International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 06 Issue: 06 | June 2019 www.irjet.net p-ISSN: 2395-0072

ARGON2id IP Core

Aparna R[1], R. Nandakumar[2], Dr. Anil Kumar C.D[3]

[1]PG Scholar & Dept. of Electronics and Communication Engineering, GEC Idukki, kerala, India
[2]Scientist/Engineer & Dept. of Electronics and Communication Engineering, NIELIT, Calicut, Kerala, India
[3]Professor, Dept. of Electronics and Communication Engineering, GEC Idukki, kerala, India

---

Abstract - *Cryptography is that the art of concealing info from eavesdroppers by suggesting a secret key and a process of converting ordinary plain text into unintelligible text and vice-versa. The data integrity assurance and data origin authentication are important security services in financial transactions, e-commerce, data storage, etc. For assuring the integrity of transmitted data we use cryptographic hash functions. This paper demonstrates cryptographic hash function Argon2id, which is a hybrid of Argon2i and Argon2d. It has been oriented for the protection of low-entropy secrets without secret keys. It decodes passwords of any length and can produce a message digest of 256 bits. Argon2id is important because it compares and stores hash much more easily than the entire original sequences. Also, it is capable to store hashes instead of passwords. When a user switches their password, the system computes the hash of it and compares it to the hashes listed. Argon2id mainly composed of using a hash function and a compression function. The Hash function is for implementing argon2id is Blake2b algorithm. The Blake2b hash function is a modified version of Dan Bernstein's ChaCha stream cipher, however a permuted copy of the input local vector, XOR with some initialization vector, is added before each round. BLAKE2b use 64-bit words and produce digest sizes of 512 bits and 384 bits, respectively.BLAKE2b is faster than SHA-3, SHA-2, SHA-1, and MD5 on 64-bit x64 and ARM architectures. Through this project, Argon2id design is discussed and is implemented on Xilinx Spartan-6 LX16FPGA (XC6SLX16-CSG324C).*

Key Words: Memory hard function, Blake2b, Internal permutation, Time-memory trade-offs, Side channel attacks, Indexing function, Compression function.

1. INTRODUCTION

A cryptographic hash function h maps an input M bit string of arbitrary length to an output string h(M) of some fixed bit-length d. Cryptographic hash functions have many applications; for example, they are used in digital signatures, time-stamping methods, and file modification detection methods. Our envisioned attacker has obtained a copy of 1 or many (possible millions of) hashed passwords from an info, and wants to find one or many passwords matching a number of these hash values. Due to Moore's law, the hash function computation becomes faster, have been called multiple times to increase the cost of password trial for the attacker by using memory hard functions. Even though all these terms stand for a lot of more general functions with rather totally different properties and functions.

•**One-wayness, or Pre-image Resistance**: It should be infeasible for an adversary to calculate an input that maps to that element.

•**Collision-Resistance**: It should be infeasible for an adversary to find distinct values M and Mo such that,

h(M) = h(Mo).

• **Second Pre-image Resistance**: It should be infeasible for an adversary, given M, to and a different value M0 such that,

h(M) = h(Mo).

•**Pseudo-randomness**: The function h must appear to be a random function (but deterministic) of its input. This property requires some care to define properly.

1.1 Problems of existing schemes

Hashing an input means the transformation of a string of input or message into a shorter or harder fixed-length value or message digest that represents the original message string. Hashing is used to index the memory either dependent or independent of the input sequence. Thus mathematically related hash functions are known as "Provably Secure Cryptographic Hash Functions". The other category is hash functions that are not based on mathematical equations but on an ad hoc basis, where the message bits are mixed to produce the digest. PHC which started in 2014 highlighted the following problems,

• Should the indexing functions be password-independent or password-dependent?
• Is it better to fill more memory but suffer from time-space tradeoff or make more passes over the memory to be more robust?
• How should the input-independent addresses be computed?
• Reading smaller random-placed blocks is slower (in cycles per byte) due to the special locality principle of the CPU cache.
•If the block is large, how to choose the internal compression function and it should be cryptographically secure or more lightweight.

1.2 Our solution

Here demonstrates a hash function called Argon2id. Argon2id is a state of the art in the design of MHFs. It is a streamlined and simple design. It provides high resistance against tradeoff attacks and working principle is optimized for the x86 architecture and exploits the cache and memory organization of the recent Intel and AMD processors. Argon2id is composed of two variants: Argon2d and Argon2i. Argon2d maximizes resistance to GPU cracking attacks. It accesses the memory array in an input dependent order, which increases resistance against time–memory trade-off [3] attacks, but introduces possible side-channel attacks. Argon2i is optimized to resist side-channel attacks. It accesses the memory array in an input independent order.

1.3 Motivation

Our main objective is to maximize the cost of password trails on ASICs or FPGA. This memory size M translates to ASIC or FPGA area A. The running ASIC time T denotes the length of the longest computational chain in the algorithm and determines the latency of ASIC memory. The memory-hard functions (MHF) can be defined using the following mode of operations. Let consider an array of memory B is filled with the compression function G. For maximizing the cost we use memory hard function, it can be modeled as flows, The memory array B[ij] is filled with the compression function G:

B [0] = H (P; S);

The addresses can be calculated by the adversaries should satisfy the following conditions they are, independent of the password and salt but dependent on the public parameters used for hashing.

1.4 Tools used

1.4.1 Xilinx ISE

Xilinx ISE (Integrated Synthesis Environment) could be a software system tool made by Xilinx. The main functions of this tool are synthesis and analysis of HDL designs, enabling the designer to synthesize and mapping their designs, perform timing analysis, examine RTL diagrams, through Xilinx device programming we can synthesize the design from design entry. Here Argon2id design is discussed and is implemented on Xilinx Spartan-6 LX16FPGA (XC6SLX16-CSG324C).

1.4 .3 IP CORE

An IP (Intellectual Property) core is a block of logic or is a reusable unit of logic. In IP core layout design that is normally developed with the idea of licensing to multiple vendors with different chip designs. IP cores mainly used in electronic design automation (EDA) industry for reusing the design logic or functionality to multiple users. An IP core should be easily portable and able to easily be inserted into any vendor technology or design methodology.

2. DESCRIPTION OF ARGON2id

In this chapter describes the specification and algorithm of the next generation of the memory-hard hash unction Argon2id[1]. The pin diagram of Argon2id is shown in fig-1.

2.1 Specification of Argon2id

It is suitable for password hashing, password-based key derivation, cryptocurrencies, proofs of work/space, etc.

Argon2id has two types of inputs:

1. Primary inputs

- Message or password 'P' - 0 to 2^{32}- 1 byte.
- Nonce or salt 'S' - 8 to 2^{32}- 1 byte.

2. Secondary inputs or parameters

- Degree of parallelism 'p' -1 to 2^{24}- 1 byte.
- Number of iterations't' -1 to 2^{32} – 1 byte.
- Version number 'v' is one byte 0x13;
- Secret value or key 'K' -0 to 32 byte.
- Associated data 'X' -0 to 2^{32}- 1 byte.
- Type 'y'- 2 for argon2id.
- Tag length or output digest size 'T' -4 to 2^{32}- 1
- Memory size 'm' represented in KiB -4p.

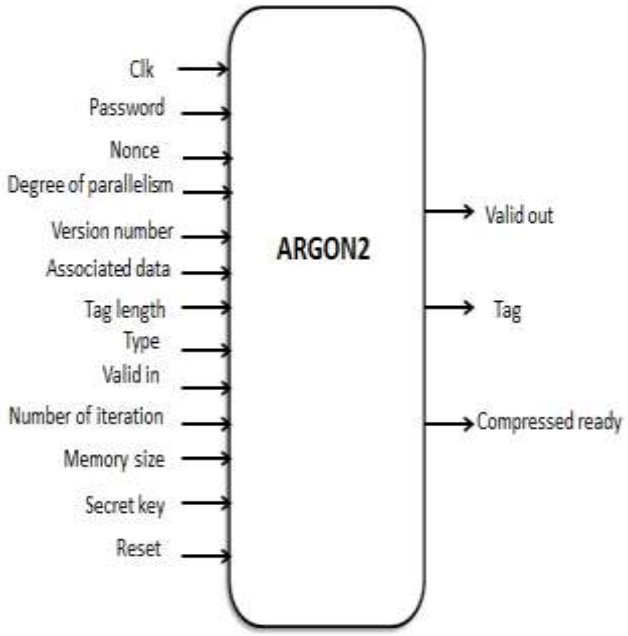

Fig -1: Argon2id entity

2.2 ARGON2id OPERATION

Argon2id uses an internal compression function G with two 1024-byte inputs and a 1024-byte output, and an internal hash function H. Here H is the BLAKE2b hash function, and the compression function G is based on Blake2b internal round function. A variable-length hash function H' built upon H is also used. G and H' are described in a later section. All the other parameters are also added to the input.

The variable length inputs Password, Salt, Key, and Associated data are prepended with their lengths. Password string P is the primary input of password hashing applications. The mode of operation of Argon2id is quite simple to illustrate when compression function G is iterated m times with no parallelism. Single-pass Argon2 with p lanes and 4 slices is shown in fig-2.

The Argon2id operation is as follows.

1. Establish Ho as the 64-bit. Here H is BLAKE2b and the non-strings p, T, m, t, v, y, length (P), length(S), length (K), and length(X) are treated as a 32-bit little-endian encoding of the integer.

2. Allocate the memory as m' and each block size is 1024 byte.

3. Compute B[i][0] and B[i][1] for all i ranging from 0 to p-1.

4. Compute B[i][j] for all i ranging from zero to p-1, and for all j ranging from two to q-1.

5. If the number of iterations t is larger than 1, we repeat the steps.

6. After t steps have been iterated, the final block C is computed as the XOR of the last column:

Algorithm

1. Generate initial 64-byte block Ho.

• buffer=(LE32(p)||LE32(T)||LE32(m) ||LE32(t) ||LE32(v) ||LE32(y)||LE32(length(P)) ||P||LE32(length(S))

• Ho = Blake2b (buffer, 64)

2. Allocate the memory as m' 1024-byte blocks, where m is the memory size.

• m'=floor (m/4p)*4p.

• q=m'/p.

• Memory is organized in a matrix B[i][j] of blocks with p rows (lanes) and q = m' /p columns.

3. Compute the first and second block of each lane

• Bi[0] =H'(Ho || LE32(0) || LE32(i))

• Bi[1] =H'(Ho || LE32(1) || LE32(i))

• H' function used here is the Variable-length hash function.

4. Compute remaining columns of each lane

• Bi[j]=G(Bi[j-1], Bi'[j'])

• Compression function G is based on blake2b permutation.

5. If t>1, we repeat the procedure and we XOR the new blocks to the old ones.

6. Compute final block C as the XOR of the last column of each row.

C = B[0][q-1]_ B[1][q-1]_:::_ B[p-1][q-1]

7. Compute output tag H'(C, tag Length)

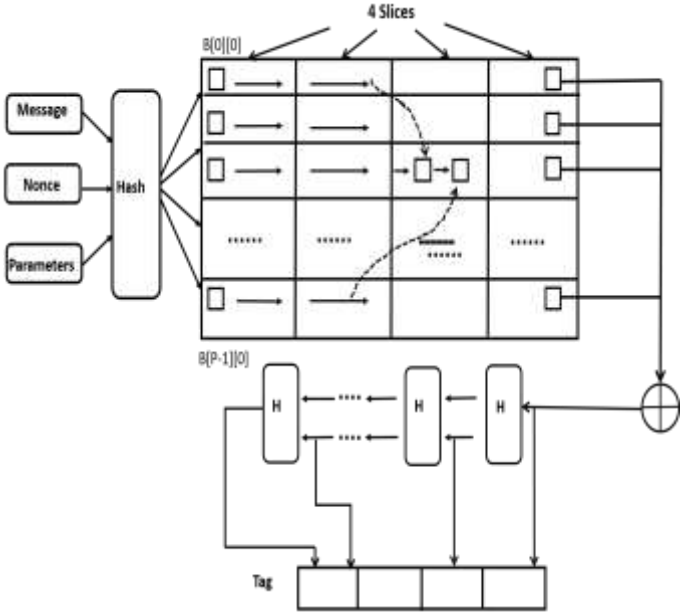

Fig -2: Argon2id mode of operation with parallelism.

2.2.1 BLAKE2b

BLAKE and BLAKE2 are the modification of Bernstein's ChaCha stream cipher, the only difference is that the input block is permuted and finally XOR with some round constants, the XOR output is added before each ChaCha round is shown in fig-3**.**

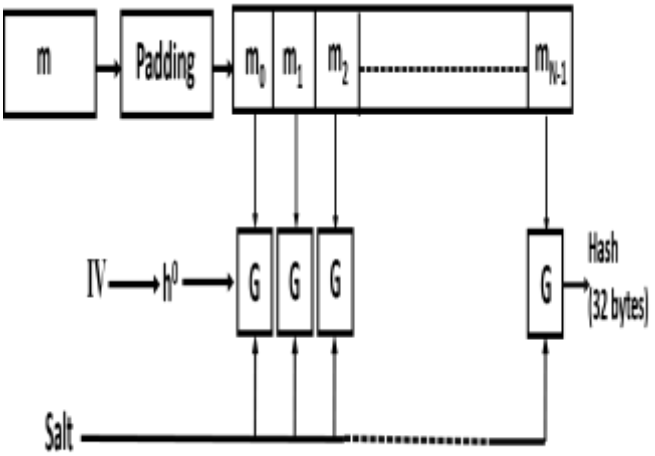

Fig -3: Blake2b operation

Like Secure Hash Algorithm, there are two variants whose word size is not equal. ChaCha round function operates on a four symmetric array of words. BLAKE generates 8-word hash value and repeatedly combined it with 16 message words and finally truncating the ChaCha result to obtain the final hash value.

Algorithm

1. BLAKE2b[2] uses an initialization vector that is the same as that used in SHA-512.

IV[i]=floor(264*frac(sqrt(prime number(i+1))))

IV0 = 0x6A09E667F3BCC908 **IV1** = 0xBB67AE8584CAA73B

IV2= 0x3C6EF372FE94F82B **IV3** = 0xA54FF53A5F1D36F1

IV4= 0x510E527FADE682D1 **IV5** = 0x9B05688C2B3E6C1F

IV6= 0x1F83D9ABFB41BD6B **IV7** = 0x5BE0CD19137E2179

2. Initialize local work vector V[0.....15]

(a) V[0...7]=h[0...7]

(b) V[8..15]=IV[0..7]

(c)V[12]=V[12] ⊕t0

(d) V[13]=V[13] ⊕ t1

(e) V[14]=V[14]⊕0Xffff….f(If last block is send)

3. Apply compression function to local work vector. The input Compress function 128-byte chunk of the input message and mixes it differently for each round:

Mix (V0, V4, V8, V12, m[S0], m[S1])

Mix (V1, V5, V9, V13, m[S2], m[S3])

Mix (V2, V6, V10, V14, m[S4], m[S5])

Mix (V3, V7, V11, V15, m[S6], m[S7])

Mix (V0, V5, V10, V15, m[S8], m[S9])

Mix (V1, V6, V11, V12, m[S10], m[S11])

Mix (V2, V7, V8, V13, m[S12], m[S13])

Mix (V3, V4, V9, V14, m[S14], m[S15])

4. The Mix function is called by the Compress function, and mixes two 8-byte words from the message into the hash state.

- Va =Va + Vb + x
- Vd=(Vd ⊕ Va)>>>32
- Vc= Vc + Vd
- Vb=(Vb ⊕ Vc) >>> 24
- Va=Va + Vb + y
- Vd=(Vd ⊕ Va)>>> 16
- Vc=Vc + Vd
- Vb=(Vb ⊕ Vc) >>> 63

5. Mix the upper and lower halves of V into ongoing state vector h

- h0..7= h0..7⊕ V0..7
- h0..7= h0..7 ⊕ V8..15
- Result_ h

2.2.2 VARIABLE LENGTH HASH FUNCTION (H')

Argon2id is capable of producing digests up to 232 bytes long. This hash function is internally built upon Blake2, Let Hx be a hash function with x-byte output. We define Ho as follows. Let Vi be a 64-byte block and Ai be its first 32 bytes. For desired hashes over 64-bytes (e.g. 1024 bytes for Argon2 blocks), then

1. Calculate the number of whole blocks:

r = Ceil(x/32)-1

2. The initial block is generated from the message:

V1 =Blake2b(x||message, 64);

3. Subsequent blocks are generated from previous blocks for i=2 to r do

Vr+1 = Blake2b(Vr, q) {q=x – 32*r}

4. Concatenate the first 32-bytes of each block Vi. Let Ai represent the lower 32-bytes of block-A1 || A2 ||... ||Ar||Vr+1

2.2.3 COMPRESSION FUNCTION

Compression function G(X, Y) operates on two 1024-byte blocks X and Y is shown in fig-4.

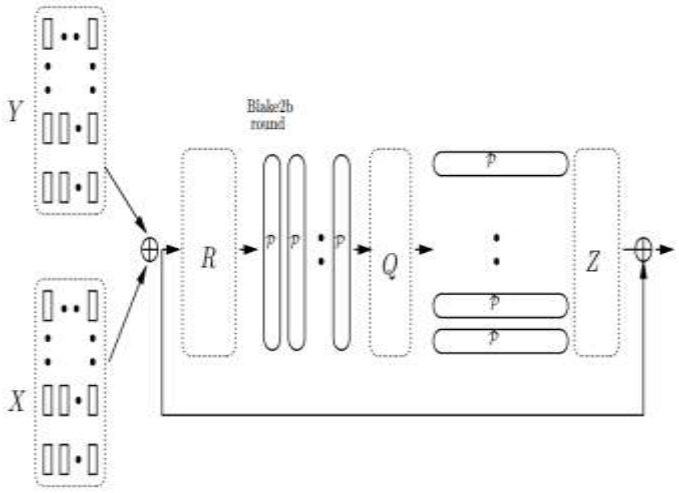

Fig -4: Block diagram of the compression function

- It first computes R = X ⊕Y
- Then P is first applied to each row to get Q.
- Then P is applied to each column to get Z.
- Finally, Output=Z _R

2.2.4 INDEXING FUNCTION

For paralleling the computational process, memory is partitioned into S = 4 vertical slices. The intersection of a slice and a lane is a segment of length q/S is shown in fig-5. All the components of the same slice are computed in parallel and there is no reference from each other. All other blocks can be referenced.

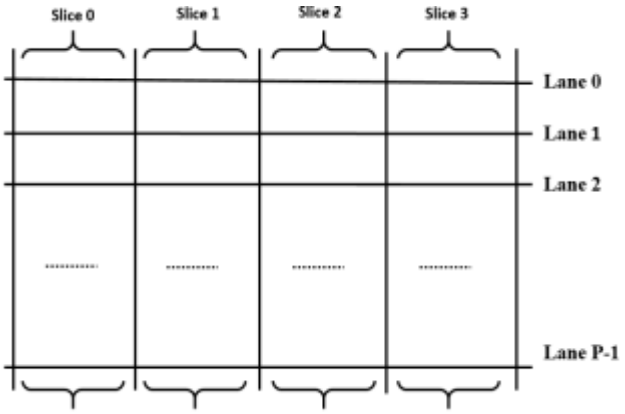

Fig -5: Indexing function

Getting the 32-bit values J_1 and J_2

(i) Argon2d

- J_1 = extract(B[i][j-1], 1)
- J_2 = extract(B[i][j-1], 2)

(ii) Argon2i

(LE64(r) || LE64(l) || LE64(s) || LE64(m') || LE64(t) || LE64(y) || LE64(i) || ZERO)

- r – the pass number
- l – the lane number
- s – the slice number
- m' – the total number of memory mapped
- t – the total number of passes
- y – the Argon2 type (2 for Argon2id)
- i – the counter

(iii) Argon2id

If the pass number is 0 and the slice number is 0 or 1, then compute J_1 and J_2 as for Argon2i, else compute J_1 and J_2 as for Argon2d.

Mapping J_1 and J_2 to reference block index

The value of l = J_1mod p gives the index of the lane from which the block will be taken. The set W contains the indices that can be referenced according to the following rules:

- W includes the indices of all blocks depending on i'.
- J_1=W(1-($J_1^2/2^{32}$))
- x = $J_1^2/2^{32}$
- y =(Wx)2^3
- j' = W - 1 - y
- i'= J_2modp

3. SECURITY ANALYSIS

Cryptanalysis[4]is the study of analyzing cryptography systems in order to study the hidden aspects of the chosen functions. Cryptanalysis is used to breach cryptographic security systems and gain access to the contents of the hashed message or encrypted message, even if the cryptographic key is unknown. In addition to the mathematical analysis of cryptographic algorithms, to study cryptographic algorithms with side-channel attacks that do not depend on target weaknesses can be performed using cryptanalysis, but exploit weaknesses in their cryptographic algorithm implementation.

3.1 RANKING TRADEOFF ATTACKS

Ranking tradeoff attack mainly used to figure out the costs of the ASIC-equipped adversary. For determining the cost first we need to calculate the time-space tradeoffs for Argon2id. This applies to both data-dependent and data-independent schemes;

- The area A determined by the amount of memory M on ASIC used for calculating the hash.
- The ASIC running time T is calculated by the time taken by the longest computational chain and also depend on the ASIC memory latency.
- Suppose that an adversary tries to compute H by creating time-space tradeoff similar to original condition using a fraction αM of memory for some α< 1.
- Using some time-memory tradeoff specific to H, the adversary has to spend C(α) times as much computation of hash and the running time used to calculate the hash increases by factor D(α).
- In order to fit the increased computation into time, the attacker has to place additional cores on a chip.
- Therefore, the time-area product changes from AT to Atα as,

ATα=AT(C(α)+ αD(α))

- The memory bandwidth limit Bw may also increase the running time, i.e cost of password trail is increased.

3.2 MEMORY OPTIMIZATION ATTACK

Memory optimization attack possible to optimize the memory use of hash functions. Therefore, for each block B[i] there is a time gap between the moment the block is used for the last time and the moment it is overwritten. We formalize this issue as follows.

- Let us denote by _(i) the reference block index for block B[i].
- Since addresses li can be precomputed, an attacker can figure out for each block B[i] when it can be discarded.

Li=(1-l^t/m)

- Our experiments show that in 1-pass Argon2id, L=.15, i.e. on average 1/7-th of memory is used.

• For t > 1, the address 'li' is precomputed and the saving strategy uses the fraction,

$$Li=((m+i-li^t)/m)$$

• For t > 1 this strategy uses 0.25 of memory on average, so the time-area product is increased.

3.3 INTERNAL COLLISION RESISTANCE

The compression function G violate collision resistant property, then it produces identical outputs for distinct inputs. Rewrite the compression G as follows:

$$G(X; Y) = P(Z) \oplus (Z); \quad Z = X \oplus Y$$

• Let us prove that all Z have different values while considering certain assumptions.

• Consider Argon2id with d lanes, s slices, and t passes over memory, then

1. P (Z) ⊕ Z is collision-resistant, i.e. it is hard to find a; b such that P (a) ⊕a = P (b) ⊕ b.

2. P(Z) ⊕Z is 4-generalized-birthday-resistant, due to this property it is hard to find distinct a, b, c and d such that it satisfies P(a) ⊕P(b) ⊕P(c) ⊕P(d) = a ⊕b ⊕ c ⊕d

4. AVAILABLE FEATURES

The key feature of Argon2id is its performance and the ability to use multiple computational cores due to parallel processing in the way that enjoins time-memory tradeoffs. Here provide an extensive list of features of Argon2id.

Table -1: Available features of Argon2id

Design rationality	Due to the streamlined and simple design of argon2id it categorized into three main operation: generic mode of operation, tradeoff-resilient parallelism with synchronization points, and tradeoff-resistant compression function. The three ideas are based on intensive research method.
Performance	Argon2id fills memory very fast. Data-independent version Argon2i securely fills the memory spending about a CPU cycle per byte, and Argon2d is twice as fast. It is mainly used for the applications that need memory-hardness but less CPU time is allowed for algorithm similar to cryptocurrency peer software
Tradeoff resilience	Despite high performance, Argon2id provides a reasonable level of tradeoff resilience. The tradeoff attacks applied to Catena and Lyra2 show the following result. With default number of passes over memory (1 for Argon2d, 3 for Argon2i,) an ASIC-equipped adversary can't decrease the AT product.
Scalability	Argon2id is scalable both in time and memory dimensions. Both parameters can be changed independently provided that a certain amount of time is always needed to fill each and every block in memory.

5. RESULT AND DISCUSSION

This chapter includes the experimental setup, simulation result and FPGA implementation of hash function Argon2id.

5.1 EXPERIMENTAL SETUP

The arrangement comprises of Blake2b hash function and Blake2b round like compression function. A 64-byte Ho is generated first. The Ho is obtained by applying all input to Blake2b hash function, the input applied is obtained by concatenating all primary and secondary inputs +-in the specified order. The H' is composed of Blake2b depending on the digest size. To find all remaining elements compression and indexing function are applied. The next step is to XOR all the elements in the last column. Again applying the hash function to the XORed output.

5.2 SIMULATION RESULT

The simulation result of argon2id (It is a hybrid of Argon2i and Argon2d, in the first pass memory is accessed in password dependent and in other passes independent of password, which gives some of Argon2i's resistance to side-channel cache timing attacks and much of Argon2d's resistance to GPU cracking attacks.) is shown in figure 6.

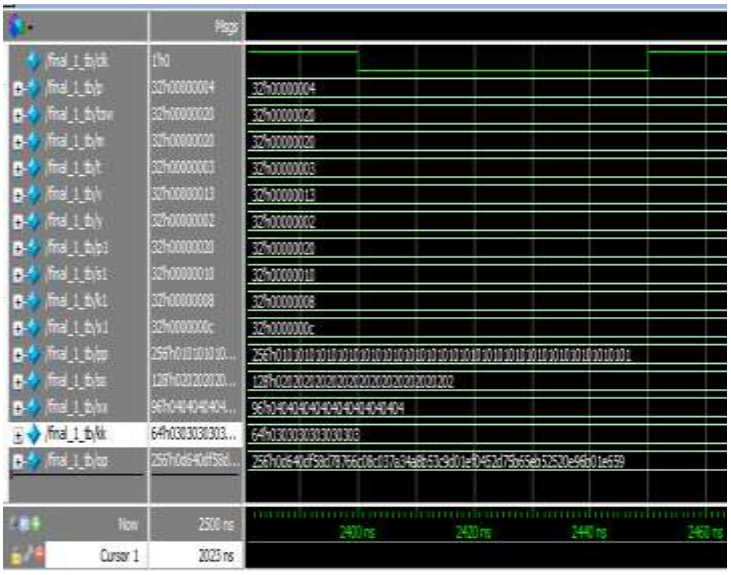

Fig -6: Simulation result of Argon2id

5.3 SYNTHESIS RESULTS

The design is implemented on a Xilinx ISE 14.3 is used as an FPGA development environment during the implementation process. The blocksize is a fixed amount of data that the algorithm will process at a time. The latency is the number of cycles that are needed to hash a password. After implementing the design on a Xilinx Spartan-6 LX16FPGA (XC6SLX16 CSG324C), when analyzed the Post-PAR static timing report the maximum check frequency is found out as 63.56MHz. The blocksize, in this case, is 256 and latency is 64 cycles. So the throughput can be easily calculated using the above equation and the calculated throughput is 603.4Mbps.

5.4 HARDWARE RESULT

Integrated Logic Analyzer (ILA) core is a physical logic analyzer core that samples various probes and displays the internal signal and used to monitor and control any internal signals used in our design.

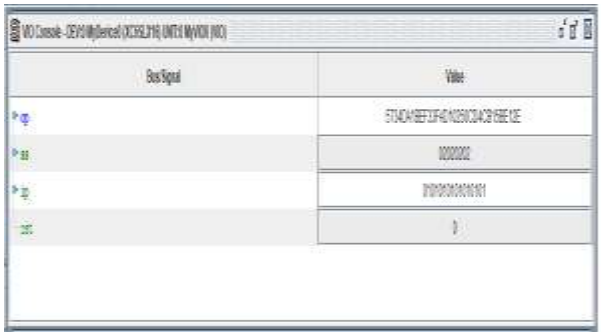

Fig -7: VIO console for 128 bit output

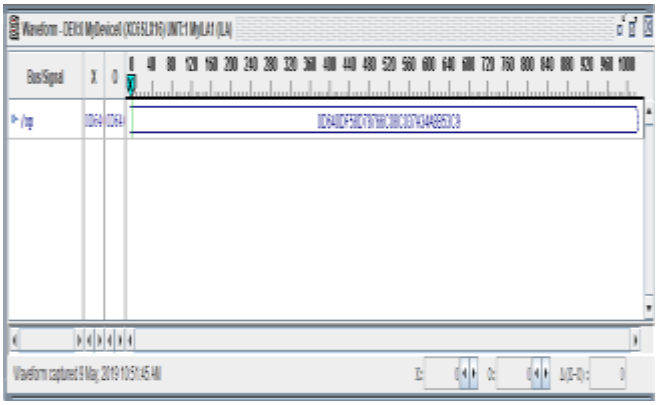

Fig -8: Waveform

6. CONCLUSION

This work proposes a memory hard function Argon2id, which maximize the ASIC or FPGA implementation cost of an adversary for a given CPU computing time. The Argon2id design has been successfully implemented on Xilinx Spartan-6 LX16FPGA (XC6SLX16CSG324C) and verification was done with the help of Chipscope ILA tool. Argon2id is a multipurpose hashing function, mainly used for password hashing, key generation, cryptocurrencies and other applications that need high memory (MHF). The power utilization is analyzed through Xilinx Power Analyzer (XPA). Compared to other hash function power utilization is very low in Argon2id. A clear and compact design is obtained by implementing this, so the operation gets a certain ratio rationale through this. The main objectives of the hash function are to increase the cost of password trail. Thus the cost of password trail can be increased through implementing argon2id. This increase is usually because of a hike in memory size.

ACKNOWLEDGEMENT

This work is supported by National Institute of Electronics and Information Technology (NIELIT), Calicut.

REFERENCES

[1] Biryukov, Alex, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications." 2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2016.

[2] Aumasson, Jean-Philippe, et al. "BLAKE2: simpler, smaller, fast as MD5." International Conference on Applied Cryptography and Network Security. Springer, Berlin, Heidelberg, 2013.

[3] Biryukov, Alex, and Adi Shamir. "Cryptanalytic time/memory/data tradeoffs for stream ciphers." International Conference on the Theory and Application of Cryptology and Information Security. Springer, Berlin, Heidelberg, 2000.

[4] Hatzivasilis, G., Papaefstathiou, I, and Manifavas, C. " Password Hashing Competition-Survey and Benchmark." IACR Cryptology ePrint Archive, 265, 2015.