VORSHA: A Variable-sized, One-way and Randomized Secure Hash Algorithm

Ripon Patgiri[†] and Laiphrakpam Dolendro Singh[‡] and Dalton Meitei Thounaojam[◇]

National Institute of Technology Silchar, Assam, India
[†]ripon@cse.nits.ac.in, [‡]ldsingh@cse.nits.ac.in,
[◇]ldsingh@cse.nits.ac.in

Abstract. In this paper, we propose a variable-sized, one-way, and randomized secure hash algorithm, VORSHA for short. We present six variants of VORSHA, which are able to generate a randomized secure hash value. VORSHA is the first secure hash algorithm to randomize the secure hash value fully. The key embodiment of our proposed algorithm is to generate a pool of pseudo-random bits using the primary hash functions and selects a few bits from the pool of bits to form the final randomized secure hash value. Each hash value of the primary hash function produces a single bit (either 0 or 1) for the pool of pseudo-random bits. Thus, VORSHA randomized the generated bit string to produce the secure hash value, and we term it as a randomized secure hash value. Moreover, the randomized secure hash value is tested using NIST-SP 800-22 statistical test suite, and the generated randomized secure hash value of VORSHA has passed all 15 statistical tests of NIST-SP 800-22. It proves that the VORSHA is able to generate a highly unpredictable yet consistent secure hash value. Moreover, VORSHA features a memory-hardness property to restrict a high degree of parallelism, which features a tiny memory footprint for legal users but massive memory requirements for adversaries. Furthermore, we demonstrate how to prevent Rainbow Table as a Service (RTaaS) attack using VORSHA. The source code is available at https://github.com/patgiri/VORSHA.

Keywords: Secure Hash Algorithm· Variable hash function· One-way hash function· Password Hashing· Random number generator· Cryptography· Security.

1 Introduction

A secure hash algorithm (SHA) is a widely used hash function in security and cryptography. Therefore, NIST held a competition on permutation-based secure hash algorithms to select an algorithm for SHA3, and the winner was Keccak [7]. Keccak is a permutation-based secure hash function that implements the sponge function based on absorb and squeeze functions [5,4,6]. Therefore, the recent development suggests that NIST is keen on developing a permutation-based secure hash algorithm. Keccak is a good hash algorithm that tries to

randomize the bits; however, there is a lack of proof of the randomness of the generated bit strings. Alternatively, Keccak is not applied to generate a random number of variable sizes, and it is not developed for a random number generation purpose.

1.1 Motivation

The permutation-based secure hash algorithm is predominantly used for string hashing, where the input is mixed with some predefined constant by addition, rotation, XOR, shift, modulus, etc., operators. Moreover, state-of-the-art secure hash algorithms require bit padding, and the bits are public. Additionally, the conventional secure hash algorithm has a fixed-sized hash value which becomes easy to attack by adversaries. There are many applications where a fix-sized hash value creates an issue due to the presence of adversaries, for instance, password hashing or digital signature. It becomes difficult for adversaries when the hash value size is variable, and the adversary does not have any clue about the size. Therefore, rainbow table attacks or other similar kinds of attacks become computationally infeasible when the hash value size is variable. Hence, a variable-sized hash value is highly demanded to prevent diverse attacks, but it still needs to incorporate into diverse applications, for instance, HMAC.

Security is a paramount feature for a secure hash algorithm, but it should not be the performance. A high-performance secure hash algorithm has the disadvantage of being fast. The state-of-the-art secure hash algorithms should make it infeasible to convert the secure hash value generation in parallel. For illustration, we assume that a set of secure hash values can be produced in parallel using a secure hash algorithm, say the algorithm is $\mathcal{Z}$; and therefore, the $\mathcal{Z}$ features extremely high performance due to high parallelism. The adversary can also take the same advantage to evade the security of the secure hash algorithm $\mathcal{Z}$. Thus, the high performance becomes disadvantageous for secure hash algorithms. Therefore, the memory-hard hash algorithms hash introduced by C. Percival [11]. Therefore, the parallelism in a secure hash algorithm is made infeasible in generating a set of secure hash values in parallel.

Most importantly, research communities believe that the secure hash algorithm and random number generator are different algorithms, and the belief continues. Partially, it is true that a secure hash value and a true-random number have a difference, but a secure hash algorithm can be modified to generate a true-random number. Each bit of a random number is influenced by external events, which is easily achieved by a secure hash algorithm. Interestingly, a true-random number generator requires a higher speed of generation of a bit string, whereas a secure hash algorithm does not require a higher speed of generation of a bit string. Instead, a secure hash algorithm restricts to a certain degree of parallelism. On the contrary, a secure hash value and a pseudo-random number do not have any significant differences. The pseudo-random numbers are completely determined by the initial input. Alternatively, a secure hash algorithm can be used to generate a pseudo-random number or vice-versa. The generated hash value should be highly random and unpredictable. Also, the generated bit

string should contain no patterns to discover by the adversaries. Therefore, a secure hash value should be highly random yet consistent. Alternatively, the secure hash value should convert a low entropy input string into high entropy hash value. Also, the output of the hash algorithm should be tested using a statistical test suite, namely, NIST SP 800-22 or any other similar tools. NIST SP 800-22 is used to test the randomness of the generated bit string. The secure hash algorithm $\mathcal{H}$ or random number generator $\mathcal{G}$ generates output $\zeta = \{0,1\}^{\eta}$ where ζ is either a secure hash value or a random number. Therefore, we argue that there is no significant difference between a pseudo-random number generator and a secure hash algorithm. Therefore, it motivates us to design a secure hash algorithm from a different perspective than a conventional way.

Notably, SHAKE128 and SHAKE256 feature variable-sized hash functions [1]; however, the hash value of a string in a lower bit size is the prefix of a higher bit size. For instance, if the hash value size of a 128-bit is h_{128} for a given input string, ω, then the hash value of 256-bit size for the ω is $h_{256} = h_{128}hex_{128}$ where hex_{128} is the remaining hash value. Alternatively, the bit size of the hash value does not influence the hash value. Theoretically, it can be derived from the lower bit-sized hash value for the higher bit-sized hash value. For instance, $h_{256} = h_{64}h_{64}hex_{128} = h_{32}h_{32}h_{32}h_{32}hex_{128}$. Therefore, it demands a bit-size dependant hash value.

Furthermore, a rainbow table attack is an attack performed by storing a set of precomputed hash values for corresponding keys at the server to evade the security of the conventional secure hash algorithm. It is a serious concern for a secure hash algorithm. It is almost impossible to prevent from rainbow attack. For instance, the rainbow attack stores all possible hash values of SHA3-256 to carry the attacks. Notably, it is impossible to store all hash values of SHA3-256, but the important hash values are stored in the hashtable. The attacker queries for a key for the corresponding hash value in the database. The database can instantly respond to the query with a key. However, the adversary needs to store $\frac{1}{2^{128}}$ hash values for SHA-256 to perform the rainbow table attack. The only way to prevent such kind of attack is by increasing the hash value size from 256 to 512 bits or beyond. Another way to make the rainbow table attack computationally infeasible is using a variable-sized hash value. For instance, if the variable-sized hash value ranges [256,1024), then the adversary needs to store all the hash values from 256-bit to 1024-bit (the hash value size can range between 256-bit and 1024-bit). It is computationally infeasible to store all such variants of hash values on a server. Moreover, a key can have $(1024 - 256) = 768$ correct hash values. However, the conventional secure hash algorithms are not designed for variable-sized hash values. Even NIST restricts the hash value size to 224, 256, 384, and 512 bits. Therefore, the secure hash algorithm is confined within fixed bit-sized hash values. It has significant drawbacks due to the presence of an adversary with more advanced computational resources. It motivates us to design a variable-sized secure hash algorithm to withstand such kinds of attacks.

Recent developments suggest that memory-hard hash functions are used to defeat the parallelism of adversaries [11]. The memory-hard hash algorithm de-

fines massive memory requirements for the adversaries to break the security, but it requires a tiny memory footprint for the legal users. It is often used in password hashing to secure the stored password [9,8]. Therefore, it motivates us to design a general-purpose secure hash algorithm that exhibits the property of memory hardness.

1.2 Key contributions

The key objectives of our proposed algorithm are outlined below-

- To prove that there is no significant difference between a secure hash algorithm and a random number generator.
- To develop a secure hash algorithm that produces a randomized secure hash value based on an input string.
- To develop a memory-hard secure hash algorithm.
- To compare our proposed algorithm with Keccak, and random oracle.
- To demonstrate how computationally infeasible to break our proposed algorithm using the brute-force method for a large-sized hash value.

To develop a randomized secure hash algorithm, we propose a variable-sized, one-way, and randomized secure hash algorithm to address our objectives; we term it VORSHA for short. The key embodiment of the VORSHA algorithm is to generate a η-bit secure hash value or a η-bit pseudo-random number using the primary hash function (primary hash function is defined in Definition 1). The primary hash function can be either a secure or non-secure hash function, but it should produce at least a $\beta \geq 32$-bit hash value. Also, the primary hash function should support a seed value in generating a hash value. VORSHA generates a set of pseudo-random bits based on an input string using the primary hash function and selects η bits from the generated random bits to form a η-bit randomized secure hash value or pseudo-random number. Therefore, we summarize our key contributions as follows-

- Our proposed algorithm, VORSHA, is a variable-sized, one-way, and randomized secure hash algorithm that generates either a secure hash value or a random number. We term the randomized secure hash algorithm (see the Definition 4). VORSHA is a randomized secure hash algorithm that produces a randomized secure hash value with variable bit sizes (Definition 6), and it is truly a one-way (Definition 5). VORSHA significantly differs from Merkle-Damgård construction mechanism. Best of our knowledge, VORSHA is the first of its kind.
- Furthermore, we derive three variants of VORSHA based on the characteristics, namely, one-dimensional VORSHA (VORSHA-1D), two-dimensional VORSHA (VORSHA-2D), and three-dimensional VORSHA (VORSHA-3D). VORSHA-1D is optimized for high performance, while VORSHA-3D is optimized for strong security requirements. VORSHA-2D is medium performance and medium security requirements. VORSHA-1D is faster than VORSHA-2D and VORSHA-3D, whereas VORSHA-3D is more secure than VORSHA-1D and VORSHA-2D. Similarly, we categorize VORSHA into two different

categories based on the input string, that is, short (S) and long (L) input string. Therefore, it becomes six total variants of VORSHA; namely, VORSHA-1D for short input string is denoted as VORSHA-1D-S, VORSHA-1D for long input string is denoted as VORSHA-1D-L. Thus, we denote other variants as VORSHA-2D-S (medium and short), VORSHA-3D-S (strong and short), VORSHA-2D-L (medium and long), and VORSHA-3D-L (strong and long).

– VORSHA is strongly dependent on a primary hash function. The primary hash function requires an input string and a seed value. The primary hash function is used to generate a set of pseudo-random bits. From the generated pool of random bits, VORSHA selects η-bit to form the final randomized secure hash value.
– VORSHA can generate a secure hash value of any size. Therefore, it can be used to conceal the size of bits from adversaries in string hashing. The final randomized secure hash value can be either a fix-sized or variable-sized bit string. Moreover, VORSHA has the properties of a memory-hard secure hash algorithm.

1.3 Our results

Table 1. Comparison among the random oracle, SHA3, SHAKE, and VORSHA.

Algorithm	**Output**	**Collision**	**Preimage**	**Second Preimage**
Random oracle	η	$2^{\eta/2}$	2^{η}	2^{η}
SHA3-224	224	2^{112}	2^{224}	2^{224}
SHA3-256	256	2^{128}	2^{256}	2^{256}
SHA3-384	384	2^{192}	2^{384}	2^{384}
SHA3-512	512	2^{256}	2^{512}	2^{512}
SHAKE128	η	$2^{min(\eta/2,128)}$	$\geq 2^{min(\eta,128)}$	$2^{min(\eta,128)}$
SHAKE256	η	$2^{min(\eta/2,256)}$	$\geq 2^{min(\eta,256)}$	$2^{min(\eta,256)}$
VORSHA-1D-S	η	$2^{\eta/2}$	2^{η}	2^{η}
VORSHA-2D-S	η	$2^{\eta/2}$	2^{η}	2^{η}
VORSHA-3D-S	η	$2^{\eta/2}$	2^{η}	2^{η}
VORSHA-1D-L	η	$2^{\eta/2}$	2^{η}	2^{η}
VORSHA-2D-L	η	$2^{\eta/2}$	2^{η}	2^{η}
VORSHA-3D-L	η	$2^{\eta/2}$	2^{η}	2^{η}
VORSHA	$\eta \in [\mu, \lambda)$	$\sum_{\eta=\mu}^{\lambda} 2^{(\eta+1)/2}$	$\sum_{\eta=\mu}^{\lambda} 2^{\eta}$	$\sum_{\eta=\mu}^{\lambda} 2^{\eta}$

Security strength Table 1 compares the collision attacks, preimage attacks, and second preimage attacks of the random oracle, SHA3, SHAKE, and VORSHA. The security strengths of VORSHA-1D-S are $2^{\frac{\eta}{2}}$, 2^{η}, and 2^{η} for collision attacks, preimage attacks, and second preimage attacks, respectively, if and

only if the size of the hash value is public and it is fixed to η. The security strengths for the other variants of VORSHA are the same as the given condition that η is public and fixed. Otherwise, the security strengths for VORSHA are $\sum_{\eta=\mu}^{\lambda} 2^{(\eta+1)/2}$, $\sum_{\eta=\mu}^{\lambda} 2^{\eta}$, and $\sum_{\eta=\mu}^{\lambda} 2^{\eta)}$ for collision attacks, preimage attacks, and second preimage attacks, respectively.

Brute-force attack The probability of reproducing the correct secure hash value using VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S are $\frac{1}{2^{(\tau+2\eta)\beta}}$, $\frac{1}{2^{\beta(2\tau+XY(\tau+1)+2\eta(\tau+1))}}$, and $\frac{1}{2^{\beta(3\tau+XYZ(\tau+1)+3\eta(\tau+1))}}$, respectively, where $8 \leq \tau \leq 64$.

Memory requirement The adversary requires $(\delta+1)\mu + \frac{\delta(\delta+1)}{2}$ memory to store a secure hash value because the size of the randomized secure hash value is secret, which is $\eta \in [\mu, \mu+\delta)$.

Lower-bound memory The lower-bound memory of VORSHA-2D-S, and VORSHA-3D-S are $\Omega(XY+\eta)$, and $\Omega(XYZ+\eta)$, respectively where X, Y, and Z are secret and η is public and fixed.

Lower-bound memory in variability The lower-bound memory of VORSHA-2D-S, and VORSHA-3D-S are $\Omega(XY + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$, and $\Omega(XYZ + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$, respectively, where X, Y, Z and η are secret and $\eta \in [\mu, \mu+\delta)$.

Space Complexity The space complexity for VORSHA-3D-S and VORSHA-2D-S is $O(^{2(\varphi-16)}C_2XY + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$, and $O(^{3(\varphi-16)}C_3XYZ + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$ for the adversary in a single system in parallel, respectively, where $\eta \in [\mu, \mu+\delta)$ and secret.

Randomness testing We prove the randomness of the generated randomized hash value using the NIST SP 800-22 statistical test suite. The generated randomized secure hash value of VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S pass all 15 test cases of NIST SP 800-22.

1.4 Organization

Our paper is organized as follows- Section 2 highlights the preliminaries with definitions and important notations used throughout the paper. Section 3 demonstrates the proposed system. Section 4 demonstrates the variability in secure hash functions. Section 5 exhibits the randomness of the generated bit string using VORSHA. Section 6 analyzes the diverse attacks on VORSHA.

2 Preliminaries

Definition 1. *The primary hash function, $\mathcal{PH}()$ or* PrimaryHash( ), *is a string hash function, either a secure or non-secure variant, which produces a hash value, h_v, of bit size β where $h_v = \{0,1\}^{\beta}$. The key requirement of the primary hash function is a seed value for hash value generation. A primary hash function $\mathcal{PH}()$ converts the input string into the hash value of β bit size, i.e., $\mathcal{PH} : (\omega, \mathcal{S}) \rightarrow h_v$ where $\omega = \{0,1\}^*$ is input string, $\mathcal{S} = \{0,1\}^{32}$ is seed value and h_v is the generated hash value by the primary hash function.*

Definition 2. *A secure hash algorithm $\mathcal{H}()$ is a function that takes an input string ($\omega = \{0,1\}^*$) of arbitrary size and produces a secure hash value $\zeta = \{0,1\}^{\eta}$, i.e., $\mathcal{H} : \omega \rightarrow \zeta$. The secure hash value can be regenerated consistently for a given input.*

Definition 3. *A random number generator $\mathcal{G}$ generates a random number, i.e., $\mathcal{G} : \omega \rightarrow \zeta$ where $\omega = \{0,1\}^*$ and $\zeta = \{0,1\}^{\eta}$. In a true-random number generation, the input string ω is not directly given by the users, and therefore, the desired output cannot be regenerated. Moreover, the input string is altered in the generation of each bit of the random number generation. For instance, the input string can be CPU clock, mouse movement, light, or any other events in each iteration. On the contrary, the input string is fixed in a pseudo-random number generator. The output of pseudo-random number generation is dependent on the initial input string, and the randomness is measured in terms of initial input.*

Definition 4. *A randomized secure hash value, $\zeta = \{0,1\}^{\eta}$, is an output of η-bit secure hash algorithm with the properties of a secure hash value (Definition 2) and a pseudo-random number (Definition 3).*

Definition 1 defines the primary hash function. The primary hash function can be either a secure hash function or a non-secure hash function, but it should support a seed value. We can use Murmur2, Murmur3, XXHash, FastHash, SuperFastHash, MD5, SHA1, SHA2, or any other algorithm, but the primary hash function must support a seed value as an input. The primary hash function should not necessarily be a secure hash function. However, the secure hash algorithm is a hash function that converts a low-entropy string into a high-entropy bit string as defined in Definition 2. Similarly, a random number generator can produce a high-entropy bit string as defined in Definition 3. The random numbers are generated based on the unpredictable input string; for instance, noises. However, the pseudo-random numbers are generated using a single input string. The randomness of a pseudo-random number is measured based on the initial input string. Similar to the pseudo-random number generator, a secure hash value can be consistently regenerated for a given input, and it is computationally infeasible to reproduce the input string from a hash value. The hash value exhibits the properties of a random number in terms of the initial input string, and it should be highly unpredictable and random. Moreover, the produced output should not have any patterns to discover by the adversaries. Thus, the randomized secure

Table 2. Important notations and symbols used throughout the paper.

Notation	Description
$\mathcal{PH}()$ or PRIMARYHASH( )	Primary hash function.
h_v	Primary hash value generated by the primary hash function.
β	The bit size of hash value h_v.
ω	Input string of arbitrary size.
$\mathscr{L}$	Length of a given string.
$\wedge$	Bit-wise AND operator.
%	Modulus operator.
$\mathcal{S}$	Seed value of size ≥ 32 bits. Initially, it is assigned to a constant. Later, it is altered to the primary hash value in the subsequent steps. Therefore, there is no difference between h_v and $\mathcal{S}$.
$\mathcal{H}()$ or $\mathcal{H}$	A secure hash algorithm.
f	Any hash function, either a secure or non-secure hash function.
ζ	It is a secure hash value or a randomized secure hash value that is generated by the secure hash algorithm or VORSHA.
η	Bit size of the secure hash value ζ.
μ	The minimum size of the secure hash value ζ.
δ	Displacement to calculate the bit size of ζ.
λ	The maximum limit of the secure hash value size ζ where $\lambda = \mu + \delta$.
$\mathcal{SV}()$	It is short form of GETSEEDVALUE( ).
τ	The number of iteration to compute seed value $\mathcal{S}$.
$\mathcal{V}$	A vector to store pseudo-random 0s and 1s.
X, Y, Z	The dimensions of $\mathcal{V}$.
θ	An extractor for dimension calculation from seed value $\mathcal{S}$.
$VORSHA - [dimension] - [short/long]$	VORSHA-[1D]-[Short], or VORSHA-1D-S, i.e., 1D-S- one-dimensional and short input string. Similarly, VORSHA-[1D]-[Long], i.e., VORSHA-1D-L- one-dimensional and long input string. Thus, 2D-S- two-dimensional and short input string, 2D-L- two-dimensional and long input string, 3D-S- three-dimensional and short input string, 3D-L- three-dimensional and long input string.

hash value is consistent, unpredictable, and random. The algorithm that produces a randomized hash value is called a randomized secure hash algorithm which is defined in Definition 4. There is no difference between a pseudo-random number and a hash value.

Definition 5. *A hash function f is said to be strictly a one-way function if the function $f : x \rightarrow y$ but there is no way to map x from y for any function, either the given function f or another function f'.*

Definition 6. *Let h_v be the non-secure hash value of the primary hash function, μ be the minimum bit size, and $\lambda = \mu + \delta$ be the maximum bit size where offset $\delta = h_v\%\delta$. A variable-sized hash function produces a secure hash value of bit size η, and the η can be within the range $[\mu, \lambda)$, where the η is a secret integer.*

Lemma 1. *The probability of generating the correct η-bit secure hash value is $\frac{1}{2^\eta}$ without knowing the input string using a brute-force method.*

Proof. Let us assume that an adversary uses the same method or other methods to generate hash value and is able to generate the η bits secure hash value correctly. The probability of correctly generating a bit is $\frac{1}{2}$ without knowing the input string. Adversaries may use brute force or other methods. It requires a probability of $\frac{1}{2^2}$ to generate two bits secure hash value correctly. Similarly, it requires a probability of $\frac{1}{2^3}$ to generate three bits secure hash value correctly. Therefore, the total probability becomes $\frac{1}{2^\eta}$ if the adversary does not know the input string. This probability is common for all kinds of secure hash values for η-bit without knowing the input string. Even it is true for random oracles too. Moreover, it is also true for VORSHA since it produces a η-bit secure hash value.

Lemma 1 shows that the probability of computing the correct hash value without the knowing input string is $\frac{1}{2^\eta}$, and it applies to all kinds of good secure hash values. Similarly, the probability of collision in secure hash value is $\frac{1}{2^{\eta/2}}$, and it also applies to all, including the random oracle. Lemma 1 shows the attack on the output bit string; however, we need to analyze the security of the algorithms and their possibilities.

2.1 Rainbow table as a service

Rainbow table as a service (RTaaS) is an assumption and black-box that it provides a lookup service to discover a set of corresponding keys for a particular hash value, i.e., $h \rightarrow x$ and $x = \{x_1,\ x_2,\ x_3, \ldots\}$ where x is a set of keys (one or more than one string) corresponding to a particular hash value h. A single hash value can have many corresponding keys due to collision. Collision is unavoidable due to the limited size of hash bits.

RTaaS is a powerful attacking service that can easily break the security of a secure hash algorithm. Therefore, RTaaS is a set of servers to store all hash values and the corresponding keys. RTaaS requires a massive space to accommodate all possible hash values and their corresponding keys. We know that it is infeasible to store all the values of η-bit hash values in real life where $\eta \geq 256$. Alternatively, it is not possible to store all 2^η hash values, but we assume that RTaaS stores all possible hash values and the corresponding keys for comparison. Notably, RTaaS can respond to a query in $O(1)$ time complexity in addition to the network latency. The RTaaS need to store more than $2^{\frac{\eta}{2}}$ secure hash values to evade the

random oracle model completely. Therefore, we need to prove the difficulties of RTaaS in storing variable-sized hash values and the hash values of the random oracle.

3 VORSHA- The Proposed Algorithm

We propose a new variant of the secure hash algorithm, called a variable-sized, one-way, and randomized secure hash algorithm (VORSHA). The VORSHA algorithm is based on the random number generator to produce a randomized hash value. Alternatively, it can be used to generate a random number or vice-versa. The algorithm generates random bits based on input string (ω). It also takes a seed value ($\mathcal{S}$) as another input. Initially, the seed value can be kept public or private depending on the design choice, but it is converted into a private value in the subsequent steps. Furthermore, it takes desired bit size η for the output ζ as input to produce a randomized secure hash value where $\zeta = \{0,1\}^{\eta}$ is the randomized secure hash value generated by VORSHA. To generate the randomized secure hash value, our proposed algorithm relies on the primary hash functions PrimaryHash( ) or $\mathcal{PH}()$. Alternatively, we devise a new hash algorithm by leveraging the properties of the primary hash functions. The primary hash function produces a hash value $h_v = \{0,1\}^{\beta}$ of β-bit, and we select one bit from the generated hash value h_v in each iteration to produce a randomized secure hash value. Notably, we can use $\beta \geq 32$-bit sized hash functions for the primary hash function.

Our proposed algorithm is categorized into two key categories based on the input string, particularly the short input string (S- short) and long input string (L- long). Moreover, both have been categorized into three categories, namely, 1D, 2D, and 3D VORSHA. We denote 1D VORSHA as VORSHA-1D, 2D VORSHA as VORSHA-2D, and 3D VORSHA as VORSHA-3D. Therefore, we have six variants of VORSHA, namely, VORSHA-1D-S (1D and short), VORSHA-2D (2D and short), VORSHA-3D (3D and short), VORSHA-1D-L (1D and long), VORSHA-2D-L (2D and long), and VORSHA-3D-L (3D and long). VORSHA-1D-S is designed to produce fast hash values for short input strings without sacrificing security; however, it is not a purely randomized secure hash algorithm. In addition, VORSHA-1D-S produces the output faster than the medium and strong. VORSHA-2D-S and VORSHA-3D-S are slower than VORSHA-1D-S, and both can produce fully randomized secure values, but VORSHA-1D-S is not a randomized secure hash algorithm. VORSHA-3D-S is designed for tight security than the rest but slower. Similarly, VORSHA-1D-L is faster than VORSHA-2D-L, and VORSHA-3D-L. For a long input string, VOSRHA-1DL, VORSHA-2D-L, and VORSHA-3D-L use VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S, respectively.

3.1 One-dimensional VORSHA for short input string

Short input strings are instrumental in hashing for diverse security systems, for instance, password hashing. A 16 characters long input string is considered as a short input. More than 16 characters input string is considered as a long input.

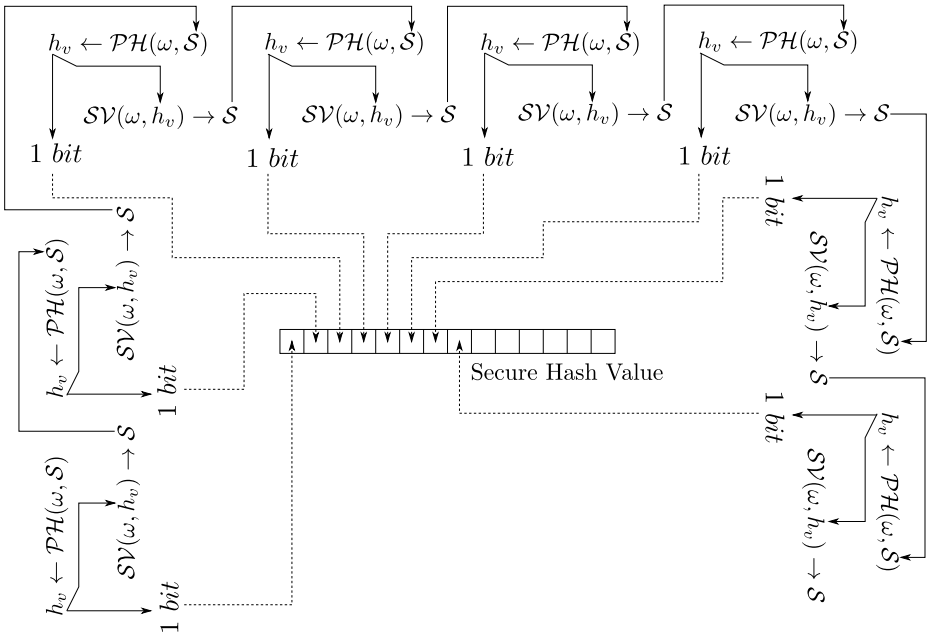

Fig. 1. A simple architecture of VORSHA-1D-S for generating 8-bit secure hash value as an example. Here, h_v denotes a hash value of the primary hash function, which is β bit size, $\mathcal{PH}()$ denotes a primary hash function, ω denotes an input string which is fixed, $\mathcal{SV}()$ denotes computation of a seed value, and $\mathcal{S}$ denotes a seed value which altered in each iteration.

VORSHA-1D for short input string Figure 1 demonstrates a simple architecture of the VORSHA-1D-S. We denote the input string as ω, seed value as $\mathcal{S}$, primary hash function as $\mathcal{PH}()$ or PrimaryHash( ), and the hash value of the primary hash function is denoted as h_v. Figure 1 demonstrates the invocation of a primary hash function and a seed value computation using getSeedValue( ) or $\mathcal{SV}()$ function to generate a single bit for the randomized secure hash value. The input string is kept fixed, and the seed value $\mathcal{S}$ is altered in every bit generation. The figure demonstrates the 8-bit secure hash value generation, for example. The length can be defined by the users as per their requirements. We do not restrict the size of the final secure hash value; for instance, it can be 1024 bits or any size. On the contrary, conventional secure hash algorithms restrict its

output size to a certain bit size; for instance, SHA512 produces a 512-bit secure hash value.

Algorithm 1 Computing seed value for utilization in the hash function.

```
1: procedure getSeedValue(ω, 𝓛, 𝒮)
2:     for i : 1 to τ do                          ▷ τ is a constant and 8 ≤ τ ≤ 64.
3:         𝒮 = PrimaryHash(ω, 𝓛, 𝒮)
4:     end for
5:     return 𝒮
6: end procedure
```

In VORSHA, we use a function to generate a seed value, which is presented in Algorithm 1. Algorithm 1 alters the publicly available seed value to private seed value using the $8 \leq \tau \leq 64$ primary hash functions. It protects the publicly available seed value and converts it into a private seed value with the help of an input string. The key idea of Algorithm 1 is to protect seed value from easy computation. However, our assumption is that the primary hash function is a non-secure hash function; therefore, a seed value can easily be computed with a probability of $\frac{1}{2^{\beta}}$, but it requires many seed values, which makes it infeasible to compute all the values.

Algorithm 2 VORSHA-1D-S for short input string and fast processing.

```
1: procedure genVORSHA-1D-S(ω, 𝒮, η)
2:     𝓛 = stringLength(ω)
3:     𝒮 = getSeedValue(ω, 𝓛, 𝒮)
4:     𝒮 = 𝒮 ⊕ η
5:     for i : 1 to η do
6:         h_v = PrimaryHash(ω, 𝓛, 𝒮)
7:         ρ = h_v % ϱ                            ▷ The ϱ = (β − c) is a prime number.
8:         bit = (h_v ∧ (1 << ρ)) >> ρ
9:         hash_bits[i] = bit
10:        𝒮 = h_v
11:        𝒮 = PrimaryHash(ω, 𝓛, 𝒮)
12:    end for
13:    ζ = convertIntoHex(hash_bits, η)
14:    return ζ
15: end procedure
```

Algorithm 2 presents the VORSHA-1D-S for high performance. Therefore, it uses an extra space of O(1), excluding the space required to write the hash output in ζ. Initially, the seed value is a public constant; therefore, it is converted into a private constant by repeatedly computing the seed value $8 \leq \tau \leq 64$ times, i.e., $\mathcal{S} = \textsc{PrimaryHash}(\omega, \mathscr{L}, \mathcal{S})$. Algorithm 2 calculates a bit position as $\rho = h_v \% \varrho$

where $\varrho = (\beta - c)$ is a prime number. For instance, if a primary hash function produces a 32-bit hash value, then $c = 1$, and $\varrho = \beta - c = 32 - 1 = 31$, where the number, 31, is a prime number. Thus, it becomes $\rho = h_v\%31$, giving a single-bit location of h_v. The ρ^{th} bit of h_v is extracted to form the secure hash value, and the bit is either 0 or 1. We consider 61 for 64-bit and 127 for 128-bit as a value of $\varrho = (\beta - c)$.

The key embodiment of VORSHA-1D-S is to generate a set of bits for the final secure hash value using the existing primary hash functions, for instance, the murmur2 hash function. Firstly, the primary hash function generates a non-secure β-bit hash value. VORSHA-1D-S extracts a single bit from the β-bit hash value, which is generated by the primary hash function. Alternatively, a β-bit primary hash function's output contributes a single bit in the final secure hash value of VORSHA-1D-S. The second primary hash function is used to produce a new seed value too. The generated new seed value is used to compute the next hash value h_v in the next step. The procedure is repeated for η times to produce η-bit secure hash value. Finally, the generated hash bits are converted into hexadecimal code, and write the hexadecimal code in ζ. However, the VORHSA-1DS is not a purely randomized secure hash algorithm to generate a randomized secure hash value. Also, VORSHA-1D-S does not exhibit memory-hardness properties.

Theorem 1. *VORSHA-1D-S requires* $(2\eta + \tau)$ *correct primary hash values to generate* η*-bit secure hash value.*

Proof. Initially, the seed value is converted into a private constant by repeatedly computing the hash value for τ times using a primary hash function. The primary hash value, h_v, is generated by inputting a secret input string ω and a public seed value $\mathcal{S}$ as $h_v = \textsc{PrimaryHash}(\omega, length, \mathcal{S})$. A single bit is extracted from the newly generated β-bit hash value h_v. The seed value is updated to the newly generated h_v as $\mathcal{S} = h_v$. Now, the seed value is altered using newly computed h_v using a primary hash function as $\mathcal{S} = \textsc{PrimaryHash}(\omega, length, \mathcal{S})$. The second h_v is correct, provided the seed value and the previous h_v are correct. Therefore, it requires $2\eta + \tau$ primary hash values to generate η-bit final secure hash value.

Corollary 1. *The time complexity to generate* η*-bit secure hash value is* $O(\beta(\eta + \tau))$.

Proof. We assume that the primary hash function takes a time complexity of $O(\beta)$ to generate a primary hash value. Therefore, the total time complexity to generate η-bit secure hash value is $O((2\eta + \tau)\beta) = O(\beta(\eta + \tau))$.

Theorem 2. *The probability of reproducing the correct secure hash value using VORSHA-1D-S is* $\frac{1}{2^{(\tau+2\eta)\beta}}$ *without knowing the input string using VORSHA-1D-S algorithm other than Lemma 1.*

Proof. This analysis shows the hardness -of regenerating the correct secure hash value without knowing the input string using VORSHA-1D-S. We assume that the adversary is willing to recompute the secure hash value using the VORSHA-1D-S algorithm without knowing the input string. Theorem 1 shows the total

number of hash values as $(2\eta+\tau)$ to produce the correct η-bit secure hash value. The probability of correctly reproducing a primary hash value without knowing the input string using the brute-force method is $\frac{1}{2^{\beta}}$. Therefore, the probability of reproducing $(2\eta+\tau)$ correct primary hash values is $\frac{1}{2^{(\tau+2\eta)\beta}}$ because the primary hash values are interdependent to each other.

Theorem 2 proves that attacking using VORSHA is more difficult than Lemma 1. VORSHA-1D requires $(\tau+2\eta)$ primary hash values to reproduce η-bit secure hash value correctly. The size of each primary hash value is β, and therefore, it is extremely difficult to reproduce all primary hash values correctly without knowing the input string. The primary hash values are strongly interdependent with each other. Therefore, it becomes computationally infeasible to reproduce all bits of secure hash value using VORSHA without knowing the input string. Therefore, it is better to attack the final secure bit string than the reproduction all the primary hash values using the VORSHA algorithms.

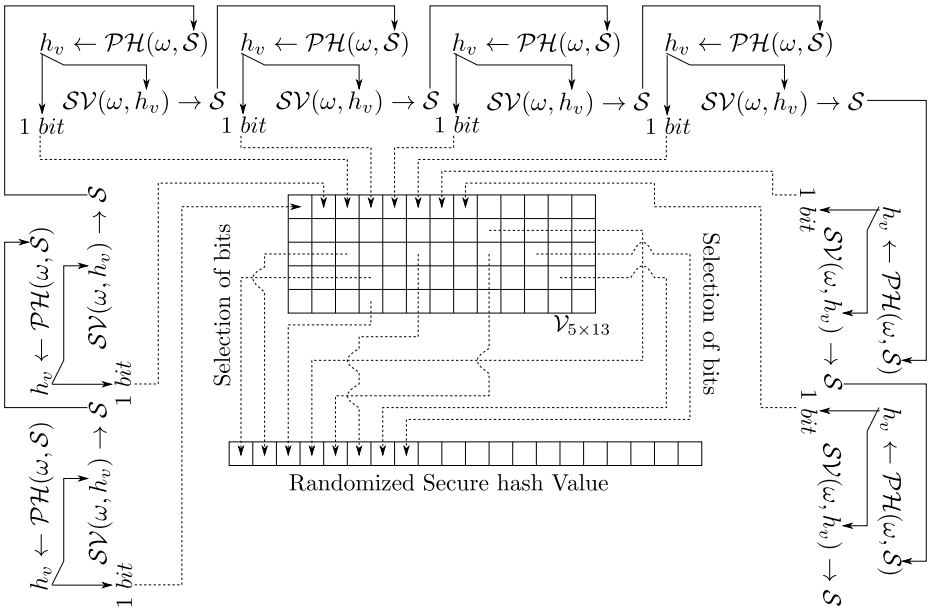

Fig. 2. A simple architecture of VORSHA-2D-S for short input string for generating a randomized secure hash value using 5×13 vector for an example. Firstly, the algorithm generates a set of bits using the primary hash function and fills the 5×13 vector with the generated bits. Secondly, select the generated bits from the vector using the pseudo-random algorithm to form η-bit of a secure hash value.

Two-dimensional VORSHA for short input string Figure 2 demonstrates VORSHA-2D-S to generate a randomized secure hash value ζ. Figure 2 is a

Algorithm 3 Generation of dimension of VORSHA for vector $\mathcal{V}$.

procedure genDim($\mathcal{S}, \theta$)
    $d = \mathcal{S}\%\theta$ ▷ Extracting the digits.
    $dim = \sqrt{d}$
    $dimension = (dim < 16)?(dim + 16) : dim$
    return *dimension*
end procedure

Algorithm 4 VORSHA-2D-S for short input string and medium security.

procedure genVORSHA-2D-S($\omega, \mathcal{S}, \eta$)
    $\mathscr{L}$ = stringLength(ω) ▷ **Stage: Dimension- Starts**
    $\mathcal{S}$ = getSeedValue($\omega, \mathscr{L}, \mathcal{S}$)
    Initialize θ ▷ The θ is used to extract a number from the computed seed value $\mathcal{S}$ for the calculation of a dimension.
    r = genDim($\mathcal{S}, \theta$)
    $\mathcal{S}$ = getSeedValue($\omega, \mathscr{L}, \mathcal{S}$)
    c = genDim($\mathcal{S}, \theta$)
    $X = prime[r], Y = prime[c]$; ▷ $X \neq Y$
    $\mathcal{S} = \mathcal{S} \oplus \eta$ ▷ **Stage: Dimension- Ends**
    for $i : 1$ *to* X **do** ▷ **Stage: Filling- Starts**
        for $j : 1$ *to* Y **do**
            $\mathcal{S}$ = getSeedValue($\omega, \mathscr{L}, \mathcal{S}$)
            h_v = PrimaryHash($\omega, \mathscr{L}, \mathcal{S}$)
            $\rho = h_v\%\varrho$; ▷ The $\varrho = (\beta - c)$ is a prime number.
            $bit = (h_v \wedge (1 << \rho)) >> \rho$
            $\mathcal{V}[i][j] = bit$
            $\mathcal{S} = h_v$
        end for
    end for ▷ **Stage: Filling- Ends**
    for $k : 1$ *to* η **do** ▷ **Stage: Retrieving- Starts**
        $\mathcal{S}$ = getSeedValue($\omega, \mathscr{L}, \mathcal{S}$)
        h_v = PrimaryHash($\omega, \mathscr{L}, \mathcal{S}$)
        $i = (h_v\%X) + 1$
        $\mathcal{S} = h_v$
        $\mathcal{S}$ = getSeedValue($\omega, \mathscr{L}, \mathcal{S}$)
        h_v = PrimaryHash($\omega, \mathscr{L}, \mathcal{S}$)
        $j = (h_v\%Y) + 1$
        $\mathcal{S} = h_v$
        $hash\_bits[k] = \mathcal{V}[i][j]$
    end for ▷ **Stage: Retrieving- Ends**
    ζ = convertIntoHex($hash\_bits, \eta$)
    return ζ
end procedure

representation of Algorithm 4 to enhance the security of the generated hash value. It provides medium security and medium performance. We introduce a vector in VORSHA-2D-S to develop a fully randomized secure hash value, and

the vector is $\mathcal{V}_{X\times Y}$ of two dimensions $X \times Y$ where X and Y are prime numbers. The key embodiment of introducing a vector is to fill the vector with 0s and 1s to form a pool of pseudo-random bits. We select η bits from the filled vector to form a randomized secure hash value. The indexes are selected using two primary hash functions. The VORSHA-2D-S algorithm is divided into three stages: a) dimension, b) filling, and c) retrieving. Firstly, VORSHA-2D-S computes the dimensions X and Y. Secondly, it fills the vector with pseudo-random bits (0s and 1s). Finally, VORSHA-2D-S retrieves η-bit from the filled vector to form η-bit randomized secure hash value.

Dimension The vector requires two prime numbers, X and Y, for defining the dimension for the vector $\mathcal{V}_{X\times Y}$. To make the X and Y prime number, we first compute the hash value with a seed value by repeatedly replacing the older seed value $8 \leq \tau \leq 64$ times. The seed value is calculated and replace τ times by computing as $\mathcal{S} = \textsc{PrimaryHash}(\omega, \mathscr{L}, \mathcal{S})$. The key embodiment of the repeatedly calculating seed value is to protect the public seed value, and it converts the public seed value into a private seed value with the help of an input string. Algorithm 3 calculates a dimension against the seed value. Therefore, the maximum size of the dimension is dependent on the digit extracted from the seed value, as demonstrated in Algorithm 4 by assigning θ to a constant, for instance, $\theta = 12967$ (assigning a prime number to θ is better). For illustration, a digit is calculated as $d = \mathcal{S}\%\theta$, and the $\sqrt{d}$ is the dimension. Notably, the performance of the VORSHA-2D-S is strongly dependent on θ, and it is tunable. Algorithm 3 is invoked two times for a two-dimensional vector, i.e., $r = \textsc{genDim}(\mathcal{S}, \theta)$ and $c = \textsc{genDim}(\mathcal{S}, \theta)$. Algorithm 3 performs square root for 2D or cube root for 3D vectors. We calculate the dimension of the row and column as $X = prime[r]$ and $Y = prime[c]$, respectively, where $prime[]$ is an array of pre-computed prime numbers. Therefore, the dimension of the vector $\mathcal{V}_{X,Y}$ is unknown to the adversaries. Alternatively, the adversaries do not know about the dimensions X and Y, which creates an extra complexity in computing the hash function by the adversaries. VORSHA-2D-S is a variant of a secure hash algorithm; therefore, we calculate the dimension of the vector $\mathcal{V}$ from the hash value produced by the primary hash function. The dimensions X and Y are kept secret to make it a memory-hard secure hash algorithm, but the X and Y must be prime numbers and $X \neq Y$.

Filling VORSHA-2D-S requires filling $X \times Y$ pseudo-random bits (0s and 1s) in the vector $\mathcal{V}_{X\times Y}$. It invokes $\textsc{getSeedValue}(\ )$ to fill a single bit in the vector by producing a hash value h_v using a primary hash function. A single bit is extracted from h_v to fill a single cell in the vector. Therefore, it computes $X \times Y \times \tau$ seed values to fill the vector.

Retrieving After filling the 0s and 1s in the vector, VORSHA-2D-S selects a set of bits from the vector to form a randomized secure hash value. To select a

cell in the vector, we need two indexes and let i and j be the two indexes for a particular cell of the vector $\mathcal{V}$. VORSHA-2D-S computes the value of i and j to retrieve a value as $\mathcal{V}[i][j]$. It requires a seed value. Therefore, it invokes GETSEEDVALUE( ) functions and computes $i = \text{PRIMARYHASH}(\omega, \mathscr{L}, \mathcal{S})\%X$. Moreover, it renews the seed value by invoking GETSEEDVALUE( ) and computes $j = \text{PRIMARYHASH}(\omega, \mathscr{L}, \mathcal{S})\%Y$. The slots are evenly and fairly distributed if X and Y are prime numbers [10]. This process is repeated η times to produce a randomized secure hash value. Therefore, VORSHA-2D-S computes the seed values $2\eta\tau$ times to retrieve η-bit from the vector.

Theorem 3. *VORSHA-2D-S requires $\tau(2 + XY + 2\eta) + (XY + 2\eta)$ primary secure hash values to generate η bits secure hash value.*

Proof. The VORSHA-2D-S algorithm has three stages, namely, dimension, filling, and retrieval. The dimension stage computes the dimensions of the vector by invoking two GETSEEDVALUE( ) functions which invoke 2τ primary hash functions. The filling stage invokes τ primary hash functions for seed values and a primary hash function for h_v to fill a single cell of the vector. Therefore, the total hash function invocations are $XY(\tau + 1)$. The retrieval stage invokes 2η times GETSEEDVALUE( ) function and computes two primary hash values h_v. therefore, the total primary hash function invocations are $2\eta\tau + 2\eta = (2\eta(\tau + 1))$. The overall primary hash function invocations of all stages are $2\tau + XY(\tau + 1) + 2\eta(\tau + 1) = \tau(2 + XY + 2\eta) + (XY + 2\eta)$.

Corollary 2. *The time complexity to generate η-bit randomized secure hash value is $O(\tau(XY + \eta))$ for legitimate users.*

Proof. Theorem 2 shows the total number of hash function calls as $2\tau + XY(\tau + 1) + 2\eta(\tau + 1)$. Rewriting Theorem 2 in Big-Oh notation, we get

$$\begin{aligned} &= O(2\tau + XY(\tau + 1) + 2\eta(\tau + 1)) \\ &= O(\tau + XY\tau + \eta\tau) \\ &= O(\tau(1 + XY + \eta) \\ &= O(\tau(XY + \eta)) \end{aligned} \tag{1}$$

Theorem 4. *The probability of reproducing the randomized secure hash value of η-bit using VORSHA-2D-S is $\frac{1}{2^{\beta(2\tau+XY(\tau+1)+2\eta(\tau+1))}}$ without knowing the input string other than Lemma 1.*

Proof. Theorem 3 exhibits the total number of primary hash function calls as $2\tau+XY(\tau+1)+2\eta(\tau+1)$. Alternatively, it shows the total number hash primary hash values to reproduce a η-bit randomized secure hash value. The probability of reproducing a correct primary hash value without knowing the input string is $\frac{1}{2^\beta}$. Therefore, the probability of reproducing $2\tau + XY(\tau + 1) + 2\eta(\tau + 1)$ primary hash values are $\frac{1}{2^{\beta(2\tau+XY(\tau+1)+2\eta(\tau+1))}}$ without knowing input string using the brute-force method.

VORSHA-2D-S requires $\tau(2+XY+2\eta)+(XY+2\eta)$ primary hash values to correctly compute a η-bit randomized secure hash value, as shown in Theorem 3. Thus, it is harder to reproduce the $\tau(2+XY+2\eta)+(XY+2\eta)$ primary hash values than VORSHA-1D-S. Theorem 4 proves that Lemma 1 is much easier to attack than attacking via VORSHA-2D-S without knowing the input string. It is computationally infeasible to reproduce all the primary hash values without knowing the input string. Thus, it proves that VORSHA-2D-S is stronger than the state-of-the-art secure hash algorithms. Lemma 1 shows that the brute-force attack probability of η-bit secure hash value is $\frac{1}{2^\eta}$ irrespective of the architecture of the secure hash algorithms.

Moreover, VORSHA-2D-S features memory hardness properties because the dimensions of the vector are unknown to the adversaries. The dimension of the vector depends on the seed value. Initially, the seed value can be a public constant; however, it is converted into a private constant in the subsequent steps. Therefore, a correct input string can produce the correct dimension of the vector. Otherwise, it becomes computationally hard to find the correct dimensions of the vector.

Theorem 5. *The lower-bound memory of VORSHA-2D-S is $\Omega(XY+\eta)$ where X and Y are secret, and η is public and fixed.*

Proof. VORSHA-2D-S computes the dimension of the vector in the dimension stage, and the dimensions are X and Y. Therefore, the size of the vector is $X \times Y$. The dimensions X and Y are not known to adversaries. Therefore, the adversary requires at least $XY+\eta$ memory if and only if the adversary succeeded on the very first attempt, but it is highly unlikely because it is hard to reproduce all primary hash values at the first attempt. Moreover, it requires the same amount of memory in a sequential computation of an instance of VORSHA-2D-S. Therefore, the minimum memory requirement is $XY+\eta$. However, the memory requirement for parallel execution is higher than the sequential execution. Therefore, the adversary computes the VORSHA-2D-S in parallel. All parallel instances need separate memory allocation in a single system. Each instance needs a memory of $XY+\eta$. Therefore, the n parallel instances requires $n(XY+\eta)$ where $n <\leq 2^\eta$. The memory of the adversary can be flooded if $n \gg \eta$.

Theorem 6. *The lower-bound time complexity of VORSHA-2D-S is $\Omega(\tau(XY+\eta))$ for the adversaries.*

Proof. Corollary 2 shows the upper-bound time complexity of VORSHA-2D-S for legitimate users. The lower-bound time complexity of VORSHA-2D-S is $\Omega(\tau(XY+\eta))$ since the adversary needs to perform many trials to evade the security of VORSHA-2D-S. The adversary requires at least $O(\tau(XY+\eta))$ time complexity for each trial. Thus, it becomes humongous for a single system. If the adversary computes VORSHA-2D-S in parallel, then time complexity remains $O(\tau(XY+\eta))$ for many instances, provided the system has huge main memory. However, it is unable to execute VORSHA-2D-S in parallel completely due to

memory-hard properties (as shown in Theorem 5). Therefore, the adversary is unable to gain the complete advantage of parallelism. Notably, the adversary can execute a few instances of VORSHA-2D-S in parallel in a single system. Therefore, the lower-bound time complexity of VORSHA-2D-S is $\Omega(\tau(XY+\eta))$ for the adversaries.

Three-dimensional VORSHA for short input string Similar to the VORSHA-2D-S algorithm, the VORSHA-3D-S algorithm is divided into three stages: a) dimensions, b) filling, and c) retrieving. Algorithm 5 computes three dimensions X, Y and Z such that $X \neq Y \neq Z$ in the dimension stage.

Dimension Algorithm 5 presents the VORSHA-3D-S algorithm. VORSHA-3D-S enhances security, but it slows down the performance of the algorithm. Here, we increase the dimension of the vector from 2D to 3D, i.e., $\mathcal{V}_{X\times Y\times Z}$. It invokes GETSEEDVALUE( ) to before computing a single dimension. Then, it computes the dimension. Therefore, it invokes GETSEEDVALUE( ) three times to produce the dimension of the vector, X, Y, and Z; i.e., the seed value is altered 3τ times. Therefore, VORSHA-3D-S features memory-hardness properties because X, Y, and Z are unknown to the adversaries. Similar to VORSHA-2D-S, the dimension of the vector is dependent on the extracted number from the computed seed value $\mathcal{S}$ using θ. Therefore, the maximum and minimum dimensions of the vector can be determined, and the dimension is strongly dependent on $\mathcal{S}$ and θ.

Filling It requires an invocation of GETSEEDVALUE( ) functions to place a single bit in the vector, as shown in Algorithm 5. Therefore, it requires a total $X \times Y \times Z \times \tau$ seed value computations using the primary hash functions to fill the vector $\mathcal{V}$.

Retrieving VORSHA-3D-S requires an invocation of GETSEEDVALUE( ) functions three times to retrieve a single bit from the filled vector. Therefore, VORSHA-3D-S invokes η times to retrieve η bits from the filled vector. Therefore, there are $3\eta\tau$ seed value computations to construct a randomized secure hash value.

Theorem 7. *VORSHA-3D-S requires* $(3\tau + XYZ(\tau+1) + 3\eta(\tau+1))$ *primary hash function invocations to generate* η *bits secure hash value.*

Proof. VORSHA-3D-S is similar to VORSHA-2D-S except for the dimensions. The dimension stage computes the X, Y, and Z dimensions by invoking GETSEEDVALUE( ) function which requires 3τ primary hash values. The filling stage invokes GETSEEDVALUE( ) for seed value, and a primary hash function calls for h_v. Thus, it requires $XYZ(\tau+1)$ primary hash values. The retrieval stage invokes GETSEEDVALUE( ) function three times and three primary hash functions. Therefore, it requires $3\tau\eta+3\eta = 3\eta(\tau+1)$ primary hash values. Hence, the overall hash value requirements or hash function calls is $(3\tau+XYZ(\tau+1)+3\eta(\tau+1))$.

Algorithm 5 Three-dimensional VORSHA for a short input string.

procedure genVORSHA-3D-S($\omega, \mathcal{S}, \eta$)
  $\mathscr{L} = \text{stringLength}(\omega)$ ▷ **Stage: Dimension- Starts**
  $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
  Initialize θ ▷ The θ is used to extract a number from the computed seed value $\mathcal{S}$ for the calculation of a dimension.
  $r = \text{genDim}(\mathcal{S}, \theta)$ ▷ The 3 is dimension, i.e., 3D.
  $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
  $c = \text{genDim}(\mathcal{S}, \theta)$
  $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
  $w = \text{genDim}(\mathcal{S}, \theta)$
  $X = prime[r], Y = prime[c], Z = prime[w];$ ▷ $X \neq Y \neq Z$
  $\mathcal{S} = \mathcal{S} \oplus \eta$ ▷ **Stage: Dimensions- Ends**
  for $i : 1$ *to* X **do** ▷ **Stage: Filling- Starts**
    for $j : 1$ *to* Y **do**
      for $k : 1$ *to* Z **do**
        $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
        $h_v = \text{PrimaryHash}(\omega, \mathscr{L}, \mathcal{S})$
        $\rho = h_v \% \varrho;$ ▷ The $\varrho = (\beta - c)$ is a prime number, for instance, 31 or 61.
        $bit = (h_v \& (1 << \rho)) >> \rho$
        $\mathcal{V}[i][j][k] = bit$
        $\mathcal{S} = h_v$
      end for
    end for
  end for ▷ **Stage: Filling- Ends**
  for $k : 1$ *to* η **do** ▷ **Stage: Retrieving- Starts**
    $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
    $h_v = \text{PrimaryHash}(\omega, \mathscr{L}, \mathcal{S})$
    $i = (h_v \% X) + 1,\ \mathcal{S} = h_v$
    $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
    $h_v = \text{PrimaryHash}(\omega, \mathscr{L}, \mathcal{S})$
    $j = (h_v \% Y) + 1,\ \mathcal{S} = h_v$
    $\mathcal{S} = \text{getSeedValue}(\omega, \mathscr{L}, \mathcal{S})$
    $h_v = \text{PrimaryHash}(\omega, \mathscr{L}, \mathcal{S})$
    $k = (h_v \% Z) + 1,\ \mathcal{S} = h_v$
    $hash\_bits[k] = \mathcal{V}[i][j][k]$
  end for ▷ **Stage: Retrieving- Ends**
  $\zeta = \text{convertIntoHex}(hash\_bits, \eta)$
  return ζ
end procedure

Corollary 3. *The total time complexity of VORSHA-3D-S to generate η bits secure hash value is $O(\tau(XYZ + \eta)$.*

Proof. Theorem 4 demonstrate the total number of hash function calls as $(3\tau + XYZ(\tau+1) + 3\eta(\tau+1))$. Rewriting Theorem 4 in Big-Oh notation, we get

$$\begin{aligned} &= O(3\tau + XYZ(\tau+1) + 3\eta(\tau+1)) \\ &= O(\tau + XYZ\tau + \eta\tau) \\ &= O(XYZ\tau + \eta\tau) \\ &= O(\tau(XYZ + \eta)) \end{aligned} \tag{2}$$

Theorem 8. *The probability of generating the secure hash value by an adversary without knowing the input string using VORSHA-3D-S is* $\frac{1}{2^{\beta(3\tau+XYZ(\tau+1)+3\eta(\tau+1))}}$ *for* β*-bit primary hash function where* η *is the number of bits of the secure hash value other than Lemma 1.*

Proof. Theorem 7 shows the total number of primary hash function invocations as $(3\tau + XYZ(\tau+1) + 3\eta(\tau+1))$. Therefore, the probability of reproducing $(3\tau + XYZ(\tau+1) + 3\eta(\tau+1))$ primary hash value for η-bit randomized secure hash value is $\frac{1}{2^{\beta(3\tau+XYZ(\tau+1)+3\eta(\tau+1))}}$ without knowing the input string using the brute-force method.

Theorem 7 shows the total number of primary hash function requirements as $(3\tau + XYZ(\tau+1) + 3\eta(\tau+1))$. Theorem 5 proves that it is computationally infeasible to reproduce $(3\tau + XYZ(\tau+1) + 3\eta(\tau+1))$ primary hash values without knowing input string, and it is massive for the brute-force attackers. Also, VORSHA-3D-S is stronger than the state-of-the-art secure hash algorithms. Also, it is better to attack Lemma 1 than VORSHA-3D-S. However, the collision probability of η-bit secure hash value is $\frac{1}{2^{\frac{\eta}{2}}}$ irrespective of algorithms.

Theorem 9. *The lower-bound memory of VORSHA-3D-S is* $\Omega(XYZ+\eta)$ *where* X*,* Y *and* Z *are secret, and* η *is public and fixed.*

Proof. Three-dimensional VORSHA for short input string or VORSHA-3D-S is similar to VORSHA-2D-S. The algorithm is divided into three stages, namely, dimension, filling, and retrieval stages. The dimension stage computes three dimensions for the vector, namely, X, Y, and Z. Therefore, the vector size is $X \times Y \times Z$. Also, it requires space of η to output the secure hash value. Therefore, the total space complexity of a legal user is $O(XYZ + \eta)$. However, the dimensions are secret and are unknown to the adversary. Thus, the lower-bound memory is $\Omega(XYZ + \eta)$ for the adversary.

Theorem 10. *The lower-bound time complexity of VORSHA-3D-S is* $\Omega(\tau(XYZ + \eta))$ *for the adversaries.*

Proof. Corollary 3 demonstrates the total time complexity for legitimate users as $O(\tau(XYZ + \eta)$. However, the adversary needs to perform many trials to evade the security of VORSHA-3D-S. If the trial is performed n times in a serial computing system, then it becomes hard to perform the trial because it takes $O(n(\tau(XYZ+\eta)))$ time complexity. The adversary can evade the security

of VORSHA-3D-S in parallel, but a few parallel instances can be executed in parallel due to memory-hard properties of VORSHA-3D-S (see Theorem 9). Thus, it is also infeasible to compute the hash value in parallel in a single system.

Trade-off 1 *There is a trade-off between the performance and security of the hash value. Alternatively, there is a trade-off that a high-bit-sized primary hash function is better in security than a low-bit-sized primary hash function but slower than a low-bit-sized hash function.*

To understand the Trade-off 1, we first ask a simple question- can we use any hash function as a primary hash function? We can use a 32-bit version of the hash function as a primary hash function, namely, murmur2, XXHash, and FastHash as a primary hash function. We can also use a 64-bit version of the primary hash functions. Intuitively, its security is better in a 64-bit version of the primary hash functions than in 32-bit versions. The probability of correctly producing a 32-bit and 64-bit hash value without knowing the input string is $\frac{1}{2^{32}}$ and $\frac{1}{2^{64}}$, respectively, where $\frac{1}{2^{64}}$ is lower than $\frac{1}{2^{32}}$. Thus, a high bit-sized primary hash value is more secure than a low bit-sized primary hash value. Moreover, we justify that the modulus operation in generated hash value as $\rho = hv\%\varrho$. As we know, that prime number is a good candidate for modulus operation; therefore, we can use 31 instead of 32, 61 instead of 64, and 127 instead of 128. The bit value of ρ^{th} position in the h_v is considered as a single hash bit for our randomized secure hash value ζ. Therefore, selecting a single bit among 61 bits is always better than selecting a bit among 31 bits. Thus, security is better in 64-bit than in 32-bit or lower-bit versions. If so, why do not we use MD5, SHA1, or SHA2 as a primary hash function? We can use it, and if we use it, the security becomes superior, but it requires a seed value to qualify as a primary hash function. However, the performance becomes slower than the rest. Particularly, the 32-bit primary hash function is faster than the 64-bit hash function. Furthermore, it is obvious that more number of primary hash function invocations is always slower than the less number of primary hash function invocations. Therefore, there is a trade-off between the performance and security in generating a randomized secure hash value by VORSHA.

Trade-off 2 *The larger the dimensions of the bit vector, better the security but slower the performance.*

Trade-off 2 is similar to Trade-off 1. The bit vector is a pool of pseudo-random bits to be selected to form a randomized secure hash value. For high security, the bit vector should be large, which forms a large-sized pool of bits. However, it associates with more primary hash function invocations in forming a large pool of bit-vector. Therefore, the time complexity rises significantly, and it impacts the performance of the algorithms.

3.2 VORSHA for long input string hashing

Long-input string hashing is useful in diverse applications, for instance, image hashing. The VORSHA for long string hashing utilizes the functions of VOR-

SHA for short input string hashing. The VORSHA for long input string creates the chunks of $16 \leq \sigma \leq 64$ characters arrays and hashes all the chunks using VORSHA for short. The hash values of all chunks are XORed to form a new hash value. VORSHA for long input string invokes corresponding VORSHA for the short hash function. For instance, VORSHA-1D-L invokes the VORSHA-1D-S at least once. The input string is split into σ characters chunks, and let the chunks be $b_1,\ b_2, b_3, \ldots$. We input each chunk into VORSHA-1D-L, VORSHA-2D-L, or VORSHA-3D-L. The first randomized secure hash value of the first block can be computed as $\zeta_1 = \mathcal{H}(b_1)$. Therefore, the final randomized secure hash value is computed as $\zeta = \zeta_1 \oplus \zeta_2 \oplus \zeta_3 \oplus \ldots$.

4 Variability in hash value

The computation of bit size η requires a secret ω, a public seed value $\mathcal{S}$, minimum bit size μ, and offset δ. The μ and δ are public, and it invokes getSeedValue( ) functions to compute the bit size. Finally, the η is computed as $\eta = \mathcal{S}\%\delta + \mu$. However, the bit size η can be input by the user or computed by other methods. Here, the bit size is controlled by the input string. Therefore, an input string can have a single value of η, which ranges $[\mu, \lambda)$ and $\lambda = \delta + \mu$.

Algorithm 6 Invoking VORSHA-3D to generate a secure hash value.

```
1: procedure VORSHA(ω)
2:     Initialize seed value S with a constant.
3:     Initialize μ, and δ                ▷ For instance, μ = 512 and δ = 997
4:     L = Length(ω)
5:     S = getSeedValue(ω, L, S)
6:     η = S%δ + μ
7:     L = Length(ω)
8:     if L ≤ 16 then
9:         S = getSeedValue(ω, L, S)
10:        ζ = genVORSHA-3D-S(ω, S, η)
11:    else
12:        S = getSeedValue(ω, L, S)
13:        ζ = genVORSHA-3D-L(ω, S, η)
14:    end if
15: end procedure
```

Algorithm 6 is an example of producing a secure hash value without the desired bit size as an input. Algorithm 6 demonstrates the VORSHA-v hash function call with VORSHA-3D-S as an example. The variable-sized hash function produces unpredictable and variable-sized secure hash values. We take $\mu = 512$, and $\delta = 997$ for demonstration purposes; however, these parameters are tunable by the users as per their requirements. For instance, a user can set $\mu = 256$, and $\delta = 1237$. Our recommendation is that the size δ should be a prime number and

large enough to produce more variability in the final secure hash value. We have achieved variability using a hash value. However, the η can be input by the user and keep it as a secret.

Theorem 11. *For a range* $[\mu, \lambda)$, $2^{\mu} + 2^{\mu+1} + 2^{\mu+2} + \ldots + 2^{\lambda} = 2^{\mu}(2^{\delta+1} - 1) - 2$ *where* $\lambda = \mu + \delta$ *and* $\lambda > \mu$.

Proof. For every natural number λ, $2^0 + 2^1 + 2^2 + \ldots + 2^{\lambda} = 2^{\lambda+1} - 1$. Therefore, $2^0 + 2^1 + 2^3 + \ldots + 2^{\mu-1} = 2^{\mu} - 1$. Thus, for the range $[\mu, \lambda)$,

$$\begin{aligned} &= 2^{\mu} + 2^{\mu+1} + 2^{\mu+2} + \ldots + 2^{\lambda} \\ &= (2^{\lambda+1} - 1) - (2^{\mu} - 1) \end{aligned} \tag{3}$$

The λ can be expressed as $\lambda = \mu + \delta$ where $\lambda > \mu$. Thus, we Equation (3) can be rewritten as

$$2^{\mu} + 2^{\mu+1} + 2^{\mu+2} + \ldots + 2^{\lambda} = 2^{\mu}(2^{\delta+1} - 1) - 2 \tag{4}$$

Theorem 11 defines the total number of required hash values to be stored by the RTaaS. It increases the size of the hashtable of the RTaaS dramatically. A correct input string has a total of $(\lambda - \mu)$ possible randomized secure hash values because the randomized secure hash algorithm can correctly compute many correct hash values with different sizes for a single input string. Hence, the adversary is unable to find the correct hash value for the key from RTaaS in a given scenario. Therefore, it creates another complexity for RTaaS. A few input strings can flood the hashtable of the RTaaS. Therefore, it becomes computationally infeasible for the RTaaS. However, RTaaS is a black box and is assumed to be capable of storing all the possible hash values. But, the variability in hash value creates another complexity for RTaaS to maintain all the possible hash values of the VORSHA.

Theorem 12. *The memory requirement to store all sizes of secure hash values, excluding the 2D or 3D vector, ranging from* μ *to* λ *for a single input string is* $((\delta + 1)\mu + \frac{\delta(\delta+1)}{2})$ *for the adversaries.*

Proof. The size of the randomized secure hash value is unable to be determined, and it ranges $[\mu, \lambda)$. Therefore, the adversary needs to compute a secure hash value using all the sizes ranging from μ to λ because η is a secret variable. Therefore, the total memory requirement of a single hash value for the adversary is the summation of all memory from μ to λ. The $\lambda = \mu + \delta$. If we sum up all, we get

$$\begin{aligned} &= \mu + (\mu + 1) + (\mu + 2) + (\mu + 3) + \ldots + \lambda \\ &= \mu + (\mu + 1) + (\mu + 2) + (\mu + 3) + \ldots + (\mu + \delta) \\ &= (\delta + 1)\mu + (1 + 2 + 3 + \ldots + \delta) \\ &= (\delta + 1)\mu + \frac{\delta(\delta + 1)}{2} \end{aligned} \tag{5}$$

Theorem 13. *The space complexity for a single hash value of VORSHA-2D-S is* $O(^{2(\varphi-16)}C_2XY + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$ *for the adversary in a single system in parallel.*

Proof. Theorem 5 and 9 show the memory requirement for an instance of VORSHA-2D-S and VORSHA-3DS in a sequential manner, respectively. VORSHA-2D-S uses a two-dimensional vector for filling pseudo-random bits, and the dimensions are X and Y. The dimensions are computed as $X = prime[r]$ and $Y = prime[c]$ where the $prime[]$ is precomputed prime number array. The r and c are computed as $\sqrt{\mathcal{S}\%\theta}$. The maximum value of r and c can be $\varphi = \sqrt{\theta - 1}$. Therefore, $r, c \in [16, \varphi)$. Therefore, the total combination of r and c by choosing two values is $\binom{2(\varphi-16)}{2}$. Moreover, each combination uses $X \times Y$ memory. Furthermore, Theorem 12 shows the total memory requirement to reproduce a single randomized secure hash value using a brute-force method. Therefore, the total memory requirement for an adversary is $O(^{2(\varphi-16)}C_2XY + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$ in a single system in parallel.

Corollary 4. *The space complexity for a single hash value of VORSHA-3D-S is* $O(^{3(\varphi-16)}C_3XYZ + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$ *for the adversary in a single system in parallel.*

Corollary 5. *The lower-bound memory of VORSHA-2D-S, and VORSHA-3D-S are* $\Omega(XY + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$*, and* $\Omega(XYZ + (\delta+1)\mu + \frac{\delta(\delta+1)}{2})$*, respectively where* X*,* Y*,* Z *and* η *are secret and* $\eta \in [\mu, \mu + \delta)$*.*

Corollary 5 replaces the η in Theorem 5 and 9 using Theorem 12. The lower-bound becomes enormous because the size of the secure hash value is unknown to the adversary.

5 Randomness analysis

True-random numbers are highly unpredictable because each bit of a random number is influenced by some other external events. However, each bit of the pseudo-random number is dependent on the initial input. Similarly, the secure hash algorithm is also dependent on the initial input string. Therefore, there is no difference between a pseudo-random number generator and a secure hash algorithm.

Table 3 and 4 show the randomness of the generated bit string of VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S for 64 and 128-bit streams, respectively, in NIST SP 800-22 statistical test suite. We generated 10M bits using VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S by setting the input string "VORSHA", $\mathcal{S} = 198899$, $\tau = 32$ and 32-bit version of the murmur2 [3] hash function. We defined the $\theta = 9973$ and $\theta = 9769$ for VORSHA-2D-S and VORSHA-3D-S, respectively. Algorithm 3 calculates the dimensions of VORSHA-2D-S as $X = 419$ and $Y = 449$. Similarly, the dimensions of VORSHA-3D-S are $X = 443$, $Y = 191$, and $Z = 313$. Our experiment proves that our proposed algorithm

Table 3. Comparison of VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S algorithms for 64-bit streams in NIST SP 800-22.

Test name	64 bits & VORSHA-1D-S		64 bits & VORSHA-2D-S		128 bits & VORSHA-3D-S	
	P-value	Pass rate	P-value	Pass rate	P-value	Pass rate
Frequency	0.437274	64/64	0.350485	62/64	0.671779	61/64
Block Frequency	0.213309	64/64	0.739918	64/64	0.500934	62/64
Cumulative sums	0.888137, 0.195163	64/64, 64/64	0.534146, 0.637119	63/64, 62/64	0.299251, 0.834308	63/64, 61/64
Runs	0.122325	61/64	0.602458	64/64	0.253551	64/64
Longest runs	0.671779	64/64	0.568055	64/64	0.066882	64/64
Rank	0.772760	63/64	0.437274	63/64	0.739918	61/64
FFT	0.862344	63/64	0.134686	63/64	0.134686	63/64
Non-overlapping Template	–	Pass	–	Pass	–	Pass
Overlapping Template	0.534146	62/64	0.148094	64/64	0.862344	64/64
Universal	0.568055	64/64	0.232760	63/64	0.213309	64/64
Approximate Entropy	0.350485	62/64	0.804337	63/64	0.134686	64/64
Random Excursions	–	Pass	–	Pass	–	Pass
Random Excursions Variant	–	Pass	–	Pass	–	Pass
Serial	0.134686, 0.090936	62/64, 64/64	0.213309, 0.437274	64/64, 63/64	0.299251, 0.706149	64/64, 63/64
Linear complexity	0.122325	62/64	0.637119	63/64	0.468595	63/64

passes all 15 tests of the NIST SP 800-22 statistical test suite. It demonstrates the generated bit strings contain highly unpredictable bit strings, which is vital for a randomized secure hash value.

6 Attack analysis of VORSHA

Attacks are inevitable in security, and we need an analysis of the proposed system. Diverse attacks on SHA are already reported, such as collision attacks, chosen-prefix attacks, preimage attacks, and second preimage attacks.

6.1 Collision attacks

Theorem 14. *The probability of collision attack of VORSHA is* $\frac{1}{\sum_{\eta=\mu}^{(\mu+\delta)} 2^{(\eta+1)/2}}$ *where the bit size* η *is secret.*

Table 4. Comparison of VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S algorithms for 128-bit stream in NIST SP 800-22.

Test name	64 bits & VORSHA-1D-S		64 bits & VORSHA-2D-S		128 bits & VORSHA-3D-S	
	P-value	Pass rate	P-value	Pass rate	P-value	Pass rate
Frequency	0.253551	127/128	0.500934	126/128	0.862344	126/128
Block Frequency	0.654467	124/128	0.534146	126/128	0.095617	128/128
Cumulative sums	0.026648, 0.337162	128/128, 128/128	0.242986, 0.422034	126/128, 126/128	0.819544, 0.772760	127/128, 126/128
Runs	0.452799	125/128	0.134686	126/128	0.568055	127/128
Longest runs	0.178278	127/128	0.875539	127/128	0.756476	127/128
Rank	0.015963	126/128	0.484646	127/128	0.287306	127/128
FFT	0.275709	127/127	0.637119	127/128	0.015963	127/128
Non-overlapping Template	–	Pass	–	Pass	–	Pass
Overlapping Template	0.350485	126/128	0.178278	128/128	0.242986	127/128
Universal	0.324180	124/128	0.311542	125/128	0.551026	126/128
Approximate Entropy	0.213309	126/128	0.054199	128/128	0.011250	125/128
Random Excursions	–	Pass	–	Pass	–	Pass
Random Excursions Variant	–	Pass	–	Pass	–	Pass
Serial	0.186566, 0.551026	123/128, 127/128	0.941144, 0.324180	127/128, 125/128	0.299251, 0.364146	128/128, 127/128
Linear complexity	0.213309	127/128	0.756476	128/128	0.517442	128/128

Proof. Let a hash function f maps input x to output y, i.e., $f : x \to y$, and it is termed as $f(x)$. For given y, we need to find two input strings x and x' such that $f(x) = f(x')$ where $x \neq x'$ [2]. Let us analyze the collision probability of the secure hash value by fixing the bit size to η where η is public. The exact probability of getting a collision in η bits hash value and k strings hashed is

$$1 - \frac{2^{\eta}!}{(2^{k\eta}(2^{\eta} - k)!)} \tag{6}$$

Our objective is to find the value of k to get a collision. From the birthday paradox, Equation (6), we get an approximation as

$$p = 1 - e^{-k^2/2^{\eta+1}} \tag{7}$$

However, Equation (7) is larger than as expected, and by rewriting Equation (7) to represent the desired probability, we get

$$\begin{aligned} p &= 1 - e^{-k^2/2^{\frac{\eta+1}{2}}} \\ 1 - p &= e^{-k^2/2^{\frac{\eta+1}{2}}} \\ -ln(1-p) &= k^2/2^{\frac{\eta+1}{2}} \\ k^2 &= -2^{\frac{\eta+1}{2}} ln(1-p) \\ k &= 2^{(\eta+1)/4}\sqrt{-ln(1-p)} \end{aligned} \tag{8}$$

Let us approximate $(1-p) = -p$ in Equation (8), then

$$k = 2^{(\eta+1)/4}\sqrt{p} \tag{9}$$

The secure hash value size is assumed as η and known to all. We get the number of strings to hash to get a collision as shown in Equation (8). Algorithm 6 uses variable-sized hash value, and the size of secure hash value η is unpredictable. The size of η depends on the primary hash value. Therefore, the minimum size of the hash value is μ, and the maximum size of the hash value is $\lambda = \mu + \delta$. The correct length of a secure hash value lies between μ and λ in Algorithm 6. The correct input string translates into the correct length of the secure hash value. Therefore, the number of strings hashing required to get a collision of in the hash value is

$$\begin{aligned} k &= \left(\sum_{\eta=\mu}^{\lambda} 2^{(\eta+1)/4} \right) \sqrt{p} \\ &= \left(\sum_{\eta=\mu}^{(\mu+\delta)} 2^{(\eta+1)/4} \right) \sqrt{p} \end{aligned} \tag{10}$$

Equation (10) shows the approximate number of string hashing required to get a collision using variable-sized hash value, ranges $[\mu,\ \lambda)$. It shows that a high range becomes highly secure in hashing a string. Alternatively, it is always better in a large difference between λ and μ, i.e., it should be at least $\delta = (\lambda - \mu) > 256$, but μ should also be sufficiently large to withstand diverse attacks. Equation (10) shows the number string to hash to get a guaranteed collision with probability p. Now, we calculate the probability of collision as

$$\begin{aligned} k &= \left(\sum_{\eta=\mu}^{(\mu+\delta)} 2^{(\eta+1)/4} \right) \sqrt{p} \\ \sqrt{p} &= \frac{k}{\sum_{\eta=\mu}^{(\mu+\delta)} 2^{(\eta+1)/4}} \\ p &= \frac{k^2}{\sum_{\eta=\mu}^{(\mu+\delta)} 2^{(\eta+1)/2}} \end{aligned} \tag{11}$$

The collision probability of a single item ($k = 1$) can be derived from Equation (11), we get

$$p = \frac{1}{\sum_{\eta=\mu}^{(\mu+\delta)} 2^{(\eta+1)/2}} \tag{12}$$

Recalling RTaaS, it is assumed to have the capability to store a massive amount of secure hash values. Still, VORSHA creates difficulty for RTaaS to conduct a collision attack because different input string translates into a wrong length of the secure hash value. Therefore, a collision attack is harder to conduct in VORSHA as compared to the state-of-the-art secure hash algorithms, but we cannot deny the possibilities. Another similar attack is chosen prefix attack. It is carried out by concatenating the message with a prefix to get collision attacks. Alternatively, $f(p_1 \mid\mid x) = f(p_2 \mid\mid x') = y$ where $\mid\mid$ is a concatenation operator and p_1 and p_2 are the chosen prefix. The probability of a chosen-prefix attack is similar to the collision attack probability, as shown in Equation (12).

6.2 Preimage attacks and second preimage attacks

Preimage attack defines that for a given hash value y, find x such that $f : x \rightarrow y$. For a large-sized hash value, it becomes infeasible to find the input item x for a certain hash function. Adversaries may correctly find input item x for the hash value y, or the adversary can find another input that maps to y due to collision.

We know that the preimage attack can be carried out in 2^η for a conventional secure hash algorithm. Here, the η is fixed and public. Therefore, the preimage attack can also be carried out in 2^η for VORSHA-1D-S, VORSHA-2D-S, and VORSHA-3D-S if and only if η is public and fixed. The η is secret and dependent on the input string in VORSHA. Therefore, the input string to perform a preimage attack may not lead to correct η or vice-versa. It requires a correct input string to perform a preimage attack. Therefore, the security strength of VORSHA is calculated as

$$\sum_{\eta=\mu}^{\mu+\delta} 2^\eta \tag{13}$$

For VORSHA, it is possible to find other input items that map to the same hash value. An adversary can find a x' which also maps to the hash value y, which is known as a second preimage attack, i.e., $f(x) = f(x') = y$ and $x \neq x'$ for given x and y. The second preimage attack defines that for given input item x, an adversary finds x' such that $x \neq x'$ and $f : x \rightarrow y$ and $f : x' \rightarrow y$; i.e., an adversary can find another input item that maps to the same output for a given input item. The security strength of VORSHA is the same as Equation (13).

7 Parallelism of generating hash value using VORSHA

Parallel processing is a way of solving problems concurrently to improve execution time. It enhances the running time significantly, but it becomes a drawback

for secure hash algorithms. If an algorithm can solve a problem in $O(n)$ time complexity, it can be reduced to even $O(1)$ time complexity in parallel execution. Particularly, if an adversary can solve the problem in $O(n)$ time complexity in a sequential computer, then the adversary can solve the problem in $O(1)$ time complexity in parallel, which is disadvantageous for secure hash algorithms. Therefore, we strongly discourage the generation of a single hash value in parallel. VORSHA hash function generates hash bits in a sequential manner, which is inefficient in generating in parallel. The key reason is that the output of the primary hash function is input for the next primary hash function for generating a bit. Thus, parallel processing becomes disadvantageous in such a problem if the input and output are strongly interdependent or it depends on the previously computed values. Thus, we are able to restrict the parallel processing of generating a single hash value using VORSHA. Noteworthy that multiple hash values can be generated in parallel using distributed computing. For instance, MapReduce can generate many hash values by spawning multiple map tasks, and one reduce task. In this case, each map task generates many hash values, and reduce task collects the generated hash values as output. But it becomes inefficient for a MapReduce job to generate multiple hash values due to the memory hardness properties because MapReduce is unable to avail the complete advantage of parallelism.

8 Conclusion

In this paper, we have demonstrated the variable-sized, one-way, and randomized secure hash algorithm. To the best of our knowledge, VORSHA is the first algorithm to feature randomization in producing a secure hash value. Also, we have presented 1D, 2D, and 3D VORSHA for diverse applications' requirements. We have also demonstrated the memory-hardness features of VORSHA to defeat the parallel processing of the adversaries. Moreover, we have illustrated the various advantages of variability in a secure hash value. Furthermore, the variability can defeat diverse attacks; for instance, the rainbow table attack.

References

1. Online Tools. [Online], https://emn178.github.io/online-tools/index.html (Mar 2021)
2. Al-Kuwari, S., Davenport, J.H., Bradford, R.J.: Cryptographic hash functions: Recent design trends and security notions. Cryptology ePrint Archive, Paper 2011/565 (2011), https://eprint.iacr.org/2011/565, https://eprint.iacr.org/2011/565
3. Appleby, A.: murmurhash. [Online], Retrieved on October 2022 from https://sites.google.com/site/murmurhash
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Advances in Cryptology – EUROCRYPT 2008, pp. 181–197. Springer, Berlin, Germany (2008). https://doi.org/10.1007/978-3-540-78967-3_11

5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Advances in Cryptology – EUROCRYPT 2013, pp. 313–314. Springer, Berlin, Germany (2013). https://doi.org/10.1007/978-3-642-38348-9_19
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Making of KECCAK. Cryptologia **38**(1), 26–60 (Jan 2014). https://doi.org/10.1080/01611194.2013.856818
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013. pp. 313–314. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 292–302. IEEE, Saarbrucken (Mar 2016). https://doi.org/10.1109/EuroSP.2016.31, http://ieeexplore.ieee.org/document/7467361/
9. Boneh, D., Corrigan-Gibbs, H., Schechter, S.: Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks. In: Advances in Cryptology – ASIACRYPT 2016, pp. 220–248. Springer, Berlin, Germany (Nov 2016). https://doi.org/10.1007/978-3-662-53887-6_8
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
11. Percival, C.: Stronger key derivation via sequential memory-hard functions. [online], Available at https://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf