2021 IEEE International Symposium on Circuits and Systems (ISCAS) | 978-1-7281-9201-7/20/$31.00 ©2021 IEEE | DOI: 10.1109/ISCAS51556.2021.9401177

High-Speed and Scalable FPGA Implementation of the Key Generation for the Leighton-Micali Signature Protocol

Yifeng Song, Xiao Hu, Wenhao Wang, Jing Tian, and Zhongfeng Wang
School of Electronic Science and Engineering, Nanjing University, Nanjing, China
Email: 141120091@smail.nju.edu.cn, huxiao@smail.nju.edu.cn, 463023464@qq.com, jingtian_nju@sina.com, zfwang@nju.edu.cn

***Abstract*—Due to the rapid progress made in quantum computers, modern cryptography faces great challenges. Many new digital signature schemes that have resistance to quantum computing are being presented for Post-Quantum Cryptography (PQC) standardization. The Leighton-Micali signature (LMS), a kind of hash-based signature scheme, is selected as a promising candidate for the PQC signature protocols by the Internet Engineering Task Force (IETF) because of its small private and public key sizes. However, the low-efficiency in key generation forms the bottleneck in practical applications. In this paper, we propose a high-speed architecture for the key generation to accelerate the LMS for the first time. The architecture is delicately devised to be scalable, supporting all the parameter sets for the LMS. The degree of parallelism is carefully designed to achieve low latency and high hardware utilization efficiency. Moreover, the control flow is well managed to accommodate different parameter sets with constant power for the consideration of anti-power analysis attacks. We code our design with Verilog language and implement it on the Xilinx Zynq UltraScale+ FPGA. The experimental results show that, compared with the optimal software implementation running on an Intel(R) Core(TM) i7-6850K 3.60GHz CPU with threading enabled, the new design achieves 55x to 2091x speedup in different parameter configurations.**

***Index Terms*—Leighton-Micali signature, hash-based signatures, post-quantum cryptography, hardware implementation, FPGA**

I. Introduction

Due to the rapid progress in quantum computing, modern cryptography is facing great challenges. Classic signature schemes which are widely used in our life, such as the RSA [1], DSA [2], ECDSA [3], and EdDSA [4], are built on the assumptions of certain mathematical problems like the integer factorization and discrete logarithm. However, Shor's algorithm [5] and its variants make it pretty efficient to solve these problems on a powerful quantum computer. Therefore, some post-quantum cryptography schemes are explored and discussed by standardization organizations and research institutions.

The hash-based scheme is one of the most promising schemes for the Post-Quantum Cryptography (PQC), whose main computations are one-way hash functions like the SHA2 [6] or SHA3 [7]. In order to satisfy the quantum security requirement, the new Hash-Based Signature (HBS) schemes need much more hashing operations in the construction of keys. Meanwhile, other schemes like the lattice-based cryptography [8], isogeny-based cryptography [9], and code-based cryptography [10], all need additional computational hardness assumptions. In terms of efficiency, the hash-based scheme shows more promises than the others in many cases.

The HBS schemes are usually constructed by using Merkle trees along with One Time Signature (OTS) [11]. In the OTS algorithm, a private key is used to sign a message and the corresponding public key is used to verify the OTS signature. An HBS tree is a binary Merkle tree whose leaves are the OTS public key hash values, and each internal node in the tree is the hash result of its two children. The root of the tree is used to construct the public key of the HBS tree. In the signature procedure, the OTS signature is firstly executed. Then, the OTS signature is packed with certain nodes, which form the path from the leaf to the root of the tree, to compose the HBS signature. Those internal nodes are used to assist in validating the HBS signature.

The LMS and XMSS are two efficient HBS schemes which have currently advanced into the standardization process hosted by the Internet Engineering Task Force (IETF) [12]. They are both developed from the classic Merkle algorithm [11]. Compared with the XMSS, the LMS has the advantage of small sizes of keys and signatures, which makes the latter be more promising in many situations, especially in the limited bandwidth. However, it requires more computations under the same security level. The key generation is the most time-consuming procedure of the LMS, which takes approximately half the time on average [13]. Therefore, accelerating the key generation process can effectively speed up the whole protocol.

In this work, an efficient hardware accelerator is firstly proposed for the key generation. The architecture is carefully designed with scalability, which can support all the parameter sets and some of them are shown in Table I. The parallelism is elaborately devised to obtain low latency and high hardware utilization efficiency. Additionally, the control flow is well planned to adapt to different parameter sets with constant power, which can help resist power attacks.

This work was supported by the National Natural Science Foundation of China under Grant 61774082, the Fundamental Research Funds for the Central Universities under Grant 021014380065, the Key Research Plan of Jiangsu Province of China under Grant BE2019003-4. (Corresponding authors: Jing Tian; Zhongfeng Wang.)

The rest of this work is structured as follows. Section II summarizes the LMS algorithm and the hash algorithm chosen in our design. The proposed hardware architecture for the key generation is presented in Section III. Experimental results and comparisons are shown in Section IV. Finally, Section V draws the conclusion.

II. Background

A. Leighton-Micali Signature

The LMS is a kind of quantum-secure HBS scheme collected as a PQC candidate by the Internet Engineering Task Force (IETF) [12]. It adopts the Winternitz one-time signature (WOTS) as its OTS scheme. The LMS consists of three core function algorithms: $key\ generation$, $signature\ generation$, and $signature\ verification$. As mentioned, $key\ generation$ consumes the most computing time and is the focus of this work. Thus, only this part is detailed in the section.

TABLE I
Parameter Set

Typecode	H	n	w	p	sig_len
SHA3_256_N32_W1	SHA3_256	32	1	265	8516
SHA3_256_N32_W2	SHA3_256	32	2	133	4292
SHA3_256_N32_W4	SHA3_256	32	4	67	2180
SHA3_256_N32_W8	SHA3_256	32	8	34	1124

Primarily, there are several parameter sets of the LMS, as shown in Table I. The whole signature system is configured with these parameters to meet different application requirements. The parameter H indicates the hash function chosen in the algorithm, and SHA3_256 [7] is adopted in our work for its high security level and flexibility. The number of bytes of the output of the hash function is denoted as n. The value of w performs a space/time trade-off: increasing the value of w will shorten the length of signature while key generation and verification will take more time to finish. p is a function of n and w, which is related to the length of the signature. The detailed algorithm is described in Alg. 1.

Algorithm 1 Private Key Generation

```
Input: $typecode$, The level value of the tree: $h$
Output: OTS private key array: $OTS\_PRIV[\ ]$
1: Set $I$ to a uniformly random 16-byte string
2: Set $SEED$ to a uniformly random 32-byte string
3: for $q = 0; q < 2^h; q = q + 1$ do
4:     $OTS\_PRIV[q] = OTS\_pri\_key\_gen(I, q,$
5:                         $typecode, SEED)$
6: end for
7: Set $q = 0$
```

As shown in Alg. 1, an LMS tree with h levels has 2^h private keys. Each of them is generated by the OTS private key generation algorithm and shares the same random strings I, $typecode$, and $SEED$. Parameters are retrieved from the $typecode$ with Table I. All these strings are combined to form the input of the OTS private key generation function, as shown in Alg. 2. And the outputs of this function become the elements of the OTS private key array.

Algorithm 2 OTS Private Key Generation ($OTS\_pri\_key\_gen$)

```
Input: $I, q, typecode, SEED$
Output: String: $\{typecode||I||q||x[0]||x[1]||......||x[p]\}$
1: Set parameter $p$ according to Table I with $typecode$.
2: for $i = 0; i < p; i = i + 1$ do
3:     $x[i] = H(I||q||i||0\text{xff}||SEED)$
4: end for
5: Return string: $\{typecode||I||q||x[0]||x[1]||......||x[p]\}$
```

The OTS public key is generated from the OTS private key, and the algorithm is described in Alg. 3. The hash functions are repeatedly applied to every single element of OTS private key array, which is denoted as x here, for $2^w - 1$ times. The outputs of these hash function form the y array. Then one more hash function is executed to get the final value K. D_PBLIC in Alg. 3 is a fixed two-byte value 0x8080.

Algorithm 3 OTS Public Key Generation

```
Input: $I, typecode, priv\_key$
Output: String: $\{typecode||I||q||K\}$
1: Set parameter $p$ and $w$ according to Table I with $typecode$.
2: Get the array $x$, random string $I$, and value $q$ from the
   $priv\_key$
3: for $i = 0; i < p; i = i + 1$ do
4:     $temp = x[i]$
5:     for $j = 0, j < 2^w - 1; j = j + 1$ do
6:         $temp = H(I||q||i||j||temp)$
7:     end for
8:     $y[i] = temp$
9: end for
10: $K = H(I||q||D\_PBLC||y[0]||......||y[p-1])$
11: Return string: $\{typecode||I||q||K\}$
```

As shown in Fig. 1, an LMS system is built with a balanced binary tree, and the public key of an LMS system is generated from the root node of that tree. Each leaf is an OTS public key hash value, which is used for signature verification. Alg. 4 shows how to calculate the internal nodes and the public key of the LMS tree. D_LEAF and D_INTR equal to 0x8282 and 0x8383, respectively. The nodes of the tree are indexed from 1 to $2^{(h+1)} - 1$, and the indexes are represented by the parameter r. The root of the tree is $T[1]$, and the index grows up from left to right, from root to leaf.

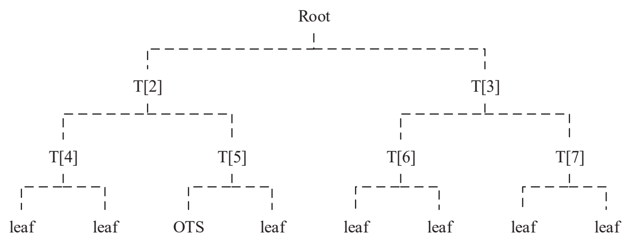

Fig. 1. Binary tree

Algorithm 4 LMS public key Generation

```
Input: OTS public key hash array $K$
Output: String:$\{typecode||ots\_type||I||T[1]\}$
1: for $r = 2^{(h+1)} - 1; r > 1; r = r - 1$ do
2:     if $r >= 2^h$ then
3:         $T[r] = H(I||r||D\_LEAF||K[r - 2^h])$
4:     else
5:         $T[r] = H(I||r||D\_INTR||T[2 * r]||T[2 * r + 1])$
6:     end if
7: end for
8: Return string: $\{typecode||ots\_type||I||T[1]\}$
```

B. SHA3_256

SHA3_256 is a kind of SHA3 algorithm whose output width is 256 bits. SHA3 is the third generation hash function standardized by the National Institute of Standards and Technology (NIST) [7], which has a high-security level. The sponge structure of the SHA3 algorithm is shown in Fig. 2.

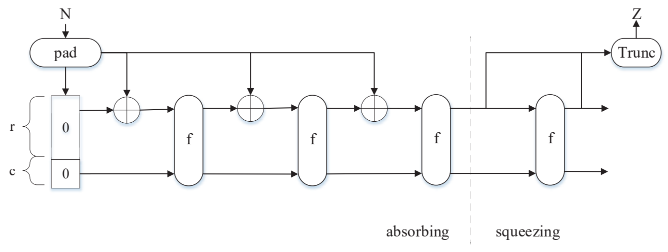

Fig. 2. SHA_3

In general, the SHA3 function consists of two phases: the absorbing and the squeezing. The input message is divided into r-byte pieces, and each of the small fragments is then xored with the updating state string. The permutation function f processes the state for 24 rounds. Each round consists of five steps: $Theta, Rho, Pi, Chi$ and $Iota$, which are explained in Alg. 5. The XOR($\oplus$), NOT and AND are the bit-wise logical operations while ROT is a bit-wise cyclic shift operation, and RC is a constant array of 24 elements.

Algorithm 5 Permutation function f (KECCAK)

```
Input: State array $A$
Output: New state array $A'$
1: for $i = 0; i < 24; i = i + 1$ do
2:     $Theta\ step : (0 <= x, y <= 4, 0 <= z < 25)$
3:         $C[x, y, z] = A[x, 0, z] \oplus A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$
4:         $D[x, z] = C[(x - 1), z] \oplus ROT(C(x + 1), 1)$
5:         $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$
6:     $Rho\ step : (0 <= x, y <= 4)$
7:         $A[x, y, z] = ROT(A'[x, y, z], r[x, y])$
8:     $Pi\ step : (0 <= x, y <= 4, 0 <= z < 25)$
9:         $B[y, (2x + 3y), z] = A[x, y, z]$
10:    $Chi\ step : (0 <= x, y <= 4, 0 <= z < 25)$
11:        $A[x, y, z] = B[x, y, z] \oplus (NOT(B[x + 1], y, z))AND(B[(x + 2), y, z])$
12:    $Iota\ step : (0 <= z < 25)$
13:        $A'[0, 0, z] = A[0, 0, z] \oplus RC[z]$
14: end for
15: Return state $A'$
```

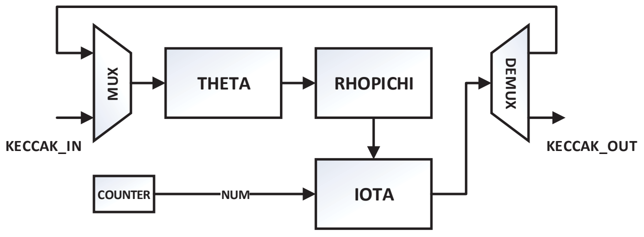

Fig. 3. KECCAK module

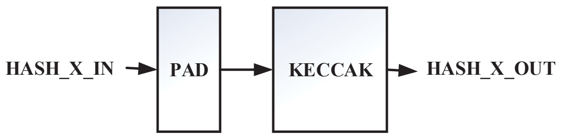

Fig. 4. HASH_X module

III. Hardware Architecture for Key Generation

According to the algorithms summarized in the last section, the core function of $key\ generation$ is the SHA3_256 hash, which constitutes almost the whole calculation procedure. Fig. 3 shows the architecture of the permutation function in SHA3_256 algorithm. Two kinds of hash modules used in our design are shown in Fig. 4 and Fig. 5. For the input strings whose width equals to X and $X < 1088$, the SHA3_256 function is achieved by the $HASH\_X$ module, consisting of a single KECCAK module along with a small padding module. And for the inputs that are longer than 1088 bits, the function is designed as the $HASH\_LS$ module where a FIFO is needed to cache and allocate the input into 1088-bit streams as the inputs of each round of KECCAK.

As shown in Table I and Alg. 2, the minimum array item parameter equals 34, and other options are nearly multiples of 34. Therefore, we set the degree of parallelism to 34 in the OTS key generation module, shown in Fig. 6. The same seed is hashed with different index i to get all the elements of array x, which are the main components of the private key. Then, these elements are sent to a loop of HASH_440 to produce the intermediate array y for the public key generation. These two

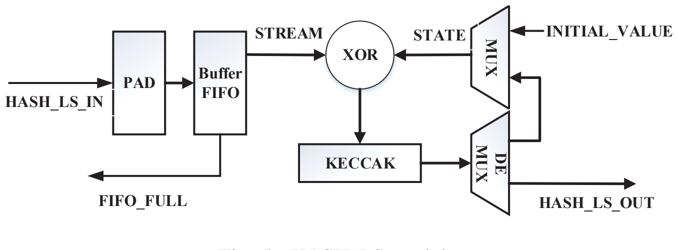

Fig. 5. HASH_LS module

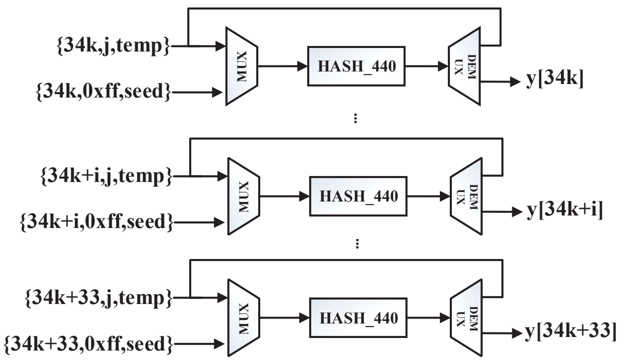

Fig. 6. OTS_Gen_y Module

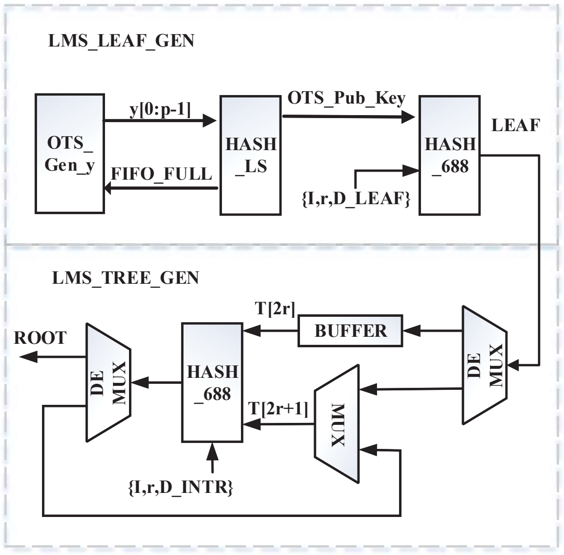

Fig. 7. LMS Public Key Generation

steps are merged to share the same hash processing element for hardware reuse. For the parameter sets where $p > 34$, this module would be executed several times to obtain all the array elements. Targeted at constant power, each hash core is activated even though there is no valid input. This method can prevent the design from being influenced by the power attack to a certain degree.

The upward side of Fig. 7 shows the architecture of the LMS leaf generation (LMS_LEAF_GEN). The intermediate array outputs y in Fig. 6 are rearranged and sent into the FIFO in the $HASH\_LS$ module whose width is 1088. Once the string y of OTS is generated, the following HASH_LS module will be started to calculate the final hash K for OTS public key. And one HASH_688 module is then required to obtain the leaf nodes of the binary tree. The downward side of Fig. 7 is the architecture of the LMS tree generation (LMS_TREE_GEN). $T[2r]$ is buffered and combined with $T[2r+1]$ as the input of the HASH_688 when $T[2r+1]$ is calculated. The root $T[1]$ of the binary tree is the expected public key of the LMS.

TABLE II
IMPLEMENTATION RESULTS OF LMS KEY GENERATION ARCHITECTURE

Resource	Number	Total	Percentage
LUT	190660	230400	82.75%
FF	126656	460800	27.49%
BRAM	17	312	5.45%

IV. Experiment Results and Comparisons

The architecture is coded with RTL and implemented on the Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. Table II shows the implementation results of the accelerator for key generation. LUT utilization is 190660, which takes about 82.75% resource of the board. Besides, 126656 FFs and 17 BRAMs are consumed, which take 27.49% and 5.45% respectively. The maximum clock speed can reach 285MHz.

Due to the fact that there is no existing hardware implementation in the open literature for the LMS system and the ARM implementation in [14] is too slow, we have to compare our scheme with software implementation. The open-source from cisco [15] is deployed on a server with Intel(R) Core(TM) i7-6850K 3.60GHz CPU with threading enabled. As shown in Table III, several parameter sets are tested. Compared to the CPU platform, 55x to 2091x speedup is achieved on FPGA. When $h = 5$ and $w < 8$, some necessary control instructions can not be shrunk since the calculation is reduced so that the accelerations are quite huge. But the speedup ratios become more stable when the h or w gets larger.

TABLE III
COMPUTING LATENCY OF HARDWARE AND SOFTWARE IMPLEMENTATION

h	w	FPGA(ms)	CPU(ms)	Speedup
5	8	0.748	1450	1938x
	4	0.097	180	1855x
	2	0.103	100	971x
	1	0.200	110	550x
10	8	23.90	46320	1938x
	4	3.059	591	193x
	2	3.229	325	101x
	1	6.689	366	55x
15	8	764.80	1599110	2091x
	4	97.83	19575	200x
	2	103.22	10422	101x
	1	202.77	11889	59x

V. Conclusion

In this paper, we proposed a high-speed and scalable architecture for the key generation of the LMS and implemented it on FPGA for the first time. Many architectural optimizations are used to increase the parallelism and scalability and decrease the latency. The control flow is well designed to fit any different parameter sets with constant power. The proposed architecture is coded with RTL and implemented on an FPGA. The experimental results show that the new hardware implementation much outperforms the conventional optimal software implementation.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[3] M. A. Simplicio Jr, E. L. Cominetti, and H. K. Patil, “Privacy-preserving linkage/revocation of vanet certificates without linkage authorities.”

[4] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, “High-speed high-security signatures,” *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[5] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[6] U. NIST, “Descriptions of sha-256, sha-384 and sha-512,” 2001.

[7] M. J. Dworkin, “Sha-3 standard: Permutation-based hash and extendable-output functions,” Tech. Rep., 2015.

[8] D. Micciancio and O. Regev, “Lattice-based cryptography,” in *Post-quantum cryptography*. Springer, 2009, pp. 147–191.

[9] L. De Feo, “Mathematics of isogeny based cryptography,” *arXiv preprint arXiv:1711.04062*, 2017.

[10] R. Overbeck and N. Sendrier, “Code-based cryptography,” in *Post-quantum cryptography*. Springer, 2009, pp. 95–145.

[11] R. C. Merkle, *Secrecy, authentication, and public key systems.* Stanford University, 1979.

[12] IETF, “rfcs,” Website, https://www.ietf.org/standards/rfcs/.

[13] D. McGrew, M. Curcio, and S. Fluhrer, “Rfc 8554–leighton-micali hash-based signatures,” IETF, Tech. Rep., apr 2019.[Online]. Available: https://datatracker. ietf . . . , Tech. Rep., 2019.

[14] F. Campos, T. Kohlstadt, S. Reith, and M. Stöttinger, “Lms vs xmss: Comparison of stateful hash-based signature schemes on arm cortex-m4.” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 470, 2020.

[15] C. Martin, “hash-sigs,” Website, https://github.com/cisco/hash-sigs.