An Efficient Full Hardware Implementation of Extended Merkle Signature Scheme

Yuan Cao, *Member, IEEE*, Yanze Wu, Wen Wang, *Member, IEEE*, Xu Lu, Shuai Chen, Jing Ye, and Chip-Hong Chang, *Fellow, IEEE*

***Abstract*— This paper presents a full hardware implementation of the eXtended Merkle Signature Scheme (XMSS), a NIST approved and IETF RFC specified post-quantum cryptography (PQC) algorithm. An optimized node traversal is proposed to enable efficient memory utilization without compromising the computational latency of the L-tree and Merkle tree construction, which are two key components used for the compression of the Winternitz One-Time Signature (WOTS) public key in XMSS. The computation of the authentication path during signature generation has also been significantly sped up by our proposed hardware implementation of the Buchmann, Dahmen, and Schneider (BDS) algorithm. Our implementation has completely avoided the use of block random-access memory, which is known to be vulnerable to side-channel attacks. The memory requirement has been highly optimized for implementation with small flip-flop chains and register counters as pointers for fast data access. To the best of our knowledge, this is the first full hardware implementation of all three *key generation*, *signing* and *verification* operations of XMSS. The design has been prototyped and evaluated on a 28 nm FPGA platform to demonstrate its performance improvements over the most efficient software and hardware/software co-design methods reported to date. Specifically, it increases the computational efficiency of the best reported XMSS implementation for *key generation* and *signature generation* by about 20% and 50%, respectively. It can also run at 10% higher clock speed than the fastest hardware implementation of *signature verification* in FPGA with 8% lower hardware resource utilization.**

Manuscript received June 7, 2021; revised August 28, 2021; accepted September 16, 2021. Date of publication October 5, 2021; date of current version January 28, 2022. This work was supported in part by the Fundamental Research Funds for Natural Science Foundation of Jiangsu Province under Grant BK20191160, in part by the Open Research of the State Key Laboratory of Computer Architecture under Grant CARCH201901, in part by QingLan Project, and in part by Changzhou Science and Technology Program under Grant CJ20200071 and Grant 2020029. This article was recommended by Associate Editor R. Azarderakhsh. *(Corresponding authors: Wen Wang; Chip-Hong Chang.)*

Yuan Cao, Yanze Wu, and Xu Lu are with the College of Internet of Things Engineering, Hohai University, Changzhou 213022, China, and also with the Rock-Solid Security Lab, Changzhou 213000, China.

Wen Wang is with the Computer Architecture and Security Laboratory, Yale University, New Haven, CT 06511 USA (e-mail: wen.wang.ww349@yale.edu).

Shuai Chen is with the Rock-Solid Security Lab, Changzhou 213000, China.

Jing Ye is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100190, China.

Chip-Hong Chang is with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639798 (e-mail: echchang@ntu.edu.sg).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2021.3115786.

Digital Object Identifier 10.1109/TCSI.2021.3115786

***Index Terms*— Post-quantum cryptography, eXtended Merkle signature scheme, hardware accelerator.**

I. Introduction

ON JANUARY 29, 2021, SpinQ Technology announced the release of the first desktop two-qubit quantum computer in the fourth quarter of 2021 for education purposes following their previous release of a hefty quantum computer, SpinQ Gemini, that weighs 55kg [1]. Given the recent advances in quantum information theory and the speed at which quantum computers are developed, a more affordable, portable, and powerful quantum computer will arrive much sooner than expected. By then the commonly used public-key cryptographic schemes such as Rivest-Shamir-Adleman (RSA) [2] and Elliptic Curve Digital Signature Algorithm (ECDSA) [3] are at high risk because of the polynomial-time realizations of the Shor's algorithm using quantum computers [4], [5]. Furthermore, Grover's algorithm can also provide a secondary acceleration for brute-force enumeration of symmetric ciphers [6]. To protect the security infrastructure, a new branch of cryptography, known as post-quantum cryptography (PQC), has drawn tremendous attention. PQC is defined as a class of cryptosystems that are deployed in classical computers but are conjectured to be secure against attacks utilizing quantum computers. At present, attacks on the 3rd round of NIST PQC candidates are still under investigation [7]. Very recently, Kamucheka *et al.* [8] built a multi-target and multi-tool platform to collect power traces from both hardware and software implementations of PQC algorithms. This platform enables them to perform power analysis on PQC algorithms based on Test Vector Leakage Assessment (TVLA).

This paper focuses on the efficient hardware implementation of one of the PQC algorithms, the eXtended Merkle Signature Scheme (XMSS). XMSS is developed based on the Merkle Signature Scheme [9]. It is a forward-secure post-quantum signature scheme, *i.e.*, previous signatures remain valid even if a secret key is compromised. XMSS is also proven to have minimal security assumption [10]–[13] - the security of XMSS depends only on the collision resistance of the hash function. In [14], a high-level synthesis design methodology was used to map high-level C specification of eleven PQC schemes under the second round of NIST evaluation into both FPGA and ASIC implementations. The evaluation showed that

the hash-based schemes, *i.e.*, SPHINCS+ always has the best latency-area product (LAP) under the same security level, which is very attractive to resource-constrained IoT devices. However, XMSS is not included in the study of [14] as it had already been standardized three years ago. Fortunately, an earlier study [15] has made a comparative analysis of prospective post-quantum hash-based digital signatures including Merkle, XMSS, LMS, and SPHINCS. This study has concluded that XMSS is the most promising digital signature algorithm in terms of security, performance, flexibility, and other metrics. XMSS is a one-time signature scheme, which means the key expires immediately after signing the current message. In [16], Kannwischer *et al.* demonstrates that XMSS can resist differential power analysis attacks. Moreover, the study of [17] also suggests that the optimized Buchmann, Dahmen, and Schneider (BDS) algorithm can limit access to the secret keys, which mitigates its overall susceptibility to side-channel attacks. The study concluded that the existing attack category will not succeed if special attention has been paid to the implementation of XMSS, such as secure caching of signatures, use of side-channel resistant PRNG, and optimization of authentication path computation. Due to its confidence-inspiring security analysis, XMSS has been standardized by the Internet Engineering Task Force (IETF) in 2018 [18] and recommended by the National Institute of Standards and Technology (NIST) in 2020 for the early use as a stateful hash-based signature scheme [19] while other PQC schemes are still undergoing the process of standardization.

Security analysis of different PQC algorithms has been examined vigorously in the standardization process. It is time to look into their implementations from the overall perspective of power, area, speed, and security (PASS) [20]. Efficient implementation of any qualified early-use PQC algorithms is in dire need for applications on resource-constrained endpoints [21]. In the nascent stage, acceleration of XMSS computation is mainly achieved through software optimizations on instruction-set architectures [22]–[24]. As a rule, their efficiency relies heavily on the performance of the high-end CPUs. Only a handful of state-of-the-art implementations have explored software-hardware (SW-HW) co-design to accelerate the XMSS computations [21], [25], [26]. In 2018, Wang *et al.* [25] proposed a SW-HW co-design method for the efficient implementation of XMSS on a RISC-V embedded processor, where the most compute-intensive SHA-256 operations are offloaded to several hardware-optimized XMSS-specific SHA-256 accelerators to speed up a large part of XMSS computations. Very recently, Mohan *et al.* [26] presented an ASIC implementation to accelerate the leaf node computations of XMSS. Unfortunately, these latest SW-HW co-design methods still cannot meet the signature verification speed required by some security-critical applications such as electronic control units. For example, the authentication time in wireless car-to-car or car-to-infrastructure communications cannot exceed 1 ms [27], but the design in [25] requires 5.8 ms to verify an incoming message. Although the most recent work [21] meets this requirement, it does not accelerate *key generation* and *signature generation* of XMSS. This is because the bottleneck Buchmann, Dahmen, and Schneider (BDS) algorithm [28] used for *signature generation* is not amenable to hardware implementation.

Besides, all existing hardware designs for XMSS leverage the on-chip random access memory (RAM) to hold temporary data during operations, *e.g.*, a dual-port RAM is used in [25]. However, RAM is a popular target of side-channel attacks. For example, in cold boot attack [29], the attacker can make use of the data remanence property of DRAM and SRAM to extract sensitive contents that remain readable for a sufficiently long duration at very low temperature upon power down. Such attack has been reported to be successfully mounted on the implementation of PQC algorithms such as Rainbow [30], Ring and Module LWE Keys [31]. Additionally, volatile memories are not dynamically allocated in custom hardware design. The number and size of memory blocks are usually determined at design time to accommodate the worst-case storage requirements. Although hardcore memory blocks provide the convenience to implement memory-demand applications, their sizes are difficult to be optimally adapted to the architectural parameters. The performance and energy efficiency can be undermined by routing constraints and memory access time than using registers and lookup tables to store the temporary data nearer to the digital signal processing modules in field-programmable gate array (FPGA) implementation. Frequent memory accesses, interleaved with intense computations, can also dissipate lots of spurious power in ASIC implementation [26], [32].

To avoid the potential security threats, our proposed XMSS hardware accelerator is RAM free, and features the following major contributions:

- To minimize the storage requirement, nodes are traversed in a depth-first manner to accelerate the performance-critical L-tree and Merkle tree computations of XMSS. In addition, an efficient register-based data access method is proposed to exponentially reduce the storage requirement. The elimination of RAM usage improves the security and power/area efficiency simultaneously.
- A new hardware implementation of BDS algorithm [28] is proposed. It greatly accelerates the authentication path generation, which is the most critical and intensive calculation in *signature generation* operation.
- It is the first fully hardware-based implementation of a complete XMSS, where the *key generation*, *signature generation* and *signature verification* are all realized in dedicated hardware blocks. The evaluation results on a 28 nm FPGA platform show that the *key generation* and *signature generation* operations have been sped up by $\sim$20% and more than 50%, respectively in comparison with the state-of-the-art XMSS implementations. As for *signature verification*, the resource consumption has been reduced by 8% with a 10% reduction in critical path delay.

The rest of the paper is organized as follows. The preliminaries of XMSS operations and parameters are introduced in Section II. Section III presents the main hardware modules of our proposed XMSS implementation. Section IV discusses the experimental results of our proposed design implemented in the FPGA platform and synthesized with ASIC standard cell

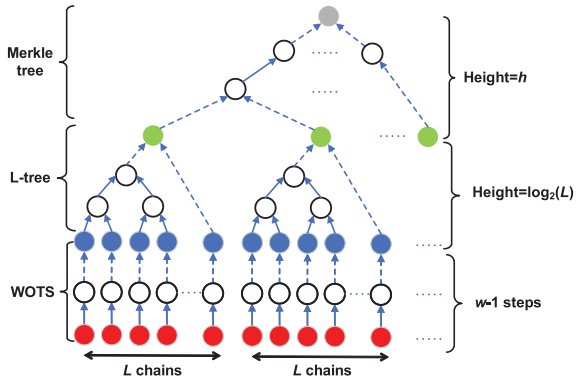


Fig. 1. Structure of the XMSS tree. The red and blue nodes are the WOTS private keys and public keys, respectively. The green nodes are the leaf nodes of the XMSS tree. The root of the XMSS tree is represented by the grey node.

libraries. Section V compares our work with the most recent related works. Section VI concludes the paper.

II. Preliminaries

XMSS is a quantum-resistant hash-based digital signature scheme that can be constructed using any second pre-image resistant cryptographic hash function. Fig. 1 shows the structure of an XMSS tree. It consists of a binary hash tree (*i.e.*, Merkle tree), L-trees, and quantum-robust Winternitz One-Time Signature (WOTS) hash chains. The WOTS is instantiated to produce 2^h private/public key pairs, where h is the height of the Merkle tree. As shown in Fig. 1, each secret key SK of the WOTS is divided into L substrings (denoted by the red leaf nodes of L hash chains). Its public key PK (denoted by the blue nodes at the L chain heads) is computed by consecutively hashing the substrings of SK through L hash chains of length $w-1$ each. To sign a message M, its message digest is first divided into L_1 integer substrings of length $log_2 w$ bits each. The differences of $w-1$ and every integer substring are summed to produce a checksum. The checksum is also divided into L_2 integer substrings of $log_2 w$ bits each. Let the integer values of these $L = L_1 + L_2$ substrings be denoted by m_i, for $i = 0, 1, \ldots L-1$. The WOTS signature of M is computed by iteratively hashing each substring of a SK m_i times through its i-th hash chain. The signature can be validated by hashing every signature substring from level m_i to the hash-head (on level $w-1$), which should match the i-th substring $\forall i \in (0, L-1)$ of the PK corresponding to SK. The L substrings of a WOTS public key are reduced into a single hash value by an L tree before a Merkle tree of height h is used to compress the individual public keys into one XMSS public key, as shown in Fig. 1. Hence. an XMSS signature signed by the j-th private key, $j = 0, 1, \ldots, 2^h - 1$ is a concatenation of strings, $M_{i,j} \| leaf_j \| Auth_j$, where $M_{i,j}$ includes a set of addresses and hash values on the m_i-th levels of the j-th L-chain, $leaf_j$ denotes the address and hash value of the j-th leaf node of the Merkel tree and the authentication path $Auth_j$ includes the addresses and hash values of up to h intermediate nodes from the j-th leaf node to the root of the Merkle tree. In what follows, the XMSS address scheme and its parameter set are briefly introduced. Three main operations of XMSS, namely *key generation*, *signature generation* and *signature verification*, will be delineated. In *key generation*, a unique pair of private/public XMSS keys is generated. The private key of the pair is used for *signature generation* and the public key is used for *signature verification*.

A. XMSS Address Scheme

A hash-function address scheme is used throughout the XMSS operation to uniquely identify each node in Fig. 1. These addresses are used by a pseudo-random function (prf) to randomize each hash function call. In our work, all the hash functions are instantiated with SHA-256. The 32-byte address is composed of eight 32-bit domains, which record the height and index of a leaf or the bitmask. The structure of the hash function address scheme is detailed in the IETF RFC 8391 document [18, Sec. 2.5].

B. XMSS Parameter Set

XMSS is defined by three parameters, h, w and n [18, Sect. 5.3]. h is the height of the Merkle tree. The maximum number of messages that can be uniquely signed and verified is 2^h. w is the Winternitz parameter. It determines the number of steps required to generate the WOTS public keys PK from the WOTS secret keys SK. $w = 4$ and 16 are most used. The length of each secret key is $L \times n$ bytes. $n = 32$ and 64 bytes are the two most widely used lengths for the hash function of the Merkle tree. An implementation of the XMSS scheme may provide support for signature generation using any of the parameter sets. Considering the most general requirements [33], our design supports the parameter sets with both $w = 4$ and $w = 16$, and $n = 32$.

C. Key Generation

The *key generation* process starts with the generation of the WOTS public key. For each WOTS instance, the WOTS private key SK is firstly generated by a prf. The WOTS public key PK is computed from SK through a hashing chain. To reduce the size of the public key, WOTS public key is compressed with an unbalanced L-tree. A tree hash function $hash_{rand}$ is used to hash a pair of child nodes, $v(i, j)$ and $v(i, j+1)$ (the index j of the left child is even), into its parent node, $v(i+1, (j+1)/2) = hash_{rand}(v(i, j), v(i, j+1))$ in the L-tree construction.

The L-tree is generated by the pseudo-code shown in Algorithm 1. It takes $L \times n$ bytes of PK, the hash address $ADDR$ and a pseudorandom number $SEED$ as input and produces an n-byte compressed PK as output. In Algorithm 1, *setTreeHeight*() and *setNodeIndex*() are address operations that set the tree height and node index, respectively. Similarly, *getTreeHeight*() and *getNodeIndex*() retrieve the tree height and node index, respectively. The function *floor*(x) returns the largest integer less than or equal to x and the function *ceil*(x) returns the smallest integer greater than or equal to x. The nodes on each level are computed with the hash function $hash_{rand}$: $\{0,1\}^{8n} \times \{0,1\}^{8n} \rightarrow \{0,1\}^{8n}$. To complete a pairwise hashing with an odd number of nodes on the same

level, the last node is promoted to a higher level until an unpaired node is available for pairing with it.

Algorithm 1 L-Tree Generation

Input: WOTS public keys: PK, address: $ADDR$, seed: $SEED$
Output: n-byte compressed public key value: $PK[0]$
1: unsigned int $L' = L$;
2: $ADDR.setTreeHeight(0)$;
3: **while** $(L' > 1)$ **do**
4: **for** (i from 0 to $floor(L'/2)-1$) **do**
5: $ADDR.setNodeIndex(i)$;
6: $PK[i] = hash_{rand}(PK[2i], PK[2i+1], SEED, ADDR)$;
7: **end for**
8: **if** ($L'/2$ is odd) **then**
9: $PK[floor(L'/2)] = PK[L'-1]$;
10: **else**
11: $L' = ceil(L'/2)$;
12: $ADDR.setTreeHeight(ADDR.getTreeHeight() + 1)$;
13: **end if**
14: **end while**
15: **return** $PK[0]$;

The 2^h WOTS verification keys are further compressed by the Merkle tree into one single node, which is the XMSS public key. The Merkle tree is constructed in a similar way by taking the root of the L-tree, *ADDR* and *SEED* as inputs to Algorithm 1. *Key generation* is the most expensive operation of XMSS. To compute the XMSS public key, the entire XMSS tree needs to be generated.

D. Signature Generation

XMSS is stateful. To ensure that each WOTS public/private key can only be used once in the *signature generation* process, a 4-byte leaf index will be stored with the private key. It indicates which WOTS key has been used and which WOTS key pair will be used for the next signature. The index is incremented after each signature. To sign a message M, the message digest of M is calculated. The digest is signed by the selected WOTS instance, which results in L n-byte hash values corresponding to the $log_2 w$-bit decompositions of the digest and its checksum. The *authentication path* (*i.e.*, one node on each level of the Merkle tree) is required for the verifier to reconstruct the XMSS public key at the leaf node of the Merkle tree. In general, the entire XMSS tree needs to be recomputed to produce an *authentication path* for a signature. Several optimized traversal algorithms to accelerate this process have been proposed, of which the BDS algorithm is most widely used [34]. Unlike most authentication path calculation algorithms [34], [35], which have a large run time difference between the best and worst cases, BDS balances the number of re-computations required in each *authentication path* with an optimal time-memory trade-off. Simplified pseudocode of the BDS algorithm [34] is shown in Algorithm 2.

Let $\varphi \in \{0, \ldots, 2^h - 1\}$ denotes the index of the current leaf node whose authentication path is to be computed. The authentication nodes required for φ on each level i are stored in $Auth_i$, $i = 0, \ldots, h-1$. To dictate the new nodes required for the authentication path of the next leaf node $\varphi + 1$, a variable τ is introduced. If φ is a left node, $\tau = 0$ and Leafcalc is called to calculate the leaf node φ from the φ-th WOTS PK (Algorithm 2, Line 6). Otherwise, τ is the level of the

Algorithm 2 BDS

Input: $\varphi \in \{0, 1, \ldots, 2^h - 2\}$, where $h \geq 2$ and even
Output: Authentication path for index $\varphi + 1$
1: $\tau \leftarrow$ minimum k s.t. $\left\lfloor \frac{\varphi}{2^k} \right\rfloor$ is even
2: **if** $(\left\lfloor \frac{\varphi}{2^{\tau+1}} \right\rfloor == 0)$ **and** $(\tau < h-1)$ **then**
3: $Keep_\tau \leftarrow Auth_\tau$
4: **end if**
5: **if** $(\tau = 0)$ **then**
6: $Auth_0 \leftarrow$ Leafcalc(φ) //calculate a leaf node from WOTS
7: **end if**
8: **if** $(\tau > 0)$ **then**
9: $Auth_\tau \leftarrow hash_{rand}(Auth_{\tau-1}, Keep_{\tau-1})$, remove $Keep_{\tau-1}$
10: **for** (i from 0 to $\tau - 1$) **do**
11: **if** $(i \leq h-2)$ **then**
12: $Auth_i \leftarrow Treehash_i.node$
13: **end if**
14: **if** $(\varphi + 1 + 3 \times 2^i < 2^h)$ **then**
15: $Treehash_i.initialize(\varphi + 1 + 3 \times 2^i)$
16: **end if**
17: **end for**
18: **end if**
19: **for** $(k = 0$ to $h/2 - 1)$ **do**
20: $s \leftarrow \min\left\{ i : Treehash_i.height = \min_{j=0,\ldots,h-2}\{Treehash_j.height\} \right\}$
21: $Treehash_s.update()$
22: **end for**
23: **return** $Auth_0, Auth_1, \ldots, Auth_{h-1}$.

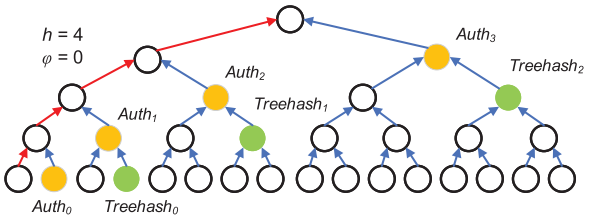


Fig. 2. An instance to demonstrate the initialization of the BDS algorithm. The yellow nodes are the authentication nodes, the green nodes are the treehash nodes and the red arrows mark the authentication path of φ.

first left node encountered in the authentication path of φ. τ can be determined by the smallest integer i such that $\left\lfloor \frac{\varphi}{2^i} \right\rfloor$ is even. A new left node on the level $j = \tau$ and new right nodes on the level $j = 0, 1, \ldots, \tau - 1$ will be required for $\varphi + 1$. If $Auth_\tau$ is a right node, then it will be stored in $Keep_\tau$. The saved node in $Keep_i$, $i \in (0, h-2)$, enables the left authentication node of $\varphi + 2^\tau + 1$ on the level $\tau + 1$ to be computed in round $\varphi + 2^\tau$ with only one hash evaluation. A node on the level i is computed by a treehash algorithm by iteratively popping its sibling node on the same level from the stack to $hash_{rand}$ and pushing the computed parent node back onto the stack. The nodes that are pushed onto the stack are called the tail nodes. At most h tail nodes are stored. Each tail node is represented by an instance structure, TREEHASH$_i$, $i = 0, \ldots, h-2$. $Treehash_i.node$, $Treehash_i.complete$ and $Treehash_i.height$ store a single tail node (*i.e.*, the first one that is pushed onto the stack), its treehash completion status and the level of the lowest tail node in the instance, respectively. $Treehash_i.height = 0$ if no tail node is stored in the treehash instance and $Treehash_i.height = \infty$ if $Treehash_i.complete$ is true or $Treehash_i$ has not been initialized. Each instance also includes a method $Treehash_i.initialize(\varphi)$ to initialize the index φ, and a method $Treehash_i.update()$ to compute the next index $\varphi + 1$ and the parent nodes of this leaf.

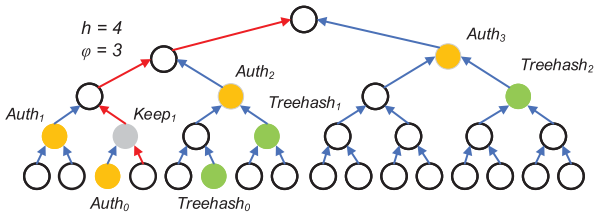


Fig. 3. The instances of BDS algorithm when the signature index $\varphi = 3$. The yellow nodes are the authentication nodes, the green nodes are the treehash nodes, the grey node is stored in $Keep_i$ and the red arrows mark the authentication path of $\varphi = 3$.

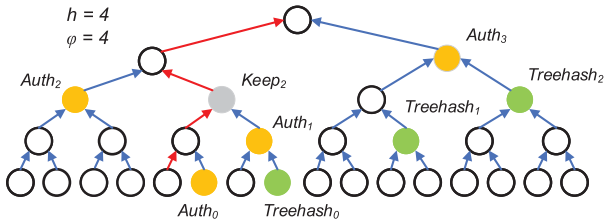


Fig. 4. The instances of BDS algorithm when the signature index $\varphi = 4$.

The BDS signature is initialized while XMSS is generating the public key. As shown in Fig. 2 for $h = 4$, upon completion of key generation process, the authentication nodes (yellow), $Auth_i, i = 0, 1, 2$ and 3, of the first index $\varphi = 0$, and the right authentication nodes (green), $Treehash_i, i = 0, 1$ and 2, for the next index $\varphi = 1$ are stored. When signing $\varphi = 0$, the BDS algorithm outputs the current authentication nodes and updates the inner nodes for the next signing.

The authentication nodes, $Auth_i$ (yellow), of $\varphi = 3$ and $\varphi = 4$ are shown in Fig. 3 and Fig. 4, respectively. τ, $Treehash_i$ (green) and $Keep_i$ (gray) $\forall i \in (0, 3)$ in both figures are the inner nodes updated by the BDS for the next signature index after exporting the authentication nodes of the current index. The transition from Fig. 3 to Fig. 4 is used to illustrate the updating process in $\varphi = 3$ for $\varphi = 4$. It should be noted that the states shown in Fig. 4 are those computed and stored during round $\varphi = 3$ but before the update is carried out in round $\varphi = 4$.

The authentication nodes (yellow) of $\varphi = 4$ are identified by its authentication path (red arrows) in Fig. 4. Among the authentication nodes, $Auth_2$ is the left node and $Auth_0$, $Auth_1$ and $Auth_3$ are the right nodes. For the left authentication node, $Auth_2$, of $\varphi = 4$, it is computed during $\varphi = 3$ by hashing (Algorithm 2, Line 9) $Auth_1$ of $\varphi = 3$ and $Keep_1$ stored during $\varphi = 1$ (see Fig. 3). In Fig. 3, $\tau = 2$ for $\varphi = 3$. Hence, $Auth_2$ is also stored in $Keep_2$ during the updating phase of $\varphi = 3$, and it will only be used in round $\varphi + 2^\tau = 7$. The right authentication nodes, $Auth_0$ and $Auth_1$ of $\varphi = 4$ on all levels lower than τ are transferred from $Treehash_0.node$ and $Treehash_1.node$, of Fig. 3, respectively (Algorithm 2, Line 12). After which, these treehash instances are initialized. The initialization begins with the leaf node index $\varphi + 1 + 3 \times 2^i$ if it is less than 2^h (Algorithm 2, Lines 14-16). The treehash instances $Treehash_i$ that are initialized but not completed (Algorithm 2, Lines 19-22) are updated by $Treehash_h.update()$ for a new round. Among the $h - 2$ treehash instances, only $h/2 - 1$ instances that have the smallest $Treehash_h.height$ are updated, such as $Treehash_0$ and $Treehash_1$ in Fig. 3 and Fig. 4.

In the existing *signature generation* process [25], all the calculations are accelerated in hardware implementation except the BDS algorithm, which makes BDS the performance bottleneck. In Section III-C, we show that the expensive BDS algorithm can be accelerated by our proposed hardware implementation.

E. Signature Verification

XMSS *signature verification* is not as computationally intensive as XMSS *key generation* or *signature generation*. When a signature is received, the verifier first calculates the WOTS public keys with the message digest and the WOTS chain. After the compression of the L-tree and the computation with the *authentication path*, the XMSS root node can be efficiently reconstructed from the WOTS public keys. If the node matches the public key, the signature is accepted. Otherwise, the signature is rejected.

III. Proposed Implementation

In the proposed implementation, the two key components *i.e.*, L-tree and Merkle tree are implemented by an optimized traversal to reduce the data storage requirement. Besides, the BDS algorithm is also implemented by an efficient control data flow to significantly reduce the memory access latency. Finally, an efficient hardware design for the complete XMSS scheme is proposed.

A. Proposed L-Tree Implementation

L-tree performs the compression of WOTS public keys PK, which is a crucial step in the *key generation*, *signature generation* and *signature verification* processes. PK consists of L n-byte WOTS public keys. Conventional implementations, *e.g.*, [21], [25], require a RAM of size $2^D \times n$ bytes, where $D = \lceil \log_2(L) \rceil$, to store PK before the L-tree computation. The data flow of the conventional L-tree computation and the memory allocation are illustrated in Fig. 5. The data in the shaded nodes of the tree are required to be stored. Initially, all the $L \times n$ bytes of the leaf node data on the bottom level are to be stored. As the level goes up, the number of stored data decreases exponentially. On reaching the top level, only one root node needs to be stored. This results in very low utilization of the RAM-based memory for the designs of [25] and [21] because the depth of the block RAM has to be a power-of-two integer. If L is not a power of two, some memory cells will never be accessed. For example, the utilization rate of their RAMs is reported to be less than 6% in Artix-7 series FPGAs. The RAM of Artix-7 is organized in blocks. Each block has a width of 36 bits and a depth of 10 bits. To read out one byte per cycle, 8 blocks are needed. Therefore, the minimum RAM size required for an L-tree is $8 \times 36 \times 2^{10} = 288$ Kb [36]. Taking into consideration of the potential risks of cold boot attack on RAM [29] and a more flexible design that can be optimized for different parameter sets, an efficient register accessing scheme is proposed to avoid the use of block RAM.

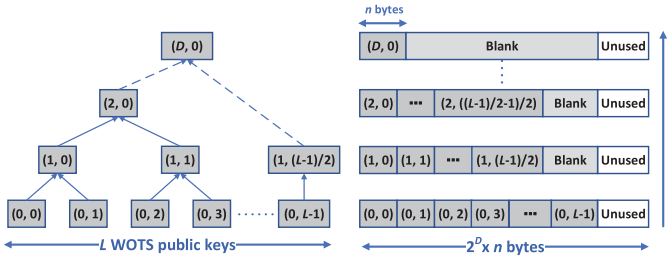


Fig. 5. Data flow and memory allocation of the conventional L-tree implementation.

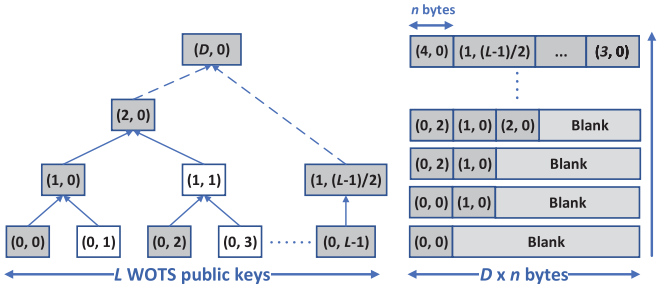


Fig. 6. Data flow and the proposed memory stack for the L-tree implementation.

TABLE I

RESOURCE UTILIZATION AND POWER CONSUMPTION OF WANG *et al.* [25] AND THE PROPOSED L-TREE IMPLEMENTATIONS ON THE SAME ARTIX-7 PLATFORM WITH $n = 32$, $w = 16$, $h = 10$ AND $L = 67$

Design	Wang [25]	Proposed
Utilization (Slices)	2069	2324
LUTs/FFs/BRAMs (36K)	5141/6543/8	5280/8751/0
Fmax(MHz)	91.7	92.2
Exec. Time (ms)	3.34	3.39
Cycles	306053	312788
Power (mW)@90 MHz	257	167

TABLE II

IMPLEMENTATION RESULTS OF WANG *et al.* [25] AND THE PROPOSED L-TREE IMPLEMENTATIONS ON 40 NM ASIC WITH $n = 32$, $w = 16$, $h = 10$ AND $L = 67$

Design	Wang [25]	Proposed
Comb. Logic (μm^2)	37147.	45290.7
Seq. Logic (μm^2)	28089.9	34834.6
Memory (μm^2)	62498.1	0
#Gates ($\times 10^3$)	84.7	55.4
Total Area (μm^2)	127735.2	80125.3
Fmax (MHz)	311.85	315.64
Power (mW)@300 MHz, 0.9 V	275.1	78.2

As there is only one hash function, each pair of nodes on the same level of the L-tree is hashed sequentially. By performing the pairwise hashing operation in a depth-first manner, a new data replacement policy is proposed to reduce the storage requirement exponentially from $2^D \times n$ bytes in conventional implementations [21], [25] to $D \times n$ bytes without increasing the number of hash function calls. For the ease of exposition, each node content (address and hash) is uniquely represented as a tuple $(i,\ j)$, where i is the level of the node and j is the position index of the node counting from 0 at the leftmost node. A node is said to be computed (or generated) when its hash value becomes available. As shown in Fig. 6, the proposed L-tree implementation employs a register stack to store the fresh inputs and outputs of treehash. The total size of the stack is only $D \times n$ bytes. Each register block of n consecutive bytes is given an address index p, where $p = 0, 1, \ldots, D-1$. The leaf nodes $(0,\ j)$ are generated from WOTS sequentially. Due to the arrival time difference between the two operands of the hash function, only the left node $(0,\ j)$ with even j needs to be stored in register block $p = 0$. Once its right node $(0,\ j+1)$ is generated, their parent, $(i+1,\ j/2)$, is computed by the pairwise hash of $(0,\ j)$ and $(0,\ j+1)$. If $j/2$ is even, $(i+1,\ j/2)$ will be stored in $p = i+1$. Otherwise, its sibling $(i+1,\ j/2-1)$ can be fetched from the register block $p = i+1$ to compute $(i+2,\ (j/2-1)/2)$. The pairwise hashing continues until $(i',\ j')$ is generated, where j' is even. This node is stored in $p = i'$. The only exception is the last node $(0,\ L-1)$. If $L-1$ is even, it has no pairing leaf node. It will be elevated to the level $(i,\ (L-1)/2^i)$, where the level i is the smallest integer that makes $(L-1)/2^i$ odd. Upon the arrival of the last data, the root node $(D, 0)$ is generated and stored in $p = 0$ as $p = D$ does not exist. The key idea behind the above data storage strategy is: once a parent node has been computed, its child node is no longer needed, so the memory it occupied can be overwritten by new node data. As only n bytes of fresh (unconsumed) data on each level needs to be kept in the register, the size of the register stack grows only linearly with the height of the L-tree. Fig. 6 provides a numerical example for $L = 13$ to illustrate how this reduced stack size can manage new data from WOTS that arrive sequentially at the leaf node of the L-tree and the update of new pairwise hash values onto the register stack. The node (0, 0) is stored in $p = 0$. When (0, 1) is available, the value of their parent node (1, 0) is computed by hashing (0, 0) and (0, 1). This result is stored in $p = 1$. At the same time, new node (0, 2) from WOTS replaces the used node (0, 0) in $p = 0$. Following the same procedure, after the nodes (0, 2) and (0, 3) have been hashed, the resultant node (1, 1) is hashed with (1, 0) in $p = 1$. The resultant node (2, 0) is then stored in $p = 2$. For this example, because $L = 13$ is odd, the last leaf node (0, 12) is unpaired ($i = 12$ is even). Hence it is elevated to (2, 3) and hashed with (2, 2) in $p = 2$. Then the result is hashed with (3, 0) in $p = 3$ to obtain (4, 0). The node (4, 0) is stored in $p = 0$. This way the nodes in the white boxes of Fig. 6 need not be stored.

TABLE I and TABLE II show the resource utilization and area consumption of [25] and our proposed L-tree implementations in FPGA and ASIC, respectively. In TABLE I, the computation data are stored in Flip-Flops (FFs) by our design in a compact fashion with only 140 extra Look-Up-Tables (LUTs). Due to the state check for each hash operation, our implementation has 2% more number of cycles. This overhead is mitigated by an increase in the maximum clock speed without the delay through an advanced peripheral bus (APB) bridge and decoder. Hence, the overall execution time is not affected (only less than 0.05 ms), yet the power consumption

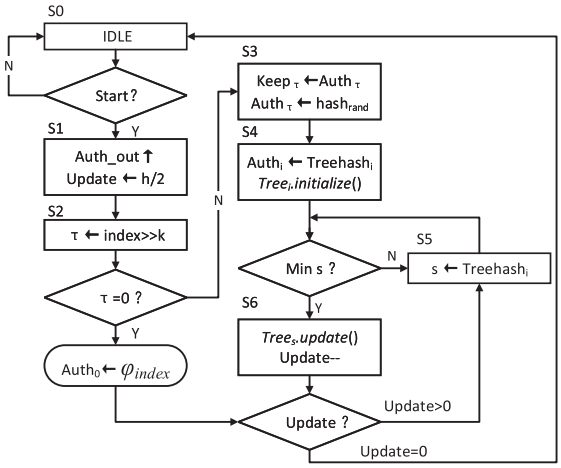


Fig. 7. ASM chart of the proposed BDS FSM.

of our proposed design has been significantly reduced by 1.5×. TABLE II shows that our L-tree implementation can save 38% of the total silicon area occupied by the L-tree implementation of [25] by getting rid of the block RAM. The technology-independent equivalent two-input NAND gate counts of [25] and our designs are also provided in the row "#Gates ($\times 10^3$)". The power consumption of the proposed design in ASIC implementation is only 28% that of [25]. The proposed RAM-free design not only reduces the silicon area but also lowers the power consumption.

B. Proposed Merkle Tree Implementation

The implementation of the Merkle tree is similar to the L-tree. The only difference is that the number of leaf nodes in the Merkle tree is a power of two. Therefore, there is no unpaired node on each level. The size of the register stack is $h \times n$ bytes. Each leaf node of the Merkle tree is the root node of an L-tree. Hence the data to each leaf node of the Merkle tree also arrive sequentially. As soon as the value of the right node is output from the root of an L-tree, the hash function is used to generate a node on the higher level of the Merkle tree before it is returned to the next L-tree. This process is repeated until the final root node (h, 0) is generated.

C. Proposed Hardware Implementation of the BDS Algorithm

The BDS algorithm is divided into six states of processing controlled by a small Finite-State-Machine (FSM). The design variables such as $Auth_i$, $(i = 0, 1, \ldots, h-1)$, $Keep_i$, $(i = 0, 1, \ldots, h-2)$ and TREEHASH$_i$, $(i = 0, 1, \ldots, h-2)$, etc. are each stored in a chain of contiguous memory elements implemented by FFs. Most of the variables are generated and consumed in block size of n bytes. Two bidirectional counters, one of length $\lceil log_2 h \rceil$ and another of length $\lceil log_2 n \rceil$ serve as a hierarchical pointer into each register chain. The pointer can be incremented or decremented to push and pop data. It is also possible to add a positive or negative offset to the pointer to nondestructively access any data in the stack without requiring successive pop operations. Fig. 7 shows the algorithmic state machine (ASM) chart of the proposed BDS FSM. The FSM is in the idle state S0 during initialization or after each signature generation. It waits for the trigger from the index φ to move to state S1, which outputs the corresponding *authentication nodes* computed in the previous index $\varphi - 1$. In State S2, τ is computed by the number of left shifting applied to the register that holds the current index φ until the first trailing 0 of φ is shifted out. States S3 and S4 are designated to update the left and right *authentication nodes* for the next index $\varphi + 1$. Each left *authentication node* is updated by retrieving its previously computed child nodes $Auth_{\tau-1}$ and $Keep_{\tau-1}$ from the stack and input them into $hash_{rand}$. The right *authentication nodes* are updated by transferring the previously computed hashed outputs $Treehash_i.node$, $i = 0, 1, \ldots, \tau - 1$ from the TREEHASH$_i$ register stack to the $Auth_i$ register stack. Once $Treehash_i.node$ has been successfully transferred, $Treehash_i.height$ and $Treehash_i.complete$ are reset to 0, which effectively performs the function of $Treehash_i.initialize()$. States S5 and S6 perform an iterative search to update the TREEHASH$_i$, $i = 0, 1, \ldots, \tau - 1$, instances in $h/2$ cycles. Specifically, up to $(h-3)$ cycles are required in State S6 to search through TREEHASH$_i$, $i = 0, 1, \ldots, h-2$ for the tail node in TREEHASH$_s$ instance with the minimum height s. When s is found, *Leafcalc*() will be invoked to calculate the leaf node $\varphi+1+3\times 2^s$ and pushed it onto the $(h \times n)$-byte stack. If its sibling node is already present in the stack, their parent node will be computed by $hash_{rand}$. Otherwise, it will be stored and waits for the next *Leafcalc*() to be invoked to calculate its sibling node. The theoretical underpinning of the sufficiency of $h/2 - 1$ updates per round with a single shared stack for all treehash instances to compute the required authentication nodes is provided in [28].

The existing SW-HW co-design implementation [37] of BDS algorithm has a few problems: 1) Large timing overhead is incurred in accessing thousands of data in the memory during the *signature generation* process. Thousands of cycles are required to manipulate the eight 32-bit domains of address with their corresponding address function individually. This also happens when updating other variables, *e.g.*, TREEHASH$_i$, $(i = 0, \ldots, h-2)$. 2) Large storage requirements make it unsuitable for resource-constrained IoT devices. For example, it takes 145 × 18 Kb RAM to store instructions and data in [25]. In our proposed hardware accelerator, memory management overhead is eliminated by using FF chains as hardware stacks. Variables stored in different stacks of FFs can be accessed concurrently and independently in the same clock cycle. In algorithms like BDS that involve a large percentage of memory access instructions, the increase in efficiency is very significant. Using our hardware stacks, the average number of cycles for the groups of 16 consecutive signatures with the lowest and highest latencies can be reduced by 23.70 % and 48.90%, respectively, as shown in Table III.

Table III shows the breakdown of the total number of cycles of the BDS algorithm implemented by the HW-SW co-design method [25] and our proposed hardware implementation. The secret key generation is independent of the public information. Other things being equal, the run time of signing with different secret keys is constant. The number of hash function calls is only dependent on the public index but not the secret key used

TABLE III

COMPARISON OF THE LATENCIES BETWEEN THE HARDWARE-SOFTWARE (HW-SW) CODESIGN [25] AND THE PROPOSED HARDWARE (HW) IMPLEMENTATION OF BDS ALGORITHM FOR XMSS WITH $n = 32$, $w = 16$ AND $h = 10$

	Platform	Method	BDS (# cycles/signature) leaf	thash_h	control	total	$\frac{\text{control}}{\text{total}}$ (%)
Group 1	Murax Soc	SW-HW	573000	458	247000	820458	30.1%
	Artix-7	HW	586732	402	215	587349	0.03%
Group 2	Murax Soc	SW-HW	1580000	1802	1730000	3311802	52.2%
	Artix-7	HW	1623293	1582	397	1789247	0.02%

* Group 1 refers to the signatures for φ = 0-15 and Group 2 refers to the signatures for φ = 254-269. # cycles/signature is the average number of cycles per signature.

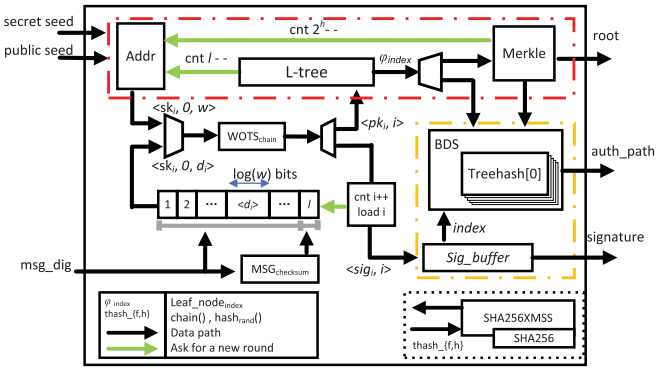


Fig. 8. Architecture of key generation and signature generation module.

for signing. The evaluation in [28, Sect. 3] shows that the performance of the BDS algorithm for signature generation varies with authentication path, *e.g.*, Leafcalc() are invoked $h/2$ times for index $\varphi = 260$ but only once for index $\varphi = 1$. Hence, the average numbers of cycles per signature for both φ = 0-15 (Group 1) and $\varphi = 254$-269 (Group 2) are presented. Considering that the control part of the BDS algorithm for the version of XMSS [25] is implemented on a soft Micro Controller Unit (MCU) core, *i.e.*, Murax SoC, the number of control cycles is calculated by subtracting the hardware accelerator cycles from the total number of cycles. For Group 2 signatures, the control cycles due to the HW-SW method account for over half of the total number of cycles, whereas only a couple of hundreds of control cycles are required by our proposed hardware accelerator. For an XMSS of height 10, our implementation can double the speed of HW-SW implementation [25]. Since the computation time and memory access rate of BDS grow with the height of the Merkle tree, the higher the Merkle tree, the larger the gain in efficiency of our implementation.

D. XMSS Architecture

In a server/client model, the key and signature generation are usually performed by the server while the signature is verified by the client. Hence, the implementation of XMSS is divided into two main modules: key and signature generation module, and signature verification module.

Fig. 8 shows the architecture of the top-level design of the *key and signature generation* module. The red dotted frame contains the `Addr`, `L-tree`, and `Merkle` submodules for key generation, whereas the yellow dotted frame consists of the `Sig` and `BDS` submodules for signature generation. The submodule $\texttt{WOTS}_{chain}$ is used in both the key generation and signature generation processes. SHA256XMSS [25] is a dedicated functional unit that is shared by these submodules. XMSS uses a modified WOTS, with a different chained hash function, *chain*(). *chain*() uses a keyed pseudo-random function *prf*() with a public seed, followed by a keyed hash function $f()$, with the key generated by *prf*(), to produce an n-byte output from the address of the current chain step. Both *prf*() and $f()$ can be realized by a SHA256 module. The hash function $hash_{rand}()$ of the L-tree and the Merkle tree also uses *prf*() and $f()$ in a similar manner but with longer input address length to produce an n-byte output. Hence, SHA256XMSS is designed to have a 1024-bit wide input and an embedded FSM to perform two or three hash iterations according to the input length, and automatically pad the input data to SHA-256. In addition, a multiplexer with a binary select input (0 for 768 bits and 1 for 1024 bits) is used to select between two input lengths.

During the XMSS key generation, the key generation module receives the secret seed and public seed as input. The submodule `Addr` generates unique addresses for the 2^h instances and produces the WOTS secret keys sk_i, $i \in (0, L-1)$. Each sk_i is converted into a WOTS public key pk_i through the submodule $\texttt{WOTS}_{chain}$. The submodule `L-tree` compresses a pk_i to a leaf node φ_{index} of the Merkle tree. The $index \in \{0, \ldots, 2^h - 1\}$ of the WOTS instance is tracked by a register counter (cnt). The public key of XMSS *root* is computed through the `Merkle` submodule. Each node compression of WOTS, L-tree and Merkle tree is performed by activating SHA256XMSS with the correct select and address inputs. Two counters (cnt) of lengths $\lceil l \rceil$, and h are used to track the computations of `L-tree` and `Merkle` submodules, respectively. Thus, `Merkle` can compute as soon as a pairing leaf node is generated without having to wait for all its leaf nodes to be generated from `L-tree`. During the computation of XMSS public key, `Merkle` also outputs the inner nodes (*e.g.*, *authentication nodes* and $Treehash_i.node$) to initialize the BDS. For signature generation, a 32-byte information digest *msg* is input into the $\texttt{MSG}_{checksum}$ submodule to compute the checksum and store the digest and its checksum as L 4-bit words d_i, $i = 1, 2, \ldots, L$. The L 4-bit words and the corresponding sk_i are input to the `WOTS` submodule to obtain the $L \times n$ bytes *signature* of *msg*. The *signature* is input to the `Sig` submodule, which sends $index$ to the `BDS` submodule to retrieve the corresponding authentication nodes *auth_path* from the stack. It should be noted that this architecture is able to generate the XMSS public keys. In other words, it can regenerate a new public key when the 2^h messages have been used up.

The architecture for the signature verification module is shown in Fig. 9. The architecture is slightly modified the implementation of [21]. The blue dashed arrows in Fig. 9 indicate that the SHA256XMSS module is a common resource shared by these modules in the implementation. The signed message consists of 4 bytes of leaf *index*, $L \times n$ bytes of *signature*, $h \times n$ bytes of *auth_path*, 32 bytes of message

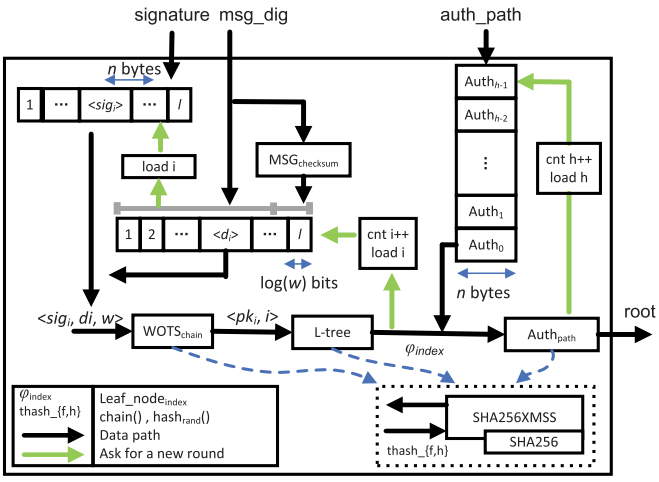


Fig. 9. Architecture of signature verification module.

TABLE IV

LUTS, FFS AND NUMBER OF CLOCK CYCLES CONSUMED BY EACH MODULE OF XMSS WITH PARAMETERS $n = 32$, $w = 16$ AND $h = 10$

Module	LUTs	FFs	Clock cycles
SHA-256	1386	1546	67
Addr (without SHA-256)	659	259	-
$WOTS_{chain}$	1534	1158	122234#
L-tree(without SHA-256)	2330	4825	30462
Merkle Tree (without SHA-256)	2154	6005	4.4×10^5
$MSG_{checksum}$	283	804	261
Sig	768	1788	148
BDS	12396	13394	215/397*
Auth_path (without SHA-256)	807	2551	-
key generation	8464	14464	3.17×10^8
signature generation	22300	31237	0.70-1.74 $\times10^6$
signature verification	6979	10860	1.60×10^5

#: average number of cycles; *: the lowest/highest number of cycles.

digest and n random bytes. In the verification process, *msg* and *signature* are stored as L $log_2(w)$-bit words. Every $log_2(w)$-bit word of *msg* is an input $Step_{d_i}$ to a $WOTS_{chain}$ instance to enable the WOTS hash chain to compute the WOTS public key pk_i by activating SHA256XMSS to perform *chain*() on sig_i $w-d_i$ times. After all L $WOTS_{chain}$ instances have completed their calculation, the WOTS public key PK is sent to the `L-tree` submodule for compression. The generated leaf node φ_{index} is input into `Auth_path` submodule to retrieve the *authentication nodes* $Auth_h$ from the stack to compute the XMSS public key $root$. Similarly, counters of appropriate lengths are used to track the current node addresses in each submodule for popping (pushing) the inputs (output) of SHA256XMSS from (to) the corresponding stack in each clock cycle.

IV. COST AND PERFORMANCE EVALUATION

The proposed techniques for the hardware implementation of XMSS are generic. The proposed design is coded in Verilog hardware description language with instantiable hardware modules according to the design parameters n, w, and h set by the designer. The parameter set $\{n, w, h\} = \{32, 16, 10\}$ is set based on the RFC 8391 recommendation [18] for the first evaluation. The design is synthesized, validated, and physically

TABLE V

NUMBER OF CYCLES FOR KEY GENERATION, SIGNATURE GENERATION AND SIGNATURE VERIFICATION OF XMSS WITH $n = 32$, $h = 10, 16, 20$ AND $w = 4, 16$

h	w	Key Generation	Group 1	Group 2	Verification
10	4	0.198×10^9	0.352×10^6	0.867×10^6	0.081×10^6
	16	0.317×10^9	0.709×10^6	1.747×10^6	0.16×10^6
16	4	10.94×10^9	0.312×10^6	1.195×10^6	0.093×10^6
	16	20.5×10^9	0.74×10^6	2.41×10^6	0.18×10^6
20	4	204.7×10^9	0.319×10^6	1.414×10^6	0.099×10^6
	16	327.7×10^9	0.752×10^6	2.838×10^6	0.191×10^6

Group 1 signatures: φ = 0 to 15; Group 2 signatures: φ = 254 to 269.

TABLE VI

IMPLEMENTATION RESULTS OF THE COMPLETE DESIGN WITH PARAMETERS $n = 32$, $w = 16$ AND $h = 10$ ON 28 NM FPGA PLATFORM AND THE SYNTHESIS RESULTS IN 40 NM ASIC

Platform	Resource & Performance	Signature Generation	Verification
FPGA	Slices	12313	3415
	LUTs/FFs/BRAM	22300/31215/0	6979/10860/0
	Fmax (MHz)	110	110
	Power (mW)@110 MHz, 1.0V	318	288
ASIC	Combination logic (μm^2)	103238.1	34480.91
	Registers (μm^2)	105889.03	38256.04
	RAM (μm^2)	0	0
	Total area (μm^2)	209127.13	72736.95
	#Gates ($\times10^3$)	130.7	45.4
	Fmax (MHz)	307.69	309.59
	Power (mW)@300 MHz, 0.9 V	168.95	96.84

implemented on a 28 nm Xilinx XC7A200T-2FBG676I FPGA chip mounted on a Xilinx Artix-7 series evaluation board for testing. Table IV lists the resources consumed by all the sub-modules shown in Fig 8. It provides a reference on the performance-cost trade-off since computational speed can be further reduced by using more than one instance of accelerator modules for XMSS and other similar hash-based post-quantum cryptographic algorithm implementation. Among them, the BDS algorithm involves many variables and iterations, which leads to large consumption of LUTs and FFs. The performance of $WOTS_{chain}$ and `BDS` are also dependent on the *msg* and *index* of the signature. Therefore, their numbers of cycles are not exact but the average/worst-case numbers. All submodules have direct and unified interfaces, which makes their interconnection costs negligible.

The performance of XMSS is different for different parameter sets $\{n, w, h\}$. Our proposed architecture is generic and can be configured for different w and h. The numbers of cycles required for the three key operations of XMSS: *key generation*, *signature generation* and *signature verification* for $h = 10$, 16 and 20 for $w = 4$ and 16 with $n = 32$ are listed in Table V. Groups 1 and 2 refer to the same groups of 16 consecutive signatures with the lowest and highest latencies as in Table III. The average number of cycles per signature is reported for each group. As expected, the number of cycles increases exponentially with h and only logarithmically with w. Among the three key operations of XMSS, *key generation* occupies the least resources although it is the most time-consuming module. This implies that XMSS will benefit most from increasing the parallelism of key generation

TABLE VII
COMPARISON WITH EXISTING WORKS ON IMPLEMENTATION OF XMSS

Design	Parameters (n, h, w)	Method	Platform	Freq. (MHz)	ALM/LUT	Key Gen. (Cyc.$\times 10^9$)	Group 1 (φ = 0-15) ($\times 10^6$) Cyc./sig.	Group 2 (φ = 254-269) ($\times 10^6$) Cyc./sig.	Verification ($\times 10^6$) Cyc.
XMSS (pub.) [37]	32,10,16	SW	Intel e5-2650	2300	-	5.089	10.458	25.503	2.447
XMSS [25]	32,10,16	SW-HW	Murax SoC	93	6530	0.323	0.93	3.427	0.541
XMSS [21]	32.10.16	SW-HW	SVU SoC	100	7544	-	-	-	0.05-0.16
XMSS [23]	32.10.16	SW	STM32 ARM-M4	-	-	23.631	23642	-	13.07
XMSS-SIMPLE [23]	32,10,16	SW	STM32 ARM-M4	-	-	7.586	7589	-	4.2
LMS [23]	32,10,16	SW	STM32 ARM-M4	-	-	3.774	3.791	-	2.65
XMSS [24]	32,10,16	SW	FRDM-K64F ARM-M4	-	-	-	-	-	6.56
XMSS [24]	32,10,16	SW	AMD	3400	-	-	14(sec)	-	-
XMSS (proposed)	32,10,16	HW	Artix-7	110	22300	0.317	0.709	1.747	0.16
XMSS (pub.) [37]	32,16,16	SW	Intel e5	2300	-	291.13	11.317	38.241	2.452
XMSS [25]	32,16,16	SW	Murax SoC	152	-	1800	70	237	15
XMSS [25]	32,16,16	SW-HW	Murax SoC	93	6530	21	0.99	5.24	0.56
XMSS (proposed)	32,16,16	HW	Artix-7	110	22300	20.5	0.74	2.41	0.18
XMSS (pub.) [37]	32,20,16	SW	Intel e5	2300	-	4574	11.6	44.931	2.493
XMSS [25]	32,20,16	SW-HW	Murax SoC	93	6530	330	1	6.48	0.58
XMSS (proposed)	32,20,16	HW	Artix-7	110	22300	327.7	0.752	2.838	0.191

* All data are selected from the best design in their articles. XMSS (pub.) refers to the publicly available implementation of XMSS on Intel E5-2650 processor, and XMSS proposed refers to our implementation.

by using more than one SHA256XMSS module with a more complex controller design. The operation *signature generation* consumes the most hardware resources due to the complexity of the *key generation* module.

Table VI summarizes the combinational and sequential (memory) usage of the complete XMSS implementation. No block RAM is required in our implementation. Our proposed data replacement policy enables the intermediate computation results to be more flexibly accessed and stored with minimal register and combinational logic area than the storage space required by the conventional approach with block RAMs. Hence, the overall area and power consumption have been reduced substantially by the saving of unused allocated RAM space and address decoding. The maximum operational frequency of our design is determined by the critical path of the hash core *i.e.*, `SHA256XMSS`, which is 110 MHz in FPGA implementation. This is different from that of [25], where its critical path also includes the RAM interface and the SW-HW interface. The results also show that *signature generation* is more area and power-hungry than *signature verification*. The proposed design is also synthesized with the standard cell library of UMC 40 nm, 0.9 V CMOS process. The logic/registers ratios required for the signature generation and verification on ASIC implementation are 0.97 and 0.90, respectively, which are close to the (slices + LUTs)/FFs ratios of 1.11 and 0.96 for the same operations in FPGA implementation. The power consumption of signature generation and verification are 168.95 mW and 96.84 mW, respectively, which are only 53.13% and 33.63%, respectively of the power consumed by the same operations in FPGA implementation.

V. COMPARISON WITH EXISTING IMPLEMENTATIONS OF XMSS

Currently, there are very few publications on the acceleration of hash-based signature schemes. Table VII lists all existing implementations of XMSS on different platforms for three different parameter sets $\{n, h, w\} = \{32, 10, 16\}$, $\{32, 16, 16\}$ and $\{32, 20, 16\}$. In 2020, Campos *et al.* [23] proposed an implementation of XMSS with slightly modification. The hash predictor technique [38] was adopted and implemented by Wang *et al.* [25]. Table VII shows that for XMSS with $\{n, h, w\} = \{32, 10, 16\}$, 23.631×10^9 cycles are required to execute the key generation of the RFC 8391 [18] version of XMSS on ARM-M4, whereas only 7.5×10^9 cycles are required for the XMSS-simple-pre version. The number of cycles is reduced to 1/3 in the signature generation and verification tests. As these implementations did not consider the BDS acceleration algorithm for signature generation, about the same number of cycles are required for signing. In [24], a large message signature acceleration was proposed to sign and verify large-scale messages instead of accelerating XMSS as a whole. To the fairness of the comparative analysis of the XMSS implementation, the same hash core of their designs is exploited in the proposed architecture. For a fair comparison, the same hash core as the designs of [21] and [25] is used in our proposed architecture. The throughput versus area trade-offs of this and other hash cores in FPGA implementation can be found in [39]. Compared with the state-of-the-art XMSS acceleration [25], our proposed hardware implementation method requires significantly fewer cycles to execute a complete XMSS and can operate at a higher clock frequency. That is because the data in the proposed structure is directly stored in cost-effective and fast DFFs, whereas RAM is used for the data storage in [25]. While the same hash core contributes to the critical paths of both designs, there is an additional delay incurred in the critical path of [25] due to the interface (APB bus) between the RISC-V core and the SHA256 module. Due to the shorter critical path delay, our proposed design has sped up the key generation process by 19∼21% regardless of the Merkle tree heights. Our proposed design has remarkably accelerated the BDS algorithm for signing. The speedup in signing depends on the height of the Merkle tree. For XMSS with $h = 10$, up to 50% reduction in the number of cycles is achieved in comparison with [25], and this reduction in the number of cycles has raised to 16× when h is increased to 20. For signature verification, our design has the same number of cycles as the fastest design [21] in the case of a single WOTS block but it can operate at a 10% higher clock rate. Meanwhile, our design also consumes about 8% lesser logic resources.

VI. Conclusion and Future Work

In the proposed implementation, pairwise hashing is computed sequentially by a depth-first traversal of the L-tree and Merkle tree to reduce the data storage requirement. The proposed XMSS hardware accelerator has been designed to avoid the use of RAM for security against side-channel attacks. The complete hardware implementation of all XMSS operations, including *key generation*, *signature generation* and *signature verification*, has been rigorously evaluated and compared against the state-of-the-art implementation [25] of XMSS. The evaluation results in the FPGA platform show that our implementation has accelerated its *key generation* and *signing* phases by around 20% and 50%, respectively. Our implementation also outperforms the fastest hardware implementation in FPGA for *signature verification* [21] with a comparable number of cycles but a 10% increase in clock speed and 8% lesser hardware resources for each WOTS block. Our future work is to explore more efficient implementation with higher parallelism by using more than one SHA256XMSS module with adaptive operation scheduling for control and data flow optimization.

Acknowledgment

The authors would like to thank Jiulong Wang, Yuhao Sun, and Jianwei Sun for help in experimental analysis.

References

[1] S.-Y. Hou *et al.*, "SpinQ Gemini: A desktop quantum computer for education and research," *EPJ Quantum Technol.*, vol. 8, no. 20, pp. 1–23, Jul. 2021.

[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.

[3] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *Int. J. Inf. Secur.*, vol. 1, no. 1, pp. 36–63, Aug. 2001.

[4] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.

[5] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.

[6] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory Comput. (STOC)*, New York, NY, USA, 1996, pp. 212–219.

[7] D. Apon and J. Howe, "Attacks on NIST PQC 3rd round candidates," in *Proc. IACR Real World Crypto Symp.*, 2021, pp. 1–26.

[8] T. Kamucheka *et al.*, "Power-based side channel attack analysis on PQC algorithms," Cryptol. ePrint Arch., Int. Assoc. Cryptologic Res., Las Vegas, NV, USA, Tech. Rep. 2021/1021, 2021.

[9] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology*, G. Brassard, Ed. New York, NY, USA: Springer, 1990, pp. 218–238.

[10] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *J. ACM*, vol. 33, no. 4, pp. 792–807, Aug. 1986.

[11] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, "A pseudorandom generator from any one-way function," *SIAM J. Comput.*, vol. 28, no. 4, pp. 1364–1396, 1999.

[12] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *Fast Software Encryption*, B. Roy and W. Meier, Eds. Berlin, Germany: Springer, 2004, pp. 371–388.

[13] J. Rompel, "One-way functions are necessary and sufficient for secure signatures," in *Proc. 22nd Annu. ACM Symp. Theory Comput.*, New York, NY, USA, 1990, pp. 387–394.

[14] D. Soni, K. Basu, M. Nabeel, and R. Karri, *A Hardware Evaluation Study of NIST Post-Quantum Cryptographic Signature Schemes*. Santa Barbara, CA, USA: NIST, 2019.

[15] O. Potii, Y. Gorbenko, and K. Isirova, "Post quantum hash based digital signatures comparative analysis. Features of their implementation and using in public key infrastructure," in *Proc. 4th Int. Sci.-Practical Conf. Problems Infocommun. Sci. Technol. (PIC S&T)*, Oct. 2017, pp. 105–109.

[16] M. J. Kannwischer, A. Genêt, D. Butin, J. Krämer, and J. Buchmann, "Differential power analysis of XMSS and SPHINCS," in *Constructive Side-Channel Analysis and Secure Design*, J. Fan and B. Gierlichs, Eds. Cham, Switzerland: Springer, 2018, pp. 168–188.

[17] J. A. Buchmann, "Physical attack vulnerability of hash-based signature schemes," M.S. thesis, Dept. Comput. Sci. Cryptogr. Comput. Algebra, TU Darmstadt, Germany, 2017.

[18] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen. (May 2018). *XMSS: Extended Merkle Signature Scheme*. RFC 8391. [Online]. Available: https://rfc-editor.org/rfc/rfc8391.txt

[19] D. A. Cooper *et al.*, "Recommendation for stateful hash-based signature schemes," *NIST Special Publication*, vol. 800, p. 208, Oct. 2020.

[20] D. Soni, M. Nabeel, K. Basu, and R. Karri, "Power, area, speed, and security (PASS) trade-offs of NIST PQC signature candidates using a C to ASIC design flow," in *Proc. IEEE 37th Int. Conf. Comput. Design (ICCD)*, Nov. 2019, pp. 337–340.

[21] V. B. Y. Kumar, N. Gupta, A. Chattopadhyay, M. Kasper, C. Kraus, and R. Niederhagen, "Post-quantum secure boot," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1582–1585.

[22] J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS—A practical forward secure signature scheme based on minimal security assumptions," in *Proc. 4th Int. Conf. Post-Quantum Cryptogr.*, 2011, pp. 117–129.

[23] F. Campos, T. Kohlstadt, S. Reith, and M. Stttinger, "LMS vs XMSS: Comparison of stateful hash-based signature schemes on ARM cortex-M4," in *Progress in Cryptology*. New York, NY, USA: Springer, 2020, pp. 258–277.

[24] J. W. Bos, A. Hülsing, J. Renes, and C. Van Vredendaal, "Rapidly verifiable XMSS signatures," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, Dec. 2020, pp. 137–168.

[25] W. Wang *et al.*, "XMSS and embedded systems," in *Selected Areas in Cryptography*. New York, NY, USA: Springer, 2020, pp. 523–550.

[26] P. Mohan, W. Wang, B. Jungk, R. Niederhagen, J. Szefer, and K. Mai, "ASIC accelerator in 28 nm for the post-quantum digital signature scheme XMSS," in *Proc. IEEE 38th Int. Conf. Comput. Design (ICCD)*, Oct. 2020, pp. 656–662.

[27] M. Knezevic, V. Nikov, and P. Rombouts, "Low-latency ECDSA signature verification—A road toward safer traffic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 11, pp. 3257–3267, Nov. 2016.

[28] B. Johannes, D. Erik, and S. Michael, "Merkle tree traversal revisited," in *Proc. 2nd Int. Workshop Post-Quantum Cryptogr.*, 2008, pp. 63–78.

[29] J. A. Halderman *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009.

[30] V. Polanco and Ricardo, "Cold boot attacks on post-quantum schemes," Ph.D. dissertation, School Math. Inf. Secur., Royal Holloway, Egham, U.K., 2019.

[31] M. R. Albrecht, A. Deo, and K. G. Paterson, "Cold boot attacks on ring and module LWE keys under the NTT," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, Aug. 2018, pp. 173–213.

[32] L. C. Stephen, "System and method for reducing power consumption during extended refresh periods of dynamic random access memory devices," U.S. Patent 7 082 073, Jul. 25, 2006.

[33] A. Shoufan, S. A. Huss, O. Kelm, and S. Schipp, "A novel rekeying message authentication procedure based on Winternitz OTS and reconfigurable hardware architectures," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Dec. 2008, pp. 301–306.

[34] M. Szydlo, "Merkle tree traversal in log space and time," in *Advances in Cryptology*, C. Cachin and J. L. Camenisch, Eds. Berlin, Germany: Springer, 2004, pp. 541–554.

[35] P. Berman, M. Karpinski, and Y. Nekrich, "Optimal trade-off for Merkle tree traversal," *Theor. Comput. Sci.*, vol. 372, no. 1, pp. 26–36, Mar. 2007.

[36] *Block Memory Generator V8.4*. Accessed: Aug. 6, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf

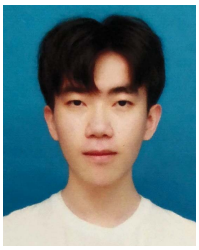
[37] *XMSS Reference Code*. Accessed: May 13, 2021. [Online]. Available: https://github.com/XMSS/xmss-reference

[38] S. F. P. Kampanakis, "LMS vs XMSS: Comparion of two hash-based signature standards," Cryptol. ePrint Archive, Int. Assoc. Cryptologic Res., Las Vegas, NV, USA, Tech. Rep. 2017/349, 2017. [Online]. Available: https://ia.cr/2017/349

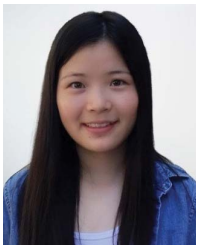
[39] E. Homsirikamol, M. Rogawski, and K. Gaj, "Throughput vs. area trade-offs in high-speed architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs," in *Cryptographic Hardware and Embedded Systems*. San Antonio, TX, USA: IACR, 2011, pp. 491–506.

Yuan Cao (Member, IEEE) received the B.S. degree from Nanjing University in 2008, the M.E. degree from The Hong Kong University of Science and Technology in 2010, and the Ph.D. degree from Nanyang Technological University (NTU) in 2015.

He worked with Advantest from 2015 to 2017 and a Research Fellow with NTU from 2017 to 2018. He was a Visiting Professor with Hamad Bin Khalifa University in December 2019. He is currently a Full Professor with the College of IoT, Hohai University. He has edited one *IET Materials, Circuits and Devices* book series 66 entitled *Frontiers in Hardware Security and Trust*, and has over 60 peer-reviewed conference and journal publications. Two of his papers has been selected as the Finalist of the Best Paper Award for AsianHOST'2017 and AsianHOST'2019. His research interests include TRNG, PUF, PQC, and analog/mixed-signal VLSI designs. He has served as an Organizing/Technical Committee Member for various IEEE conferences, such as AsianHOST, ASHES, DSP, and CTC.



Yanze Wu received the B.S. degree from Hohai University, where he is currently pursuing the M.S. degree. His research interests include applied cryptography and hardware security.



Wen Wang (Member, IEEE) received the B.S. degree from the University of Science and Technology of China, and the M.S., M.Phil., and Ph.D. degrees from Yale University in 2015, 2017, and 2021, respectively. She currently works as a Research Scientist with Intel Lab, USA. Her research interests include applied cryptography, computer architecture, and hardware security.

Xu Lu received the B.S. degree from Jinling Institute of Technology. She is currently pursuing the M.S. degree with Hohai University. Her research interests are in post-quantum cryptography and hardware security.

Shuai Chen received the joint Ph.D. degree from Southeast University and Yale University in 2018 and the Ph.D. degree from Southeast University in 2021. He currently works as the Co-Founder and the Director of the Rock-Solid Security Lab. His research interests include applied cryptography, computer architecture, and hardware security.

Jing Ye received the B.S. degree from Peking University in 2008 and the Ph.D. degree from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), in 2014. He currently works as an Associate Professor with the State Key Laboratory of Computer Architecture, ICT, CAS. His research interests include VLSI test and security.

Chip-Hong Chang (Fellow, IEEE) received the B.Eng. degree (Hons.) from the National University of Singapore in 1989, and the M.Eng. and Ph.D. degrees from Nanyang Technological University (NTU), Singapore, in 1993 and 1998, respectively.

He served as a Technical Consultant in industry prior to joining the School of Electrical and Electronic Engineering (EEE), NTU, in 1999, where he is currently an Associate Professor. He holds joint appointments with the university as the Deputy Director of the Center for High Performance Embedded Systems from 2000 to 2011, the Program Director of the Center for Integrated Circuits and Systems from 2003 to 2009, and the Assistant Chair of Alumni of the School of EEE from 2008 to 2014. He has edited and coedited five books, published 13 book chapters, more than 100 international journal articles (80 are IEEE) and more than 180 refereed international conference papers (mostly IEEE), and delivered over 40 colloquia. His current research interests include hardware security and trustable computing, artificial intelligence security, low-power and fault-tolerant computing, residue number systems, and application-specific digital signal processing algorithms and architectures. He is an IET Fellow. He served as an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS from 2016 to 2019, IEEE ACCESS from 2013 to 2019, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS from 2010 to 2013, *Integration, the VLSI Journal* from 2013 to 2015, *Journal of Hardware and Systems Security* (Springer) from 2016 to 2020, and *Microelectronics Journal* from 2014 to 2020. He has guest edited several special issues and served in the organizing and technical program committee of more than 70 international conferences (mostly IEEE). He serves as a Senior Area Editor for IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, and an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. He was a Distinguished Lecturer of the IEEE Circuits and Systems Society for the period 2018–2019.