ISSN 2007-9737

Processing Time Optimization for XMSS on an Object Oriented SPHINCS+ Implementation

Rafael Soto Landa, Octavio Ortiz Ortiz, Juan Manuel García García

Instituto Tecnológico de Morelia,
Tecnológico Nacional de México,
México

ranfael@hotmail.com, {octavioortiz, jmgarcia}@itmorelia.edu.mx

Abstract. Computation paradigms are changing due to the idea of quantum computers. Although today this kind of technology is not available and has not reached its full potential, public-key cryptography is at risk since its security depends on problems that classical computers cannot solve in polynomial time, while quantum computers can. Quantum-resistant cryptography has been developed for years, and this work proposes a new implementation that reduces the processing time of XMSS signature generation of SPHINCS+, a stateless scheme designed for digital signature and is in the process of standardization by the National Institute of Standards and Technology of the United States. The proposed implementation has a tradeoff between time and memory space. While original documentation is focused on space optimization, this new implementation needs an additional 20 kB of memory for each tree during the signing process in order to generate a signature but optimizes the time the internal process requires in the authentication path generation, from exponential to linear complexity. Digital signatures generated by both ways are the same.

Keywords. Digital signatures, hash-based scheme, post-quantum cryptography, public-key cryptography.

1 Introduction

The quantum supremacy race is a matter of fact and has serious contenders like Google [1], IBM [18], and governments around the world [7].

Although it has not been possible to build large-scale quantum computers yet, quantum algorithms already exist, like Shor's [19] and Grover's [8] that have the potential of breaking public-key cryptography through the resolution of integer factorization, discrete logarithm, and elliptic curve discrete logarithm problems in polynomial time and the quadratic speed-up on database searches [21].

Digital signature schemes represent an important part of public-key cryptography used on the internet since they provide authenticity, integrity, and non-repudiation services [6]. Current digital signature schemes are vulnerable to quantum attacks as their security lies on the lack of feasibility to solve problems, such as listed before, on a classical computer.

Post-quantum cryptography is a recently created study field that researches cryptographic schemes that are resistant to known quantum attacks and can be implemented on classical computers. Several post-quantum schemes have been generated in the last few years and are categorized in hash-based, code-based, lattice-based, and multivariate-quadratic [3]. The security level of code-based, lattice-based, and multivariate-quadratic schemes is still uncertain because they have not been researched enough, and there are no known attacks for them [20]. The National Institute of Standards and Technology of the United States works in evaluating post-quantum cryptography schemes [16].

On the other hand, Hash-based schemes have more certainty in their security levels, which lie mainly on the security of their underlying hash functions, and standardized hash functions have been tested for unpredictability, first and second preimage attacks resistance, and collision resistance [17]. Thus, if a standardized hash function is used as the base of a scheme, it is just needed to prove the higher part of such scheme. Within hash-based schemes, there are two main classes: stateful and stateless; the first class

requires more secure implementation assumptions but shows better performance than the second class.

This work is focused on SPHINCS+, a stateless hash-based scheme for digital signatures that is in the process of standardization by the NIST [2], and more specifically in XMSS, a scheme implemented inside SPHINCS+.

At the moment of this writing, no complexity analysis of implementations other than those shown in this work were found.

2 SPHINCS+

SPHINCS+ [2] is the evolution of the original SPHINCS (Stateless Practical Hash-based Incredibly Nice Cryptographic Signatures) [4], submitted to the NIST in 2019 for standardization. The scheme consists of the orchestration of the other four nested schemes, as shown in Fig. 1. Through the signing and verification processes, these schemes call for a family of tweakable hash functions that share the same underlying standard hash function, that must have second preimage resistance.

2.1 WOTS+

The base scheme is WOTS+ [10], the Winternitz One-Time Signature is an extension of Lamport one-time signature [14] used to sign a fixed-length group of bytes through a recursive chaining function that calls the tweakable hash function **F**, and a checksum with the purpose of making the signature existentially unforgeable [5].

At the highest level, this scheme has four routines:

– A secret key generation that calls the tweakable hash function **PRF**.
– Public key generation, which calls the chaining function.
– Sign, which calls the chaining function. A signature is a bidimensional array of bytes, as described in [10].
– Public key generation from a signature, which calls chaining function.

I.e., in the context of SPHINCS+, WOTS+ has not a direct signature verification routine, but a way to compute the public key from a given signature and a message; if the generated public key is the known public key of the sender, the signature is valid.

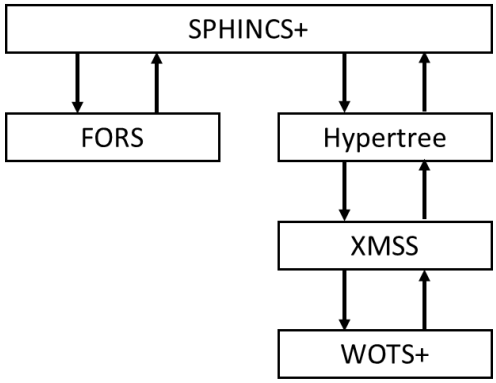

Fig. 1. Inner schemes of SPHINCS+

2.2 XMSS^{MT}

XMSS is a scheme proposed in [11] and analyzed in [12] with the goal of generating parameter sets suitable for different implementations.

XMSS is an eXtended Merkle Signature Scheme and takes from Ralph Merkle [15] the idea of a tree made of hashes. MT means multi-tree.

For a single tree, the leaves are WOTS+ public keys, i.e., in the context of SPHINCS+, XMSS instances sign WOTS+ public keys instead of actual messages and use the WOTS+ signing routine; hence the message must be a fixed-length group of bytes, but it is possible to sign more messages, according to the number of leaves.

2.2.1 Treehash Routine

A tree of height h is generated from h^2 WOTS+ public keys. The nodes in each level are computed using a bitmask on the concatenation of the hashes contained in the nodes below and applying the tweakable hash function **H** as shown in Fig. 2, taken from [13]. This routine is known as *treehash* and is detailed in [2]. The public key of an XMSS instance is the root node of the tree.

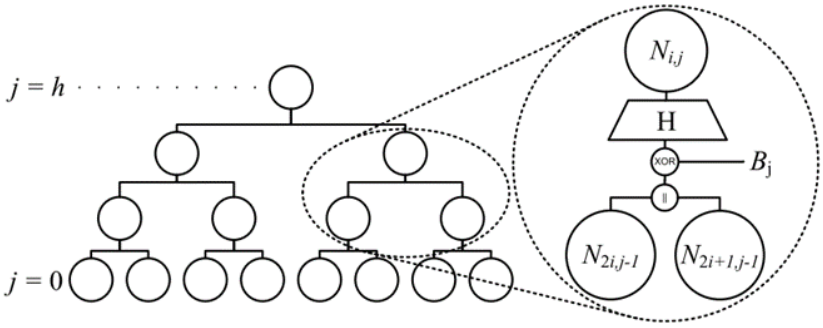

Fig. 2. XMSS nodes generation

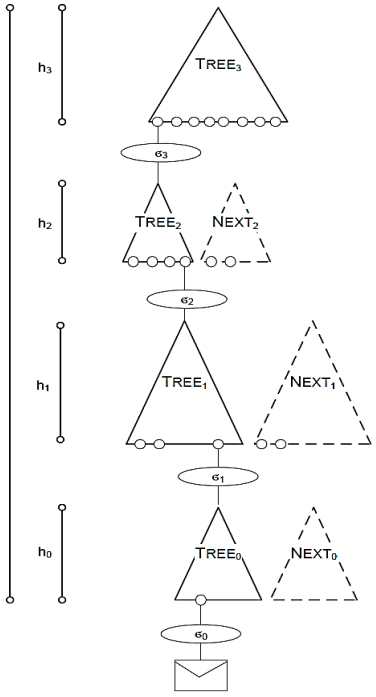

Fig. 2. SPHINCS+ hypertree

2.2.2 Signing Routine

An XMSS signature consists of a selected leaf, i.e., a WOTS+ signature, and the authentication path, which is the set of hashes of the intermediate nodes needed to generate the root node, which is the public key of the instance.

In [9], it is recommended to use the treehash routine to generate the authentication path, and in [2], it is used in this way. This method is memory-efficient but also time-expensive since it is necessary to regenerate the entire XMSS tree.

2.2.3 Signature Verification

XMSS signature verification follows the same approach as WOTS+ because there is not a direct verification method but a public key generation method that needs the signature (WOTS+

signature and authentication path) and the message. If the public key generated by this method matches the known sender's public key, the signature is valid.

2.3 Hypertree

An instance of XMSS can sign up to 2^h messages so, the greater h is the more messages that can be signed, increasing the execution time of the treehash routine.

The processing time growth is circumvented, splitting a big tree into layers of certification trees. The trees on the lower level sign WOTS+ public keys, and the trees in higher levels sign the root node of each tree below. This construction is known as hypertree, see Fig. 2, taken from [13], and gives the *MT* to XMSS^{MT}.

The hypertree scheme constructs intermediate layers of trees. This method is more time-efficient than generating a tree with the total height of the hypertree.

As XMSS instances, the hypertree has routines for public key generation (the higher root node) and signature generation, but this scheme does have an explicit signature verification routine.

2.4 FORS

So far, there is a need to maintain a state, so the FORS scheme is implemented to make SPHINCS+ stateless.

FORS is the acronym for Forest Of Random Subsets and is the successor of HORS and HORST schemes present in the previous versions of SPHINCS. All of these are few-time signature schemes used to sign at random leaves, so the SPHINCS+ key pair may be used several times without degrading security.

FORS scheme has routines for private key generation, public key generation, signature generation, and public key generation from a signature. As XMSS, it uses its own version of the treehash routine.

2.5 SPHINCS+ Orchestration

SPHINCS+ is the higher scheme and is used for the orchestration of the other schemes.

It has routines for key pair generation, signature generation, and signature verification.

SPHINCS+ generates a FORS instance and adds randomness to it for signing the original message. Then signs the FORS public key with an instance of the hypertree to get the SPHINCS+ signature.

3 Object-Oriented Implementation

This work is about an object-oriented implementation of SPHINCS+ that aims to optimize the processing time of the original one. In addition, the fact of working with objects facilitates the understanding, scalability, and maintenance of code.

Each scheme can be instantiated as an object of the class with its name and makes use of related supplementary classes.

Some schemes have common functionalities, i.e., generic routines that are organized in the classes:

- Hash: Contains every tweakable hash function used by the schemes. If a change on the underlying hash function is needed, like the implementation of new standard digest functions, this is the single place to make the modifications.
- Utils: It has the byte operations needed in the schemes, like number base changes, XOR, array comparisons, and the like.
- Address: along the schemes implemented by SPHINCS+, it is necessary to maintain indexes of leaves, trees, and chains. These indexes are stored in byte arrays called addresses.
- Random: Used for the generation of pseudorandom bytes.

As this paper is focused on XMSS signing optimization, the relevant structs will be detailed in the following section.

3.1 XMSS Objects

As XMSS consists of a tree, there is a struct for nodes. Since the tree is for signing messages and verifying signatures, there is a struct for signatures; both are described below.

3.1.1 TreeNode Struct

An XMSS node is a struct with three relevant values:

– **hash:** The hash is a fixed-length byte array that may contain a WOTS+ public key for leaf nodes or the node construction shown in section 2.2.1 Treehash Routine for intermediate and root nodes.
– **height:** It is an unsigned integer number that represents the vertical position of the node inside the tree. Leaf nodes have a zero (0) value, and the root node has a value h.
– **index:** The horizontal position of the node within a level is represented by this unsigned integer number. The leftmost node of each level has a zero (0) index.

3.1.2 XmssSignature Struct

As described in section 2.2.2 Signing Routine, a signature is the bidimensional byte array that represents the WOTS+ signature on the selected leaf and the authentication path from that leaf to the root node. In Fig. 3., taken from [13], the WOTS+ signature is represented with i, and the dark nodes are conforming the authentication path. In this context, the authentication path is a list of the hashes generated in the construction of TreeNode objects.

Therefore, this struct has:

– **authentication_path:** This is a list of byte arrays that represent the hashes of the TreeNode objects. It could be an array too.
– **wots_signature:** Bidimensional byte array.

3.2 XMSS Variant

In [2], it is recommended to generate the authentication path by using the Treehash method directly, although this routine was already used before to generate the XMSS public key.

The suggested way to sign is the best-known solution to improve space performance since it does not store in memory every node but the currently needed hash.

The problem with using Treehash again at this point is that it is the heaviest time processing routine in XMSS because it must generate every single node again.

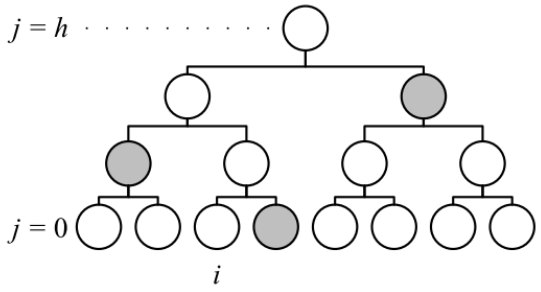

Fig. 3. XmssSingature struct construction

The solution proposed in this work has the opposite tradeoff, less processing time at the expense of more memory to do the same task. This decision is based on the current computers' capabilities and the need for immediacy when providing a signing service.

Tests on an Intel® Core™ i7-8550U processor at 1.99 GHz, signing and verification time was reduced from about 5.2 seconds to 2.7 seconds on average for trees of height 8, with $n = 32$ and $w = 16$ as shown in section *5 Implementations Testing*. The complexity analysis is in the next section, where the tradeoff is formally demonstrated.

To get the proposed improvement it was necessary to make changes in the scheme implementation. These changes are described in the next three subsections.

3.2.1 Treehash Routine

The original idea of the Treehash routine was preserved, but a *hashtree* argument was included and filled inside the routine, as shown in Algorithm 1. The algorithm was taken from [2], and the added elements are distinguished in bold letters.

3.2.2 TreePathGeneration Routine

TreePathGeneration (tree_path_generation in pseudocode, see Algorithm 2) is a routine added to the original scheme. This algorithm computes the index of all the nodes that must be appended to the authentication path on each level of the XMSS tree.

Algorithm 1. Treehash algorithm

```
# Input: Secret seed SK.seed, start index
s, target node height z, public seed
PK.seed, address ADRS, list of XmssNode
elements hashtree
# Output: n-byte root node - top node on
Stack

treehash(SK.seed, s, z, PK.seed, ADRS,
  hashtree)
{
  if (s % (1 << z) != 0 ) return -1;
  for (i = 0; i < 2^z; i++)
  {
    ADRS.setType(WOTS_HASH);
    ADRS.setKeyPairAddress(s + i);
    node = wots_PKgen(SK.seed, PK.seed,
      ADRS);
    ADRS.setType(TREE);
    ADRS.setTreeHeight(1);
    ADRS.setTreeIndex(s + i);
    hashtree.append(node);
    while (Top node on Stack has same
      height as node)
    {
      ADRS.setTreeIndex(
        (ADRS.getTreeIndex() – 1) / 2);
      node = H(PK.seed, ADRS,
        (Stack.pop() || node));
      ADRS.setTreeHeight(
        ADRS.getTreeHeight() + 1);
      hashtree.append(node);
    }
    Stack.push(node);
  }
  return Stack.pop();
}
```

Algorithm 2. TreePathGeneration algorithm

```
# Input: Tree height z, index of leaf
selected for WOTS+ signature idx
# Output: Stack of integers that represent
the index of the nodes in the path

tree_path_generation(z, idx)
{
  path = new Stack();
  hi_bound = 2^z;
  low_bound = 0;
  srch_idx = 0;
  for (i = z; i > 1; i--)
  {
    half = (hi_bound + low_bound) / 2;
    if (idx >= half)
    {
      low_bound = half;
      srch_idx = srch_idx * 2 + 1;
    }
    else
    {
      hi_bound = half;
      srch_idx *= 2;
    }
    path.push((srch_idx % 2 == 0) ?
      srch_idx + 1 : srch_idx - 1);
  }
  path.push((idx % 2 == 0) ?
      idx + 1 : idx - 1);
  return path;
}
```

The routine returns a stack of integers that is used in the signing routine, where each number is popped out from the stack and is used to get the node at the corresponding index and height of the tree.

3.2.3 Signing Routine

The signing routine implementation is different from the original one only in the *build authentication path* section. The logic is in Algprithm 3. It is originally computed by the treehash function. In the new implementation, it is calculated through the hashtree list, and the indexes are returned by the TreePathGeneration routine.

The strikethrough text was removed from the original implementation, while the bold text was added. The values of *z* and *hashtree* are taken from the XMSS object.

4 Complexity Analysis

As mentioned in earlier sections, the aim of this work is to optimize the processing time, and this is done by losing memory efficiency. In the next subsections, there is an analysis of how these two variables in both implementations have been

affected. In this section, z represents the height of the XMSS tree.

4.1 Time

4.1.1 Treehash Routine

Treehash has two new lines for adding every single node to the list hashtree.

There are $2^{z+1}-1$ nodes on a tree, and each assignation has three internal assignations (hash, height, and index). So, this routine has added $3*(2^{z+1}-1)$ operations to the scheme. Since these operations are in the same structure as the rest of the computation of Treehash, they do not increase the complexity of the routine.

4.1.2 TreePathGeneration Routine

TreePathGeneration is new to the scheme, so all its operations count in the total.

There are up to 12 operations inside the for loop that is repeated $z-1$ times. There are $nine$ operations outside the for loop. Therefore, this routine adds $12*(z-1)+9$ operations. The complexity of this routine is linear.

4.1.3 Signing Routine

Original singing routine calls for the original version of Treehash routine, which calls 2^z times wots_PKgen, whose complexity is quadratic because it calls the recursive chain function in a for loop, while new implementation calls for the TreePathGeneration routine, which has linear complexity as it is demonstrated in previous subsections.

In conclusion, by adding those operations to Treehash and TreePathGeneration routines, the number of operations is reduced since it is not necessary to regenerate the entire tree but just go through it.

4.2 Space

New implementation requires storing the list of instances of TreeNode. The list must contain $2^{z+1}-1$ nodes, and each node contains a 32-byte hash, a 4-byte height, and a 4-byte index, giving a total of $(2^{z+1}-1)*40$ bytes.

Algorithm 3. XMSS signing algorithm

```
# Input: n-byte message M, secret seed
SK.seed, index idx, public seed PK.seed,
address ADRS
# Output: XMSS signature SIG_XMSS = (sig ||
AUTH)

xmss_sign(M, SK.seed, idx, PK.seed, ADRS)
{
  path = tree_path_generation(z, idx);
  // build authentication path
  for (j = 0; j < z; j++)
  {
    k = floor(idx / (2^j)) XOR 1;
    AUTH[j] = treehash(SK.seed, k * 2^j,
      j, PK.seed, ADRS);
  }
  for (j = 1; j < z; j++)
  {
    srch_idx = path.pop();
    AUTH[j - 1] = (hashtree node with
      height == j and index == srch_idx);
  }
  ADRS.setType(WOTS_HASH);
  ADRS.setKeyPairAddress(idx);
  sig = wots_sign(M, SK.seed, PK.seed,
    ADRS);
  SIG_XMSS = sig || AUTH;
  return SIG_XMSS;
}
```

The complete SPHINCS+ implementation uses trees of height 8, so the total needed storage for a tree is 20,440 bytes that are not necessary at all in the original implementation.

Note that this use of memory is not persistent since the hashtree object is discarded once the signing processing is finished.

5 Implementations Testing

The proposed solution was implemented besides the original one in a library using C# on .NET Core 2.1 framework and tested on an Intel® Core™ i7-8550U processor at 1.99 GHz. Tests were performed with Xunit.

Two performance scenarios were tested 50 times over both implementations. The first scenario aimed to compare the whole process of generating

Table 1. Time optimizations

Implementation	Tree and signature generation	Authentication path generation
Original	5211.07	2637.85
Object-oriented	2738.29	0.02
Time reduction percentage	**47.45%**	**99.99%**

the XMSS object, generate the public key with the Treehash routine, and to sign a 32-byte message.

The second scenario compares just the authentication path generation inside the signing routine. This subroutine in the optimized implementation assumes that a hashtree object was created at a cost of $3 * (2^{z+1} - 1)$ extra operations, i.e., 1,533 extra operations for implementation with $z = 8$, and does not call for the Treehash routine at all. Table 1 shows the resulting times and time reduction percentage.

6 Conclusions

Current applications need immediacy, and this implementation improves it compared with the original proposed one.

The tradeoff between time and memory space proposed in this work seems affordable for modern computers since the space cost of signing 32 bytes in almost half of the original time is about 20 kB, which are released once the signing process is finished.

Test and complexity analysis of this work demonstrate that Treehash is the time costliest routine in the overall process of XMSS and should be avoided when possible.

XMSS is just a sub-scheme inside SPHINCS+; in higher scheme FORS, there is a version of Treehash and is implemented in a similar way inside the FORS signing process.

Future research could be about the elimination of hashtree objects by implementing the Treehash routine just once and get the signature and the public key in a single run, but this could imply greater coupling between classes and, therefore, less maintenance ease.

Acknowledgments

The authors thank Tecnológico Nacional de México that funded this project and especially the Technologic Institute of Morelia (ITM) for the facilities granted to carry out the investigation.

References

1. **Arute, F., Arya, K., Babbush, R. et al. (2019).** Quantum Supremacy Using a Programmable Superconducting Processor. Nature, Vol. 574, No. 7779. pp. 505–510. DOI: 10.1038/s41586-019-1666-5.
2. **Bernstein, D., Dobraunig, Ch., Schwabe, P., et al. (2019).** SPHINCS+ Submission to the NIST post-quantum project. pp. 1–62.
3. **Bernstein, D.J. (2009)**. Introduction to post-quantum cryptography. In **Bernstein, Buchmann, and Dahmen, (eds.)** Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–14. DOI: 10.1007/978-3-540-88702-7_1.
4. **Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R. (2015).** SPHINCS: Practical stateless hash-based signatures. Advances in Cryptology. EUROCRYPT´15, In **Oswald, Fischlin, (eds.)** Springer Berlin Heidelberg, Vol. 9056, pp. 368–397. DOI: 10.1007/978-3-662-46800-5_15.
5. **Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M. (2011).** On the Security of the Winternitz One-Time Signature Scheme. In **Nitaj A., Pointcheval D. (eds.)** Progress in Cryptology – AFRICACRYPT´11, Lecture Notes in Computer Science, Vol. 6737. DOI:10.1007/978-3-642-21969-6_23.
6. **Buchmann, J., Dahmen, E., Szydlo, M. (2009).** Hash-based Digital Signature Schemes. In **Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.)** Post-Quantum Cryptography. Springer Berlin Heidelberg, pp. 35–39. DOI: 10.1007/978-3-540-88702-7_3.
7. **Griffiths, J. (2019).** The US just moved ahead of China in quantum computing.
8. **Grover, L.K. (1996).** A fast quantum mechanical algorithm for database search. Proceedings of the Annual ACM Symposium

on Theory of Computing, pp. 212–219. DOI: 10.1145/237814.237866.

9. **Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A. (2018).** XMSS: eXtended Merkle Signature Scheme. Vol. 15. DOI: 10.17487/RFC8391.
10. **Hülsing, A. (2013).** W-OTS+ - Shorter signatures for hash-based signature schemes. Lecture Notes in Computer Science, Vol. 7918, LNCS, pp. 173–188. DOI: 10.1007/978-3-642-38553-7-10.
11. **Hülsing, A., Busold, C., Buchmann, J. (2013).** Forward Secure Signatures on Smart Cards. **In Knudsen L.R., Wu H. (eds.)** Selected Areas in Cryptography. SAC´12, Lecture Notes in Computer Science, Vol. 7707, pp. 66–80. DOI: 10.1007/978-3-642-35999-6_5.
12. **Hülsing, A., Rausch, L., Buchmann, J. (2013).** Optimal Parameters for XMSS MT in Security Engineering and Intelligence Informatics, pp. 194–208. DOI: 10.1007/978-3-642-40588-4_14.
13. **Hülsing, A., Rausch, L., Buchmann, J. (2013).** Optimal Parameters for XMSS MT. International Conference on Availability, Reliability, and security, pp. 194–208.
14. **Lamport, L. (1979).** Constructing Digital Signatures from a One Way Function. Computer Science Labotatory, SRI International, pp. 1–7.
15. **Merkle, R.C. (1979).** A certified digital signature. Advances in Cryptology - CRYPTO' 89, Lecture Notes in Computer Science book series (LNCS, volume 435), Springer.
16. **National Institute of Standards and Technology (2019).** Post-Quantum Cryptography - Workshops and Timeline.
17. **National Institute of Standards and Technology (2015).** SHA-3 Standard: Permutation-based hash and extendable-output functions. DOI: 10.6028/NIST.FIPS.202.
18. **Pednault, E., Gunnels, J., Maslov, D., Gambetta, J. (2019).** On Quantum Supremacy. IBM Research Blog.
19. **Shor. P.W. (1997).** Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing, Society for Industrial & Applied Mathematics, Vol. 5, pp. 1484–1509. DOI: 10.1007/3-540-58691-1_68.
20. **Sjöberg, M. (2017).** Post-quantum algorithms for digital signing in Public Key Infrastructures. KTH Royal Institute of Technology.
21. **Yan, SY. (2015).** Classical and quantum computation, in quantum computational number theory. Cham: Springer International Publishing, pp. 33–58. DOI: 10.1007/978-3-319-25823-2_2.

Article received on 13/01/2020; accepted on 06/11/2020. Corresponding author is Rafael Soto Landa.