High Performance of Hash-based Signature Schemes

Ana Karina D. S. de Oliveira
FACOM-UFMS
Federal University of Mato Grosso do Sul
MS, Brazil

Julio López
IC-UNICAMP
University of Campinas
SP, Brazil

Roberto Cabral
UFC-CRATEÚS
Federal University of Ceará
CE, Brazil

***Abstract*—Hash-based signature schemes, whose security is based on properties of the underlying hash functions, are promising candidates to be quantum-safe digital signatures schemes. In this work, we present a software implementation of two recent standard proposals for hash-based signature schemes, Leighton and Micali Signature (LMS) scheme and Extended Merkle Signature Scheme (XMSS), using a set of AVX2 instructions on Intel processors. The implementation uses several optimization techniques for speeding up the underlying hash functions SHA2 or SHA3, and other building block functions which lead to high performance for signature operations on both schemes. On an Intel Skylake processor, using a tree of height 60 with 12 layers, the signing operation for XMSS takes 3,841,199 cycles (1,043 signatures per second) at 128-bit security level (against quantum attacks). For an equivalent security, the LMS system computes a signature in 1,307,376 cycles (3,065 signatures per second). We also provide the first comparative performance results for signing and verification of both schemes using different parameters. The results of our implementation indicate that both schemes LMS and XMSS can achieve high performance using vector instructions on modern processors.**

***Keywords*—post-quantum cryptography; digital signature; Merkle signature; LMS; XMSS**

I. Introduction

A digital signature scheme is an important cryptographic tool for public-key cryptography. Digital signature scheme are widely used for providing authenticity, integrity, and non-repudiation of data. Nowadays, the most commonly used digital signature schemes are ECDSA [1], RSA [2] and DSA [3]. These schemes have their security based on the difficulty of factoring large integers or computing discrete logarithms. In [4], Shor introduced a polynomial-time quantum algorithm for factoring and computing discrete logarithms. Thus, digital signature schemes that can resist an attack by quantum computers are an active area of research.

A One-Time Signature (OTS) scheme allows using a key pair to sign exactly one message [5]. These schemes are inadequate for the most practical situations since each key pair is used only for a single signature. In [6], Merkle proposed N-Time Signature (NTS), that are built out of one-time signature schemes. The Merkle Signature Scheme (MSS) makes one-time signatures practical by combining $N = 2^h$ of them in a single structure, which is a complete binary tree of height h. These systems have regained interest since 2006 because of their resistance against quantum-computer-aided attacks. Since the security of these schemes is based on the underlying cryptographic hash function, they are called hash-based signature schemes.

Independently of the actual realization of quantum computing, governmental and standardization organizations are encouraging the transition to post-quantum cryptography, i.e. cryptographic schemes not known to be vulnerable to quantum computer attacks [7]. Standardization efforts are under way, for example, the National Institute of Standards and Technology (NIST) is now accepting submissions for quantum-resistant public-key cryptographic algorithms [8].

Some MSS variants were proposed: An improved Merkle signature scheme (CMSS) [9] builds two chained trees allowing the signature of 2^{40} messages and also reduce the runtime of key pair and signature generation. The Merkle signatures with virtually unlimited signature capacity (GMSS) [10] allow to sign a significant number of messages (2^{80}) with one key pair. XMSS [11] introduced a signature scheme with minimal security requirements. A hierarchical based-hash signature XMSS^{MT} [12] allows signing a large but a fixed number of messages. SPHINCS [13] is a practical stateless hash-based signature scheme and introduces a new method to randomize tree-based stateless signatures. SPHINCS has significantly larger signatures, which could make it impractical in some scenarios. In 2016, the work [7] analyzes the state management in hash-based schemes N-times and proposes a hybrid stateless/stateful scheme to protect against unintentional copies of the state of the private key and has smaller and faster signatures.

There are two proposals for standards of hash-based signature schemes: the first one [14] describes the LMS, an adaptation of the one-time signature scheme Lamport-Diffie-Winternitz-Merkle [15]. The second one [16] describes XMSS. Therefore, the design and efficient implementation of secure and practical digital signature schemes are crucial for applications that require data integrity assurance and data origin authentication.

Our Contribution. In this work, we present a software implementation of two recent standard proposals for hash-based signatures schemes, LMS and XMSS, using the Intel AVX2 vector instruction set. We use parallel optimization techniques for improving the performance of the underlying hash functions SHA2 and SHA3. We also show how to speed up the main building blocks of LMS and XMSS by taking advantage of the fastest implementation of SHA2 and SHA3. We provide a comparative performance analysis of both schemes.

Organization. The rest of this paper is organized as follows. We describe: the Winternitz One-Time Signature (WOTS) and (WOTS^{+}) in Section II, the MSS and XMSS in Section III, Hierarchical Signatures Scheme (HSS) in Section

IV and LMS and XMSS drafts in Section V. We present the target microarchitectures in Section VI. We discuss the software optimizations in Sections VII and VIII. In Section IX we show the performance results. In Section X we present the conclusions.

II. Winternitz One-Time Signature (WOTS) and (WOTS$^+$)

The OTS are used to validate the authenticity of a message by associating a secret private key with a shared public key [14]. In these one-time signatures, each private key must be used only one time to sign any given message. As a part of the signing process, a digest of the original message is computed using a cryptographic hash function H, and the resulting digest is signed. The WOTS [15] is a modification of the Lamport One-Time Signature (LOTS) [5]. WOTS uses a parameter w which is the number of bits to be signed simultaneously. This scheme produces smaller signatures than Lamport, but increases the number of one-way function evaluations from 1 to $2^w - 1$, in each element of the signing key. Hülsing [17] proposed WOTS$^+$, a modification of WOTS that uses a chaining function f^e starting from random inputs. This modification allows eliminating the requirement to use a collision-resistant hash function.

WOTS uses a one-way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ and a cryptographic hash function $g : \{0,1\}^* \rightarrow \{0,1\}^n$, where n is positive integer. The WOTS chaining function f^e computes e iterations of f on input $x \in \{0,1\}^n$ where $e \in \mathbb{N}$ ($e < w$).

The WOTS chaining function is defined as:

$$f^e(x) = \begin{cases} x & \text{if } e = 0; \\ f(f^{e-1}(x)) & \text{if } e > 0. \end{cases}$$

Similarly, the WOTS$^+$ chaining function f^e computes e iterations of f_K on inputs key $K \in \{0,1\}^n$ chosen randomly, $x \in \{0,1\}^n$ and bitmask $bm = (bm_1, ..., bm_{w-1})$ chosen randomly, with $e \in \mathbb{N}$. Then, the chaining function f^e is defined as:

$$f^e(x, bm) = \begin{cases} x & \text{if } e = 0; \\ f_K(f^{e-1}(x, bm) \oplus bm_e) & \text{if } e > 0. \end{cases}$$

These schemes are parameterized by a security parameter n and the Winternitz parameter $w \in \mathbb{N}$, for $w > 1$. The values n and w are used to compute len (number of elements of the signature), where $len = len_1 + len_2$.

In WOTS: $len_1 = \lceil n/w \rceil$, $len_2 = \lceil (\lfloor log_2 len_1 \rfloor + 1 + w)/w \rceil$.

In WOTS$^+$: $len_1 = \lceil n/(log_2(w)) \rceil$, and $len_2 = \lfloor (log_2(len_1(w-1)))/(log_2(w)) \rfloor + 1$.

A. WOTS/WOTS$^+$ key pair generation

The private keys $sk = (sk_0, \ldots, sk_{len-1})$ can be generated uniformly at random, or via a pseudorandom process. The public verification key is $pk = (pk_0, ..., pk_{len-1}) \in \{0,1\}^{(n,len)}$.

In WOTS: $pk_i = f^{2^w-1}(sk_i)$.

In WOTS$^+$: $pk_i = f^{w-1}(sk_i, bm)$.

B. WOTS/WOTS$^+$ signature generation

To generate the signature of a message M, first compute the message digest $d = g(M)$. Then, d is split into len_1 binary blocks, resulting in $d = (m_0||...||m_{len_1-1})$, where $||$ denotes concatenation. The checksum c is computed and added to d, where c can be divided into len_2 blocks $c = (c_0|| \ldots ||c_{len_2-1})$.

In WOTS: $c = \sum_{i=0}^{len_1-1}(2^w - m_i)$.

In WOTS$^+$: $c = \sum_{i=0}^{len_1-1}(w - 1 - m_i)$.

Let $b = d||c$ be the concatenation of the extended string d with the extended string c. Thus $b = (b_0||b_1|| \ldots ||b_{len-1}) = (m_0|| \ldots ||m_{len_1-1}||c_0|| \ldots ||c_{len_2-1})$. The signature of the message M is $sig_ots = (sig_0, ..., sig_{len-1})$, where:

In WOTS: $sig_i = f^{b_i}(sk_i)$.

In WOTS$^+$: $sig_i = f^{b_i}(sk_i, bm)$.

C. WOTS/WOTS$^+$ verification

To verify the signature sig_ots of the message M, we compute $(b_0, ..., b_{len-1})$ in the same way as it was calculated during signature generation. Then, we compute: $temp_sig = (sig'_0, \ldots, sig'_{len-1})$.

In WOTS: $sig'_i = f^{2^w-1-b_i}(sig_i)$.

In WOTS$^+$: $sig'_i = f^{w-1-b_i}(sig_i, bm)$.

If $temp_sig = pk_i$ for $i = \{0, 1, ..., len-1\}$, then the signature is valid, otherwise is invalid.

III. Merkle Signature Scheme (MSS)

MSS [15] is a digital signature scheme that consists of three algorithms: key generation, signing and verification. This scheme constructs a binary tree where the leaves are the verification keys, and the public key is the root of the tree. This key pair can sign/verify messages. A tree of height h and 2^h leaves will have 2^h one-time key pairs. The digest of the one-time verification public key ($g(pk_0||...||pk_{t-1})$) will be a leaf of the Merkle tree.

A. MSS key pair generation

First, the signer must select the tree height $h \in N$, $h \geq 2$. Merkle uses a cryptographic hash function $g : \{0,1\}^* \rightarrow \{0,1\}^n$, where n is a positive integer. The treehash algorithm [15] is used to generate the public key that is the root of the tree. The authentication path (Aut) is formed by the sibling right nodes, connecting the leaf up to the tree root, which is used to validate the public key. Aut is saved during the execution of the treehash algorithm.

B. MSS signature generation

The signature generation consists of two steps: first, the signature of the message digest $g(M)$ is generated using the WOTS signature algorithm and the corresponding secret key sk_s of the leaf s. Then, the signature $SIG = (s, sig_s, Aut)$ contains the index of the leaf s, the WOTS signature sig_s, and the authentication path Aut. In the second step, the next authentication path Aut is generated. This step can be

done efficiently with the algorithm proposed by [18] which is a modification of the classic authentication path algorithm proposed by Merkle [6].

C. MSS verification

The signature verification consists of two steps: first, the signature sig_s is used to recover a leaf of the tree. Second, the public key of the Merkle tree is validated in the following way. The receiver can reconstruct the path $(p_0, ..., p_h)$ from leaf s to root. The index s is used to decide the order in which the authentication path is reconstructed. Initially, $p_0 = Y_s$. For each $i = 1, 2, \ldots, h$, p_i is computed using the condition (if $\lfloor s/(2^{i-1}) \rfloor \equiv 1 \bmod 2$) and the recursive formula:

$$p_i = \begin{cases} g(Aut_{i-1} \| p_{i-1}) & \text{if } \lfloor s/(2^{i-1}) \rfloor \equiv 1 \bmod 2; \\ g(p_{i-1} \| Aut_{i-1}) & \text{otherwise.} \end{cases}$$

Finally, if the value p_h is equal to the public key pub, the signature is valid.

D. Extended Merkle Signature Scheme (XMSS)

XMSS [11] is a modification of MSS. This scheme uses a slightly modified version of Winternitz WOTS^+ described in Section II. XMSS is provably forward-secure and efficient when instantiated with two secure and efficient function families: one second-preimage resistant hash function family G_n and the other a pseudorandom function family F_n, where $G_n = \{g_K : \{0,1\}^{2n} \rightarrow \{0,1\}^n | K \in \{0,1\}^n\}$ and $F_n = \{f_K : \{0,1\}^n \rightarrow \{0,1\}^n | K \in \{0,1\}^n\}$.

The parameters of XMSS are: $n \in \mathbb{N}$, the security parameter; $w \in \mathbb{N}(w > 1)$, the Winternitz parameter; $m \in \mathbb{N}$, the message digest length; and $h \in \mathbb{N}$, the height of the binary tree.

An XMSS binary tree is constructed to generate the public key pub. The XMSS tree is a modification of the Merkle tree. A tree of height h has $h + 1$ levels. The nodes on level j are $node_{i,j}$, for $0 < j \leq h$ and $0 \leq i < 2^{h-j}$. XMSS uses the hash function g_K and bitmask (bitmaskTree) $bm \in \{0,1\}^{2n}$, chosen uniformly at random, where bm_{2i+2j} is the left bitmask and $bm_{2i+2j+1}$ is the right bitmask. The bitmasks are the main difference among the others Merkle tree constructions since they allow to replace the collision-resistant hash function family by a second-preimage resistant hash function family [11]. The nodes are computed as: $node_{i,j} = g_K((node_{2i,j-1} \oplus bm_{2i+2j}) || (node_{2i+1,j-1} \oplus bm_{2i+2j+1}))$.

To generate a leaf in the XMSS tree, a Ltree is used. The Ltree [11] uses bitmasks in the same form as in the XMSS tree. The WOTS^+ public verification keys $(pk_0, \ldots, pk_{len-1})$ are the first len leaves of a Ltree. If len is not a power of 2, then there are not sufficiently leaves to build a binary tree. Therefore, a node that has a no right sibling is lifted to a higher level of the Ltree until it becomes the right sibling of another node.

IV. Hierarchical Signatures Scheme (HSS)

A hierarchical signature scheme is an N-time signature scheme that uses other hash-based signatures in its construction [7]. Some schemes use this constructions as in CMSS [9], GMSS [10], XMSS^{MT} [12], LMS [14] and SPHINCS [13]. The basic construction of HSS consists of a tree with d layers of subtrees, for $i = 0, \ldots, d-1$, where the lower layer is $i = d-1$. The trees on top and intermediate layers are used to sign the root nodes of the trees on the respective layer below. Trees on the lowest layer are used to sign the actual messages. All trees can have equal height.

An HSS private key consists of the private keys of each level. The public key is the root of the top level. A signature HSS consists of the public keys of levels 1 to $(d-1)$, along with the signatures in each level, and the signature of the message M with the private key of the lower level $(d-1)$. Hierarchical signatures allow for shorter signing time of a message M while offering a larger number of signed messages.

V. LMS and XMSS Drafts

Among the variants of the Merkle scheme, we chose the two standard proposals for hash-based signatures to implement: LMS [14] and XMSS[16]. LMS system is an adaptation of the original Lamport-Diffie-Winternitz-Merkle one-time signature system [15] and uses the WOTS and the HSS. XMSS specifies the one-time signature scheme (WOTS^+), a single-tree (XMSS) and a multi-tree variant (XMSS^{MT}) of XMSS.

A. LMS

Leighton and Micali [14], introduce a "security string" that is distinct for each invocation of H to improve security against attacks that amortize their effort against multiple invocations of the hash function H. The following fields can appear in a security string: (I, D_ITER, D_PBLC, D_MESG, D_LEAF, D_INTR, C, r, q, i, j) as described in [14]. The values I, $D_$ and C must be chosen uniformly at random, or via a pseudorandom process; r is the node number associated with a particular node of a hash tree; q is set to be the leaf number of the hash tree; i is the index of the private key element $(pk[i])$; and j is the iteration number used when the private key element is being iteratively hashed. To generate a leaf $(leaf[q])$ in the LMS tree, the hash functions are used: $tmp = H_leaf(X)$=$Hash(X)$, where $X = (S||pk[0]|| \ldots ||pk[p-1]||D_PBLC)$ and $leaf[q] = H_node(Y)$=$Hash(Y)$, where $Y = (I||tmp||u32str(r)||D_LEAF)$.

B. XMSS

XMSS [16] randomize each hash function call; this means that aside of the initial message digest, for each hash function call a different key and different bitmask is used. These values are pseudorandomly generated using a pseudorandom function that takes a key $SEED$ and a 32-byte address $ADRS$ and outputs a n-byte value, where n is the security parameter. There are three different types of addresses; one type for the hashes used in one-time signature schemes, one for hashes used within the main Merkle-tree construction, and one for hashes used in the Ltrees.

C. Functions used in LMS and XMSS

This section describes the differences between the main functions of both schemes LMS and XMSS. Let F be the chain function used to generate the private keys, sign and verify messages. Let G be the function used to generate the inner nodes of the tree. Let I,r,i,q,j be the security strings defined in Section V-A; S=I+q; $Left$ and $Right$ be nodes left and right; KEY be a key; BM, BM_0 and BM_1 be bitmasks; sk_i be the WOTS secret key. The function $u\mathbf{Y}str(X)$ takes a nonnegative integer X as input and return $\mathbf{Y}/8$ byte strings.

In the LMS, the main functions have the following input sizes:

- $F(X)$ = Hash(X), where X is composed of $(S||sk_i||u16str(i)||u8str(j)||D_ITER)$ and $|X|$ is $(3n+8)$ bytes.
- $G(Y)$ = Hash(Y), where Y is composed of $(I||node[2.r]||node[2.r+1]||u32str(r)||\ D_INTR)$ and $|Y|$ is $(4n+5)$ bytes.

In the XMSS, the main functions have the following input sizes:

- $F(X)$ = Hash(X), where X is composed of $((toByte(0,n)||KEY||sk_i\ XOR\ BM))$ and $|X|$ is $(3n)$ bytes.
- $G(Y)$ = Hash(Y), where Y is composed of $((toByte(1,n)||KEY||(Left\ XOR\ BM_0)||(RightXORBM_1))$ and $|Y|$ is $(4n)$ bytes.

D. Keys LMS and XMSS

The sizes of the private key(SK), the verification key (PK) and the signature (Sig) are described below.

In LMS:

- $SK = (q, SEED_sk, SEED_I)$ has $2n+4$ bytes, given that q requires 4 bytes, the seed to generate the secret key and the seed to generate the identifier I have n bytes.
- $PK = (I, T[1])$ has $3n$ bytes, given that the identifier I has $2n$ bytes and the root of the tree $(T[1])$ has n bytes.
- $Sig = (q, sig_ots, auth[0], \ldots, auth[h-1])$ has $(p+1+h)n+4$ bytes, given that the index q has 4 bytes, the WOTS signature has a random value C with n bytes and sig with pn bytes; the authentication path has hn bytes.

In XMSS:

- $SK = (idx, wots_sk, SK_PRF, root, SEED)$ has $4n+4$ bytes, given that the index leaf idx requires 4 bytes, and the secret key $wots_sk$, the key SK_PRF, the root $root$ and the seed $SEED$ require n bytes.
- $PK = (root, SEED)$ has $2n$ bytes, given the $root$ and $SEED$ require n bytes.
- $Sig = (idx_sig, r, sig_ots, auth[0], \ldots, auth[h-1])$ has $(len{+}h{+}1)n{+}4$ bytes, given that the idx_sig has 4 bytes, the random value r has n bytes, the WOTS+ signature require $lenn$ bytes, and an authentication path require hn bytes.

E. Security considerations

LMS is provably secure in the random oracle model, as shown by Katz [19]. From Theorem 8 of that reference: *for any adversary attacking arbitrarily many instances of the one-time signature scheme, and making at most q hash queries, the probability with which the adversary can forge a signature with respect to any of the instances is at most* $q2^{(1-8n)}$ [14]. The format of the inputs to the hash function have the property that each invocation of that function has an input that is distinct from all others, with high probability. This property is important for a proof of security in the random oracle model. Let n be the number of bytes in the output of the hash function. Therefore, we use n = 32 to have a security level of 128 bits, even assuming that there are quantum computers that can compute the input to an arbitrary function with a computational cost equivalent to the square root of the size of the domain of that function.

XMSS provides strong security guarantees and is even secure when the collision resistance of the underlying hash function is broken. Parameters are accompanied by a bit security value. The meaning of bit security is that a parameter set grants b bits of security if the best attack takes at least $2^{(b-1)}$ bit operations to achieve a success probability of $1/2$. Hence, to mount a successful attack, an attacker needs to perform 2^b bit operations on average [20]. According to the security proof in [16], it is not sufficient to break the collision resistance of the hash functions to generate a forgery. More specifically, the requirements on the used functions are that F and G are post-quantum multi-function multi-target second-preimage resistant keyed functions, F fulfills an additional statistical requirement that roughly says that most images have at least two preimages, PRF is a post-quantum pseudorandom function, H_msg is a post-quantum multi-target extended target collision resistant keyed hash function.

VI. Target Microarchitectures

In this section, we describe the microarchitecture details of the Intel processors (Haswell and Skylake) used in this work. The Haswell microarchitecture, launched in 2013, supports the AVX2 vector instruction set, which expanded the integer arithmetic instructions of 128-bit to 256-bit registers. A single AVX2 instruction can operate eight 32-bit values or four 64-bit values at the same time. These instructions allowed four hashes could be processed concurrently for the SHA2-512/SHAKE128 and eight hashes for SHA2-256.

The Skylake microarchitecture, released in 2015, is based on the Haswell and Broadwell microarchitecture [21]. Skylake improved the latency of some instructions. Some instructions in Skylake (such as `vmov, vpand` and `vpsllq`) have better throughput and can be used to better schedule the instructions.

In the following, we describe general aspects of these micro-architectures, and the most relevant vector instructions used in this work.

A. Vector operations

In the late 1990s, processor manufacturers focused their efforts on exploiting data parallelism rather than instruction parallelism. Thus, they incorporated functional units that could execute a single instruction over a set of data. This processing fits into the paradigm of parallel computing known as Single Instruction Multiple Data (SIMD) [22].

In 1997, Intel launched its first set of instructions to implement the SIMD paradigm; called Multimedia eXtensions (MMX). MMX added 64-bit registers and vector instructions that enabled the processing of two 32-bit operations; at that time, the architectures had native 32-bit registers [21].

In 1999, Intel released the Streaming SIMD Extensions (SSE) that included eight 128-bit registers (XMMs); the number of registers was doubled in the next year when the size of native registers increased to 64. In the following years, the SSE has evolved with the launch of the new instructions sets SSE2, SSE3, e SSE4 [21].

In 2011, it was launched the Advanced Vector eXtensions (AVX) instruction set, which introduced significant contributions to the architecture; were included 256-bit registers, called YMMs, that are overlapped on XMMs registers. Also, AVX introduced a new encoding format that allows the use of three-operand assembly code, making the assignment of registers more flexible.

The code that is compiled for an instruction set can be executed only if both the CPU and the operating system support such set. Some compilers, like GCC, Clang and ICC can perform vector operations automatically (without programmer interference); however, it is not easy to determine whether the code can be vectorized. It is possible to vectorize code explicitly, by writing the code in assembly or using intrinsic functions. The intrinsic functions are primitive operations in the sense that each intrinsic function is translated into one or more machine instructions.

B. Haswell

The Haswell microarchitecture, Intel's 4th generation Core processor family, was launched in early 2013 and presenting a series of improvements on performance and also new instructions. There are instructions in the Bit Manipulation Instruction (BMI), feature group that aid in SHA2 (RORX) and RSA (MULX) performance increases. Also besides, the new instruction set AVX2 that promote vector operations from 128 bits to 256 bits, increasing performance of integer operations [23]. AVX2 has permutation and combination instructions that allow moving the words contained in vector registers [24].

C. Skylake

The Skylake microarchitecture was launched in 2015. Skylake offers the following enhancements: larger internal buffers, higher cache bandwidth, higher throughput, better branching predictor, low power consumption, throughput balancing, and reduced floating point. A significant portion of the SSE, AVX, AVX2 and general purpose instructions also had latency improvements [21].

D. Relevant instructions

According to Agner Fog [24], the latency of an instruction is the delay that the instruction generates in a dependency chain, the unit of measure is clock cycles. Another factor that influences performance is throughput, which is the maximum number of instructions of the same type that can be executed per clock cycle when the operands of each instruction are independent of the previous instructions.

In Table I are highlighted some instructions of the AVX2 set that are relevant to the context of the efficient implementation of XMSS and LMS. In this table are shown the latency, the throughput and the execution ports in Haswell and Skylake [24].

TABLE I. LATENCY, THROUGHPUT AND EXECUTION PORT [24], [25]

Haswell			Execution port			
Vector instruction	Latency	Throughput	0	1	2...4	5
`vmov`	1	3	x	x		x
`vpadd`	1	2		x		x
`vpand/vpor/vpxor`	1	3	x	x		x
`vpsllq`	1	1	x			
Skylake			**Execution port**			
Vector instruction	Latency	Throughput	0	1	2...4	5
`vmov`	1	4	x	x		x
`vpadd`	1	3	x	x		x
`vpand/vpor/vpxor`	1	3	x	x		x
`vpsllq`	1	2	x	x		

The ports 0, 1 and 5 in Table I are the most used ports, and therefore, the most critical in determining the efficiency of the implementation. Note that some instructions in Skylake have better throughput; this can be used to schedule instructions to take advantage of this fact.

VII. SOFTWARE OPTIMIZATIONS

In this section, we will discuss the software optimization aspects applied in this work for Intel micro-architectures: Haswell and Skylake. Software optimization is committed to making software faster and smaller and goes beyond of writing a program with few lines of code. One must consider the costs of software development, the programming language used, the security of the code, and the computing power of processors. We will show the most critical parts of our program and how we apply optimizations using AVX2 instructions.

One of the general objectives of this work was to provide techniques that enable the efficient use of vector instruction sets in the implementation of the XMSS and LMS. Because both schemes are based on hash functions, this work shows the results of an efficient implementation that uses 256-bit registers to compute four hash values using SHA2-512/SHAKE128 or eight hash values using SHA2-256 concurrently.

The first optimization for improving both signature schemes uses the computation of multiple hashes at the same time. We call this approach multi-buffer optimization. As data buffers are independent of each other and have messages of the same size, it is possible to take advantage of the data-level parallelization of the hash algorithms. In addition, once the data is loaded into the registers, the data is processed several times by the hash function, performing several iterations of the hash algorithm on the same data, avoiding memory accesses. The result of hash is also returned in the same order as it

was sent, making it easy to implement. This optimization was applied in both hash functions SHA2 and SHA3.

A. SHA2 optimizations

The 256-bit vector instructions can process four 64-bit words or eight 32-bit words using only one instruction. The SHA2-512/SHAKE128 algorithm works internally with 64-bit words and SHA2-256 with 32-bit words. Thus, taking advantage of the 256-bit registers, four hashes could be processed concurrently for the SHA2-512/SHAKE128 and eight hashes for SHA2-256.

For example, in the calculation of $W[t]$:

$$W[t] = \sigma_1^{256}(W[t-2]) + W[t-7] + \sigma_0^{256}(W[t-15]) + W[t-16].$$

The operations with the values of $W[t]$ for the eight messages can be performed in parallel using 256-bit registers. Each $W[t]$ receives and processes 32-bit values. Thus, it is possible to compute the operations for the calculation of eight hashes at the same time with SHA2-256.

1) Optimizations based on processor execution ports: Another optimization was to reformulate the code of the functions that compute the hash of the messages, scheduling the execution of the instructions to improve the throughput. The core strategy of our implementation was to analyze the main functions that process the messages to generate the hash. We look for the instructions for these functions, the ports available to execute them, the latency, and throughput, and the dependencies in the code. We have eliminated several dependencies in the SHA2 algorithm code. This approach allows us to parallelize calculations when there is no dependence between instructions.

The most critical functions are:

- $T_1(h, e, f, g) = h + Sig1(e) + Ch(e, f, g) + K^{256} + W.$
- $T_2(a, b, c) = Sig0(a) + Maj(a, b, c).$

Analyzing the dependency chain in the function T_1:

- $Sig1(x) = Rot^6(x) \oplus Rot^{11}(x) \oplus Rot^{25}(x).$
- $Rot^n(x) = (SL) \vee (SR) = (x \ll (32-n)) \vee (x \gg n).$
- $Ch(e, f, g) = (((f \oplus g) \wedge e) \oplus g).$

As an example, the function $Sig1(x)$ makes three calls to the rotation function $Rot^n(x)$. $Sig1(x)$ performs three shifts to the left (SL), three shifts to the right (SR) and three OR operations. If each call to the $Rot^n(x)$ is performed separately, then the instructions of this function will also be executed separately, underutilizing the available ports on the processor.

Both microarchitectures offer three ports to execute the OR instruction; then we unroll the $Rot^n(x)$ function to perform all (SL) first, followed by (SR). Then, three SL values and the three SR values will be available to execute three OR operations in parallel. In particular, Skylake has one additional port to perform the shift operation; then it is possible to execute two shifts at the same time.

This analysis of the logical functions of the SHA2 algorithm has resulted in an implementation that takes advantage of the available ports by the processor used and improves the throughput. As an example, we illustrate in Figure 1, the execution sequence of the T_1 function instructions in the Haswell processor; the graph represents dependency in the bottom-up design; where the nodes represent the operations of the instructions and the numbers below each node represent the time in clock cycles.

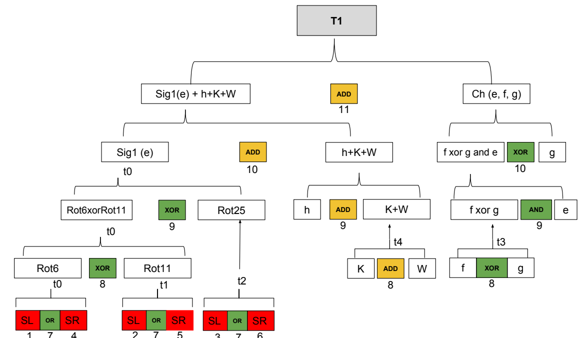

Fig. 1. Instruction scheduling of the function T_1

.

The shift (SH) to the left (SL) and right (SR) of Figure 1 can be calculated in time 1 to 6. According to Table I, this instruction has one cycle of latency and throughput one on Haswell. The three operations OR are executed at time seven because the throughput of this logical operation is three. Since the latency of the OR instruction is one cycle, at time eight the next instructions already can be executed.

Overall, the latency will be one cycle if we look isolated instructions, but if we look at a long chain of instructions of the T_1 function, the total latency will be eleven cycles, where the most critical parts are bit rotations. We can calculate, on average, the total latency in Haswell of the T_1 function, with the following formula:

$$\text{Lat_Haswell} = 6(SH) + 4/2(ADD) + 8/3(LOGIC).$$

$$\text{Lat_Haswell} \approx 11 \ cycles.$$

Analyzing the latency and the throughput in the Skylake processor for the T_1 function, we observe that some operations have the same latency, but a better throughput. The SH operation has a throughput of two and the operation ADD a throughput of three. Thus, the latency calculation for the Skylake processor can be expressed as:

$$\text{Lat_Skylake} = 6/2(SH) + 4/3(ADD) + 8/3(LOGIC).$$

$$\text{Lat_Skylake} \approx 8 \ cycles.$$

Section IX shows how these optimizations improved the performance of SHA2 in Haswell and Skylake.

B. SHA3 optimization

The Secure Hash Algorithm-3 (SHA-3) is a family of functions that was standardized by NIST in 2015 [26]. This family consists of four cryptographic hash functions and two extendable-output functions (XOFs), called SHAKE128 and SHAKE256. The permutation function used in the SHA-3 family is KECCAK-p [1600,24] and is the one responsible for algorithm efficiency.

The permutation function KECCAK-p [1600,24] is composed of five steps that are processed 24 times. The steps are: the θ step, where is computed an XOR of each word of the state with the parity of the left column and the right column rotated one bit; the ρ step, where each word of the state is rotated a fixed amount of bits; the π step where the words of the state are permuted; the χ step, where is processed a non-linear function between the elements of the same row; the ι step, where is computed an XOR between the first element of the state with a constant value. The KECCAK-p [1600,24] function uses a state of 25 words of 64 bits. The use of AVX2 instructions allows to gather four words in the same register and to process four states at the same time. To map four states are required 25 variables of 256 bits; after the mapping, each one of the 25 variables will be composed by one word of each state.

To implement the θ step, we need only XORs and rotations; the AVX instructions `vpsllq` and `vpsrlq` can be used to emulate rotation instructions. The other four steps can be implemented in blocks of five words at the same time; it is important to process these words together to avoid a large number of memory accesses because the Intel architecture has 16 256-bit registers and this implementation uses 25 variables.

In the ρ step is required to rotate a different amount of bits in each word of the state. It is possible to process this step in parallel using the AVX2 instructions `vpsllvq` and `vpsrlvq` to emulate a variable rotate. The π step permutes the words of the state; as each word of each state was mapped in the same variable, the permutation just change the name of the variables, that in fact, no instruction is required.

The χ step is processed in parallel by using one XOR and the `vpandn` instruction and the ι step is just one XOR of the first word of the state with a constant. The complete code can be found in [27].

VIII. Optimizations in LMS and XMSS

The following software optimizations were applied to the standard proposal LMS and XMSS. We will show how these optimizations improved the algorithms of key generation, signature, and verification of these schemes. Each of these operations is based on hash functions. Thus, by optimizing the underlying hash functions, we speedup the execution of the signature operations of both schemes.

The optimized functions, based on hash algorithms, were:

- the keyed hash functions of LMS and XMSS;
- the function F of LMS and XMSS;
- the functions PRF and PRG of LMS and XMSS;
- the function H of XMSS.

A. Optimization of keyed hash functions

The keyed hashed functions of both schemes LMS and XMSS always work with message blocks of the same size. Then, in order to accelerate the computation of the keyed hash function, we made a specialized implementation based on the size of the message input to be processed and set the $block$ values and the pad values. Since the pad values are fixed, there is no need to calculate the pad each time the function is called. Figure 2 shows the optimization of the function F of the XMSS with fixed pad for SHA2-256.

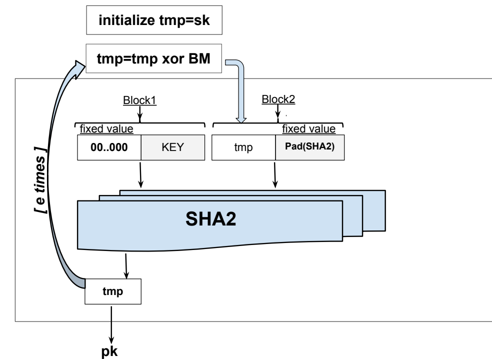

Fig. 2. Function F of the WOTS+.

In the specialized implementation of these functions, we have created an interface to receive and processes 32-bit message blocks on the SHA2-256 and 64 bits on SHA2-512/SHAKE128. The creation of an implementation of these functions with input, processing, and output with values of 32/64-bits, has significantly reduced processing time over generic functions that receive 8-bit characters because the conversion from 32/64 bit to 8 bits is time-consuming for the processor.

The hash function SHA2 processes blocks of 512 bits while the SHA3/SHAKE128 can handle up to 1344 bits of the message at the same time. An implementation of the function F of XMSS with SHAKE128 needs to process just a single block while in SHA2 must process two blocks to generate the hash value.

B. Optimization of the function F

The function F is used in the chaining function algorithm to generate the verification keys OTS. In the signature, the chaining function algorithm is also used to update the leaves in the authentication path. Thus, reducing the execution time of the function F reflected a significant improvement in the performance of both schemes LMS and XMSS.

Figure 3 shows our implementation of the function F with SHA2-512 which computes four instances in parallel, generating four public keys pk at the same time. We load four secret keys sk into the 256-bit vector registers, perform e iterations of the function F and then store four private keys pk on memory. We can compute the private keys pk in parallel because its generation is independent. Additionally, we store four instances of the pad value in a 256 register because these values will be used multiple times.

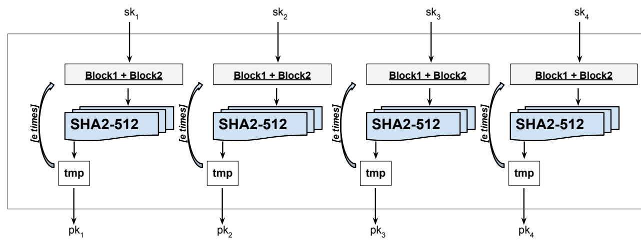

Fig. 3. Implementation of the chaining function F with SHA2-512.

The use of SIMD instructions helped to reduce the runtime of the function F, which are the computationally most expensive parts for key and signature generation.

1) The Function F in the signature and verification: For the generation of OTS keys, the optimization of the function F was simple, because in the generation of keys pk, the function F is performed the same number of times on all elements of the secret key sk. However, for OTS signature and verification generation, the application of the function F in each signature element depends on the message digest M. So we made a small change in the one-time signature and verification algorithms to apply the function F in parallel in elements of the signature.

We have added a sort algorithm before the function F. The vector msg, which contains how many times the function F will be performed, is sorted according to the number of applications of the function F. After sorting, we select the msg elements that have the same value to run the F in parallel. The function F is executed, and at the end, the signature elements are scaled according to the original order.

C. Optimizing the functions PRG and PRF

The PRG pseudo random generator generates the secret key elements sk using the PRF function. The secret keys are calculated as $sk[i] = PRF(S, toByte(i, 32))$, to $0 \leq i \leq len$. The string S is a secret value generated randomly and is used as a seed to generate all keys sk. The value i is concatenated with the value S to generate the values of sk. Since there is no dependence on the generation of the elements of sk, we could generate eight values of sk at the same time with SHA2-256 and four values of sk with SHAKE128, reducing the execution time of these functions.

The PRF function is used to generate the pseudorandom values. This generator was implemented in key generation and the signature of the LMS and the XMSS. The $PRF : Hash(toByte(3, n)||KEY||M)$ function receives the values of KEY and M as input. We created the function PRF_SIMD, which receives eight values of KEY and eight values of M in SHA2-256 and four values in SHAKE128. Then, processes these values in parallel and returns eight or four pseudorandom values.

D. Optimizing the implementation of the Ltree

The Ltree from XMSS [11] is used to generate the leaves of XMSS. In this section, we show an optimization in the generation of the Ltree for improving the performance of generating each leaf of the XMSS tree. This optimization was suggested in [13]. The function G is applied to each concatenation of children nodes to generate the parent node. Then, we modified the Ltree algorithm to perform eight evaluations of the function G at the same time. We generate eight internal nodes at the same time, from 16 children nodes, which are concatenated two by two. If the number of remaining nodes is not multiple of 16, we generate the next internal nodes one by one as the traditional way. If len is not a power of two, then there are not sufficient leaves to build a binary tree. Therefore, a node that has not a right sibling is lifted to a higher level of the Ltree until it becomes the right sibling of another node. Figure 4 shows the optimization performed in the generation of the Ltree for $w = 16$ and $l = 67$.

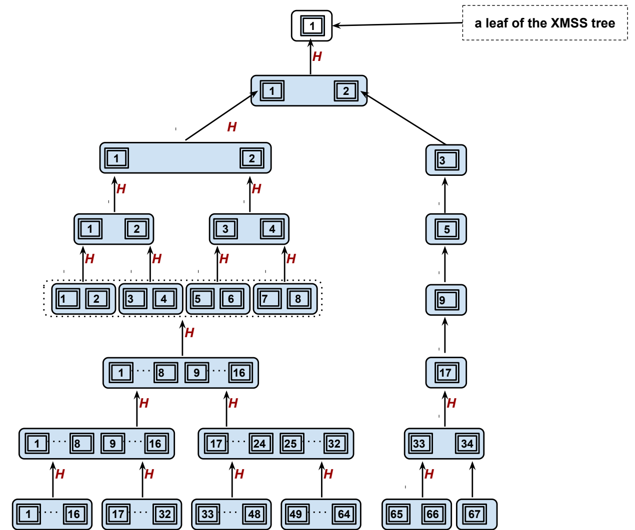

Fig. 4. Construction of the Ltree.

IX. Performance Results

This section shows the experimental results for LMS and XMSS of our implementation using AVX2 instructions. These results were obtained by running benchmarks on a Haswell processor Core i7-4770 at 3.4 GHz and a Skylake processor Core i7-6700K at 4.0 GHz. The Intel Turbo Boost and the Intel Hyper-Threading technologies were disabled to ensure the reproducibility of the results. Our implementation was written in C language and compiled using the GNU C Compiler v6.2.0. In our work, the runtimes for signing and verifying for $H > 20$, are calculated using the arithmetic average of the first one million signatures.

A. Scheme parameters

We have selected a set of parameters provided by the drafts LMS [14] and XMSS [16]. The parameters selected were: $w \in \mathbb{N}$, the Winternitz parameter; $h \in \mathbb{N}$, the total height of the tree; d, the number of layers; n, the output of the hash function.

The output of the chosen hash function influences system security. Considering classic computers, the parameter $n = 32$ provides a 256-bit security level and $n = 64$ provides 512-bit security level. Considering quantum computers, for 128-bit

security, we use the SHA2-256 and SHAKE256 functions in the LMS and the SHA2-256 and SHAKE-128 functions in the XMSS. For 256-bit security, we use the SHA2-512 and SHAKE256 functions in XMSS.

The value w influences the execution time and the size of the signature. Larger values for w imply larger execution times, but smaller signature sizes. The H and d values affect the signature size. The output n of the chosen hash function also influences the size of the public, private, and signature keys. We used $w = 2$ or $w = 4$ for LMS and $w = 16$ for XMSS. The maximum height of XMSS^{MT} was $H = 60$ and the maximum number of layers was $d = 12$ according to [16].

B. SHA2/SHA3 implementation results

In Table II, we show the performance of the implementation of the SHA2-256 and the SHAKE128 single-buffer (64-bit) and multi-buffer(256-bit) on Haswell and Skylake. The input sizes of these functions have been selected according to the size of the functions F and G of LMS and XMSS.

TABLE II. PERFORMANCE FIGURES OF SHA2-256 AND SHAKE128.

runtime(cycles/bytes)					
		Single Buffer		Multi Buffer	
hash	function	Haswell	Skylake	Haswell	Skylake
SHA2-256	F(104 bytes)	15.28	14.64	3.31	2.98
SHAKE128	F(104 bytes)	12.75	12.37	5.24	4.73
SHA2-256	G(133 bytes)	17.17	16.52	4.20	3.50
SHAKE128	G(133 bytes)	9.96	9.67	4.09	3.48

The function F processes message of 104 bytes in LMS and 96 bytes in XMSS. The G function processes message of 133 bytes in LMS and 128 bytes in XMSS. Then, the function F on SHA2-256 processes two data blocks (512 bits) and the functions G processes three data blocks (512 bits). In SHAKE128, the functions F and G processes a single block (1344 bits). Therefore, the implementation of F and G single-buffer functions with SHAKE128 has better results.

The computation of the functions F and G with SHA2-256 were faster than the versions using SHAKE128, for multi-buffer implementations. The speedup of the function F with SHAKE single-buffer is $1.2\times$ compared to SHA2-256 single-buffer implementation. SHA2-256 processes 8 independent hash values simultaneously and SHAKE128 processes only four independent hash values in parallel. Also, the speedup with SHA2 multi-buffer is approximate $4.6\times$, and with SHAKE multi-buffer is approximate $2.4\times$ compared to the single-buffer.

We also note in Table II that the function F with SHA2-256 presents a speedup of $4.6\times$ per hashing on Haswell and $4.9\times$ per hashing on Skylake. Performance on Skylake is better because of the computer architecture features presented in Section VI.

In the following sections, we will show that the performance obtained in the multi-buffer implementation of the hash functions impacts on the performance of the key generation, signature, and verification algorithms.

C. XMSS/LMS implementation results

In this section, we present the results of our XMSS/LMS single-buffer (64-bit) and multi-buffer (256-bit) implementation with the hash functions SHA2 and SHAKE.

In Table III, we compare our XMSS single-buffer (64-bit) implementation with the single-buffer implementation presented on the author's website [28]. The results were obtained on a Haswell processor. A speedup of $1.4\times$ is observed for key generation, signature, and verification of our implementation over the implementation [28]. This improvement was due to the specialized implementation of each function of XMSS.

TABLE III. PERFORMANCE FIGURES OF XMSS FROM [28] AND OUR XMSS IMPLEMENTATION ON HASWELL

runtime(ms) XMSS SHA2-256						
	Single Buffer [28]			Single Buffer(our)		
h	KeyGen	Sig	Ver	KeyGen	Sig	Ver
10	2241	9.78	1.2	1613	7.04	0.87
16	143329	16.31	1.22	103268	11.76	0.88
20	2286505	20.63	1.22	1652284	14.91	0.89

Table IV presents the results of the single-buffer (64-bit) and multi-buffer (256-bit) implementation of XMSS with SHA2 and SHAKE for different security levels. We compare our results using single-buffer and using a multi-buffer for $h = 20$.

We show that the speed up due to the multi-buffer optimization, for key generation, signing, and verification respectively, is: with SHA2-256 ranges from $4.4\times$, $4.2\times$ and $2.4\times$; with SHAKE128 ranges from $2.6\times$, $2.5\times$ and $2.0\times$; with SHA2-512 ranges from $2.4\times$, $2.4\times$ and $2.0\times$; and with SHAKE256 ranges from $3.3\times$, $3.3\times$ and $2.8\times$.

TABLE IV. PERFORMANCE FIGURES OF XMSS FOR PARAMETERS $h = 20$ AND $w = 16$ FOR DIFFERENT SECURITY LEVELS ON SKYLAKE

runtime(ms) XMSS							
	Function	Single Buffer			Multi Buffer		
security	F/G	KeyGen	Sig	Ver	KeyGen	Sig	Ver
128	SHA2-256	1369770	12.36	0.73	312702	2.92	0.30
128	SHAKE128	1056084	9.49	0.60	410818	3.74	0.30
256	SHA2-512	3537106	32.49	1.85	1452600	13.57	0.90
256	SHAKE256	5339130	49.12	2.77	1586750	14.70	0.99

For key generation, the performance is greater because the function F_SIMD executes the same amount of times in all elements of the secret key. However, in the WOTS signing and verification process, it was necessary to sort the elements of the signature before of the function F_SIMD, because the number of applications of the function depends on the bits of the message.

The runtimes with SHA2-512/SHAKE256 are larger than using SHA2-256/SHAKE128, but we get a higher level of security (256-bit security level). For 128-bit security level, the performance of the XMSS single-buffer with SHAKE128 is better than with SHA2-256. However, the multi-buffer version of SHA2-256 has better runtimes than the multi-buffer version of SHAKE128, due to the performance of these functions presented in Table II.

Table V represents the timing results of our software for the multi-buffer version of LMS with SHA2-256 and SHAKE256 at 128-bit security level. We observed that the acceleration obtained with SHA2-256 multi-buffer and $w = 4$ is $4.2\times$, $4.1\times$ and $2.0\times$ for key generation, signature, and verification respectively. The implementation with SHAKE256 multi-buffer and $w = 4$ ranges from $2.7\times$, $2.6\times$ and $1.8\times$ for key generation, signing, and verification.

A larger value of w results in shorter signatures but slower overall signing operations; it has little effect on security. For

TABLE V. PERFORMANCE FIGURES OF LMS 128-BIT SECURITY LEVEL FOR DIFFERENT VALUES OF w ON SKYLAKE

runtimes(ms) LMS								
			Single Buffer			Multi Buffer		
HASH	h	w	KeyGen	Sig	Ver	KeyGen	Sig	Ver
SHA2-256	20	2	225069	2.06	0.12	61987	0.56	0.06
SHA2-256	20	4	436627	3.96	0.23	103053	0.96	0.11
SHAKE256	20	2	186187	1.73	0.10	76870	0.71	0.06
SHAKE256	20	4	353786	3.20	0.18	130802	1.20	0.10

keys and signature generation, performance is higher for the same reasons as for XMSS implementation. As in XMSS, the LMS single-buffer with SHAKE256 is faster than SHA2-256, and the LMS multi-buffered with SHA2-256 is faster than with SHAKE256.

D. Hierarchical signatures scheme implementation results

In this section, we show the performance results of our software for both schemes HSS and XMSS^{MT} on the Skylake processor. We use the parameter $w = 4$ for LMS and $w = 16$ for XMSS, then the length of the signature OTS is $len = 67$ for both schemes. In Table VI, are given the runtimes of HSS multi-buffer using the hash functions SHA2-256 and SHAKE256 at 128-bit security level.

TABLE VI. PERFORMANCE FIGURES OF HSS MULTI-BUFFER FOR $w = 4$ AND DIFFERENT VALUES OF h AND d ON SKYLAKE

runtimes(ms) HSS multi-buffer					
HASH	h	d	KeyGen	Sig	Ver
SHA2-256	40	4	402	0.57	0.39
SHA2-256	40	8	27	0.32	0.73
SHA2-256	60	6	602	0.57	0.60
SHA2-256	60	12	40	0.32	1.11
SHAKE256	40	4	552	0.76	0.42
SHAKE256	40	8	36	0.41	0.79
SHAKE256	60	6	827	0.76	0.62
SHAKE256	60	12	55	0.41	1.19

Table VII presents the results of our implementation of XMSS^{MT} with SHA2-256 and SHAKE128 for the 128-bit security level.

TABLE VII. PERFORMANCE FIGURES OF XMSS^{MT} MULTI-BUFFER FOR $w = 16$ AND DIFFERENT VALUES OF h AND d ON SKYLAKE

runtimes(ms) XMSS^{MT} multi-buffer					
HASH	h	d	KeyGen	Sig	Ver
SHA2-256	40	4	1236	1.74	1.15
SHA2-256	40	8	83	0.96	2.18
SHA2-256	60	6	1883	1.75	1.76
SHA2-256	60	12	124	0.96	3.32
SHAKE128	40	4	1667	2.28	1.14
SHAKE128	40	8	111	1.23	2.23
SHAKE128	60	6	2498	2.28	1.67
SHAKE128	60	12	167	1.23	3.33

Notice that by increasing the number of layers the runtime is reduced for key generation and signing; however, the runtime for verification increases because the signatures of all layers must be checked. For subtrees that have the same height, the signature time remains constant. Then, increasing the tree height allows producing more signatures without impacting the performance of signing and verifying. Additionally, by increasing the number of layers the size of the secret key and signature is larger because they store information of each layer.

E. Analysis of results

In this section, we examine the results with AVX2 and compare the schemes LMS and XMSS.

Figure 5 shows the performance of XMSS/LMS multi-buffer × single-buffer with AVX2. The multi-buffer implementations with SHA2-256 have better performance because it allowed executing eight hashes at the same time whereas the SHAKE allowed to perform only four hashes in parallel.

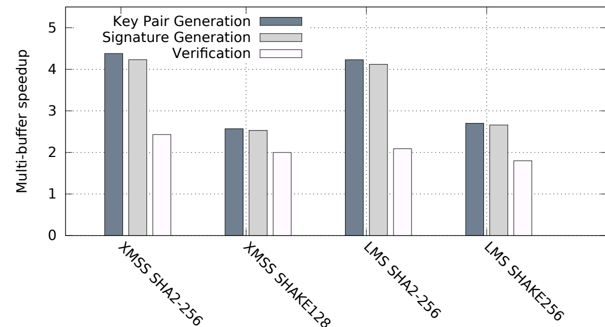

Fig. 5. Performance multi-buffer x single-buffer implementation.

Table VIII shows a summary of the code size, in C language, for the main functions of the schemes LMS and XMSS. We execute the GNU command nm in the Linux compiler, and it returned the size of the objects in a file. Note that the LMS has code size approximately $1.23\times$ greater than the XMSS code for the single-buffer implementation because the LMS uses two hash functions H_leaf and H_node for the generation of the leaves of the tree and XMSS uses only Ltree.

TABLE VIII. SIZE OF LMS AND XMSS CODES

	size (bytes)			
scheme	F	G	H_leaf/Ltree	H_no
LMS	9413	14168	15577	9552
XMSS	9664	13996	15819	

Table IX shows the keys length and runtimes of both schemes LMS and XMSS for a tree with height $h = 20$ and 128-bit security level ($n = 32$ bytes). If one uses $w = 4$ for LMS and $w = 16$ for XMSS, then the length of the signature OTS is $len = 67$ for both schemes.

TABLE IX. SIZES AND RUNTIMES OF THE LMS AND XMSS WITH SHA2-256 FOR 2^{20} SIGNATURES

		sizes (bytes)			runtimes(ms)		
draft	w	SK	PK	Sig	KeyGen	Sig	Ver
LMS	4	68	96	2820	103053	0.96	0.11
XMSS	16	132	64	2820	410818	3.74	0.30

Note that for the selected parameters, LMS secret key size is shorter than XMSS secret key. On the other hand, LMS public key size is larger than XMSS public key, and the signature key of the both schemes has the same size. Since LMS has fewer calls to the underlying hash function than XMSS, the implementation of LMS with SHA2-256 is approximate $2.7\times$ faster than the implementation of XMSS with SHA2-256. In addition, LMS does not use a Ltree tree and performs fewer operations on generating the internal nodes of the binary tree.

According to the results presented, the use of AVX2 contributes significantly to the implementation of the proposals standard for the Merkle scheme and its variants. For 128-bit security level, if the computer does not have instructions $AVX2$, then the implementation of the schemes LMS/XMSS with the hash function SHAKE128 single-buffer is a good option for presenting better runtimes. However, if these instructions are available on the computer, the implementation of the LMS/XMSS multi-buffer with SHA2-256 would be the best option.

Also, if the choice of the signature scheme is based on the runtimes, the LMS could be used because of better execution times. However, if there is a greater preoccupation with information security, XMSS would be a better option because the XMSS scheme provides strong security guarantees, XMSS is existentially unforgeable under adaptively chosen message attacks (EUCMA), it is forward security, and it is considered safe even when the collision resistance of the underlying hash function is broken.

X. Conclusion

The emerging transition to post-quantum cryptography requires digital signature schemes that are immune to quantum computers. Hash-based signatures schemes are promising candidates for replacing the current signatures schemes because they do not depend on arithmetic operations such as the problem of factorization of integers. These schemes are the object of current standardization efforts. Many improvements have already been made to the MSS making it feasible for many nowaday applications. However, some additional issues also appear as some signatures, storage resources, state management and slow generation of the key pair, leading to an important question: How can we apply the Merkle scheme in current applications?

New variants have emerged to improve the storage resource problem, such as the use of pseudo-random generators, reducing key size. The use of multi-trees, allowed to increase the number of signatures and reduce the time of generation of signature and verification keys.

In this work, we present an efficient software implementation of the Merkle scheme proposals (LMS and XMSS) using the set of vector instructions AVX2 on Intel processors. We show that our implementation presents significant improvements in the execution times of the key generation algorithms, signature, and verification of these standards. We have used several optimization techniques for increasing the performance in the software of both schemes. Our results show the feasibility of using these post-quantum schemes in practical applications.

Acknowledgment

The authors would like to thank the anonymous referees for their valuable comments and suggestions to improve the quality of this paper. The second author was partially supported by a research productivity scholarship from CNPq Brazil.

References

[1] D. Johnson, A. Menezes, and S. Vanstone, “The elliptic curve digital signature algorithm (ecdsa),” *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001. [Online]. Available: http://dx.doi.org/10.1007/s102070100002

[2] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[3] N. FIPS, “186 digital signature standard,” 1994.

[4] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. IEEE, 1994, pp. 124–134.

[5] L. Lamport, “Constructing digital signatures from a one-way function,” Technical Report CSL-98, SRI International Palo Alto, Tech. Rep., 1979.

[6] R. C. Merkle, “A certified digital signature,” in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[7] D. McGrew, P. Kampanakis, S. Fluhrer, S.-L. Gazdag, D. Butin, and J. Buchmann, “State management for hash-based signatures,” in *Security Standardisation Research*. Springer, 2016, pp. 244–260.

[8] NIST. (2016) Post-quantum cryptography standardization. [Online]. Available: http://csrc.nist.gov/groups/ST/post-quantum-crypto/index.html

[9] J. Buchmann, L. C. C. García, E. Dahmen, M. Döring, and E. Klintsevich, “Cmss–an improved merkle signature scheme,” in *International Conference on Cryptology in India*. Springer, 2006, pp. 349–363.

[10] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume, “Merkle signatures with virtually unlimited signature capacity,” in *Applied Cryptography and Network Security*. Springer, 2007, pp. 31–45.

[11] J. Buchmann, E. Dahmen, and A. Hülsing, “Xmss-a practical forward secure signature scheme based on minimal security assumptions,” in *International Workshop on Post-Quantum Cryptography*. Springer, 2011, pp. 117–129.

[12] A. Hülsing, L. Rausch, and J. Buchmann, “Optimal parameters for xmss-mt,” in *International Conference on Availability, Reliability, and Security*. Springer, 2013, pp. 194–208.

[13] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O’Hearn, “Sphincs: practical stateless hash-based signatures,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 368–397.

[14] D. McGrew, M. Curcio, and S. Fluhrer, “Hash-based signatures,” 2016, work in progress, draft-mcgrew-hash-sigs-05.

[15] R. C. Merkle, “Secrecy, authentication, and public key systems,” 1979, ph.D. thesis, Electrical Engineering, Stanford.

[16] A. Hülsing, D. Butin, S. . Gazdag, and A. Mohaisen, “Xmss: Extended hash-based signatures,” 2016, work in progress, Crypto Forum Research Group, Internet Draft, draft-xmss-06.

[17] A. Hülsing, “W-ots+-shorter signatures for hash-based signature schemes.” *Africacrypt*, vol. 7918, pp. 173–188, 2013.

[18] J. Buchmann, E. Dahmen, and M. Schneider, “Merkle tree traversal revisited,” in *International Workshop on Post-Quantum Cryptography*. Springer, 2008, pp. 63–78.

[19] J. Katz, “Analysis of a proposed hash-based signature standard,” 2015. [Online]. Available: http://www.cs.umd.edu/ jkatz/papers/HashBasedSigs.pdf

[20] A. Hülsing, J. Rijneveld, and F. Song, “Mitigating multi-target attacks in hash-based signatures,” in *Public-Key Cryptography–PKC 2016*. Springer, 2016, pp. 387–416.

[21] INTEL. (2016) Intel 64 and ia-32 architectures optimization reference manual. [Online]. Available: www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[22] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[23] S. Gulley and V. Gopal, “Haswell cryptographic performance,” 2013.

[24] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, 2016.

[25] A. Faz-Hernández, R. Cabral, D. F. Aranha, and J. López, "Implementação Eficiente e Segura de Algoritmos Criptográficos," in *Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - Minicursos*, vol. XV. Sociedade Brasileira de Computação, 2015, pp. 93–140.

[26] National Institute of Standards and Technology, *FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Gaithersburg, MD, USA: National Institute for Standards and Technology, Aug. 2015. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[27] R. Cabral. (2017) Implementation of the sha-3 family using avx/avx2 instructions. [Online]. Available: https://github.com/rbCabral/SHA-3

[28] A. Hülsing and J. Rijneveld. (2016) Implementation of xmss and xmssmt as specified in draft-huelsing-cfrg-hash-sig-xmss-06. [Online]. Available: https://huelsing.wordpress.com/code