team/keccak.html

[24] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche," Keccak and the SHA-3 Standardization",February-2013

[25] Khushboo Bussi, Dhananjoy Dey, Manoj Kumar, B.K. Dass," Neeva: A Lightweight Hash Function",February-2016

[26] Andrey Bogdanov , Miroslav Kne˘zevi´c , Gregor Leander , Deniz Toz , Kerem Varıcı , and Ingrid Verbauwhede," SPONGENT: The Design Space of Lightweight Cryptographic Hashing"

[27] Mohamed Ahmed Abdelraheem , C´eline Blondeau , Mar´ıa Naya-Plasencia , Marion Videau , and Erik Zenner ," Cryptanalysis of ARMADILLO2"

[28] Muhammad Rakha Rafi Bayhaqi, Bety Hayat Susanti, Mohamad Syahral, "Correcting Block Attack on Reduced NEEVA"

[29] Hirotaka Yoshida, Dai Watanabe, Katsuyuki Okeya, Jun Kitahara, Hongjun Wu, Ozgul Kucuk, Bart Preneel, "MAME: A compression function with reduced hardware requirements", in ECRYPT Hash Workshop 2007 Barcelona, Spain May 24, 2007

[30] T. P. Berger , J. D'Hayer , K. Marquet , M. Minier , G. Thomas, "The GLUON family: a lightweight Hash function family based on FCSRs", in Africacrypt 2012, Ifrane, 10-12 July 2012

[31] Ashish Kumar and Somitra Kumar Sanadhya, "Attacking the Tav-128 Hash function", in IIIT-Delhi Technical Report Date: 28-July-2010"

[32] Tobias Meuser, Larissa Schmidt, and Alex Wiesmaier, "Comparing Lightweight Hash Functions – PHOTON & Quark"

[33] Honorio Martin, Pedro Peris Lopez, Enrique San Millan, and Juan E. Tapiador" A lightweight implementation of the Tav-128 hash function" in IEICE Electronics Express, Vol.14, No.11, 1–9