Neeva: A Lightweight Hash Function

Khushboo Bussi[1], Dhananjoy Dey[2], Manoj Kumar[2], B.K. Dass[1]

[1]Department of Mathematics, University of Delhi,
Delhi-110 007, INDIA.
khushboobussi7@gmail.com, dassbk@rediffmail.com
[2]Scientific Analysis Group, DRDO, Metcalfe House Complex,
Delhi-110 054, INDIA.
{dhananjoydey, manojkumar}@sag.drdo.in

February 3, 2016

Abstract

RFID technology is one of the major applications of lightweight cryptography where security and cost both are equally essential or we may say that cost friendly cryptographic tools have given more weightage. In this paper, we propose a lightweight hash, *Neeva-hash* satisfying the very basic idea of lightweight cryptography. Neeva-hash is based on sponge mode of iteration with software friendly permutation which provides great efficiency and required security in RFID technology. The proposed hash can be used for many application based purposes.

Keywords: Hash function, Lightweight cryptography, Present, Quark, Sponge construction, Spongent

1 Introduction

When we think about the advancement of today's world, the very first thing comes into our mind is the 'Internet' and with it, 'how secure it is'. One of the important tasks in front of cryptographers is to preserve the privacy, authentication as well as integrity of the messages which we want to send through insecure channel like internet. Hash functions have a significant role in cryptography. It maps messages of arbitrary length into a fixed length digest. They were used to check the integrity of message initially but now they are being used in each and every field of online world. They are the building blocks of today's digital world. The online banking transactions would not be secured enough without hash functions, password protected systems would not exist without it, even the android phones would have lost their charm without that extra feature of pattern lock which is an application of hash functions (SHA-1). In the mentioned examples, we don't only need to manage privacy and integrity but also sometimes authentication (keyed hash) and many other features and this is done by standard dedicated hash functions such as SHA-224, SHA-256, SHA-384, SHA-512 (collectively called SHA-2 [SS]) and now SHA-3 [RPCK] can also be counted in this list. MD5 [Ber] and SHA-1 [CMR] are outdated due to their low security and SHA-2 are excessively used worldwide for data integrity, privacy, digital signatures and for many more purposes. SHA-3 has got standardised in Aug, 2015 by NIST and now can be used in the applications of hash function. Dedicated hash functions usually need more memory size hence cost to achieve the normal efficiency and required security. A decent trade-off between efficiency and security is what we are looking in a standard hash function.

Size plays an important role in today's world as it can be seen in many small computing devices that are used in day-to-day life. These small embedded devices like barcodes, biometrics, smart cards, RFIDs (Radio frequency identification) based devices which play a very prominent part of today's modern world where integrity, authentication and compactness are needed. The "Internet of Things" which comprises a network of physically small things needed in electronics, software and sensors for exchanging data between the objects. The compactness can be understood by the limitation on size or battery power and here comes lightweight cryptography. As the name suggests, it would be light in the sense that the memory part is restricted, i.e., low cost in the hardware but performance doesn't get halted. In this case, we need to have a good trade-off between storage and security. Lightweight cryptography ensures that the algorithm would be efficient and secured (majorly preimage resistant) at the lower cost. Its main aim is to achieve "low cost" subjected to good efficiency in hardware as well as software and security of algorithm. We don't want to keep the information secret forever, hence low-embedded devices do not need security more than what is required. Many lightweight ciphers have been proposed since then which includes many block cipher as well as stream cipher like HIGHT [HSH], IDEA [WPS], NEOKEON [DPR], PRESENT [BKLP], Grain-128a [HJM] and many more can be added in the list. Most of them are not a novel design, instead constructed on the existed designs to assure their security and performance are upto the mark.

Lightweight hash functions are integral part of lightweight cryptography. It has vast applications in smart cards technology, RFID tags [Fin] not just for tracking the shipping materials but also for every grocery item and many others. There excess demand in every sector, viz., industries, education, services etc. says it all. These hash functions target on low cost of implementation without compromising much on efficiency and provides necessary security. We have sponge based lightweight hash like QUARK [Aum], PHOTON [GPP], SPONGENT [BKL] which provide good trade-off between efficiency and security. There are few designs of lightweight hash functions which are based on stream cipher, i.e., BEAN [KOJ] which could be used as random number generators and have been successfully used in many applications.

Lightweight cryptography is still under exploring phase. We don't have proper standard lightweight hash or lightweight block ciphers but as already mentioned many have been introduced lately. A lot of research and analysis are going on for their betterment so that it can be used for further applications and providing an upcoming area of research.

In this paper we propose a lightweight hash function, *Neeva-hash*[1]. From now onwards we call this hash as Neeva-hash. This paper is organised in the following manner: In section 2, lightweight cryptography is briefly discussed, we describe our proposed lightweight hash scheme in section 3 and its analysis is shown in section 4.

2 Lightweight Cryptography

Cost, security, and efficiency are the three major dimensions of lightweight cryptography. The standardization of lightweight ciphers has not taken place yet, so any design whether it's a conventional one or unconventional novel structure is welcomed with keeping in mind its related cost, power consumption and cryptographic security and performance needed in today's world. Few of the lightweight ciphers have been designed initially like Crypto1 is a stream cipher with 48-bit key designed for RFID tags [Sol], Cryptomeria cipher is a Feistel based block cipher used in encryption on DVD audio discs, Kindle is a stream cipher with 128-bit key and used in Amazon kindle. Because of their specified use and low cost they are used very much but been broken recently [BLR]. In the last decade, massive designs been introduced in lightweight ciphers. Many lightweight block ciphers and stream ciphers are there in the usage. Present, is one of the important ultra lightweight block cipher and been extensively used in many lightweight block cipher based hash. After SHA-3 competition many sponge based hash functions, for example Quark – sponge construction based hash with non-linear Boolean functions available in the digest size of 256-bit, 176-bit and 136-bit, Photon – sponge like construction and AES like primitive as internal permutation available in the digest size 256-bit and 160-bit, Spongent – sponge construction with Present permutations and counter available in the digest size 88-bit, 128-bit, 160-bit 224-bit and 256-bit, even Keccak (small) can be used as lightweight hash with $r = 40$, $c = 160$ and digest size is 160-bit where r is the rate and c is capacity, are designed and used extensively for their good efficiency and software friendly designs.

The security of sponge based designs depends on their capacity. A sponge function is been found indifferentiable from a random oracle upto $2^{c/2}$ computation. For n-bit sponge based hash function and $n \geqslant c$, the security bound for collision resistant is $2^{c/2}$ and for second preimage resistant, it has been reduced to $2^{c/2}$. If the sponge construction is hermetic (underlying permutation doesn't have structural distinguishes) and has a reasonable small rate (r) then security bound for preimage resistant has been reduced to 2^{n-r} [BKL].

3 Proposed Scheme

The scheme proposed in this paper is based on sponge construction. Here initial register is of b-bit and $b = r + c$ where r is rate and c is capacity of the state b. The state b is of 256-bit. The rate and capacity is 32-bit and 224-bit respectively, viz.,

[1] *"Neeva"* is a Punjabi word which literally means "low". Here we are using Neeva to explain the small memory size required for this hash hence the low cost

$r = 32$, $c = 224$. As in sponge construction, message block will be xored to the most significant 32-bit of the initial register. Message is padded so it can be divided into the blocks of 32-bit. The initial state s_0 is of all zeroes i.e., $s_0 = 0^r||0^c$, first message block is xored and initial register is updated. Here we have sixteen 16-bit words in the updated register. We apply Present S-box (4×4) on these sixteen words in parallel for confusion. Then first, second and third word is updated by xoring with fourth word and keeping fourth word unchanged. This unbalanced Feistal structure is applied in parallel on four 64-bit strings of the register to provide word wise diffusion but this alone would not be enough. For overall diffusion throughout the 256-bit register, there is a 8-bit left rotation of the register and then we add round constants to it. Addition is done as addition modulo 2^{16}. The register is updated and this whole comprises to one round. So here we have three basic operations for confusion and diffusion, i.e., S-Box layer for confusion, xoring and rotation will provide overall diffusion and modular addition will remove fixed patterns. The whole process will be repeated for 31 more rounds i.e., in total 32 rounds. So, every message block will go through these 32 rounds.

The sponge construction processed in three phases as follows:

(i) **Initialization Phase**

The message can be of arbitrary length. The padded message should be a multiple of 32-bit. For that we need an unambiguous padding rule. Given an input message M of length ℓ bits, append the bit 1 to end of M, and then append k '0' bits and 1 at the end of it where k is the smallest positive integer satisfying the following equation.

$$(-\ell - 2) \equiv k \; mod \; 32$$

For sponge based hash function, we need to have a message blocks of size of rate.

$$M||Pad(M) = M_1||M_2|| \cdots ||M_t,$$

where each M_i is a 32-bit block.

(ii) **Absorbing Phase**

The first message block is xored with most significant 32-bit of initial register, s_0 (i.e., 256-bit register of all zeroes) and then permutation f applies on it. Permutation f is 32 times composition of function g.

i.e.,

$$g(x) = (rotl_8(F(S(x)))) \boxplus_{2^{16}} RC_j$$

$$f = \underbrace{g \circ g \cdots \circ g}_{32-times}$$

where S is the Present S-box (4×4) acting in parallel, F is the Feistal structure defined earlier (shown in the figure 2) and RC_j, $0 \leq j \leq 31$ are the 32 round constants written in the last

In the processing of one message block, 32 times g acts on the register. This keeps happening till all message blocks are absorbed. In the absorbing phase, the initial register as_i is updated by the following manner:

$$as_i = f(as_{i-1} \oplus M_i||0^c),$$

for $1 \leq i \leq t$ and $as_0 = s_0$ is initial register.

(iii) **Squeezing Phase**

The most significant 32-bit of the final register of absorbing phase are taken out and then permutation f applies on the same register and the most significant 32-bit are taken again. This process continues and every time most significant 32-bit are taken out. These bits then concatenated till we get 224-bit required digest i.e., we need to apply f permutation six more times to get a digest size of 224-bit. In the squeezing phase, ss_i is updated by the following manner:

$ss_i = f(ss_{i-1})$ for $1 \leq i \leq 7$ and $ss_0 = as_t$.

Figure 1 describes the phases of sponge construction [BDPA].

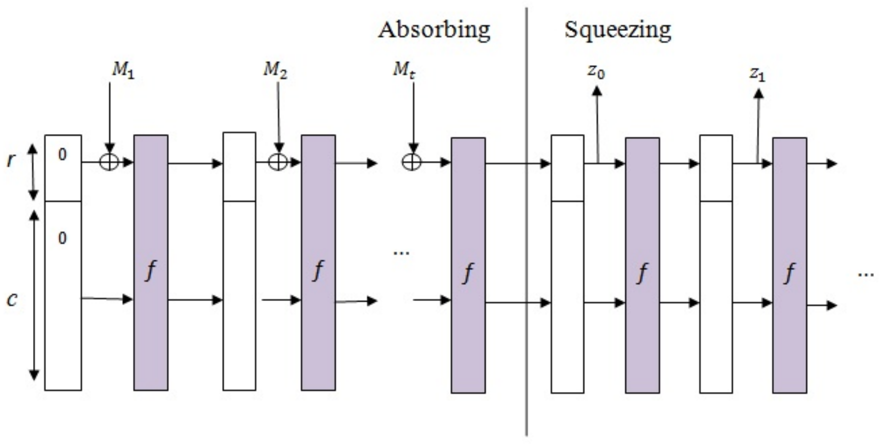

Figure 1: Three Stages of sponge construction

3.1 The Compression Function

The compression function of proposed scheme is based on sponge construction [BDPA]. Suppose we want to compute Neeva-hash of a message M. Message M is padded and divided into the 32-bit blocks and then the first message block M_1 is xored to the state. After applying the Present S-box in parallel, the updated register is divided in sixteen 16-bit words on which we apply Feistel structure on every 64-bit parallely. After a 8-bit left rotation, it is added to a round constant. The updated register after modular addition is the output of first round. It keeps feeding to the next round till 32 rounds. This is the absorbing phase of compression function as message is getting absorbed here. If we have more than one message block, then after the first block absorption, its output will be taken as an updated register and the next message block will be xored to updated register and again whole of the steps would be repeated for this message block. We keep processing all the message blocks in this manner till they get exhausted and this would end absorption phase. In squeezing phase, the most significant 32-bit of final register of absorbed phase is taken out. Then we apply permutation f on the updated register and every time we take out the most significant 32-bit. This is done six more times and all these seven 32-bit are concatenated to get the output of 224-bit. Hash digest of 224-bit would usually provide security of 2^{112} and here $c = 224$ which implies that collision security would be 2^{112} which is good with respect to today's security requirement.

3.2 Mode of Operation

After initialization, all the message blocks are first absorbed on which we apply the permutations. Each message block is processed by f which is a composition of 32 times of g. Once message blocks are exhausted then we start taking first 32-bit after every f application, to have a digest of 224-bit. This would need application of f seven times. The following figure describes the processing of one message block with Neeva-hash.

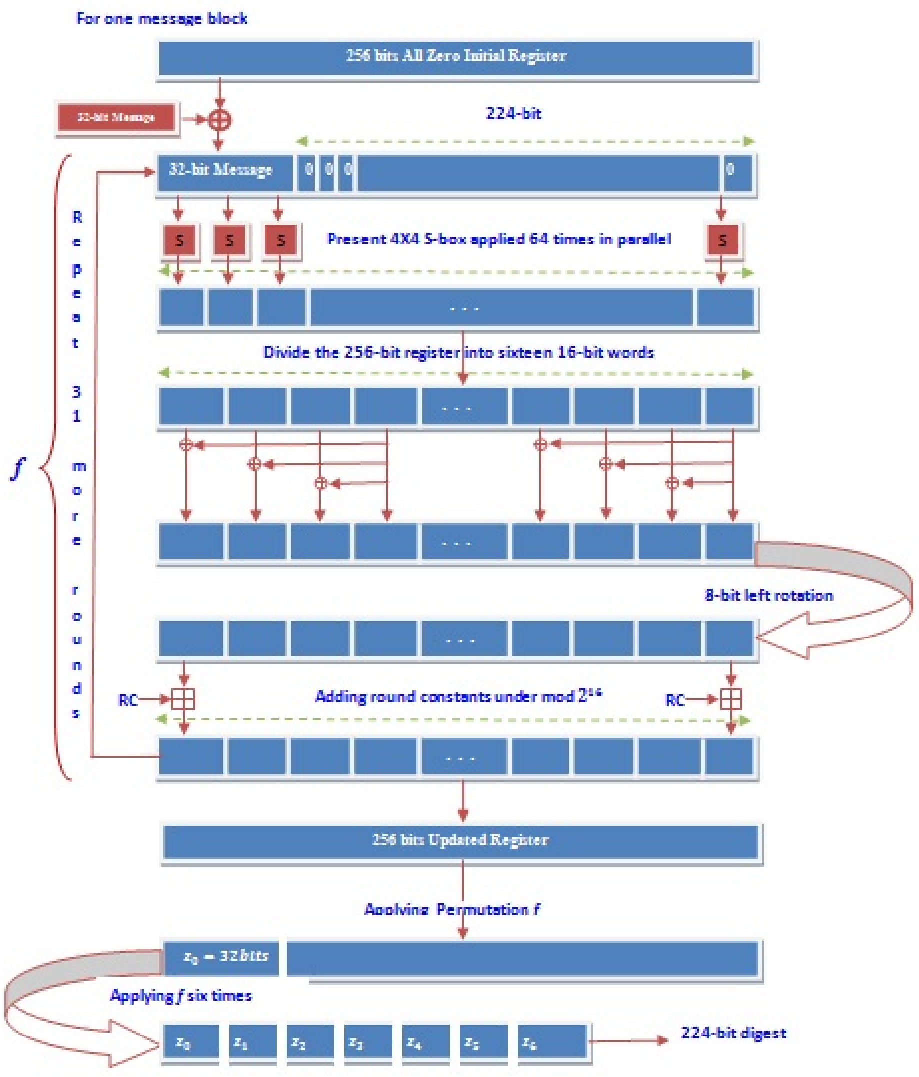

Figure 2: Neeva-hash function

```
input : $M_1, M_2, \ldots, M_t$
for $i = 1$ to $t$ do
    $A_i \leftarrow CV_{i-1} \oplus M_i$, $CV_0 = 0^{32}||0^{224}$
    for $j = 0$ to $31$ do
        $B_i^{(j)} \leftarrow S^{(j)}(A_i)$ Apply S-box 64 times in parallel
        $B_i^{(j)} = b_0^{(j)}||b_1^{(j)}||\cdots||b_{15}^{(j)}$, $b_m^{(j)} \rightarrow$ 16-bit words, $0 \leq m \leq 15$
        for $k = 0$ to $3$ do
            for $l = 0$ to $2$ do
                $b_{4k+l}^{(j)} \leftarrow b_{4k+l}^{(j)} \oplus b_{4k+3}^{(j)}$
            end
            $B_i^{(j)} \leftarrow b_0^{(j)}||b_1^{(j)}||\cdots||b_{15}^{(j)}$
        end
        $B_i^{(j)} \leftarrow rotl_8(B_i^{(j)})$
        $B_i^{(j)} \leftarrow B_i^{(j)} \boxplus_{2^{16}} RC_j$
    end
    $CV_i \leftarrow B_i^{(31)}$
end
return $A_t \leftarrow B_t^{(31)}$;
$r = 0, 1 \ldots, 6$;
$z_r \leftarrow MSB_{32}(f^r(A^t))$
$H = z_0||z_1||z_2||z_3||z_4||z_5||z_6$
```

Algorithm 1: Neeva-hash Function

4 Analysis of Neeva-hash

In this section we will discuss the analysis of Neeva-hash. The test values have been given in the end of the paper.

4.1 Efficiency of Neeva-hash Function

In this subsection we are analysing the efficiency of Neeva-hash with one of the popular lightweight hash functions, Spongent-224. The following table provides the comparison in the efficiency of Neeva-hash with Spongent-224 on an Intel core 2 duo 32-bit OS E8400 @ 3 Ghz processor with 1 GB RAM.

File Size (in MB)	**Spongent-224** (in Sec)	**Neeva-hash** (in Sec)
1	1.44	**0.65**
5	3.06	**2.53**
10	5.87	**4.99**

This shows that the efficiency of Neeva-hash is better than Spongent-224 which makes it really software friendly.

The speed of this hash algorithm has been calculated as almost 12067 cycles/bytes which seems fairly good in context to hardware.

4.2 Avalanche Effect

A message M of length 1024-bit is taken and its Neeva-hash(M) is calculated. While calculating Neeva-hash it will be padded before dividing it into the blocks of 32-bit which means we would have 33 message blocks. Now change in the i^{th} bit in message M will generate M_i files where $1 \leq i \leq 1024$. The hamming distance of M_i from M exactly 1. We then calculate Neeva-hash(M_i), thus compute the hamming distance d_i, between Neeva-hash(M_i) and Neeva-hash(M) for $1 \leq i \leq 1024$ and finally the hamming distances in the corresponding 32-bit words of the hash values are computed.

The results have been shown in the following table with the maximum, the minimum, the mode and the average value of hamming distances.

Changes	W_1	W_2	W_3	W_4	W_5	W_6	W_7	*Spongent-224*	***Neeva-hash***
Max	24	24	24	25	25	26	23	136	**137**
Min	8	7	5	7	7	8	8	89	**89**
Mode	16	15	17	16	16	16	18	113	**113**
Mean	16.03	15.77	16.03	16.06	15.97	16.07	15.98	111.84	**111.91**

Table 1: **Hamming Distances**

The following figure shows the hamming distance range of 1024 files which is coming out to be almost uniform i.e change in one bit of the input brings ≈ 50% change in output digest of hash.

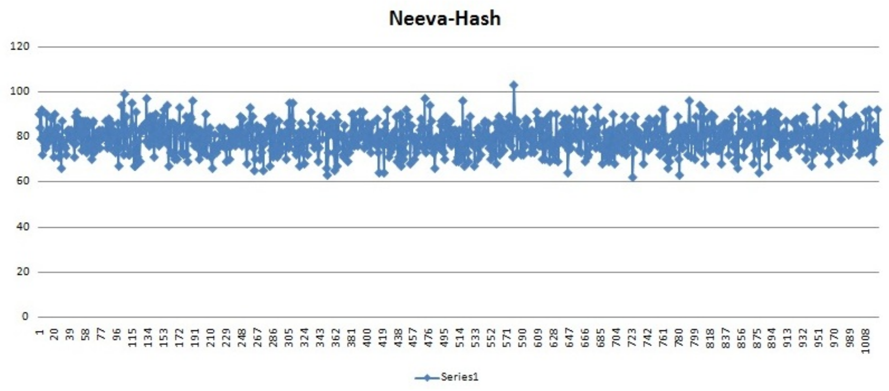

Figure 3: Hamming Distances range of the 1024 files

4.3 Differential Characteristics

Differential attack was presented by Biham and Shamir in 1990 on DES block cipher [BS].This attack exploits the high probability differences in input and output of an encryption scheme. High probability input and output differences of the non-linear components (e.g., S-box in this case, as modular addition is not counted here because we try to apply differential cryptanalysis of the modified version of our proposed hash function by replacing modular addition of constants with xor operation with the round constants) are used to form differential trail of the sponge f function by joining 1 round high probability differentials trails. The modified f function uses a 4×4 Present S-box as its only non- linear component now. Maximum differential probability for arbitrary input difference producing a output difference in a single S-box application is $\frac{4}{16} = 2^{-2}$ [Wan]. This value ensures that even if there is only one active S-box in each round, still differential attack will require 2^{64} chosen plaintexts to distinguish the first 32-bit of Neeva-hash.

Now, we see whether differential cryptanalysis is applicable on Neeva-hash. This hash function is using PRESENT S-box in their construction. In the modified version of Neeva-hash, atleast one bit input difference causes an active S-box after one round and 1 more active S-box after second round if it occurs in the most significant byte of the word of updated register. The number of active S-boxes is 12 after 9-rounds. If we keep continuing in this way till all 32-rounds, the number of active S-boxes would reach to 50. Then, the minimum number of active S-boxes is 50 (Here we have taken the input difference in the first sixteen most significant bits). Hence the maximal probability of finding a differential characteristic is $(2^{-2})^{50}$, i.e., 2^{-100}. This means we require 2^{100} chosen plaintexts to distinguish the most significant 32-bit of the output whereas by birthday paradox this chosen plaintexts is 2^{16} (for most significant 32-bit of the output) which is comparatively very small as compared to 2^{100}. It shows that differential cryptanalysis is not applicable to modified version of Neeva-hash hence it will not applicable on Neeva-hash.

4.4 Bit-Variance Test

Bit Variance is one of the statistical tests for the randomness of the hash digest. It basically tests the change in the hash output when input bits are changed. It measures the uniformity in each bit of the output. Given an input message, all the changes in the input are accounted and output is calculated for each of the change. Then, for each digest bit the probabilities of taking on the values of 1 and 0 are measured considering all the outputs produced by applying input message bit changes. If the probability, $P_i(1) = P_i(0) = 1/2$ for all digest bits $i = 1, \ldots, 224$ then, the Neeva-hash function has attained maximum performance in view of bit variance test [KZ]. Since it is computationally infeasible to consider all input message bit changes, we have evaluated the results for only up to 1024 files, viz. $M, M_1, M_2, \ldots, M_{1024}$ which we have generated for conducting avalanche effect, and found the following results:

The experiments were performed on 1024 different messages (which we have taken in avalanche test). This hash function passes the bit-variance test. Plotting the probability (Figure 4) of each of the bits (224-bit), we see that the average probability is approximately 0.50.

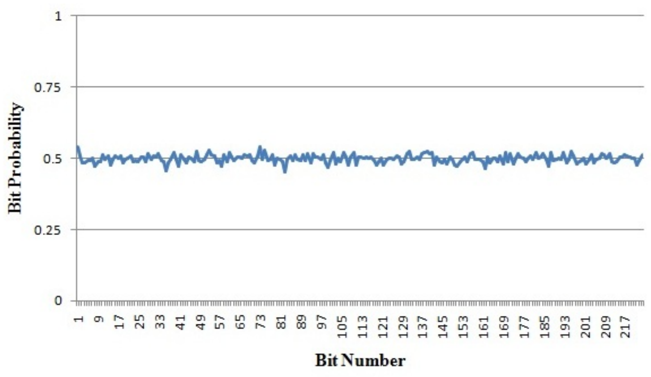

Figure 4: The probability of the bit position

4.5 Near-collision resistant

A hash function is called near-collision resistant if it is computationally infeasible to find two different inputs with hash output differ in small number of bits i.e., for two different messages M and M', their Neeva-hash values are almost same (differ in small number of bits), which means hamming weight of $(H(M) \oplus H(M'))$ is relatively small (upto 16-bit). To check the near-collision resistance we have taken 100,000 files and checked xor of hamming distance of *Neeva-hash* of two random files from the lot. So, we need to choose two files out of 100,000 files that would be $\binom{100000}{2}$ which is 4,999,950,000 files. After analyzing, it shows the minimum and maximum hamming weight of the hash of these files comes as 66 and 162 respectively. The hamming weight 113 comes maximum number of times i.e. 266,272,292.

No. of files having the difference between 92 and 132

$$(92 \leq \#files \leq 132) \ = 4,967,495,075(i.e.,\ 99.35\%)$$

That means more than 99% of files give hamming distance between 112 ± 20 which is good with respect to the fact that they won't give any near-collision attack as for near-collision the hamming distance of two files needs to be really small viz. upto 16-bit. Hence by the analysis results *Neeva-hash* shows resistant to near-collision attack.

5 Conclusion

In this paper we have proposed a lightweight hash function, Neeva-hash, based on sponge construction with a software friendly permutation. It gives the collision resistance 2^{112} which is good with respect to today's security requirements. Analysis of this hash function includes avalanche effect, a heuristic proof of differential characteristics, bit variance test and near-collision resistant test. It satisfies all the properties needed for an RFID tag and pseudo random number generator. It seems a good option for lightweight hash and can be used in many of its applications.

References

[Aum] J. P. Aumasson, *Quark: A Lightweight Hash*, 2012. Available online at `https://131002.net/quark/quark_full.pdf`

[BDPA] G. Bertoni, J. Daemen, M. Peeters & G. V. Assche, *The Keccak SHA-3 Submission*, Jan 2011. Available online at `http://keccak.noekeon.org/`

[Ber] T. Berson, *Differential Cryptanalysis mod* 2^{32} *with Applications to MD5*, in Advances in Cryptology Eurocrypt'92, LNCS, vol.0658, Springer, 1993.

[BKLP] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin & C. Vikkelsoe, *PRESENT: An Ultra-Lightweight Block Cipher*, CHES, LNCS, vol. 4727, Springer 2007.

[BKL] A. Bogdanov, M. Knežević & G. Leander et. al., *Spongent : A Lightweight Hash Function*, Cryptographic Hardware and Embedded Systems, LNCS, vol. 6917, Springer, 2011. Available online at `https://eprint.iacr.org/2011/697.pdf`

[BLR] A. Biryukov, G. Leurent & A. Roy, *Cryptanalysis of the "Kindle" Cipher*, Selected Areas in Cryptography, LNCS, vol. 7707, Springer, 2012.

[BS] E. Biham & A. Shamir, *Differential Cryptanalysis of the Full 16-round DES*, Advances in Cryptology- CRYPTO'92, LNCS, vol. 740, Springer, 1993.

[CMR] C. Canniére, F. Mendel & C. Rechberger, *Collisions for 70-Step SHA-1: On the Full Cost of Collision Search*, Selected Areas in Cryptography, LNCS, vol. 4876, Springer, 2007.

[DPR] J. Daemen, M. Peeters, V. Rijmen & G.V. Assche, *Nessie Proposal: NEOKEON*, 2000. Available online at `https://gro.neokeon.org/neokeon-spec.pdf`

[Fin] K. Finkenzeller, *RFID Handbook*, third edition, Wiley publications, 2010.

[GPP] J. Guo, T. Peyrin & A. Poschmann, *The PHOTON Family of Lightweight Hash Functions*, 2011. Available online at `https://eprint.iacr.org/2011/609.pdf`

[HJM] M. Hell, T. Johansson & W. Meier, *Grain: A Stream Cipher for Constrained Environments*, International Journal of Wireless and Mobile Computing, vol. 2(1), Inderscience, 2007.

[HSH] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim & S. Chee, *HIGHT: A New Block*

Cipher Suitable for Low-Resource Device, Cryptographic Hardware and Embedded Systems, LNCS, vol. 4249, Springer, 2006.

[KOJ] N. Kumar, S. Ojha, K. Jain & S.Lal, *BEAN: A Lightweight Stream Cipher*, International conference on Security of information and networks, ACM, 2009.

[KZ] D. Karras & V. Zorkadis, *A Novel Suite of Tests for Evaluating One-Way Hash Functions for Electronic Commerce Applications*, IEEE, 2000.

[RPCK] A. Regenscheid, R. Perlner, S. Chang, J. Kelsey, M. Nandi & S. Paul, *Status Report on the First Round of the SHA- 3 Cryptographic Hash Algorithm Competition*, NIST Internal Reports 7620, 2009. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf`

[Sol] H. Soleimany, *Studies in Lightweight Cryptography*, PhD Thesis, Aalto University, 2014.

[SS] S. Sanadhya & P. Sarkar, *Attacking Step Reduced SHA-2 Family in a Unified Framework*, Available online at `http://eprint.iacr.org/2008/271.pdf`

[Wan] M. Wang, *Differential Cryptanalysis of PRESENT*,

[WPS] L. Wie, T. Peyrin, P. Sokolowski, S. Ling, J. Pieprzyk & H. Wang, *On the (In)Security of IDEA in Various Hashing Modes*, Foundations of Software Engineering, LNCS, vol. 7549, Springer, 2012.

Test Vectors

Test values of the three inputs are given below:

$$\begin{array}{rclllll}
\mathit{Neeva\text{-}hash}(a) & = & 52ca54ca & ad4617dc & b051b2c4 & cc6c1c9e \\
 & & 92753d16 & 47a22405 & aa912c08 & \\
\mathit{Neeva\text{-}hash}(ab) & = & 0a163ca8 & 02692371 & b2d1a303 & 5da3bb8f \\
 & & 5e9b08ee & 82e2d5f4 & 1e532c1a & \\
\mathit{Neeva\text{-}hash}(abc) & = & b0c8be3d & fcbc3886 & 439256e1 & fe568253 \\
 & & 5d58c7dd & 9124dbc3 & 6cc37c91 &
\end{array}$$

Round Constants

RC_0	$=$	`c7b119402be75b5fe34230e1c6de7511503b802a96a7f546fd02a80d8cb27863`
RC_1	$=$	`6990c02e24cf9ab94c057e4e08726162dccb97ca280e1ccb6db961615a126f97`
RC_2	$=$	`ff223911f7f604c272d7ec72db58b760669de33dee6be0202550c439d270f05e`
RC_3	$=$	`f5a6c2820cac1ab3b263f3f68b1d3c53118bb9d52521bd520eb7a1e5a3cb9e5b`
RC_4	$=$	`1612115e8201b0311ea4d23d2bb3f906832a60191b4181d9f3f2a22b9671f3ba`
RC_5	$=$	`d299ae33da1d4ed5ed9c5c77047b758fe01bb24d4801a33b8050013fbb396b14`
RC_6	$=$	`1d18fe11cd6aa678cfe053451418e7dbb8b382220290ebd42291a6ff6c4c1743`
RC_7	$=$	`4afc5e1277a7355ec0b5a2231a9e2ccc02f555d4739836567bcdef91d914cfe2`
RC_8	$=$	`ece8b0d3361a8b569fe8cecb31b9ecd7e730d51ab9f94b620357d728fdbeda72`
RC_9	$=$	`1e5d2b7bfca2f0cce303b2bf33be3dc4ce60882398bb64f60b7adb092bface29`
RC_{10}	$=$	`89a2a6a2baf87b8705ead75447d16334479ad1f87a467e1245e036f2119df0eb`
RC_{11}	$=$	`96b970981eb889eb988a96bf01fc1dd13a0c119519ffe34590a0fe36c225749e`
RC_{12}	$=$	`10f20d64be3da2783114fe4dfaef826db18e6e25cf42ff6f22a604a3496878d6`
RC_{13}	$=$	`104d1cdde66f47312729c321e0ca3b99d39b754672e3910d6a4ddc204a7989f6`
RC_{14}	$=$	`3b346ce05703de7eb2719130af1b426660aac3243e43b2234b95c10d28d13528`
RC_{15}	$=$	`786d780921f9490b94476162609fd9e100c2fdb347fe2208086b1d8fc2459661`
RC_{16}	$=$	`888460b5299cee14e2095e0676c4ee73aef17819767cd8ee9223162928c83763`
RC_{17}	$=$	`e80f465c9f7cfc78a49539b737812cbcdcd37347cf4d4025ac70a24356ef05d3`
RC_{18}	$=$	`ce366bd878a9218786f4fddef33e2ad51012edbde19085f0ebcee84638fa7126`
RC_{19}	$=$	`76a45e9feb2c4123370448278054b494b62d481b5c8403a1cab5529bea62b745`
RC_{20}	$=$	`adf6d3e93166a6f892b0a9d59d55a1a51ca11b9cb530d7f5d50946dd9ceeda2c`
RC_{21}	$=$	`3246b10c987b174fd9f598444a5c42e9ea390cf5c4c5a5fdba7e0a08f59d2f10`
RC_{22}	$=$	`9f3903e5338b6415d92b4707462d4ef82844f7897dcf8f702e131c062682a99a`
RC_{23}	$=$	`70ff29c4c11f18008dd533acd7248c9b0a642ebaf42b4fb20898288b394e5f33`
RC_{24}	$=$	`cb8befdfdf5b238b1c730c0bf30855bbc7a0bfa5ae3516ab7edd326f5611ae48`
RC_{25}	$=$	`dfeb28672f6bcfc1afb3d11a97bbe65fc0ffb97d526913fca74d7e995ba9a3a6`
RC_{26}	$=$	`9f7f4896467352c824c941af49866c11246f4529d55c0b1110b9047575249533`
RC_{27}	$=$	`79990702621c531145378996444dc267629c221a9d6fc3d75be71d704ae1bac2`
RC_{28}	$=$	`5f6731bf692923f1b6d1dce74905c7ca504acba3d0b95bc79d7787025783e5cf`
RC_{29}	$=$	`ec1d0d8ddd6b5d8dcf1c5a759fae7dc0c206489bc8f14d8d9e4a6bcb2287c7c3`
RC_{30}	$=$	`fc2d8fd04b8f582fadd6205ca979b648a2c6fc9b00ca8b389cd94a3ef90ad435`
RC_{31}	$=$	`40e308b38501c4273130a587906a0ccc5461f947f201759b50b61dd32adedb9a`