JOIN (Jurnal Online Informatika)
p-ISSN: 2528-1682, e-ISSN: 2527-9165
Volume 10 Number 2 | December 2025: 384-394
DOI: 10.15575/join.v10i2.1471

Modified Hash to Obtain Random Subset-Tree (MHORST) Using Merkle Tree and Mersenne Twister

Faidhil Nugrah Ramadhan Ahmad[1], Ari Moesriami Barmawi[2]
[1]Graduate School of Informatics, School of Computing, Telkom University, Indonesia
[2]Graduate School of Forensic Science and Cyber Security, School of Computing, Telkom University, Indonesia

Article Info

Article history:

Received September 19, 2024
Revised March 06, 2025
Accepted March 22, 2025
Published November 10, 2025

Keywords:

Digital Signature
HORST
HORSIC
Merkle Tree
Mersenne Twister
Quantum Computing
Security Level

ABSTRACT

The development of quantum computing triggers new challenges in data security, particularly in addressing attacks that can solve complex mathematical problems on the fly. Several hash-based data security methods have been proposed to deal with this threat, one of them being Hash to Obtain Random Subset-Tree (HORST). However, HORST has drawbacks, such as low security, because it only uses one hash round. The security of HORST is already improved by Hash to Obtain Random Subset and Integer Composition (HORSIC). However, HORSIC's execution time is significantly increased. The problem of this research is the low-security HORST and the high execution time of HORSIC. This research proposes a new method, Modified Hash to Obtain Random Subset-Tree (MHORST), which aims to improve the security of HORST and reduce the execution time to less than HORSI's. MHORST uses Merkle tree, SHA-256 hashes, and Mersenne Twister to build public keys and digital signatures. Based on the experiment results, MHORST reduces the signing time by more than 3.3 times compared to HORST. MHORST reduces the verification time by more than 1.1 times HORST and 17 times HORSIC. Although the security level of MHORST decreases slightly compared to HORSIC, this method is still more secure than HORST against signature forgery.

Corresponding Author:

Ari Moesriami Barmawi
Graduate School of Forensic Science and Cyber Security, School of Computing, Telkom University
Jalan Telekomunikasi No.1, Bandung 40257, Indonesia
Email: mbarmawi@melsa.net.id

1. INTRODUCTION

Several hash methods that are resistant to quantum attacks are Hash to Obtain Random Subset (HORS) [1][3] and Hash to Obtain Random Subset-Tree (HORST) [3][5]. Hash to Obtain Random Subset-Tree is a development of the HORS method[15], which combines the Merkel Tree [8][9] in creating public keys and signatures. Since the signature is distributed, an attacker can guess the pattern of the signature using the public key[6][10].

Hash to Obtain Random Subset and Integer Composition combines the HORS and Integer Composition methods [13], where the signature creation process uses the Integer Composition Algorithm. The Integer Composition Algorithm is the process of combining several positive integers to form a specific integer[11]. However, the time required for creating compositions from the Integer Composition algorithm is quite large, which is a weakness of this method [12]. Therefore, a method is needed to improve the level of security and the relatively lower time execution of the two methods[4][16].

HORSIC is a method that combines HORS and integer composition[7]. However, adding the integer composition method increases the processing time of the signature creation. Additionally,

HORSIC key generation uses repeated hashing, which increases the execution time required [3]. Therefore, it remains a challenge to design a digital signature method that addresses the shortcomings of HORST in terms of the security level and HORSIC in terms of time execution[17].

The proposed method aims to increase security for HORST, and decrease time execution for HORSIC. In the proposed method, the key generation uses a Merkel tree, where each leaf that has been concatenated will be hashed to the top of the Merkle tree. The hash uses SHA-256, which is considered safe and resistant to many cryptographic attacks[2][14]. Public keys are taken from some leaf of each level of the Merkel tree. In the message section, the message will be hashed and converted into an index value without going through the Integer composition process.

This research assumes the seed is a random value from 0 - 255, generated with a Pseudo Random Number Generator (PRNG). The hash process uses the SHA-256 or 256-bit Secure Hash Algorithm[14]. The sender and Receiver have the same PRNG. The results show that the key generation time for HORST is more than 9 times faster than that of HORSIC and about 1.2 times faster than that of MHORST. Regarding the signing, HORST reduces the time by more than 2.1 times compared to HORSIC, while MHORST reduces the signing time by more than 3.3 times compared to HORST. HORST shows a time reduction of more than 17 times for verification compared to HORSIC, while MHORST reduces the verification time by more than 1.1 times. Although the security level of MHORST decreases slightly compared to HORSIC, this method is still superior to HORST in reducing the probability of signature forgery.

2. METHOD

The main idea of this research is to overcome HORST's weakness related to its low security[4]. This research proposes modifications by adding a hashing method to the Merkle tree for public key and signature generation. In addition, public key randomization is performed using a seed obtained using Mersenne Twister PRNG to increase the security of the transmitted public key. The randomized key is referred to as the Published Public Key.

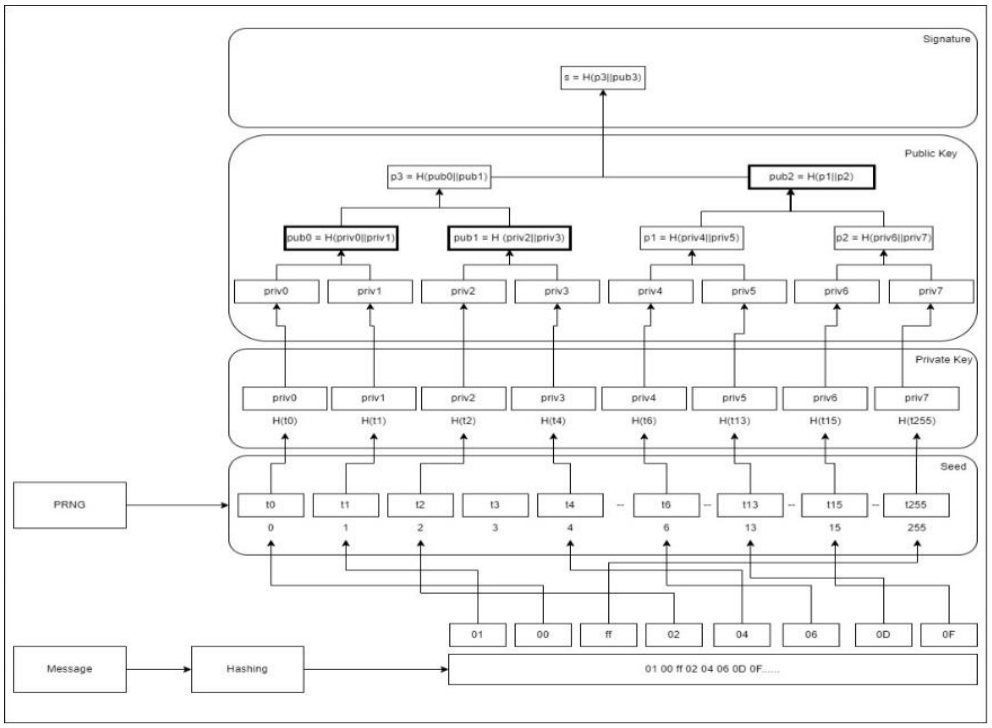

Figure 1. Overview of the MHORST

Creating a digital signature using MHORST goes through five stages: seed generation, private key generation, signing process, public key generation, and signature verification (see Figure 1). The seed generation process uses PRNG for public key randomization. This process will produce a seed in the form of random numbers that will be used for private key generation. The next stage is the signing process, which involves creating a digital signature involving private and public keys. The last stage is

p-ISSN: 2528-1682
e-ISSN: 2527-9165

the signature verification process, which is the process of validating the signature using the public key and message. By using the methods described, an increase in the security of the MHORST method will be obtained.

2.1 Seed Generation

The seed generation process uses a Pseudorandom Number Generator (PRNG) based on the Mersenne Twister algorithm. A PRNG is a program or tool designed to generate a sequence of numbers or symbols in a non-deterministic manner, producing what appears to be a random series of values. The Mersenne Twister algorithm has been selected for this purpose due to its efficiency and reliability in generating pseudo-random numbers[20]. The seed generated in this process results from a random number generator of 256 numbers with values between 0 and 255. The algorithm of the random generation process can be seen in Algorithm 1.

Algorithm 1: PRNG Mersenne Twister

```
Input: z (number of seed), m (max range), f(initial number for values), w (word
size exp.64 bit), a (multiplication constant for transformation), s, t (shift
values), b, c (bitmask values for xor transformation), l (final transformation
shift), r is the binary number of r 1's
Output: t_k (seed)
index = 1
ll = (1 << r) - 1
u = lowest w bits of (NOT ll)
MT[] = (0,1,2,3,4,…,255)
i = 1
while i < z
    MT[i] = lowest w bits of (f * (MT[i-1] XOR (MT[i-1] >> (w-2))) + i)
    i = i + 1
end while
Function extract number()
for i from 1 to z-1
  if index >= z then
      if index > z then
          error "Generator was never seeded"
      end if
      twist()
  end if
   y = MT[index]
   y = y XOR (y >> u)
   y = y XOR ((y << s) AND b)
   y = y XOR ((y << t) AND c)
   y = y XOR (y >> l)
   t_i = lowest w bits of (y)
   index = index + 1
end for
Function twist()
i = 0
while i < z
   x = (MT[i] AND upper_mask) OR (MT[(i+1) mod z] AND lower_mask)
   xA = x >> 1
   if (x MOD 2) != 0 then
       xA = xA XOR a
   end if
    MT[i] = MT[(i + w) mod k] XOR xA
   i = i + 1
end while
t_k = (t_1,t_2,t_3,…,t_z)
return t_k
```

Note: z (number of seed), m (max range), f (initial number for values), w (word size exp. 64 bit), u (bitmask upper bits), ll (bitmask lower bits), a (multiplication constant for transformation), s, t (shift values), b, c (bitmask values for xor transformation),l (final transformation shift)

2.2 Private Key Generation

This section discusses hashing messages using SHA-256, the indexing process, where the message is cut as an index value and generates a private key.

2.2.1 Hashing Message

SHA-256 will be used in the HORST modification process to generate private keys, public keys, and signatures. In the HORST modification process, the message will be hashed using SHA-256 to produce a hash value, which will then be indexed. Here is an example of a message that has been hashed using SHA-256:f54c3cb3e19e0604e5210c136756462e59cd4ec4d8b442575c5a139d15ec 42a2

2.2.2 Indexing Process

The indexing process divides the message into parts of 8 bits and assigns an index for each part of the message. In this case, the parts are h_1, h_2,..., h_k. The seed is selected based on the index. In this case, the data that the index will show is the selected seed, which is then hashed to generate the private key. Seed is a random value generated by PRNG using Mersenne Twister. Suppose the indexed parts of the message are as follows:
[245,76,60,179,225,158,6,4,229,33,12,19,103,86,70,46,89,205,78,196,216,180,66,87,92,90,19,157,21, 236,66,162]

2.2.3 Generating Private Key

In the private key generation process, the data used is the seed selected in the indexing process that will be used in the creation of the Merkle tree. Next, the seed is processed using SHA-256 for one hash round. The private key resulting from the indexing process is as follows. The algorithm of the key generation process can be seen in Algorithm 2.

Algorithm 2: Private Key Generation

Input: m (Message), t_k (Seed) = (k, t_1, t_2,..., t_k), and k (Key length)
Output: $priv_k$ (Private Key)
h = Hash(m)
Split h into k substrings h_1, h_2,..., h_k, of length $\log_2$ t bits each
For j = 1 to k:
    convert each h_j to an integer i_j
End For
For j = 1 to k:
    $priv_j$ =Hash (t_{i_j})
End For
priv=($priv_1$,$priv_2$ …$priv_k$)
Return priv

Note: m is the message, t_k is the seed, k is the key length, and $priv_k$ is the private key

Suppose the seed is as follows: [44,214,171,180,19,29,101,49,152,112,77,210,209,105 ,48,145,29,237,90,156,193,167,93,142,236,220,210,13,207,228,93,6]. The result of seed hash process: [71ee45a3c0db9a9865f7313dd3372cf60dca6479d46261f3542eb9346e4a04d6,802b906a18591ead8 a6dd809b262ace4c65c16e89764c40ae326cfcff811e10c,284de502c9847342318c17d474733ef468fbd be252cddf6e4b4be0676706d9d0,7b69759630f869f2723875f873935fed29d2d12b10ef763c1c33b8e0 004cb405,……..,9d693eeee1d1899cbc50b6d45df953d3835acf28ee869879b45565fccc814765,6 e4001871c0cf27c7634ef1dc478408f642410fd3a444e2a88e301f5c4a35a4d,e7f6c011776e8db7cd330 b54174fd76f7d0216b612387a5ffcfb81e6f0919683]

2.3 Signing Process in MHORST

The signing process begins by generating a Merkle Tree using the private key. Suppose the first node of the Merkle Tree is 71ee45a3c0db9a9865f7313dd3372cf60dca6479d46261f3542eb9346e4a0

p-ISSN: 2528-1682
e-ISSN: 2527-9165

4d6. Furthermore, the second node of the Merkle Tree is 802b906a18591ead8a6dd809b 262ace4c65c16e89764c40ae326cfcff811e10c. Then, the first node and the second node should be concatenated. The concatenated node should be processed using SHA-256 and placed as the first branch of the first-level Merkle tree. The third node of the Merkle Tree is 284de502c9847342318c17d474733ef 468fbdbe252cddf6e4b4be0676706d9d0, the fourth node of the Merkle Tree is 7b69759630f869f2723875f873935fed29d2d12b10ef763c1c33b8e0004cb405. Then, the third node and the fourth node should be concatenated. The concatenated node should be processed using SHA-256 and placed as the second branch of the first-level Merkle tree. This mechanism will be used until the last private key node pair is reached. The algorithm of the signature generation process is shown in Algorithm 3.

Algorithm 3: Signing-MHORST

Input: $priv_k$ (Private Key)
Output: s (Signature) and tpb_n (Temporary Public Key)
l =length(priv)
$tpb_0 = priv_0$
$tpb_1 = priv_1$
$temp_x = priv$
while l > 2
  i = 0
  x = 0
  while i < *l*
    sk_x= Hash (concatenate ($temp_i$+$temp_{i+1}$))
    $temp_x = sk_x$
    x = x+1
    i = i+2
  $tpb_x = (temp_x)$
  End While
  l = x
end while
i=0
s = Hash (concatenate ($temp_i$+$temp_{i+1}$))
tpb= (tpb_1,tpb_2 … tpb_n)
return s,tpb

Note: s is signature, tpb is temporary public key, priv is private key

2.4 Public Key Generation

The public key is a randomly selected leaf on the Merkle tree created in the previous process. Selecting multiple public keys aims to reduce processing time. The verification process can be shortened by taking representative public keys in the Merkle tree (see Algorithm 4). In this case, to determine the number of leaves selected as public keys, it is necessary to know the number of levels of the Merkle tree level, which is equal to $\log_2 d$, where d is the number of nodes.

Algorithm 4: Choosing the Public Key

Input: tpb(tpb_1,tpb_2,…)(temporary public key),ck(ck[1],ck[2]…, ck[length(ck)](seed)
Output: pub (public key)
q =length(ck)
For i = 0 to q:
  pb_i =$tpb_{ck[i]}$
End For
pub= (pb_1,pb_2 … pb_q)
Return pub

Note: tpb is temporary public key, ck is chosen key, pub is public key

Suppose d = 32 which means the number of Merkle tree levels is 5. In the leaf selection, at least the leaves that will be used to form paths to higher levels and have connections to build roots are selected so there

is no interference in key construction and verification. Thus, the number of public keys that must be stored and processed is optimized, making the process more efficient than taking all the leaves in the Merkle tree.

2.5 Published Public Key Generation

The purpose of publishing the published public key is to randomize the order of the public key so that an attacker cannot easily guess it. This method is necessary because creating a signature from a public key requires combining the data and hashing each public key sequentially, which allows an attacker to obtain the public key[18][19]. Since the signature can be constructed if the public key has been obtained, the attacker can easily calculate the signature from the public key. To overcome this problem, MHORST randomizes the public key (see Table 1) is necessary so the attacker cannot build a signature. The published public key is generated by randomizing the public key based on seeds generated by Mersenne Twister (see Algorithm 1). The result is shown in Table 2.

Table 1 Seed-based public key resulted by MHORST

Seed	Public Key
69	71ee45a3c0db9a9865f7313dd3372cf60dca6479d46261f3542eb9346e4a04d6
120	802b906a18591ead8a6dd809b262ace4c65c16e89764c40ae326cfcff811e10c
188	5aa7dc47cd2b4322ea163252f7ac0e0799e276a0e6cc649782487876ca407f91
11	725a5987ce617e8a24d0f89c4ce8176d92ec56c0cde164896413f9cbcc8e3b80
49	ed536c62f2c313a1f4906599ae2b17db8d94ffbca7fc2cc2dd54f5fd1a3f39e8
115	2f309c8228f9e2797659de4569374e30fae23c6a3656c25c11698363b07d22b9

2.6 Verification Process

The verification process will involve three processes: generating seeds, obtaining public keys from published public keys, and verifying signatures.

2.6.1 Seed Generation

In this process, the receiver conducted seed generation using PRNG (Mersenne twister) so that it can be assumed that the seed result of 256 numbers obtained by the receiver is the same as the seed result created by the sender. For example [69,120,188,11,49,115,101,117,71,28,190,239,77,197,140, 117,78,31,174,210,241,207,231,24,…,170,62,169]

2.6.2 Obtaining Public Key from Published Public Key

This process aims to return the published public key to its original public key. It is conducted by sorting the seeds obtained in step 2.2.3. from smallest to largest. The published public key will be arranged based on the sorted seeds. Next, the seeds will be returned to their original sorted position. Then, the published public key with the corresponding seed will be returned to the public key order based on the original seed position. For example, in Table 2, the seed is 11, while the randomized published public key is shown in the column published public key.

Table 2 Randomized published public keys as a result of randomization using Mersenne Twister

Seed	Published Public Key
11	725a5987ce617e8a24d0f89c4ce8176d92ec56c0cde164896413f9cbcc8e3b80
49	ed536c62f2c313a1f4906599ae2b17db8d94ffbca7fc2cc2dd54f5fd1a3f39e8
69	71ee45a3c0db9a9865f7313dd3372cf60dca6479d46261f3542eb9346e4a04d6
115	2f309c8228f9e2797659de4569374e30fae23c6a3656c25c11698363b07d22b9
120	802b906a18591ead8a6dd809b262ace4c65c16e89764c40ae326cfcff811e10c
188	5aa7dc47cd2b4322ea163252f7ac0e0799e276a0e6cc649782487876ca407f91

2.6.3 Signature Verification

The verification process uses the public key data sorted in the published public key verification process. The first data, namely 71ee45a3c0db9a9865f7313dd3372cf60dca6479d46261f354 2eb9346e4a04d6, and the second data, namely 802b906a18591ead8a6dd809b2 62ace4c65c16e89764c40ae326cfc40ae326cf811e10c, the results of the public key arrangement are then combined and processed using SHA-256. Next, the data is combined with the third-order public key data, namely 5aa7dc47cd2b4322ea163252f7ac0e0799e276a0e6cc649782487876ca407f91, and processed using SHA-256 until all the last public keys are processed. This process will continue until a combined data of all public keys processed using SHA-256 is obtained. The results of this data will then

p-ISSN: 2528-1682
e-ISSN: 2527-9165

be matched with the signature data. If it is the same, the signature is considered valid, and if it is not, the signature is considered invalid.

3. RESULT AND DISCUSSION

This section describes the experimental results for each module and the analysis. The experiment has two purposes. First, it evaluates the time execution of the three methods, namely HORST, HORSIC, and MHORST, by calculating the time required at the key generation, signing, and verification stages. Furthermore, the analysis was conducted to assess the security level of the three methods.

3.1 Experiment Result

This subsection discusses the experimental results and analysis of the HORST, HORSIC, and MHORST methods.

3.1.1 Key Generation

Based on the results of 30 experiments with different private keys, the HORST method's key generation time ranges from 0.00662 ms to 0.02397 ms. Meanwhile, the HORSIC method requires a longer time, between 0.09764 ms and 0.20331 ms. Using the MHORST method, the time required ranges from 0.00850 ms to 0.03863 ms. HORSIC takes the highest key generation time among these three methods, while HORST takes the fastest. Based on the results shown in Figure 2, there is a significant difference in public key generation time between the HORSIC method with HORST and MHORST. In contrast, the difference between HORST and MHORST is not significant. This condition occurs because the HORSIC method performs 256 SHA-256 processes for 256 public key data, while the HORST and MHORST methods only require one SHA-256 process for the same amount of data. Thus, the HORST and MHORST methods are more optimal for public key generation compared to HORSIC.

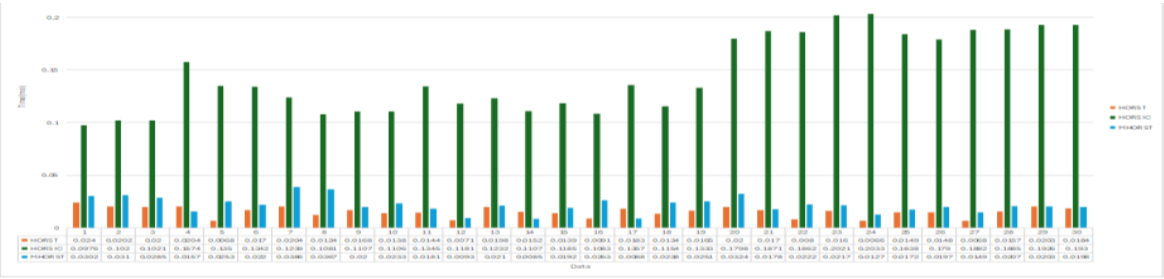

Figure 2. Execution time of the key generation process

3.1.2 Signing

The results of signature generation time for the three methods, namely HORST, HORSIC, and MHORST, show significant differences. MHORST is the fastest method, with less signing times ranging from 0.0008 to 0.00576 milliseconds. Although slower than MHORST, HORST performs well compared with HORSIC, with signing times between 0.00509 to 0.01915 milliseconds. In other words, HORSIC has the longest signing time, ranging from 0.01845 to 0.04059 milliseconds, indicating that it is less efficient than the other two methods. The research results show a significant difference in signature generation time between the HORSIC, HORST, and MHORST methods. HORSIC shows a much longer signature generation time compared to HORST and MHORST. This difference is caused by the indexing process in HORSIC, which must generate a unique index value for each entry, thus increasing the overall processing time. In HORST and MHORST, no unique index is required, so the time required for indexing is relatively lower than when using the HORSIC method.

In addition to the time difference caused by indexing, the signature generation process in HORSIC requires 256 minus n SHA-256 runs, where n is the resultant integer composition value of the message. In contrast, the HORST and MHORST methods require only one SHA-256 run for signature generation. Thus, additional time is required for signature generation in addition to the time for index

generation. Based on the explanation above, the time needed to create a signature using the HORSIC method will be much greater than the time needed to create a signature using HORST and MHORST.

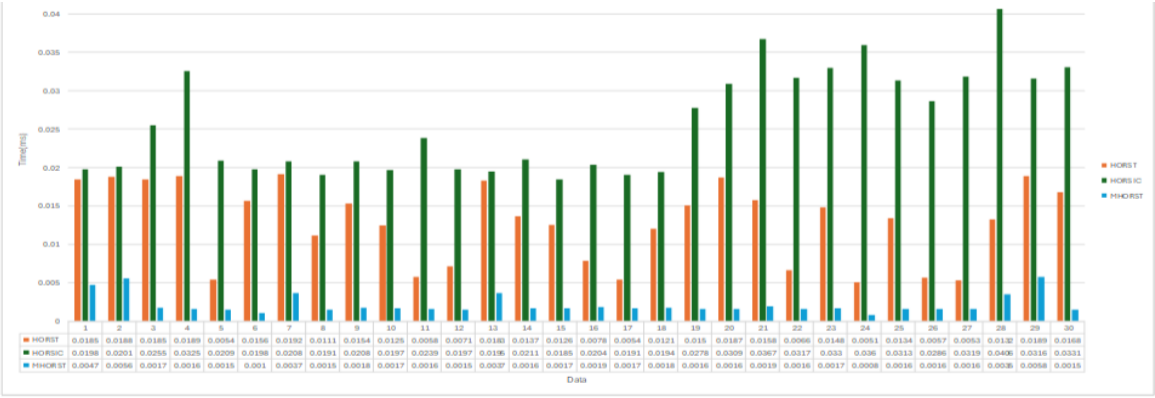

Figure 3 Execution time of signing process

3.1.3 Verification

The results of the three methods, HORST, HORSIC, and MHORST, based on the time taken for signature verification (in milliseconds), show significant performance differences. The HORST method shows efficient verification performance with times ranging from 0.000074 to 0.000211 milliseconds. In contrast, HORSIC shows much slower verification times, ranging from 0.00131 to 0.017383 milliseconds, making it less efficient than HORST and MHORST. MHORST also shows short verification times, similar to HORST, ranging from 0.000067 to 0.000097 milliseconds. In fact, in some cases, MHORST shows a shorter verification time compared to HORST, thus making it a highly efficient algorithm for digital signature verification.

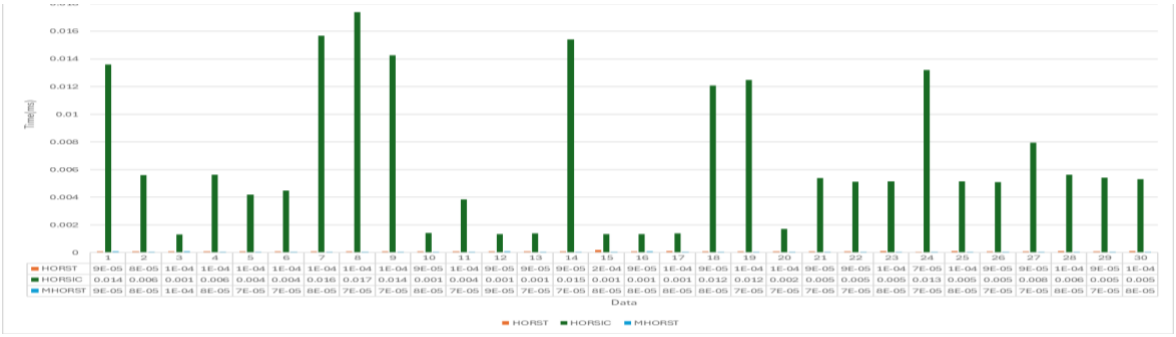

Figure 4 Execution time of the verification process

Based on the signature verification time results, the MHORST method proved to be the most superior, with the shortest key generation and signing time. This condition is due to the reduction in the number of SHA-256 rounds and optimization in public key randomization. In contrast, HORSIC takes the most time due to the repetitive SHA-256 process. Although faster than HORSIC, the HORST method still requires more verification time due to the large public key length. Due to the public key length (about 8192 bytes), HORST needs more time for the verification process.

3.2.2 Security Analysis

This section discusses the security analysis of successful guessing attacks on private keys and signatures in HORST, HORSIC, and MHORST.

3.2.2.1 Security Analysis of Successful Brute Force Attack on Private Keys

One attack method on private keys is brute force, in which the attacker tries every possible secret key until they find the right one. This section discusses the security analysis of private keys in HORST, HORSIC, and MHORST methods against brute force attacks.

p-ISSN: 2528-1682
e-ISSN: 2527-9165

a. Security Analysis of Private Key Against Brute Force Attack in HORSIC

Based on the analysis result, security level usually refers to how difficult it is for an attacker to crack a system successfully. In this case, if a guessing/brute force attack is used, then the formula for the probability of success in HORSIC is shown in Equation (1).

$$\text{The probability of success private key guessing in HORSIC} = \prod_{i=1}^{m} \left(\frac{1}{2^{(m-i)+1}}\right) \quad (1)$$

where *m* is the number of unique values, and *i* is the iteration. Since the value of $2^{(m-i)+1}$ indicates that the chance of getting identical keys is minimal, this guarantees the security level in HORSIC. Suppose m = 6, then the probability of success private key guessing in HORSIC based on equation (1) is as follows: the probability of success private key guessing in HORSIC = $\frac{1}{1} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4} \times \frac{1}{5} \times \frac{1}{6} = \frac{1}{720}$

b. Security Analysis of Private Key in HORST and MHORST

Based on the analysis result, the security level of the HORST and MHORST methods in terms of private key guessing differs from that of HORSIC because each index does not need to be unique. Thus, the probability of success in private key guessing in HORST and MHORST can be calculated using Equation (2).

$$\text{The probability of success private key guessing in HORST and MHORST} = \prod_{i=1}^{m} \left(\frac{1}{(m-r_{(m-i)+1})+1}\right)^{l_{(m-i)+1}} \quad (2)$$

where *m* is the total number of unique values. *i* is the iteration, $r_{(m-i)+1}$ is the sequence of unique values, $l_{(m-i)+1}$ is the number of unique values for each m. This equation illustrates that the security level of a private key depends on the number of unique elements in each iteration. The more unique values, the smaller the probability of private key guessing. This condition means that the security level of the private key in the HORST and MHORST methods increases as the value of m increases. For example, six data indexes exist (4, 3, 5, 3, 3, 4). The same value is found in the data and it becomes (3, 3, 3, 4, 4, 5) when sorted. By referring to Equation (2), the equations for HORST and MHORST are obtained: The probability of success private key guessing in HORST and MHORST = $\frac{1}{3} \times \frac{1}{3} \times \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{1}$ = $(\frac{1}{3})^3 (\frac{1}{2})^2 \frac{1}{1} = \frac{1}{9} \times \frac{1}{4} \times \frac{1}{1} = \frac{1}{36}$

3.2.2.2 Security Analysis Against Signature Brute Force Attack

Signatures are important in maintaining message integrity and authentication in cryptographic systems. The security of a signature depends on its complexity and the hashing algorithm used. This section will discuss the security level of HORST, MHORST, and HORSIC methods in the context of signature guessing.

a. Signature guessing analysis for HORST

Based on the analysis results, the security level of the HORST method in terms of signature guessing is equal to 1. This condition means that the signature can be guessed with a high probability because the signature is part of the public key taken from the Merkle tree leaves and combined without going through additional steps to increase security. Thus, it makes the security of the HORST method vulnerable in this aspect, significantly if the number of signatures distributed is increasing, which makes it easier for an attacker to guess the signature through public key analysis.

b. Signature guessing analysis for MHORST

In MHORST, the public key is obtained by selecting nodes or leaves of the Merkle tree. Based on equation 1, with *d* = 256 nodes, resulting in 8 levels. The number of public keys (*n*) used ranges between max-*n* and min-*n,* where *n*=max-*n* is used when all nodes at the lowest level are used to generate the public key, while *n*=min *n* is used if one node is used in one level is used to generate the public key. If all

nodes are used, n = 256, it means that 256 nodes will be processed, which causes a higher processing time than the processing time for an n=min-n value. The selection of min-n needs to be adjusted to the number of levels to ensure that the path from the leaf to the root is not manipulated. This process ensures the integrity of the initial leaf to the root. In this case, the probability of guessing the public key in MHORST depends on the number of public keys, as shown in Equation 3:

$$\text{Probability of Public Key guessing} = 1/(n+1)! \quad (3)$$

Using public key randomization makes it difficult for an attacker to guess it because the permutation of n means the probability of public key guessing becomes very small. Since the probability for signature guessing depends on the probability of public key guessing, the MHORST signature guessing probability can be calculated using Equation (4).

$$\text{MHORST Signature Guessing Probability} = 1/(n+1)! \quad (4)$$

The parameter n represents the total number of public key, which means the number of key permutations is n factorial. Thus, the probability of guessing the public key from the published public key is equal to $1/(n+1)!$ or it means that the probability of forging a signature is $1/(n+1)!$, which is less than the probability of creating a signature using HORST. Suppose n (the total number of the public key) is 8, then based on equation (4), the MHORST Signature Guessing Probability equals 1/9!

c. Signature guessing analysis for HORSIC

Equation (5) shows the security level of the HORSIC method in terms of signature key guessing based on the analysis results.

$$\text{HORSIC Signature guessing probability} = (1 / (2^{64})^{256-c} \quad (5)$$

$\left(1/_{2^{64}}\right)$ is the maximum length of data processed using SHA-256, where the maximum length of a message is 2^{64}– bits. 256 is the number of SHA-256 processes, and c is a compositional integer value. So, this equation shows that the probability of guessing the HORSIC signature is small, providing a very high level of security. Based on the discussion about signature guessing in HORST, HORSIC, and MHORST, it can be concluded that the probability of signature guessing in MHORST is less than in HORST but greater than in HORSIC.

4. CONCLUSION

In this research, the main problem with the HORST method is that it is easy to guess the signature since the public key is generated from the concatenation of all signatures. This problem has been overcome by the HORSIC method. However, the execution time increased due to the added complexity of integer composition and repetitive SHA-256 processing. The MHORST method is proposed to overcome HORST's vulnerability and optimize HORSIC's execution time. The first contribution of MHORST is reducing the execution time using SHA-256 and the Merkle tree. The second contribution is decreasing the probability of signature guessing by randomizing the public key using the Mersenne Twister.

Based on the experiment result, it is proven that the MHORST method can overcome the problem of HORST because the probability of signature guessing in MHORST is less than in HORST. The MHORST method overcomes the problem of the HORSIC method, which is the high execution time. The problem of HORSIC is overcome by MHORST using Merkle trees and the SHA-256 process to reduce the execution time. Based on the experiment results, the execution times of the key generation, signing, and verification are faster than those of HORSIC. Although the probability of signature guessing in MHORST is greater compared to HORSIC, the MHORST method is still better than HORST. Based on the analysis result, the suggestion for further research is to reduce the execution time on key generation in the MHORST method.

REFERENCES

[1] V. Srivastava, A. Baksi, and S. K. Debnath, "An overview of hash-based signatures," *National Institute of Technology Jamshedpur & Nanyang Technological University, Singapore*, 2020.

[2] F. Shahid and A. Khan, “Smart Digital Signatures (SDS): A post-quantum digital signature scheme for distributed ledgers,” *Future Generation Computer Systems*, vol. 111, pp. 241–253, Oct. 2020, doi: 10.1016/j.future.2020.04.042.
[3] L. Li, X. Lu, and K. Wang, “Hash-based signature revisited,” Jul. 01, 2022, *Springer Science and Business Media B.V.* doi: http://dx.doi.org/10.1186/s42400-022-00117-w.
[4] E. Fathalla and M. Azab, "Beyond classical cryptography: A systematic review of post-quantum hash-based signature schemes, security, and optimizations," *IEEE Access*, vol. 12, pp. 175969-175986, Dec. 2024, doi: 10.1109/ACCESS.2024.3485602.
[5] P. Lafrance, *Digital Signature Schemes Based on Hash Functions*, M.Math. thesis, Univ. of Waterloo, Waterloo, ON, Canada, 2017.
[6] K. Zhang, H. Cui, and Y. Yu, "SPHINCS-α: A compact stateless hash-based signature scheme," *IACR Cryptology ePrint Archive*, Jun. 2023.
[7] J. Lee and Y. Park, “HORSIC+: An efficient post-quantum few-time signature scheme,” *Applied Sciences (Switzerland)*, vol. 11, no. 16, Aug. 2021, doi: 10.3390/app11167350.
[8] R. Hu, “Visual Blockchain Using Merkle Tree,” Thesis, 2019. Accessed: Sep. 16, 2024. [Online]. Available: https://hdl.handle.net/10292/12529
[9] W. Wirachantika, A. M Barmawi, and B. A. Wahyudi, “Memperkuat Fawkescoin Melawan Serangan Pembelanjaan Ganda Menggunakan Merkle Tree,” pp. 49–54, Jan. 2019, doi: https://doi.org/10.1145/3309074.3309105.
[10] J. P. Aumasson and G. Endignoux, “Improving stateless hash-based signatures,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 2018, pp. 219–242. doi: 10.1007/978-3-319-76953-0_12.
[11] M. Alegri, "Identities involving sum of divisors, integer partitions and compositions," *Online Journal of Analytic Combinatorics*, Feb. 2024, doi: 10.61091/ojac-1703.
[12] Q. He, “Improved Algorithms for Integer Complexity,” Department of Computer Science, University of Illinois at Urbana-Champaign, Sep. 2023.
[13] K. Algazy, K. Sakan, A. Khompysh, and D. Dyusenbayev, "Development of a new post-quantum digital signature algorithm: Syrga-1," *Computers*, vol. 13, no. 26, Jan. 2024, doi: 10.3390/computers13010026.
[14] B. U. I. Khan, R. F. Olanrewaju, M. A. Morshidi, R. N. Mir, M. L. M. Kiah, and A. M. Khan, "Evolution and analysis of secure hash algorithm (SHA) family," *Malaysian Journal of Computer Science*, vol. 35, no. 3, pp. 179-200, 2022, doi: 10.22452/mjcs.vol35no3.1
[15] S. Bouaziz–Ermann, A. B. Grilo, and D. Vergnaud, "Quantum security of subset cover problems," *LIP6, Sorbonne Université*, 2023
[16] M. Yehia, R. Altawy, and T. A. Gulliver, “Hash-based Signatures Revisited: A Dynamic FORS with Adaptive Chosen Message Security,” A. Nitaj and A. Youssef, Eds., Springer International Publishing, Jul. 2020, pp. 239–257. doi: https://doi.org/10.1007/978-3-030-51938-4_12.
[17] J.-P. Aumasson and G. Endignoux, “Clarifying the subset-resilience problem,” Kudelski Security, Switzerland, 2017.
[18] H. Li, Y.-M. Xie, X.-Y. Cao, C.-L. Li, Y. Fu, H.-L. Yin, and Z.-B. Chen, “One-Time Universal Hashing Quantum Digital Signatures without Perfect Keys,” National Laboratory of Solid State Microstructures and School of Physics, Nanjing University, Oct. 2023
[19] A. Shafarenko, “Winternitz Stack Protocols for Embedded Systems and IoT,” Cybersecurity, vol. 7, no. 34, 2024. doi: 10.1186/s42400-024-00225-9
[20] N. Boutsioukis, "Comparative analysis of pseudorandom number generators: Mersenne Twister, middle square method, and linear congruential generator through Dieharder tests," *SSRN*, Jan. 2023, doi: 10.4761542.