JOURNAL OF NORTHEASTERN UNIVERSITY

Volume 25 Issue 04, 2022 ISSN: 1005-3026 https://dbdxxb.cn/ Original Research Paper

PERFORMANCE EVALUATION OF HASH FUNCTION USING NEW DECENTRALIZED BLOCKCHAIN-BASED SECURE KEYLESS HASH ALGORITHM

Bhagvant Ram Ambedkar

Research Scholar, Department of Computer Science and Engineering, Shri Venkateshwara University, Amroha, Uttar Pradesh, India Email: brambedkar@mjpru.ac.in

Pawan Kumar Bharti and Akhtar Husain

Department of Computer Science and Engineering, Shri Venkateshwara University, Amroha, Uttar Pradesh, India Department of CS&IT, MJP Rohilkhand University, Bareilly, Uttar Pradesh, India, Email: padutt, husainakhtar@gmail.com

Abstract— Nowadays, the implementation of advanced technology is using the principle of blockchain technology to secure the data communication of social media through the public network. However, each user wants to secure data communication and modern secure technique to protect their data. The integrity, confidentiality, and digital signatures are verified by secure hash algorithms (SHA). The hash code is generated from a variable length input message by a hash function that is processed by SHA. Several researchers have proposed many hash algorithms that used additive keys constant and initial value as the basic parameter of the cryptographic hash function. These key constants for all input messages are openly known and these are constant for any variable length input message. In this research, we are eliminating the key concept of the cryptographic hash function by using the decentralization principle of the blockchain. So that the design and implementation of a new decentralized blockchain-based secure keyless hash algorithm (NDBCKSHA) to evaluate the performance of the hash function by generating 384-bit fixed-length hash code and comparing its analytical analysis and experimental results in python 3.9.5 programming language. □

Index Terms— blockchain implementation model, hash function, how to verify the data communication through the public network, key constants, security issues with SHA

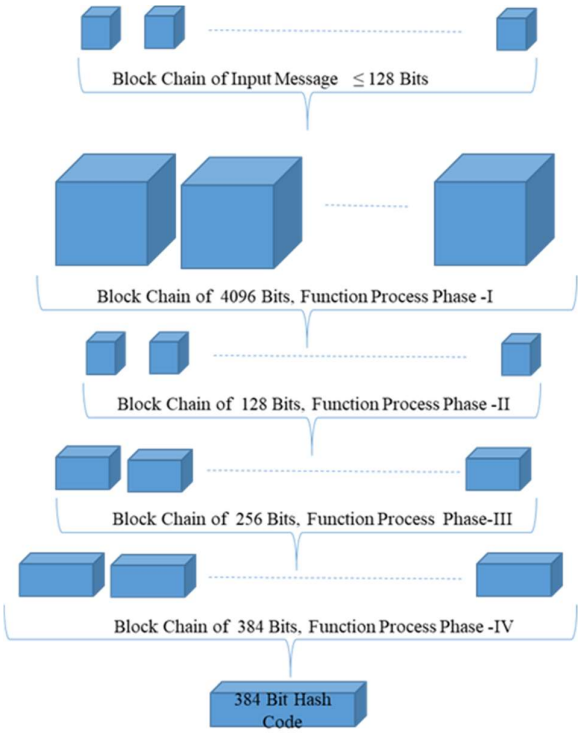
Introduction

This research is introduced to implement the keyless hash algorithm based on a decentralization principle blockchain. By using it we are eliminating the key concept of cryptographic hash algorithms. The cryptographic hash algorithms are using one-way keys to map the fixed-size hash codes. Many existing cryptographic hash algorithms are using a centralized hash function and one-way additive keys for the generation of hash code. Therefore we are using the decentralization principle of blockchain technology and implementing a hash algorithm NDBCKSHA-384 and which used a normal register to hold hash value. The function process of NDBCKSHA-384 is computed in four phases of function processing based on the decentralization principle of blockchain technology [1]. Decentralization means that the mapping of hash code is not processed by the single or centralized hash function. The

Submitted: 17/10/2022 **Accepted**: 16/11/2022

cryptographic hash function algorithms use the additive key constant and initial value as a basic parameter and it holds in the buffer registers [2]. The cryptographic hash function in the blockchain is a way to secure the message block by decentralization of function processing. In the blockchain, each block contains its block hash code and a hash code of its previous block. The modernization of technologies and the development of blockchain-based hash algorithms are support for development to protect the integrity of digital data and password storage. The double blockchain-based model to secure data storage and data communication through the public network and the need for an accurate security mechanism to avoid dangerous attacks [3]. Such a blockchain-based technique is very useful for generating hash codes to verify the integrity of data transmitted over the public network [4]. The hash function is used to protect web information from unauthorized users and to check information during transmission [5]. Hash algorithms are provided to use fixed-length pseudorandom bits that are presenting adequate entropy [6]. Hashing algorithms produce fixed-length code from variable-length input messages [7]. In this proposed algorithm, we are using fifty-five step processes in each round to generate 512-bit fixed-length hash code which is independent of key constants, it is very difficult to reconstruct the original message from the hash code because in this algorithm we are not using any openly known keyword.

The basic blockchain model of NDBCKSHA-384 is shown in Fig. 1.



Basic Blockchain Model of NDBCKSHA-384

Each block of the input blockchain is directly connected to the process of the hash function and maps the hash code from the input blockchain [8]. The hashing algorithms implement to secure the integrity of the digital signature and continuously increase the computing power that

attackers have at their throwing away [9]. The efficient characteristics are recently drawn by fault and enhanced hash security by the principle of blockchain technology [10]. The avoiding use of basic parameters is based on the cryptographic hash key used by the hash function [11]. Hash code 256 bit generated by proposed hash algorithms that enhance the processing of hash function by elimination of key constant and initial value as a basic parameter [12]. In this algorithm, we expand the variable length input message into the size of block 4096 bits by appending zero bits with one-time padding after the message bit. The expanded block is divided into 32 sub-block and the size of each block has 128-bits. The designed and implemented hash function generates the 384-bit hash code from the input blockchain and is used to verify the integrity and confidentiality of digital information. Verification of digital information is very difficult because it is varying between 0 and 1. The implementation of a hash algorithm based on a secure keyless blockchain is compression and expansion-based secure hash algorithms, that are generating the secure hash code [13]. The hash function confirmed the authenticity of the message during communication between the sender to the receiver. The security strength of the hash function can be increased by the circular shift operation. In this implementation of the hash function, we are using a left circular shifts operation with a hash function enhancing the performance of hash values [7]. The hash function has used a set of logical and arithmetical operations to generate a fixed-size hash code it was computed by SHA. It is used to verify the integrity of digital signatures and passwords during information communication through the public network. Design and Implementation of a hash function are required in case hashing applications are in low power and efficient memory space [14]. The hash function is the main center of SHA is provide high security and confidentiality to the user [15]. Therefore the hash function is required for the improvement of hash algorithms to secure the integrity of user information [16]. The principles of blockchain technology are very significant for verifying the processing of data integrity [17]. By designing and implementing the hash function, we can secure user information against an illegitimate person that can recreate the original message by using openly known information and input variable [18]. The implementation of the hash function has a big challenge in the case of the device having limited memory space [19]. The SHA family is providing an updated version of the secure hash algorithms (SHA) [20]. The properties of a “good” cryptographic hash function will satisfy all security requirements of the hash function [21]. The limited memory-spaced-based devices have big issues because hash algorithms have to provide security with an appropriate storage and processing capability [22]. The hash function is providing a big role to protect public network information from an unsanctioned user and check information integrity during information transmission through the public network [23]. Hash algorithms are providing users with fixed-length pseudorandom bits that are presenting adequate entropy [24]. Hashing algorithms produce fixed length code from variable length input messages and increase the security strength by left circular shift [25]. We are proposing a keyless blockchain-based secure hash algorithm to produce 384-bit hash code by designing and implementing the hash function. It is used low function processing time to generate hash code from the input message. The implementation of the hash function, we are

using four function phases and it used fifty-seven steps in a single round to generate 384-bit hash code in a fixed size. The generated hash code is satisfying all security requirements.

Application of Hash Function and Issues

Information technologies cause several issues with the security of public network assets that require enhancing the performance of public networks with new methodologies [26]. A lot of existing cryptographic hash algorithms are producing hash codes based on key constants. The hash function used 64 key constants to produce a hash code by a new hash algorithm [16]. An innovative secure algorithm is using six registers, each register has a 32-bit word length which is used for the initial value to produce a 192-bit hash code [17]. The attacker can predict the original message by using the key constant and initial value. Hashing for message authentication is describing the single-key and double-key versions for security proof [18]. Hash functions are enhancing the security strength during message exchange through public networks [19]. Hash algorithms have designed hash code by using higher-order two-variable polynomial functions [20].

Basic of Hash Function

The hash function process is used logical and arithmetic operation and map the fixed-length hash [21]. Faultless hash functions satisfy the property of security and require that the results of applying the function to the enormous set of inputs blockchain and input variables that produce outputs are matchlessly casual. The innocent hash function is required for safety to satisfy all security requirements of the cryptographic hash function.

SHA Family

The National Institute of Standards and Technology (NIST) commercialized safe hash algorithms, and the flaws in SHA were known as SHA-0 in 1993, these well-defined three new variants of SHA, known as SHA-256, SHA-384, and SHA-512, are known as SHA-2 hashing algorithms [13].

The main objective of the implementation of NDBCKSHA-384 is to avoid the use of basic parameters are the initial value and additive key constant Kt of cryptographic hashing algorithms.

Computation of Initial Value and Additive key Constant

Step1: Compute the square root value of the first eight prime number
Step2: Select the fractional part of step 1
Step3: Convert in hexadecimal value of step 2
Step4: compute the binary form of the square root of the first 16 prime numbers and select the first 16 Hexa- decimal digit fractional parts of these square root prime numbers [8] first sixteen prime numbers are hexadecimal values of step 3
Step5: Step four should be the Initial value or Initial vector or H0, as follows :

6A09E667F3BCC908 BB67AE854CAA73B 3C6EF372FE94F82B A54FF53A5F1D36F1
570E527FADE682D1 9B05688C2B3E6C1F 1F83D9ABFB41BD6B 5BEDCD19137E2179

We are analyzing step 5, there are required buffer registers to hold the initial value and function processing is processed in "n" steps and process of each step is using "n" additive constant key Kt and these values are equal to the size of buffer register in bits of the fractional part of the cube root of the first "n" prime numbers, which is constant for all input messages and these are required some extra memory space. So we are concluding this step and seeing the security strength of hashing algorithms is increasing by using these basic parameters. Many cryptographic hash algorithms are used initial values and key constants as the one-way cryptographic hash function.

Therefore we are implementing NDBCKSHA-384 based on the decentralization principle of blockchain technology and holding its hash value in a normal register. Verifying the security strength of the hash algorithms are required three basic properties one-way, weak collision resistance, and strong collision resistance, hash satisfying the design and implementation of good secure hash algorithms [13].

Padding

The padding is a technique used for expanding the input message bit in a constant fixed size of blocks by appending zero or one bit. We are designing and implementing the keyless input blockchain of 4096-bits with padding bits. The one-time padding is providing a very important role in the hash function to implementation of secure hash algorithms. We are using one-time padding for a large number of zeros with input messages to avoid, hash function issues. The padding is appending zero followed by one bit [22]. The security strength of NDBCKSHA-384 is increased by one-time padding and the length of padding bits is computed by the following relation:

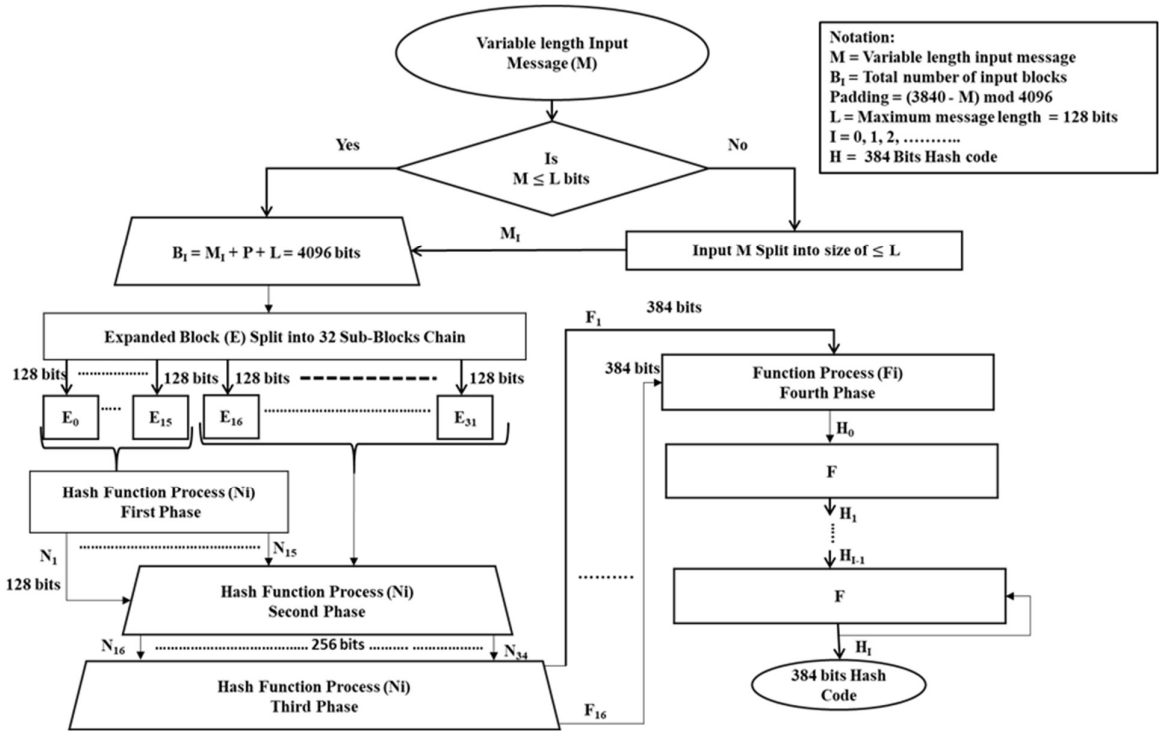
Padding bits (P) = (3968 – M) mod 4096) where M is the variable length input message size and also append 128 zeros with padding, which is L = 128 is the maximum input message size.

IMPLEMENTATION OF NDBCKSHA-384

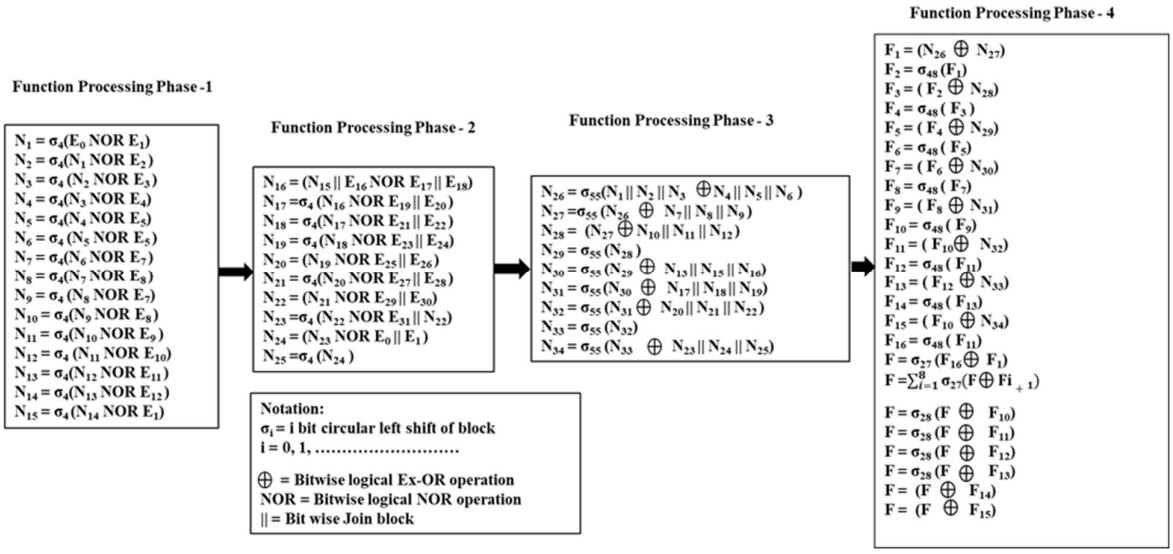
The basic architecture of NDBCKSHA-384 is shown in Fig. 2. It shows the decentralization function processing of the hash function. The following steps are required to implementation of the NDBCKSHA-384 for producing 384-bit hash code:

1. Convert the variable-length input message (M) into bits
2. Maximum size of input message length (L) = 128 bits
3. If M ≤ L, use P to expand the input message
4. If M > L split M into the length of M ≤ L, use P to expand the input message
6. Append the length of padding bit (P) = (3968 – M) mod 4096 bits with M.
7. Now the length expanded input block E = M + P + L = 4096 bits
8. Now E is divided into blockchains of thirty-two blocks (E0, E1, ……. E32), and the size of each block has 128 bits.

9. The hash function process is computed in four phases and the function processing of each phase is shown in Fig. 3. This blockchain model is using four hash function process phases to generate 384-bit hash code. First, we expand the variable length input message to a size of 4096 bits in fixed length. The expanded block is split into 32 sub-blocks chain and the size of each sub-block is 128-bit applying the function process used by these sub-blocks chain. The function process is computed in four phases as shown in Fig. 3.



Decentralized Hash Function Processing architecture of NDBCKSHA-384



Four Phases of Single-Round Function Processing of NDBCKSHA-384

The hash function processing is computed into four phases and holds computed results in the hold normal register shown in Fig 3. In the first phase function map, the 128-bit fixed length hash code. Description first phase is computed as follows:

N1 = (¬E0 * ¬E1) where: ¬ = inverse of given variable and * = product of input blocks

Ni = σ4 (¬Ni-1 * ¬ Ei), i = 2,3,.....15

σi = left circular shift "i" bits block of Ni, i = 0, 1, …….

In the second phase of function processing, we map the 256-bit hash code from the 128-bit input blockchain. Description second phase is computed as follows:

N16 = σ4 (¬(N15 || E16) * ¬(E17 || E18)), Where || = Bitwise Join operation

Ni = σ4 (¬Ni-1 * ¬ (Ei+1 || Ei+2)), i = 17, 18,.....31

In the third phase function processing map the 384-bit hash code from the 256-bit blockchain. Description third phase is computed as follows:

N26 = σ55 ((N1 || N2 || N3) (N4 || N5 || N6))

Ni = σ55 ((Ni-1 (Nj || Nj+1 || Nj+2)), where i = 27, …….., 34, and j = 7, ……, 23

Where = Exclusive-OR operation

The final fourth phase is to map 384-bit hash code from 384 input blockchain. Description fourth phase is computed as follows:

:

F1 = σ48 (N34 N26),

Fi = σ48 (Fi-1 Nj) Integer i = 2, ……,15 and integer j = 27, ……,33

Final hash function

F = σ27 (F16 F1)

F = σ27 (F Fi) where i = 2, ……13

F = (F F14)

F = (F F15)

For example, the Variable length input message executed by python-3.9.5 programming software is as follows:

Input Message = xyz

Length of input message in bits = 24

Length of input message in bits = 24

Length of Padding bit = 3944

Length of Input Block with Padding Bits= 4096

Length of Hash Code in Bits = 384

Hash code or Message Digest = fa3abc1f790e67d5eea3a0d10c13093a9aa489f6a37e8c532427ddd958812a34c1c9193454ae815adbd1929c557f561c

Length of Hash Code in Hexadecimal Digit = 96

Elapsed time in second = 0.0005008999999915886

We are calculating the executed elapsed time by following syntax which is executed by python:

import timeit

start = timeit.timeit()
NDBCKSHA -384 python code
end = timeit.timeit()
print("Elapsed time in second =",end – start)

RESULT AND DISCUSSION

In this section, all the results and the discussions should be made. Performance of NDBCKSHA-384 based on security requirement condition.

Security Requirements Condition: Our proposed algorithm is satisfying the resistance properties of various data integrity and security requirements [7]. These are mainly three types of security requirements as follows:

Preimage Resistance: The limitation of preimage is the maximum length of work < 2n , for the length of hash code n=384 bit, 384 is preimage resistance

Collision Resistance: the work is < 2n/2 for hash code 384 bit: 192 bits of collision resistance [27].

Input Message = x
Length of input message in bits = 8
Length of input message in bits = 8
Length of Padding bit = 3960
Length of Input Block with Padding Bits= 4096
Length of Hash Code in Bits = 384
Hash code or Message Digest = 5ab163861182e1b07486768a8f3c8f639119330e04d5992aef392ae5535a63ba846f9d6af0d99af8938b8ebebb0bac1f
Length of Hash Code in Hexadecimal Digit = 96
Elapsed time in second = 0.00040919999997868217

The input message is the preimage of the hash code.

Collision Resistance:
Input Message = y
Length of input message in bits = 8
Length of input message in bits = 8
Length of Padding bit = 3960
Length of Input Block with Padding Bits= 4096
Length of Hash Code in Bits = 384
Hash code or Message Digest = 6d35fed231ba85b59f9ced62ffcabb70f275d187daf1d5ae1a9318f0b30c1418ab86fb0033595ab1456c7ff352cc8a91
Length of Hash Code in Hexadecimal Digit = 96

Elapsed time in second = 0.0008744000000433516
As shown in the above-executed example hash code H (x) ≠ H (y), so it is collision-resistant.

Second Preimage: Therefore, it is impossible to find the message (x, y) which has the same hash code H(x) = H(y).
For example, we are taking 3552 bits of the input message split into blockchain lengths of ≤ 128 bits and producing 384 bits hash code of each block and final hash code as follows:
Input Message = "the implementation of advanced technology is using the blockchain model to secure the data communication of social media through the public network. However, each user wants to secure data communication and modern secure technique to protect their data. The hash code is generated from a variable length input message by a hash function that is processed by SHA. A lot of researchers have proposed many hash algorithms that used keys constantly"
The total length of the input message = 3552 bits
All security requirements of the hash function are satisfied by executed results shown in Table I and Table II

Experimental Results of Internal Blockchain of Input Message length of 3552 Bits

Block Chain of Internal Hash Code	Total No. of 1's in Hash Code	384-bit Hash code of internal blocks in Hexa-decimal digit
H_1	208	fced144bbf1fbff2b93951d33b8d10d05ccd105fec072229971da0debceb33e05f95e7d5da8e336c9c7ba150c2bfb8ce
H_2	182	5139894c1c6a382caec19f26568a077b34451db9f42566b4086309496b61548ca55033b76b9bec4ba9e1b8eb5a4097ba
H_3	188	670df993e43471c21b0481d3816a484774db3ec5de4ee23bce604ac4fbcb0e0d8239fc8ce0b1f7ecabf2134d2940ea2a
H_4	182	94d803dfe00aad0a6ca7c5117908ea7bd50c4c50f73032d5909d79dc9dd2b2bad11015df7b344c04cd8c8acf387b04d
H_5	192	c49bd55d0cbe25f617f6af51c1069b43312383272ab6ea0ef594ffe721289d9a1c0353bcc59894ee2efab2292854b4c5
H_6	198	519fd80a37295d1a3764fccfb1e01a9337e6f8f383f85d45cc742a531971f141fbe2c327fbf0792260f380c7d948b1ae
H_7	184	dca95fd23e47c35d435ec2a2916dd8e5ec8e76e58f492107231d7980cb08529d0de1eeb3cb03280a1c02d1576ac81736
H_8	176	d2852f4ec26e65ef455c6617a4bb856d822500508ca331af86c06f8a750b8bd25268251074007f1f21226e49d2575d1e
H_9	198	b5accf0e71275944872c17fa178237059e51860ffa0807bd90d7dbd6dedaaa50ff02b79f20c729d4aa71254a2ac02d18
H_{10}	192	7ec9204fbbc5f074cfb110940133c432521974612198a94f0a21bdcd4afb70f9a86efa3f3dad76c02c375bbd35b6b0c9
H_{11}	190	30d44d5da98b8a032a52c2ed7c9ce5903819bf9c6aacbae7686626ddc8c4d639ffae0abb098e7c04a24c781f1736dc8a
H_{12}	186	1537f2a48aaf3a8082f0f8390a245098ae96f51e67315ab3abe48d6243296bd1bbc7496a746fa9035c46d4c8b873ec38
H_{13}	180	2dfd391831ebf3d5af6cd9e841a05804be940ed58c1458b4909426364bc53e2aaae0c4c18f5333ca279abde133310232

H_{14}	196	48bcbc194a9cc35d25c7e971e27c8e4edbcb373d8f489e06eae8692bcd06510fa9369b56fff48806ee24bcd01bf82c31
H_{15}	194	82ccf8eb405e4f2668e16963e7738edb417b25c700564259187d6a4f7dfbad83ca8dc1217854512257d0daf8aee30f9f
H_{16}	178	4816c72b5d80d1d293851d5c03783c4873915dd8049e67201c335063070217fc1a4c3361de2c2a1ec933df76b3ce8a4f
H_{17}	198	7945d54d3b4eb07b9b4fce642bca3a14fa7891acf587ed6f70ac65314a054b299c25e59c1ec121bf9dded53455743895
H_{18}	198	81a6cfdfa728fb37ea9975851c46a0876ed78312c56c1f755f1a59cd80b9eba2d1abe3bef45c8166b2a128191c71e277
H_{19}	194	c08bf34c8f4f79cd6db7c25ea918e2c6cbec8deb32b2c8449aa4c1ea9227ae18c69e59298ff0a81e3df88b2fa9d46a11
H_{20}	210	a26d98f11f94db9d3759f588deef256234bbd9fc34a2bce6d3b50edbbc0f2c566ae9d7f1f3108bb2611630326ff8ede9
H_{21}	194	9af802346182ddeca7a748839447a04adba6ebcd5c5964ea301ecd5c03b216a91d8ed8f83db2e92710dabf4a975ddfd3
H_{22}	184	a0ff5791132b35a761ae5349f34a622214fc3c31eccea934a0d78ca8f49020a9703bd47e41aa249eee2a9f8b6611a586
H_{23}	204	267968ba711ceaff9061dc16b08b5b5d3bea1958011f4d7faddd43fc0326f1f68291cbf829e38abe0799bf364b5c9daf
H_{24}	204	7b6fa3dd53b147e332fd59e307b43f4fbd51779d00b1a632ced8fdc3b47f7be2052547e68f414b51d5068113259a37a5
H_{25}	192	4eceb760bb9ba10b27269b4df714e63fea2280a7c80c80a32ad9c86aad52697d6e715292ceee6f1f2a28949e19ea9556
H_{26}	202	28bb5d89ab4b26ef2e654bde444fa37ad7ef69c707146cdaa4ec40bfc1e762a3f1274a92c39485ecfbe0d1d33d48c3da
H_{27}	172	93e9f24f5f404e29d6b418b2145552f083443b0a11442110a6a370ab8aed407a17351aa5eb17b1201a4e8d3056fae916
H_{28} Final Hash Code	184	56b207c4066811ab8595c1eff1d08973f2717a24e53344d57b650f541b02c19c55212abe559661db5582c107fdae1e39

For a map of the n-bit size of the hash code, the length of the input message will be < 2n to prevent the preimage and second preimage attacks. The brute-force attack is to pick values of x at random and try each value until a collision occurs. For an n-bit hash value, the level of effort is proportional to 2n and it tries, on average, 2n-1 values of x to find one that generates a given hash value h [28].

For a collision-resistant attack, the length of the input message can't exceed 2n/2. If we take the random variables in the range 0 through M - 1, then the probability that a repeated element is encountered exceeds 0.5 after √M choices have been made. Thus, for an n-bit hash value, if we pick a block of input message at random, we can expect to find two data blocks. If collision resistance is required then the value 2n/2 determines the strength of the hash code against brute-force attacks with an identical hash value within √(2^m) = √(2^(m/2)) attempts [28].

The above security requirements are satisfying our proposed algorithm so our proposed algorithm is secure and time efficient because it takes to order one complexity O(1) during all

phases of function processing. The comparative analysis hash algorithms based on basic parameters and NDBCKSHA-384 are shown in Table II.

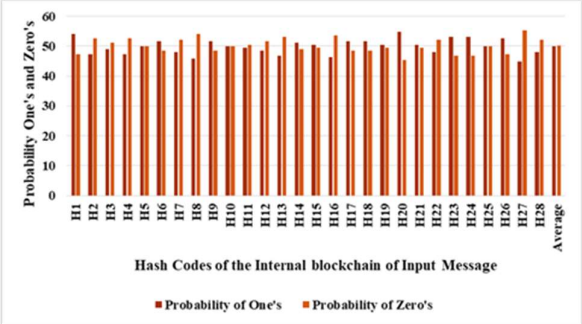
Comparison with Basic Parameters of Hash Algorithms with NDBCKSHA-384

Hash Function	Hash Code in Bits	Input Block Size in Bits	Security Requirement		Hash Function Processing Steps in Single Round	Reference
			Preimage and Second Preimage in Bits	Collision Resistance in Bits		
MD5	128	384	128	64	64	[13, 24, and 29]
SHA-0	160	384	160	80	80	
SHA-1	160	384	160	80	80	
SHA-224	224	224	224	112	64	
SHA-256	256	256	256	128	80	
SHA-384	384	384	384	192	80	
AIVPSHA64	64	2048	64	32	31	[11]
AIVPSHA 256	256	2048	256	128	22	[12]
NDBCKSHA-384	384	4096	384	192	65	NDBCKSHA-384

The experimental results of NDBCKSHA-384 and hashing algorithms are shown in Table I, which we are using for the execution of Python-3.9.5 programing language, on Windows 10, 64-bit Operating system, 4GB RAM platform, Intel® processor. Statistical testing for executed results of NDBCKSHA-384 with input variable length is as follows:

Statistical Testing of NDBCKSHA 384

Frequency (Mono-bits) Test: The focus of the test is the proportion of zeroes and ones for the entire sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence is approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to ½, that is, the number of ones and zeroes in a sequence should be about the same. All subsequent tests depend on the passing of this test [29]. Aims to determine the relationship between zeros and ones in a binary sequence of a certain length. For a truly random binary sequence, the number of zeros and ones is almost the same. The test estimates how close the unit is to 0.5 and the generators of "random" sequences, that is, with a high probability to confirm whether the generated sequence is statistically secure [28].



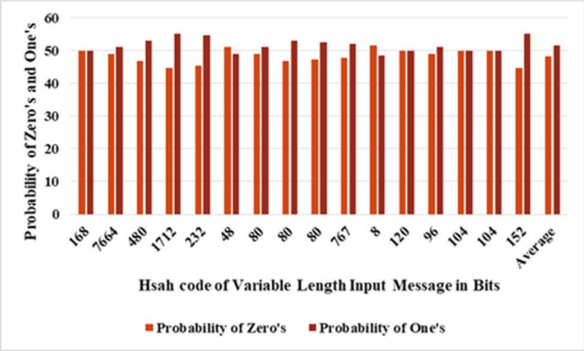
Frequency (Mono-Bit) Test of Internal Blockchain of Input Message

The order of testing of the randomness of the proposed algorithm is based on a frequency (mono-bit) shown in Table III.

Frequency (Mono-Bits) Test of NDBCKSHA 384

Length of Input Message	Hash Code	Total No. 1s in hash code
168	afabef988fea49837c58a6491cefc1a21a47d3 ee00a996401c19172dc7e9e0a8c8ba61922d32 fa2cc2b7afb601f7a5c7	192
7664	9792eb4e401a18cdf6aed8eafd4b83a529bf33 a67a30308ef3d615dd9bcbe77ae6a4932a60e9 c3f98c224ccec04ceb20	196
480	f511e672bd738f9d6f5470348cf8ccf0fbb052 34dc79d0ed4fadaad5a98bb6fd3acd290185dc 8554ead4a1c94be6e633	204
1712	26dacb2b09f7972cff4dd550cea45ceeb8da51 39375b39394eb635977eeaa38f3f472827e7f6 5e64ca93cde8b52693b8	212
232	4fbb762d5fb4988d5cf0367a5132d093f26869 dbec3c952c6c3ff3c704dcd8efcb28ca68b83e efb4cb38ae5befc9815f	210
48	58f69cb0f414f66e8b0e93fe053f2e8014051b 7a2f5b344e25fe69d492bd9d20268cadd48707 29e9c8960653159357ba	188
80	68e6f5073cfcb14f115abaae13f0bc24d48f3d 2952dab720f98daa99902a5407aea017c8cc4f 50db77eb7d0b5c9533a7	196
80	e2d42b90635eb0f3730edfc5bc75125383739c 13d6dd27be4cca6a2dba746166d98cfea0fd55 5bf1ea1346aa5638cb7a	204
80	13a63e7198b014ff8e450715c37d3f61548adb 50ee45f97a897742f08ae6f34b779106896cfd 7e0423d9f4f45d57ebb3	202
767	ca31f9a3c6149bdceb6520d037b450401accfe ed0ad0a5a7bf196677dcf9312b1d356d08566e 373df877f88fa02bf561	200

8	b0fea11112b9e5c47a52c39925a1bc5186fe8d74e8d179c6ed698b8b9d44026fa2d3aefa17535c668ca78c600122e2c9	186
120	2129c0c490cc1d292f45fbe717f681c679b97bbb98e3909d37ea00787c69194a088572603f157fb5674f9727f3cd5e882	192
96	15ccc9b6b9da989513bc45d0eb6f7b1b5f3a9ba9b5987ee983495e203397249d9edf07b0a0d30909f275a55350684fa6	196
104	59cd6b288ea38871c7f9dc57068ad25c8e739179a0edcf954e75d4f36294503458306ff1fae9b20aad0fcda2d413b881	192
104	71e82162ccc170dd2cd74d2c239acdf462c427c6feac3b8d51b3096b3be182d4a5cae99cb3e20d8c07e52fd71f5c578	192
152	e08ebbe118a2f2aa598dcedb32fdb2269c6a471eace440fa95eff713a37aa182f76a94df56f88d3dad7d8eb88fed1f9b	212



Frequency (Mono-Bit) Test of NDBCKSHA 384 with Variable Length Input Message Sensitivity Analysis Test: The sensitivity of the hash function by comparing the effects of seven variants to the input message, which means that a small change in its input message will generate a considerable change in the hash value [29].

For example, we are computing the small change in the input message "abc" and find the change bits compared with the hash value for "abc"

Structure 1

Change (C1): abc

Change (C2): "abc#".

Change (C3): "aec".

Change (C4): "ABC"

Change (C5): "7abc"

Change (C6): "Abc"

Change (C7): "aBc"

Structure 2

Change (C1): efg
Change (C2): “efg#”.
Change (C3): “edg”.
Change (C4): “EFG”
Change (C5): “7efg”
Change (C6): “Efg”
Change (C7): “eFg”

Structure 3
Change (C1): hij
Change (C2): “hij#”.
Change (C3): “hfj”.
Change (C4): “HIJ”
Change (C5): “7hij”
Change (C6): “Hij”
Change (C7): “hIj”

Structure 4
Change (C1): klm
Change (C2): “klm#”.
Change (C3): “kxm”.
Change (C4): “KLM”
Change (C5): “7klm”
Change (C6): “Klm”
Change (C7): “kLm”

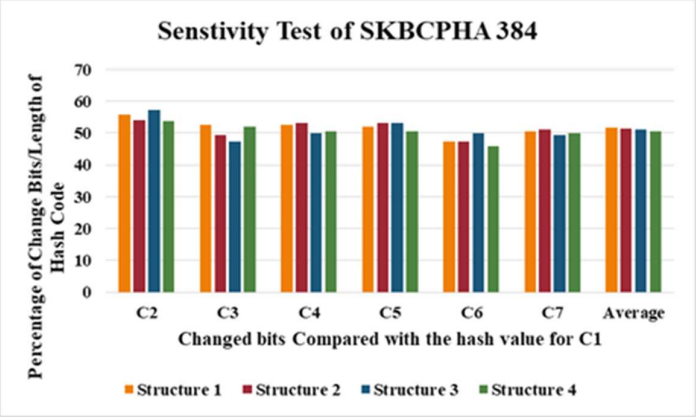
The resulting hash values are listed in hexadecimal format for all cases, followed by the number of changed bits compared with the hash value for C1. The experimental comparative result of statistical testing of the frequency (mono-bit) test and sensitivity test are shown in Table IV.

Statistical Testing of NDBCKSHA 384

Input Message	Hash Code	Change Bit	Total No. of 1’s
Structure 1			
C_1	fca788de40730b709bdb77a20f263103d965659fe987789223728ec6a492a13edb95c761f391bf5e89c15a075388c28b	----------	192
C_2	c36f8e10b0a7af1b1323da777a31ebf95ef619121c459f61a8c3331c66c134e3dbff4f8ede1bc2319a0a90643d15fd34	214	198
C_3	1ffedc12e2fe824c824aa4db73f34fd32c003c12b47b5775bdbaa6adf7b2e75dc96bfc7c0950cf1f74e1a50d20312f7b	202	206

C_4	4c485877ef1acc41d50467842b05f1a819bf306501798dd5980fca89898025a87c0df65056cfbab567ce27f6bb207524	202	182
C_5	3b78e62b70a0e08883a4113a33c8c8b18d9e6f3c0f53a4ebb60896d7b0fa52ec9ad747e4fe4593a1766860d270e96fbb	200	192
C_6	7ff0621a05f38bcde8cb028ad820b4ef86adf4e46f82e80c96240b66ddf536f63fa10a398389b664542922fb4a3e3345	182	188
C_7	ab461b74c47407f026a855185e3edf68a72db5cad66b3102bdc7e94345c74620855db7a8afc96757b31ca07fab83bc72	194	196
Structure 2			
C_1	b14378c9659e3396b02a3f0e26f59e67a74e0f00b498a63ad41d267f823cc7846ee5c247c73a7fe3f4221e16f51dd9eb	-----	198
C_2	2d016607854a97fd38d292db53e2449d20dd738d415a41c95fac8be4543c4e6bf7aecba9eab0028ce7e9d4749e94b734	208	192
C_3	1f893cf7c9f6835ebbfc1935c2c5928900ba8205e76662a3ddf640ccb1b59891886c89cf56d73263678fe06666c655f6	190	194
C_4	8bb4a870caf7f4a7fef52f2802d65ecc67945afa5c66537d6f606230af2e4312c97df37662647a081a2d62e209e40ee7	204	196
C_5	92b409972eca7e686d93508719e34e4f08731bb141c8dfc7c5866cb394fff8e3abfbe3eab22cc4018a8f5103f11c0ed7	204	194
C_6	3214920d201eb32bc33a4a26f1f31b8bf8869e7b329d36a4614ba3dffb5b504c8ad10f1fb72276d929ca66eaecab2825	182	192
C_7	eeb2fb63e1993f160d591db477ed700cd906df558b74efaa4aa841fa6369209a302db28e9b62a 7eaceffe56a1d578690	196	202
Structure 3			
C_1	65417c03eeba844d49cc28ca3e80e0636a89238bd701f6f0f9997fbfca18201f100105ac860183af2cd62c6427336c5f	----------	176
C_2	5a897acd1e6e2026c13485176b153395fd5adf8622c311037228c265006b37cb1c7bcdc32b8bfec03f1de60749ae53e0	220	186
C_3	7deb5411fd95ce517f013e2905564fb324ba8e13c506cb7340fce59a5f38024a9b5a92933b75c5ec9f412e3c1be4351b	182	194

C_4	a1effdaac1d3437c07122db94fd19b6233 5266713fff03b742e4b9f8c79a658dbf91 349d235f8644c2d958a934756f7f	192	206
C_5	cbef9cf57703ab2b0984213058825a75b1 ad6aea3f260d33b253b2d52f2747514bde f616cff33da95215e3445af8569	204	196
C_6	a69616c7ab3a04f03adc5de2e986658f35 41b2f05104666e4ccf781793ef76d3fc3d c8f4f6198a95f13e5cb8bc8c9181	192	196
C_7	2af07f296abd88cdf4be1e212aab942085 d0f3dee8edbf60672c183a2b4dc7014ec9 7565da595ba6160bd710effb3a28	190	196
Structure 4			
C_1	d6f215f53c534e3e5a2ff7a81df1c3fdcb b6b146d8b9c62c97da5fd590a7554dd3ea d333ba3b1fe4fa8fb00e57f4ecb8	----------	218
C_2	119c9222dc87ea55d2d75a6c2df24a3eee acdcdb2d7b21df1c6bf24e5ec65ab95e91 1b5c17b1628be9447a6daa24e7c4	206	202
C_3	eb94464d2681a9ee133202207618b145fb 7b4366e0b26bd6cb5621a24a2bd2509711 714637cde8e140ba33faaaceefea	200	186
C_4	de9dd45c933a890f15e5a38e1e3e8275a3 ecf5bc3047336b2ca61f8afc9453d36462 e2021f651a0f149199beb932d224	194	190
C_5	5ded687b7eca46dbad772736a58aa47f5b e938c0d40a0f4f78311f75676b89e081f9 f6d14b7ff6d3c5949d8ad4922fdf	194	212
C_6	54a5ff3179d3ce83283b92c1eb754e0184 7e203d5ebc56b2228dde65a8e1408d27ce 1e6bca2316de277789f65c5a3df7	176	198
C_7	1812965fb85442bee74d80471bc2973f25 ef6113e7558fbc096f385071f2b2538d22 a3fae663c7edc0535f23fa8d29eb	192	198

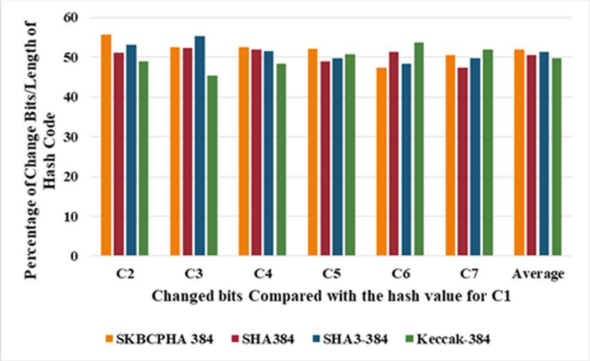


Sensitivity Test of NDBCKSHA 384 with Small Change Structure of Input Message and Compared with Hash Value C1

Sensitivity and Frequency(Mono-Bit) Testing of Hashing Algorithms

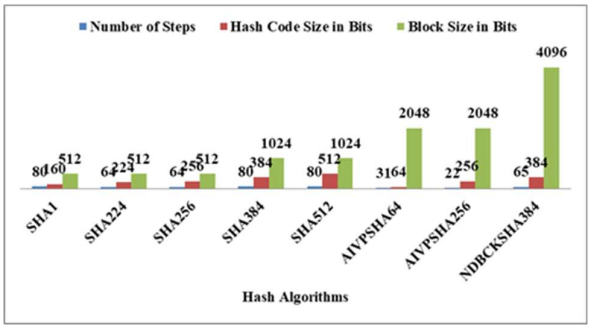
Hash Algorithms	Input Message	Hash Code	Change Bit	Total No. of 1's
Structure 1				
SHA384	C_1	cb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7	----------	185
SHA384	C_2	b267bcec3d57bb9153fd3322817e724c2660a62ebcf936bd58aa4295f369bc90ec66a5ed9f7a755d69d516e9d982657c	196	207
SHA384	C_3	23f61f91d8b921baf8243f39742fc20a6821864e6e7a4fa96e6316a51289ca9a552afa99799b6cbe897c02a5988dd8cb	201	190
SHA384	C_4	1e02dc92a41db610c9bcdc9b5935d1fb9be5639116f6c67e97bc1a3ac649753baba7ba021c813e1fe20c0480213ad371	199	188
SHA384	C_5	2bc7e76188109faae9d3fc5e0172a492f9b584a906aeed0f0f43d4732f0f8201b696484d16f46270a7ad444c64d5efa7	188	189
SHA384	C_6	3903757a5a73c197222ec4fec36eb6b891f2a75779feb8daffcfe70522527282f7792e1aa1745e052ef8ed56b920e49e	197	206
SHA384	C_7	8ee5836ad7940c11570e8124aa732133a649b7c25ea54afa6f944f10b32e5b1c8facce294db237a552981895350e1847	182	181
	Structure 1			
SHA3-384	C_1	ec01498288516fc926459f58e2c6ad8df9b473cb0fc08c2596da7cf0e49be4b298d88cea927ac7f539f1edf228376d25	-----	194
SHA3-384	C_2	91330740291cbd664d5d7b20c192fc29d5e1c8376ffc607e57fd714d23379a9d6eef1340552d445167aaea4dd97f789e	204	200
SHA3-384	C_3	2cfeb1d1c6a3500442452eb35b7c7b155b5e6d4c1297238445	212	192

		0bcf3d6b3fbdcaeb8d138c1cb63e1d5cd18829df285ec0		
SHA3-384	C_4	38078331baaa86dbe9b38224a0780e9661daa35b42066a804efd5215b2487b9728a19ae4940ddbcbda39b697f13ebebb	198	188
SHA3-384	C_5	d86f2cee799f13a7244a1a433d14ea1ed82c77923d0dbe7964fbcea926244fd40d5ed07f515b9c918fee376c4fe3680a	191	201
SHA3-384	C_6	437614b6126e0e8e8963d64cdcc652a8114d0f4b2d376001ae364dac55d8c069ea1433f4a07e9ec61d424112e515ef71	186	178
SHA3-384	C_7	6254cf7e23bf62a2445408e6a127dbe1d611fd8f3dbc935d559d4858e0b4ecc98205ed67f9fa536c62dc3c5f19c86333	191	197
	Structure 1			
Keccak-384	C_1	f7df1165f033337be098e7d288ad6a2f74409d7a60b49c36642218de161b1f99f8c681e4afaf31a34db29fb763e3c28e	----------	198
Keccak-384	C_2	84d9dad3f15ee979c19722e23ce4c205665c05ef89b0eaab533ee1a4ac3446001cfa67a9b7a63818abe448ac90599b35	188	186
Keccak-384	C_3	f964d583ef3d2f99321ca5dbbaf63e7257c2b8fcc1460c34be75bbb4a78b16ea789ca33ba4b25ea4ad38818b38e96df0	174	204
Keccak-384	C_4	a786995442b0677bdcde1f0187c971518a79c65b2726ed9ba0098d6227560c768258db19d9a7e2842d80dc3e50a8a630	186	180
Keccak-384	C_5	2e04a4590bf03e593e81995e975678d4d57cc228b6c72c2763d5359fb054e696e36176a4a9a8b1622586e995663e61c8	195	187
Keccak-384	C_6	4d6c269c94776693071379a9822e11ff9cd6cdaf174963410c4701900bf62d3de6ccda2a21f5f89468eac9cdd8f2fb29	206	192
Keccak-384	C_7	9857b23058ae5d462d5c571faf219795eb6cab8d6d6e77578010c6e679c7962da6bfa7dddc6885f37099998bc3a1b6c4	199	203



Comparative Sensitivity Testing NDBCKSHA-384 with Hashing Algorithm

We are enhancing the performance of the hash function by avoiding the basic parameter additive key constant and one-way key concept of cryptographic hash algorithms. The performance of NDBCKSHA 384 is tested by the statistical test of the hash function. By using it we got good experimental results frequency (mono-bit) test shown in Table I. The probability of one's zero's is near 50 percent by frequency (mono-bit) test shown in Fig. 4 and the variable length random input message is shown in Fig. 5. Experimental results of random variable length input message shown in Table III. The sensitivity test of statistical testing for NDBCKSHA-384 and Hashing Algorithms are shown in Table IV and Table V. The comparative sensitivity test results of NDBCKSHA-384 are shown in Fig. 6. The comparative sensitivity test of NDBCKSHA-384 with hashing algorithms are shown in Fig. 7. Experimental results of NDBCKSHA-384 are executed by python-3.9.5 and existing algorithms are executed by python inbuilt tools.



Comparison with Basic Parameters of Hash Algorithms

The comparative results of the basic parameter of NDBCKSHA-384 with hashing algorithms are shown in Fig. 8.

CONCLUSION

In this research, we implemented NDBCKSHA-384 to produce a 384-bit hash code from the input message. It satisfied the security strength of hash code by statistical test and we are getting the evaluate the performance of a hash function by using the decentralization principle of blockchain technology. We have increased the security strength and complexity of NDBCKSHA-384 by one-time padding bit and circular left shift. The main advantage of NDBCKSHA-384 has mapped the hash code without the need for a cryptography key and it is

very sensitive in case of a small change in input message to produce 384-bit fixed-length hash code with a high probability change of bits therefore it is statistically secure. The main disadvantage of the NDBCKSHA-384 is it used limited logical and arithmetical operations to produce 384-bit hash code.

Conflict of Interest

The authors declare there is no conflict of interest.

Author Contributions

B. R. Ambedkar think of the presented idea and executed the blockchain-based keyless hash function computations. P. K. Bharti and Akhtar Husain verified the comparative results and supervised the findings of this work. All authors discussed the results and contributed to the final manuscript.

References

[1] M. Pilkington, "Blockchain Technology: Principles and Applications." Rochester, NY, Sep. 18, 2015. Accessed: Sep. 27, 2022. [Online]. Available: https://papers.ssrn.com/abstract=2662660

[2] N. El-Meligy, T. Diab, A. Mohra, A. Hassan, and W. Alsobky, "A Novel Dynamic Mathematical Model Applied in Hash Function Based on DNA Algorithm and Chaotic Maps," Mathematics, vol. 10, p. 1333, Apr. 2022, doi: 10.3390/math10081333.

[3] K. Aldriwish, "A double-blockchain architecture for secure storage and transaction on the Internet of Things networks," International Journal of Computer Science and Network Security, vol. 21, no. 6, pp. 119–126, Jun. 2021, doi: 10.22937/IJCSNS.2021.21.6.16.

[4] A. Almutairi, O. Alrumayh, S. Alyami, N. Albagami, and M. Hossein, "A blockchain-enabled secured fault allocation in smart grids based on μPMUs and UT," IET Renewable Power Generation, vol. n/a, no. n/a, doi: 10.1049/rpg2.12332.

[5] J. Polpong and P. Wuttidittachotti, "Authentication and password storing improvement using SXR algorithm with a hash function," IJECE, vol. 10, no. 6, p. 6582, Dec. 2020, doi: 10.11591/ijece.v10i6.pp6582-6591.

[6] A. Visconti and F. Gorla, "Exploiting an HMAC-SHA-1 Optimization to Speed up PBKDF2," IEEE Transactions on Dependable and Secure Computing, vol. 17, no. 4, pp. 775–781, Jul. 2020, doi: 10.1109/TDSC.2018.2878697.

[7] S. Mathew and K. P. Jacob, "Performance Evaluation of Popular Hash Functions," World Academy of Science, Engineering and Technology, pp. 449–452, 2010.

[8] R. Shinde, S. Patil, K. Kotecha, and K. Ruikar, "Blockchain for Securing AI Applications and Open Innovations," Journal of Open Innovation: Technology, Market, and Complexity, vol. 7, no. 3, Art. no. 3, Sep. 2021, doi: 10.3390/joitmc7030189.

[9] L. V. Cherckesova, O. A. Safaryan, N. G. Lyashenko, and D. A. Korochentsev, "Developing a New Collision-Resistant Hashing Algorithm," Mathematics, vol. 10, no. 15, Art. no. 15, Jan. 2022, doi: 10.3390/math10152769.

[10] S. Abbas, M. A. Talib, A. Ahmed, F. Khan, S. Ahmad, and D.-H. Kim, "Blockchain-Based Authentication in Internet of Vehicles: A Survey," Sensors, vol. 21, no. 23, Art. no. 23, Jan. 2021, doi: 10.3390/s21237927.
[11] B. R. Ambedkar, P. K. Bharti, and A. Husain, "Design and Analysis of Hash Algorithm Using Autonomous Initial Value Proposed Secure Hash Algorithm64," in 2021 IEEE 18th India Council International Conference (INDICON), Dec. 2021, pp. 1–6. doi: 10.1109/INDICON52576.2021.9691602.
[12] B. R. Ambedkar, P. K. Bharti, and A. Husain, "Enhancing the Performance of Hash Function Using Autonomous Initial Value Proposed Secure Hash Algorithm 256," in 2022 IEEE 11th International Conference on Communication Systems and Network Technologies (CSNT), Apr. 2022, pp. 560–565. doi: 10.1109/CSNT54456.2022.9787561.
[13] S. William, Cryptography and Network Security - Principles and Practice | Seventh Edition | By Pearson, Seventh edition. Uttar Pradesh, India: Pearson Education, 2017.
[14] X. Zheng, X. Hu, J. Zhang, J. Yang, S. Cai, and X. Xiong, "An Efficient and Low-Power Design of the SM3 Hash Algorithm for IoT," Electronics, vol. 8, no. 9, Art. no. 9, Sep. 2019, doi: 10.3390/electronics8091033.
[15] A. Mohammed Ali and A. Kadhim Farhan, "A Novel Improvement With an Effective Expansion to Enhance the MD5 Hash Function for Verification of a Secure E-Document," IEEE Access, vol. 8, pp. 80290–80304, 2020, doi: 10.1109/ACCESS.2020.2989050.
[16] L. Singh, A. K. Singh, and P. K. Singh, "Secure data hiding techniques: a survey," Multimed Tools Appl, vol. 79, no. 23, pp. 15901–15921, Jun. 2020, doi: 10.1007/s11042-018-6407-5.
[17] Faculty of Computer Science and Information Technology, Riga Technical University Riga, Latvia, V. Stepanova, and I. Eriņš, "The Blockchain-Based Model for Professional Growth Data Processing," JAIT, vol. 12, no. 4, 2021, doi: 10.12720/jait.12.4.319-326.
[18] F. E. De Guzman, B. D. Gerardo, and R. P. Medina, "Implementation of Enhanced Secure Hash Algorithm Towards a Secured Web Portal," in 2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS), Feb. 2019, pp. 189–192. doi: 10.1109/CCOMS.2019.8821763.
[19] A. A. Yavuz and M. O. Ozmen, "Ultra Lightweight Multiple-time Digital Signature for the Internet of Things Devices," IEEE Transactions on Services Computing, pp. 1–1, 2019, doi: 10.1109/TSC.2019.2928303.
[20] M. Samiullah et al., "An Image Encryption Scheme Based on DNA Computing and Multiple Chaotic Systems," IEEE Access, vol. 8, pp. 25650–25663, 2020, doi: 10.1109/ACCESS.2020.2970981.
[21] X. Fei, K. Li, W. Yang, and K. Li, "A secure and efficient file protecting system based on SHA3 and parallel AES," Parallel Computing, vol. 52, pp. 106–132, Feb. 2016, doi: 10.1016/j.parco.2016.01.001.
[22] K. Ideguchi, T. Owada, and H. Yoshida, "A Study on RAM Requirements of Various SHA-3 Candidates on Low-cost 8-bit CPUs," 260, 2009. Accessed: Nov. 23, 2021. [Online]. Available: http://eprint.iacr.org/2009/260

[23] V. Melnyk and A. Kit, “Basic operations of modern hashing algorithms,” undefined, 2013, Accessed: Dec. 17, 2021. [Online]. Available: https://www.semanticscholar.org/paper/Basic-operations-of-modern-hashing-algorithms-Melnyk-Kit/f423b1e8a5365b713e14a2d7d95b8c2a94c9aebf

[24] B. Madhuravani and D. S. R. Murthy, “Cryptographic hash functions: SHA family,” International Journal of Innovative Technology and Exploring Engineering (IJITEE), vol. 2, no. 4, pp. 326–329, 2013.

[25] Y. Li and X. Li, “Chaotic hash function based on circular shifts with variable parameters,” Chaos, Solitons & Fractals, vol. 91, pp. 639–648, Oct. 2016, doi: 10.1016/j.chaos.2016.08.014.

[26] A. A. Abdugafforovich, G. S. Rajaboevich, and A. Z. Ildarovna, “Development a Model of a Network Attack Detection in Information and Communication Systems,” JAIT, vol. 13, no. 4, 2022, doi: 10.12720/jait.13.4.312-319.

[27] S. Chang et al., “Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition,” National Institute of Standards and Technology, Gaithersburg, MD, NIST IR 7896, Nov. 2012. doi: 10.6028/NIST.IR.7896.

[28] A. Kuznetsov, M. Lutsenko, K. Kuznetsova, O. Martyniuk, V. Babenko, and I. Perevozova, “Statistical Testing of Blockchain Hash Algorithms,” p. 13.

[29] A. Rukhin et al., “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” NIST Special Publication 800-22 (revised May 15.” 2002.