Lightweight Post-Quantum-Secure Digital Signature Approach for IoT Motes

Santosh Ghosh, Rafael Misoczki, and Manoj R. Sastry

Security and Privacy Research, Intel Labs
Intel Corporation
2111 NE 25th Ave, Hillsboro, OR 97124
{santosh.ghosh, rafael.misoczki, manoj.r.sastry}@intel.com

Keywords: Hash-Based Signature (HBS) · XMSS · WOTS+ · Keccak · SHA-3 · Post-Quantum Cryptography · Security · IoT · Public-Key Cryptography

Abstract. Internet-of-Things (IoT) applications often require constrained devices to be deployed in the field for several years, even decades. Protection of these tiny motes is crucial for end-to-end IoT security. Secure boot and attestation techniques are critical requirements in such devices which rely on public key Sign/Verify operations. In a not-so-distant future, quantum computers are expected to break traditional public key Sign/Verify functions (e.g. RSA and ECC signatures). Hash Based Signatures (HBS) schemes, on the other hand, are promising quantum-resistant alternatives. Their security is based on the security of cryptographic hash function which is known to be secure against quantum computers. The XMSS signature scheme is a modern HBS construction with several advantages but it requires thousands of hash operations per Sign/Verify operation, which could be challenging in resource constrained IoT motes. In this work, we investigated the use of the XMSS scheme targeting IoT constrained. We propose a latency-area optimized XMSS Sign or Verify scheme with 128-bit post-quantum security. An appropriate HW-SW architecture has been designed and implemented in FPGA and Silicon where it spans out to 1521 ALMs and 13.5k gates respectively. In total, each XMSS Sign/Verify operation takes 4.8 million clock cycles in our proposed HW-SW hybrid design approach which is 5.35 times faster than its pure SW execution latency on a 32-bit microcontroller.

1 Introduction

IoT will play an extremely important role in the 21^{st} century by comprising millions of smart and connected devices offering benefits in a wide range of situations. A smart city whose street light, energy, water and waste managements rely on thousands of tiny smart sensors (IoT motes) is one example of them. These mote devices will be responsible to collect and transfer related information to the gateway/cloud; the cloud will perform data analytics and send the results to the particular management system who takes suitable action. This forms

the end-to-end network for IoT technology. Protecting this complete network against malicious events is one of the toughest challenges for deploying IoT technology. [14] provides a good analysis on feasible threats for a smart city. To allow cost and energy effective end-to-end IoT solutions, motes are usually comprised of a tight power and area constrained system-on-a-chip (SoC) which also help them to last in the field for a long period of time (either by means of a battery or by harvesting energy from the environment). However from the security perspective, these motes can be exploited by attackers as the weakest link of an IoT end-to-end network to undermine city infrastructure. Therefore, inbuilt security feature that can survive for a long period of time is a fundamental requirement for these IoT motes to provide IoT end-to-end security, although this contradicts the power and area constraints.

Typically, a SoC mote consists of a 8/16/32-bit microcontroller connected with peripheral hub to collect data from sensors and to transfer data to the cloud [23]. For ensuring that the transported data are genuine, it is critical to have the integrity and authenticity of the code running on the mote. Data collected and transfered by malicious code cannot be trusted. Secure boot is used to check the integrity and authenticity of firmware (FW) and software (SW) by means of public key signature verification. It comprises of multiple stages during the platform booting process. Typically, it starts with Root-of-Trust (RoT) [31], which is usually hard-coded in bootROM or One Time Programmable (OTP) flash to authenticate the first firmware. One firmware image checks the subsequent firmware or OS, thus establishing a chain of trust. A mote with authentic FW and SW is more reliably used to enable security operations that preserve end-to-end IoT security like secure communication, attestation etc.

The RSA [27] and Elliptic curve cryptography (ECC) [24] based public-key signature verification are traditionally used for FW/SW authentication during secure boot flow as well as remote attestation services. Both schemes rely their security on number-theory mathematical problems, such as integer factorization and the discrete logarithm problem. These problems are considered untractable for current technology but are expected to be solvable by future large-scale quantum computers due to the polynomial-time Shor's quantum algorithm [28]. Therefore, applications aiming at long-term security must not rely upon RSA or ECC, and alternative quantum-resistant schemes must be investigated.

In this context, Hash-Based Signatures (HBS) appear as a very attractive quantum-resistant approach. Their security is solely based on some well-known security notions related to hash functions. Hash functions are expected to be only marginally affected by quantum computers by means of Grover's attack [16]. This is the optimal case when compared to all literature on quantum resistant algorithms because these other alternatives depend not only on hash functions (used to map an arbitrary-length message into a fixed digest that is signed) but also on other less-studied mathematical problems (e.g., short vectors in a lattice). The Merkle signature scheme [22] is the best known hash-based signature scheme and dates back to the same period when RSA cryptosystem was proposed. Modern constructions based on Merkle scheme have been recently proposed. The XMSS

(eXtended Merkle Signature) scheme offers interesting benefits such as smaller signatures when compared to the classic Merkle scheme [11]. However, it still suffers from the somewhat intensive signing/verification procedure that require thousands of hash computations, a limitation to be deployed in constrained devices.

Contributions: This paper introduces a lightweight solution for post-quantum secure public key Sign/Verify technique which can preserve end-to-end security of IoT technology while the IoT edge devices can be deployed in the field for more than a decade. Our solution is based on the XMSS scheme with lightweight hash function. We use the Keccak-400 hash function in eXtended Output Function (XOF) mode as the underlying hash function in order to achieve an acceptable performance of the XMSS scheme for IoT motes while providing 128-bit preimage resistance against Grover's attack [16]. We chose suitable XMSS parameters that achieve 128-bit security against quantum attacks and ensures that the motes can not be exploited as the weakest part in the end-to-end IoT security. Furthermore, this paper provides an area-latency optimized HW-SW hybrid architecture for enabling proposed lightweight XMSS scheme on resource constrained IoT motes. To reduce the gap between idea and the practical deployment of the technology, we designed and implemented the proposed architecture in FPGA as well as in Silicon, and demonstrated its feasibility for IoT motes in terms of area overhead and performance.

The paper is organized as follows. A brief introduction on hash-based signatures and the XMSS HBS scheme is given in Section 2. Our proposed solution for lightweight XMSS scheme is described in Section 3. Section 4 describes the proposed HW-SW architecture followed by Section 5 that provides detailed descriptions of the internal HW modules. The implementation results and comparison with state of the art solutions are captured in Section 7. We conclude the paper in Section 8.

2 Hash-Based Signatures

Hash-Based Signature schemes are very promising quantum-resistant cryptographic constructions. Their security rely solely on the security of hash functions. In comparison, any other digital signature scheme (pre-quantum or post-quantum) with appended message relies not only on the security of hash functions (used to map messages of arbitrary length into fixed length message representatives) but also on some other (likely less studied) computational problem. In fact, even if a well-known hash function is broken, the security of HBS schemes is not affected at all as this would only suggest that such particular hash function is not recommended. In summary, HBS schemes can be made secure as long as there exists at least one secure hash function, which is a minimal assumption when compared to all assumptions required by other schemes.

The efficiency of HBS schemes inherently depends on the efficiency of the underlying hash function since all operations (key-generation, signing and verify-

ing) require a large number of hash computations. Selecting different underlying hash function allows meeting different performance requirements.

The best known HBS schemes are either *one-time signature (OTS)* or *multi-time signature (MTS)* schemes. OTS schemes limits the use of a private key to sign only a single message. If the same private key is used to sign a second message, the scheme loses its security guarantees. MTS schemes allows signing multiple messages with a same key pair but are less efficient than OTS schemes. In fact, MTS schemes use as a building-block an OTS scheme.

2.1 XMSS Scheme

In this work, we will be particularly interested by the XMSS (eXtended Merkle Signature) scheme [11], which uses the WOTS+ scheme [18] as a building block. The WOTS+ scheme [18] is an improvement on top of the classic Winternitz scheme [22] as it allows shorter hash lengths for a same security level. Winternitz-like signature schemes have a straightforward rationale. In a simplified description, the private key is a set of random bits. The public key is computed from applying a hash function on the private key a fixed number $N \in \mathbb{N}$ of times (the output of one iteration works as the input of the next iteration). To sign a message, seen as an integer $m \in \mathbb{N}$, $0 < m < N$, the hash function is applied m times on the private-key. To verify a signature σ, the hash function is applied $(N-m)$ times on σ. If the result matches the public key, the signature is authentic; otherwise, it should be rejected. From this simple rationale, it is possible to design OTS schemes that solely rely on well-known security properties of hash functions (e.g. pre-image or collision resistance).

The XMSS scheme [11] improves the classical Merkle scheme [22] by allowing shorter hash lengths for a same security level, thus leading to smaller signatures. Merkle-like signature schemes focus on enabling multi-time signatures for the limited one-time schemes. They bind a large number of one-time public keys into a single public key using a data-structure called Merkle tree, which is a binary tree. In this way, a one-time key is used to sign a single message but all signatures are verified using a same public key. The root of the Merkle tree is the overall public key, while the leaf nodes are constructed from the one-time public keys. The rule to build this tree (from the leaf nodes up to the root node) depends on the particular scheme. Figure 1 shows a toy-size Merkle tree for the classical Merkle scheme, which has the leaf nodes computed as the hash of the one-time public keys and the non-leaf nodes are constructed by hashing the concatenation of the two children nodes.

The signing process consists of signing the message with a one-time key and then computing the so-called *authentication path*. The authentication path consists of $\mathcal{H}$ nodes needed to reconstruct the root node. As an example, consider Figure 1 and assume that the first (the left most, of index 1) one-time key pair is used. The authentication path will include nodes h_2 and h_6. The tree height determines the number of maximum signature per Merkle key pair: $2^{\mathcal{H}}$. To verify a signature, one needs to at first verify the one-time signature in order

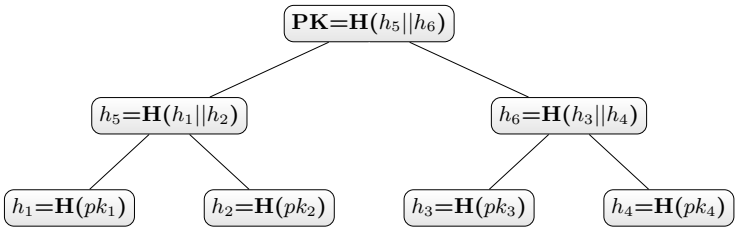

Fig. 1: Merkle-Tree of height 2 and 4 leaf nodes.

to produce the corresponding one-time public key. Then, with the assistance of the authentication path and the hash of the one-time public key, it is possible to reconstruct the root of the tree by following the tree construction rule. If the result matches the Merkle public key, then the signature is accepted; otherwise, it should be rejected.

Merkle-like schemes are stateful because it is necessary to keep track which leaf nodes have already been used to sign messages (and thus should not be used anymore, given the one-timeness property of the underlying OTS scheme). In other words, a state needs to be securely maintained by the signer in order to prevent duplicate usages of one-time keys. This represents an additional requirement for the HBS when compared with traditional signature schemes, and there are works describing how to satisfy this requirement in practice [21].

Now that we presented the intuition on how Merkle-like signature schemes work, we will define the XMSS scheme. The XMSS scheme uses a different rule to build the tree and relies on the WOTS+ scheme, instead of the classical Winternitz. The parameters are $(w, n, m \in \mathbb{N}, \mathcal{H})$, which represents the Winternitz parameter, the underlying hash digest length, the digest length of the hash used to create message representatives and the tree height. The XMSS tree is built using the following rule: $Node_{i,j} = h_k((Node_{2i,j-1} \oplus bm_{l,j})||(Node_{2i+1,j-1} \oplus bm_{r,j}))$, $0 < j < \mathcal{H}$, $0 < i < 2^{\mathcal{H}-j}$. The bitmasks $(bm_{l,j}, bm_{r,j}) \in \{0,1\}^{2n}$ are sampled uniformly at random and h_k is chosen randomly from the set $\{h_k : \{0,1\}^{2n} \to \{0,1\}^n | k \in \{0,1\}^n\}$.

Key generation consists of generating $2^{\mathcal{H}}$ WOTS+ key pairs. Each WOTS+ public-key is used to build an L-Tree, which is a binary tree of ℓ leaf nodes that follows the tree building rule described above, and is used to compress an Ln bits one-time public key into an n-bits. The root nodes of the L-Trees are used as leaf nodes of the (main) Merkle tree. The XMSS public key is the root of the Merkle tree plus the bitmasks. The private key is either the $2^{\mathcal{H}}$ WOTS+ private keys or, more conveniently, an n-bit seed used to generate all one-time private keys assuming that a secure pseudo random number generator is available.

To sign the i-th message, the i-th WOTS+ private key is used and the signature is $SIG = (i, \sigma, Auth)$, where i is an index between $0 < i < 2^{\mathcal{H}} - 1$, σ is the the WOTS+ signature, and $Auth \in \{0,1\}^{\mathcal{H} \times n}$ is the authentication path. Although there are several candidate techniques to compute the authenti-

cation path, [12] is suggested as it provides optimal balanced runtime with little memory requirement.

To verify a signature SIG, the WOTS+ signature σ is verified in order to produce the WOTS+ public key **pk**. Then using both WOTS+ public key and the authentication path, the verifier is able to reconstruct the root of the Merkle tree. If the result matches the XMSS public key, the signature is valid; otherwise, the signature is rejected.

3 Lightweight PQ-Secure XMSS for IoT Motes

IoT motes are considered as ultra resource constrained devices that harvest energy from the environment. To meet the energy budget, a lightweight XMSS solution is of great relevance. We considered three key configuration knobs for optimizing XMSS algorithm for resource constrained IoTs.

- **Smaller Parameters:** The WOTS+ parameters (w, n, m) and tree height play important roles with respect to the computational cost of the XMSS Sign or Verify operations. By reducing length of those parameters, we can achieve a lightweight XMSS scheme constructions.
- **Lightweight Hash Function:** The underlying hash algorithm is the major operation which executes repeatedly for several thousand of times while computing each XMSS Sign/Verify operation. A suitable lightweight hash function linearly reduces the compute intensity of an XMSS scheme.
- **Design Optimization:** To meet the energy budget and smaller-die area constraints of IoT motes, specific design tread-offs can be taken in each architecture level of XMSS Sign/Verify computation.

In applications of IoT technology, there is a very limited scope for compromising length of the WOTS+ parameter because the motes are essentially deployed in the field for long period of time. These parameters should be sufficiently long for surviving against attacks by quantum computers. Later two are the major knobs for enabling the PQ-security features to IoT motes. This section provides a brief description about the WOTS+ parameters and energy constrained hash function that are used in this work for enabling XMSS hash based signature scheme in IoT motes; whereas, the following sections describes design optimizations and the final XMSS solution for IoT motes.

3.1 The XMSS/WOTS+ Parameters

Table 1 provides the value of the XMSS/WOTS+ parameters that are used in this work.

Table 1: XMSS/WOTS+ Parameters

n	w	h	len_1	len_2	len
256	16	16	64	3	67

3.2 Energy Constrained Hash Function for XMSS

We investigated several hash algorithms in terms of their performance and HW costs based on existing analysis reported in [30]. Thereafter, we implement and perform explicit area and latency analysis of four hash candidates on same technology. The goal of this analysis is to find out a good hash function for energy efficient computation of XMSS Sign/Verify operations with selected WOTS+ parameters. The first hash candidate was SHA-256 which is one of recommended hash functions by XMSS authors. We implement SHA-256 based on the smallest implementation reported in [19]. Our second candidate is SHAKE-256 which is part of SHA-3 standard and also recommended by the XMSS authors as alternate hash function. We considered the round based implementation as reported in [13, 17] with 30500 gates of area and 24 clock cycles of latency per block. The smallest SHA-3/SHAKE implementation is reported in [26] using only 5898 gates which is $5.17x$ smaller than round based one. This design takes 15427 clock cycles to process one input blocks which is $643x$ higher than the latency of a round based design. Therefore, this smallest design will consume significantly higher energy than a round based design which is not appropriate for energy constrained IoT motes for hashing a boot-image and performing HBS operations. Further, we select s-quark which is a popular non-standard lightweight hash algorithm [4]. Our third candidate is Keccak-400 [7]. We consider the keccak-400 for two reasons. First, the algorithm Keccak is the new SHA-3 standard. Other than the 1600-bit state size used in the SHA-3 standard, Keccak provides flexibility to use smaller state sizes (800, 400, 200 etc) based on particular security and bitrate requirement. Second, we observe that if we choose its 400-bit state size and configure as (eXtended Output Function) XOF with 128-bit bitrate, 128-bit digest and 256-bit extended-output then it can provide 128-bit pre-image and second-preimage resistance against quantum computers. We are able to process the PQ-secure WOTS+ parameters with this Keccak configuration very efficiently by using two absorb and two squeeze operations.

Figure 2 provides the area and latency comparison when we implemented them on a 14nm technology [1, 8, 9]. These round based implementations span out 30.5k 6.8k, 2.5k and 3.6k gate equivalents (GE) with 24, 64, 1024 and 20 clock cycles latencies for SHAKE-256, SHA-256, S-quark and Keccak-400 respectively. It concludes that the keccak-400 provides 10.2x, 6x and 18x lower $area \times latency$ product compared to SHAKE-256, SHA-256 and s-quark which is linearly proportional to the energy consumptions for XMSS Sign/Verify operations.

4 Architecture for XMSS Operations

Our design favors minimal hardware area & latency, simplicity and flexibility. To achieve these goals, at first, we identified what are the operations from the XMSS and WOTS+ algorithms that would benefit the most from hardware acceleration, and what are the ones which will occupy hardware area without significant gains in terms of performance.

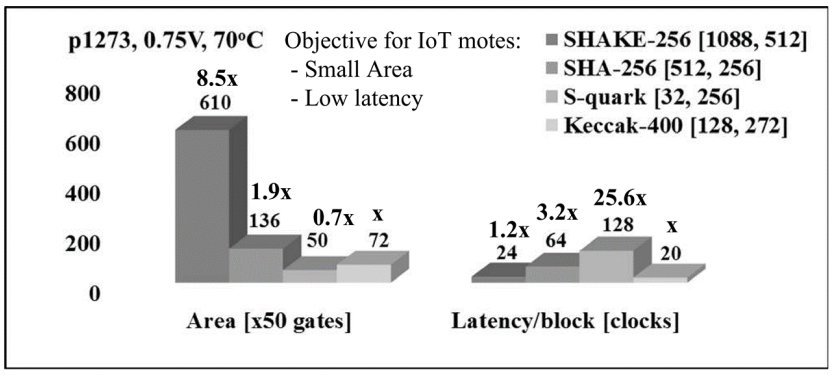

Fig. 2: Latency of Keccak-400 vs standard SHA-256 vs lightweight s-quark.

WOTS+ operations, such as key generation, signing and verification have been identified as the most suitable for hardware acceleration since they require a significant amount of repetitive computations (hash calls). On the other hand, the XMSS operations, in particular, sign and verify, operates in a higher level of abstraction that would benefit less from hardware acceleration but still requiring additional area overhead. Authentication path computation algorithms, such as [12], also have significant logical complexity. To favor flexibility, we left the WOTS+ parametrization being defined in the software level.

Therefore, to preserve the simplicity of the design with the most significant latency gains without penalizing hardware area, we opted to offload to hardware the WOTS+ operations, and leave in software the XMSS operations and WOTS+ parametrization control. Figure 3 shows the pictorial view of the proposed HW-SW design layers. A HW driver module is collocated within the XMSS SW library that configures the underlying WOTS+ HW Engine for specific operations.

SW Layer 1: XMSS Key generation, Sign, Merkel tree management, and Verify
SW Layer 2: WOTS+ HW Driver
HW: WOTS+ Key generation, Sign, Verify

Fig. 3: The SW and HW boundary and architecture for XMSS.

4.1 XMSS SW and WOTS+ HW Driver

Regarding the XMSS software library, it is a software library that assumes four logical modules:

- xmss.c: responsible for providing all XMSS operations, such as key generation, sign, verify and authentication path update.
- wots.c: responsible for defining the WOTS+ parameters, and either calling a hardware API of the WOTS+ operations (as in the work here described) or providing the WOTS+ operations, such as key generation, sign and verify.
- functions.c: responsible for providing the underlying XMSS/WOTS+ functions (the ones that actually call the underlying hash function). Note that this module is not needed in software when the WOTS+ operations are assumed to be offloaded in hardware.
- hash.c: responsible for providing the underlying hash function operations. Note that this module is not needed in software when the WOTS+ operations are assumed to be offloaded in hardware.

In this work, we assumed that the WOTS+ operations are offloaded to a hardware acceleration. For this reason, the XMSS module (xmss.c) is the only complete module implemented in software. The WOTS+ module (wots.c) is still needed in software to control the parameters chosen for the scheme, however all operations such as WOTS+ key generation, signing and verification are called through the hardware API, thus benefiting from a better performance in those operations.

The WOTS+ HW Driver is implemented in C and integrated as the second layer of our XMSS SW library which can be viewed as the closest SW layer to the WOTS+ HW Engine. Fig 4 describes the execution flow of the driver. The driver module communicates with the WOTS+ HW Engine by executing Memory-mapped I/O (MMIO) instructions. It writes specific instructions, special control signals and data into the address space of the WOTS+ HW Engine in the platform. After configuring the HW, a typical driver can move to either of the two states. It can set the interrupt mask to the WOTS+ HW Engine and return control to the upper SW layer. Once the HW completes the current execution, it then generates an interrupt to the microcontroller (MCU) unit and so the control goes through the interrupt vector routine and then the XMSS SW again initiate the driver module to read back the result. The MCU can perform some other task in between. The This is little complicated process that requires an interrupt generation logic in the WOTS+ HW Engine and a dedicated interrupt line to the microcontroller unit. Only advantage of this process is that MCU can utilize its cycles for executing some other tasks in parallel with WOTS+ HW Engine. Though, the context switch and execution of interrupt vector routine neutralizes a significant latency benefit as the WOTS+ HW Engine completes each operation very quickly.

We implemented the simpler alternate technique that is described in the flow diagram. After configuring the WOTS+ HW Engine, the driver move into a

busy loop where it continuously polls (MMIO Read) the status of the WOTS+ HW Engine. Once it receives a satisfactory value of the status register which indicates that the engine has completed the current execution then it reads out the result from specific addresses and returns the control to the upper SW layer. This technique saves a lot of context switchings as well as saves HW design complexity which is more suitable for an SoC targeting for IoT motes.

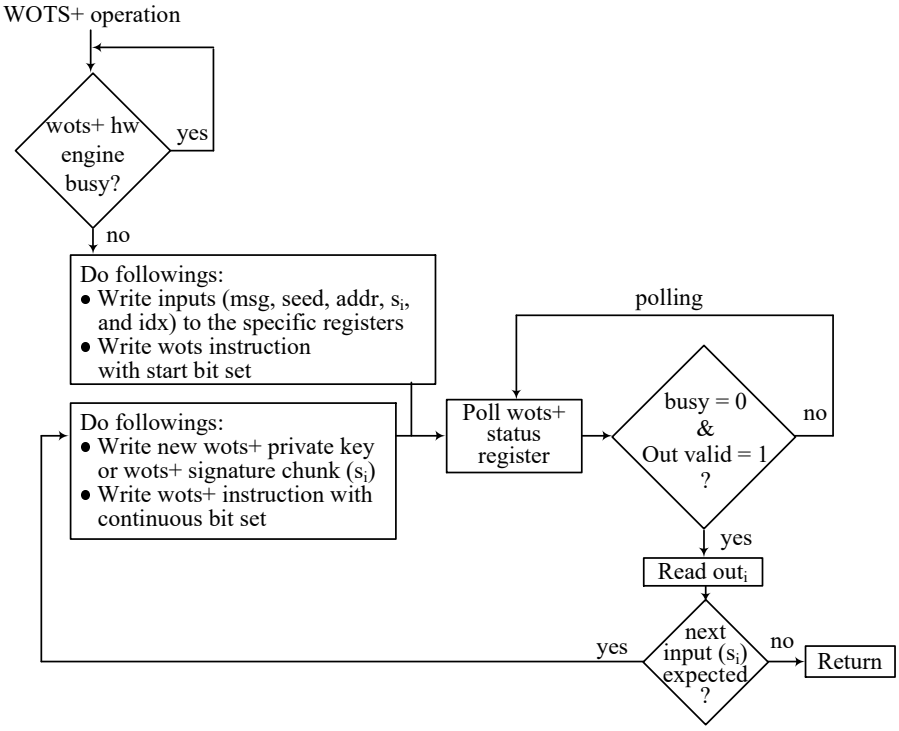

Fig. 4: Execution flow of the HW Driver.

5 The WOTS+ HW Engine

Figure 5 depicts the top level architecture diagram of the WOTS+ HW Engine which comprised of the WOTS+ Functional Block, a set of addressable registers and additional interface logic. The XMSS SW stack is expected to be executed in a small microcontroller which is connected with the WOTS+ HW Engine through a bus protocol, like: AMBA [3]. It has an inbuilt driver module that configures the HW Engine by writing a set of specific values into the addressable registers. There is a set of instructions for offloading WOTS+ operations

and stand alone Hash function to the HW Engine which are described in the following section. The interface logic emulates the bus protocol for participating the communication between the WOTS+ HW Engine and the microcontroller unit.

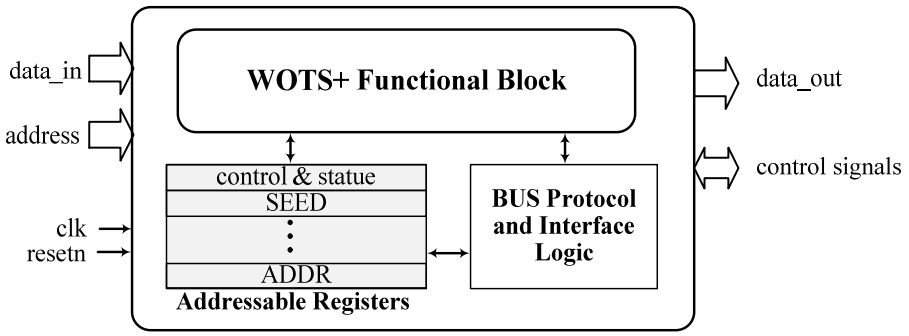

Fig. 5: The WOTS+ HW Engine for XMSS PKC solution for IoT motes.

Figure 6 depicts the top level view of the WOTS+ Functional Block within the WOTS+ HW Engine. It consists of a Keccak400 hash primitive, hash_chain computation block, wots_gensk block and wots_genpk/sign/verify block. The later one is a combined block for computing WOTS+ public key generation, sign and verify operations all boiling down to calls to the hash_chain block. Both hash_chain block and wots_gensk block executes the respective operations by invoking Keccak400−128/256 primitive which operates as XOF with 128-bit bitrate and 256-bit output. Table 2 describes the functionalities of each of the input and output interface.

5.1 WOTS+ Private Key Generation

Algorithm 1 generates the WOTS+ Private Key which has been adapted to be called from the XMSS signatures scheme. It receives the SK_KEY which is part of the XMSS private key, the index idx that specifies which WOTS+ private key should be generated among the 2^h possible, the len_1 and the len_2 inputs. Our PQ-secure WOTS+ HW Engine is based on 256-bit SK_KEY and 32-bit idx inputs. It uses $\text{len}_1 = 64$ and $\text{len}_2 = 3$. The proposed WOTS+ HW implements both G and PRF functions using Keccak-400 XOF hash operations. The $toByte$ function extends the 32-bit idx and 8-bit i to 256-bit value by perpending zeros which is implemented by wiring into the inputs to the Keccak-400 module. We compute the seed S by invoking the keccak-400 XOF HW module for five times. First four invocations absorb 256-bit SK_KEY followed by 256-bit padded idx; whereas, the final one squeezes XOF for once more for generating 256-bit S output.

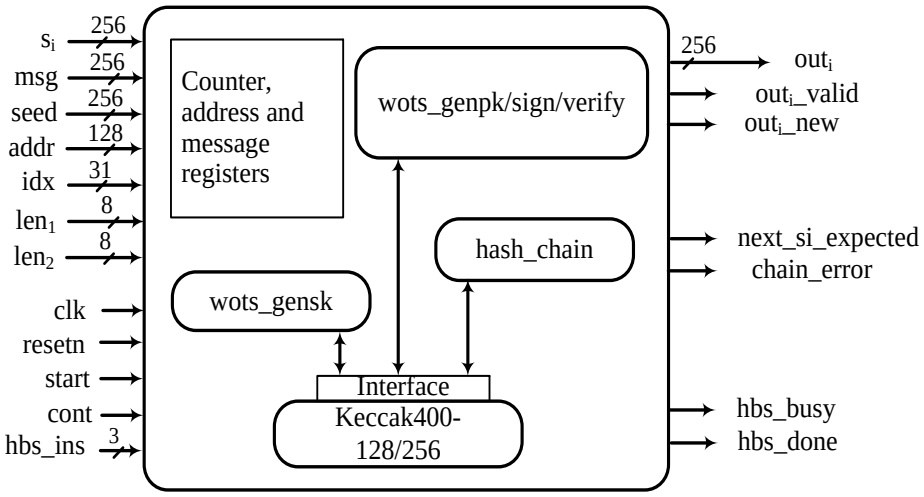

Fig. 6: Architecture and interface of WOTS+ Functional Block.

Algorithm 1 The WOTS+ private key generation

Require: SK_KEY, idx, len_1, len_2
Ensure: The Private key sk
  $S \leftarrow G(SK\_KEYS,\ toByte(idx, 32))$
  for $i = 0$ to $len_1 + len_2 - 1$ **do**
    $sk[i] \leftarrow PRF(S,\ toByte(i, 32))$
  return sk;

Table 2: Interface of the WOTS+ Functional Block.

Signal Name	**In/Out**	**Width (bits)**	**Functionality**
s_i	Input	256	Provides the following inputs: a) The sk_keys for wots_gensk b) sk_i during wots_genpk and wots_sign c) $Sign_i$ during wots_verify.
msg	Input	256	Hash of the whole message.
seed	Input	256	WOTS+ parameter used in wots_genpk, wots_sign and wots_verify.
addr	Input	256	HBS parameter used in wots_genpk, wots_sign and wots_verify.
idx	Input	32	Specifies which WOTS+ private key should be generated (among the 2^H possible).
len1	Input	8	$\lceil (8M/lg(W)) \rceil$
len2	Input	8	$\lfloor (lg(len1(W-1))/lg(W)) \rfloor + 1$
clk	Input	1	Functional clock input.
resetn	Input	1	Active low asynchronous reset.
start	Input	1	It is a pulse to starts a new WOTS+ operation which may contain multiple elements.
cont	Input	1	It is a pulse input to continue the same WOTS+ operation (which is already started) with consecutive intermediate sk_i or $sign_i$.
hbs_ins	Input	3	HBS instruction which specifies the target operation. They are defined as: wots_gensk = 001; wots_genpk = 010; wots_sign = 011; wots_verify = 100; hash_chain = 101; keccak400_new = 110; keccak400_continue = 110;
outi	Output	256	Provides 256-bit outputs of ski, pki, signi, verifyi, hash_chain and keccak-400_hash.
outi_valid	Output	1	1 output in this port indicates that there is a valid data in the outi port
new_outi	Output	1	This is a pulse, indicates that the new output is available in the outi port.
next_si_expected	Output	1	1 in this port indicates that the HBS engine is waiting for the next s_i input.
chain_error	Output	1	1 means hash_chain error, i.e., $(i+s) > (W-1)$.
hbs_busy	Output	1	1 means the engine is busy.
hbs_done	Output	1	1 means the engine has completed the assigned operation.

The 67 private key components are computed iteratively by hashing the S followed by current iteration number that varies from 0 to 66. For all 67

iterations, the first 256-bit input to the hash function is same S. Therefore, it can be either pre-compute the hash of S and utilize it for 67 times or execute for all 67-times. Former one provides latency benefit by reducing 132 keccak-400 executions which take 2640 clock cycles in our design. However to accommodate this latency benefit there are following two HW overheads. First, an additional 400-bit register is necessary inside the WOTS+ HW for storing the pre-computed Keccak-400 state. Second, keccak-400 engine has to be additional feature for loading its initial state variable. The later approach on the other hand, takes 2640 clock cycles overhead which is 60% slower than former one. However, it has no HW overhead which is equivalent to 75% lower register counts than former design approach. We took the later design approach for current WOTS+ HW engine that targets resource constraints and cost effective IoT motes.

5.2 WOTS+ Public Key Generation

Algorithm 2 generates the WOTS+ public key for a corresponding input private key. It takes additional three inputs: the WOTS+ address variable $ADDR$, the 256-bit $SEED$, and the parameter w. Our proposed hardware computes the whole public key in 67 iterations. It takes $SEED$, $ADDR$ and w at the beginning for one time and each 256-bit $sk[i]$ input at each iteration. It updates the 32-bit chain-address field in the $ADDR$ (shown in Fig. 7) based on current iteration number and starts the hash_chain module into the WOTS+ HW (Fig. 6) for computing the chain function on current inputs. The hash_chain module returns the respective 256-bit public key components $pk[i]$ which is stored into the output register. The WOTS+ HW then sets the respective fields into its status register with hbs_busy = 0, outi_valid = 1 and next_si_expected = 1. The XMSS SW module polls the wots+_status register periodically after sending the current $sk[i]$. Once the SW module receives the status with above values, it reads out the corresponding $pk[i]$. The XMSS SW then sends the next $sk[i]$ to the WOTS+ HW for its next $pk[i]$ computation.

Algorithm 2 The WOTS+ public key generation

Require: sk, $SEED$, $ADDR$, w
Ensure: The WOTS+ Public key pk
  for $i = 0$ to $len_1 + len_2 - 1$ **do**
    $ADRS.setChainAddress(i)$
    $pk[i] \leftarrow chain(sk[i],\ 0,\ w-1,\ SEED,\ ADRS)$
  return pk;

5.3 Computation of WOTS+ Sign

The WOTS+ sign function (Algorithm 3) takes the WOTS+ private key and the message as the input. For longer messages it is first hashed and the sign

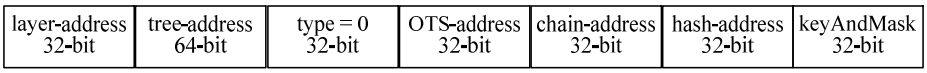

Fig. 7: The $ADDR$ format for WOTS+.

function takes the fixed length digest of the message for generating the WOTS+ signature. The current design supports 256-bit M with 64 chunks of base w where $\lg(w) = 4$. The WOTS+ engine computes the checksum of the input message based on the respective base w value of each 4-bit chunk, which is then formated in base w forms. In our HW, checksum is computed in 64 clock cycles on a 16-bit adder circuit. The base w computation is easy as $w = 16$. In the HW, we compute it on-the-fly by accessing 4-bit at a time from the circular msg and checksum registers. The signature components are computed iteratively by adjusting the chain_address field in the $ADDR$ input followed by staring the hash_chain module with current chunks of the private key, the base w message value, the $SEED$ and the $ADDR$. The hash_chain module returns the respective 256-bit signature components $sig[i]$ which is stored into the common output register that is already described in the previous sub-section for storing the public key.

Algorithm 3 The WOTS+ signature generation

Require: sk, M, $SEED$, $ADDR$, len_1, len_2, w
Ensure: The WOTS+ Signature sig for M
  $csum \leftarrow 0$
  $msg \leftarrow base_w(M,\ w)$
  for $i = 0$ to $len_1 - 1$ **do**
    $csum \leftarrow csum + w - 1 - msg[i]$
  $csum \leftarrow csum << (8 - ((len_2 * lg(w))\%8))$
  $len\_2\_bytes \leftarrow \lceil((len_2 * lg(w))/8)\rceil$
  $msg \leftarrow msg \mid\mid base_w(toByte(csum,\ len\_2\_bytes),\ w)$
  for $i = 0$ to $len_1 + len_2 - 1$ **do**
    $ADRS.setChainAddress(i)$
    $sig[i] \leftarrow chain(sk[i],\ 0,\ msg[i],\ SEED,\ ADRS)$
  return sig;

The WOTS+ HW Engine then sets the respective fields into its status register with hbs_busy = 0, outi_valid = 1 and next_si_expected = 1. The XMSS SW module polls the wots+_status register and reads out the corresponding $sig[i]$. The XMSS SW then sends the next $sk[i]$ to the WOTS+ HW for its next $sig[i]$ computation. Note that the XMSS SW module sends all other inputs of the signature generation algorithm are applied only once at the beginning to the proposed WOTS+ HW. It only sends the private key chunks one at a time for each iteration and receives one signature components from each iteration. In our

HW each such iteration is based on 120 keccak-400 executions which in total takes 2400 clock cycles. Therefore, the computation of the complete WOTS+ signature takes $161 \times 10^3 + \delta$ clock cycles, where δ varies on system interface and data transfer delay from SW accessible memory to the WOTS+ HW registers.

5.4 WOTS+ Signature Verification

The verification of WOTS+ is performed by regenerating the public key from the signature and the message M. If the generated public key matches with the one time WOTS+ public key that corresponds the WOTS+ signature then we say signature is valid otherwise it is considered as an invalid signature. In the XMSS signature scheme, WOTS+ public keys are plugged into a Merkle tree where the root node is used as the multi-time public key. The signature verification in XMSS scheme therefore does not keep track of the WOTS+ public keys rather it uses the root of the Merkle tree as the public key. It reconstructs the root-node from the regenerated WOTS+ public key by Algorithm 4 and corresponding authentication path that comprised of intermediate nodes necessary for computing the root node from a leaf node.

Algorithm 4 Computation of WOTS+ public key from signature

Require: sig, M, $SEED$, $ADDR$, len_1, len_2, w
Ensure: The WOTS+ public key pk for sig
  $csum \leftarrow 0$
  $msg \leftarrow base_w(M, w)$
  for $i = 0$ to $len_1 - 1$ **do**
    $csum \leftarrow csum + w - 1 - msg[i]$
  $csum \leftarrow csum << (8 - ((len_2 * lg(w))\%8))$
  $len\_2\_bytes \leftarrow \lceil ((len_2 * lg(w))/8) \rceil$
  $msg \leftarrow msg \mid\mid base_w(toByte(csum,\ len\_2\_bytes),\ w)$
  for $i = 0$ to $len_1 + len_2 - 1$ **do**
    $ADRS.setChainAddress(i)$
    $tmp_pk[i] \leftarrow chain(sig[i],\ msg[i],\ w - 1 - msg[i],\ SEED,\ ADRS)$
  return tmp_pk;

Our WOTS+ HW Engine regenerates the public key iteratively from a given signature and message. It takes one 256-bit of signature component in every iteration and starts the hash_chain module for computing the respective public key chunk. In total, 67 public key components are computed in 67 iterations in similar input/output flow that is already described in previous sub-sections. The public key regeneration on our WOTS+ HW takes same amount of time with signature generation on average.

5.5 Chain Function

In our WOTS+ HW, there is a shared hash_chain module that computes the underlying chain functions for public key generation, sign and verify procedures. Algorithm 5 provides the recursive definition of the WOTS+ chain function as defined in [18]. In this work, we converted the chain function in its equivalent iterative form. A counter is used to keep track of each iterations from 0 to j. The register hc_out is initialized with input S and updated with the new intermediate result of each iteration. We compute the KEY, BM and new hc_out by utilizing Keccak-400 module. Each of the PRF and F functions in our design consists of 512-bit inputs which are absorbed by the current HW module in four consecutive Keccak-400 operations. Thereafter, for producing 256-bit output we use one additional squeeze step. In total, each of the PRF and F function are computed by five Keccak-400 operations which takes 100 clock cycles in the current design. Note that, alternatively one can pre-compute the Keccak-400 on the $SEED$ and reload the resultant state which saves 40 clock cycles for each PRF execution. As we described before, this technique requires additional 400-bit registers and related combinational circuits to load and handle the intermediate Keccak state. We avoided the incremental area overhead in this work for scope in the proposed XMSS solution for IoT motes.

Algorithm 5 chain: The WOTS+ hash chain function

Require: S, i, j, w, $SEED$, $ADDR$
Ensure: The hc_out
  if j is equal to 0 **then**
    return S;
  if $i + j > w - 1$ **then**
    return $NULL$;
  $hc\_out \leftarrow chain(S,\ i,\ j-1,\ SEED,\ ADRS)$
  $ADRS.setHashAddress(i + j - 1)$
  $ADRS.setKeyAndMask(0)$
  $KEY \leftarrow PRF(SEED,\ ADRS)$
  $ADRS.setKeyAndMask(1)$
  $BM \leftarrow PRF(SEED,\ ADRS)$
  $hc\_out \leftarrow F(KEY,\ tmp\ XOR\ BM)$
  return hc_out

Figure 8 demonstrates the hash_chain module of the proposed WOTS+ HW engine. The state machine controls and sequences the intermediate operations. It generates the start and continue signals for the Keccak-400 module where a start pulse indicates a new Keccak-400 operation with all zero initial state and continue pulse indicates to absorb the input on the existing Keccak state. After initiating a Keccak-400 operation, the state machine wait in a state for receiving a "1" in keccak_done signal on which it restores output into the specific register.

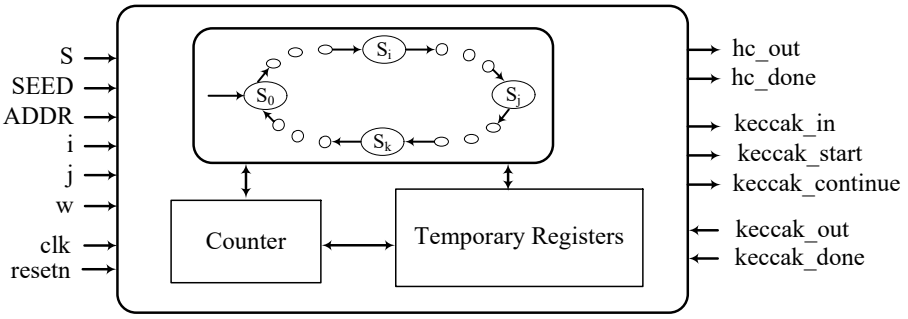

Fig. 8: The hash_chain module in WOTS+ HW engine.

6 Area-Latency Optimized Keccak-400 HW Design

Figure 9 shows our design architecture of the Keccak-400 module. A 400-bit state register is used to store the state variable. The sub-round functions $(\theta, \rho, \pi, \chi, \iota)$ are computed back to back on fully combinatorial datapath. Several logic level optimizations have been taken place in the current design for reducing area. For example, round constant (RC) are generated through optimized Boolean functions of a 5-bit LFSR based round counter output, which eliminates the additional storage for RCs in the IoT motes. A logic high input to start or continue signal initiates the Keccak-400 module for executing a new operation. During ip_busy period, at every clock the state register is updated by the computed round output. However, all other time the state register holds the old state, so that through a high value on continue signal the Keccak-400 module initiates its next execution. During absorb steps, we apply the specific 128-bit input block on the msg input port; whereas additional squeezes are performed by applying 128-bit zero input to the msg port and a high pulse to the continue signal.

The internal datapath of the Keccak-400 module comprised of linear XOR gates and non-linear AND gates. The rotations (in θ and ρ steps) and the permutation (in π step) are implemented by simple rewiring which do not consume any logic cell. Each round is computed within one clock period by the current datapath and hence the latency of one keccak-400 operation is 20 clock cycles in our proposed design.

7 Results

We developed RTL of the WOTS+ HW engine in Verilog (HDL) and SW stack in C for XMSS operations. For demonstrating the functionality, the HW design has been synthesized, place & route and implemented on Intel (Altera) Stratix IV FPGA by integrating it with a 32-bit microcontroller and additional peripherals. The WOTS+ HW module consumes 2963 combinational logic cells and 2337 register cells within that the Keccak-400 module spans out to 838 and 406

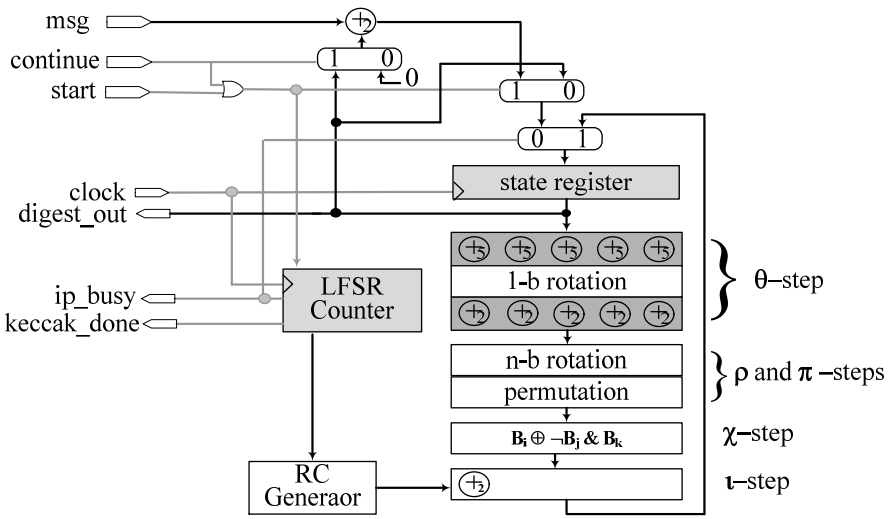

Fig. 9: The Keccak-400 module in WOTS+ HW engine.

combinational and register logic cells. All combinational and register cells are fit within 1521 Adaptive Logic Modules (ALMs) of a Stratix IV device. Table 3 and Table 4 provide latency numbers that are measured form our implementation. We assume that the message to be signed in both WOTS+ and XMSS experiments is the one which splits the effort equally between the signing and verification processes.

First we compared the latency improvement from SW execution of WOTS+ operations on a low-power 32-bit Intel Quark microcontroller [15] to the execution on our WOTS+ HW engine. The WOTS+ HW engine provides more than two order of magnitude speed up for computing each of the WOTS+ operations. In the current design, WOTS+ HW is connected with microcontroller unit by 32-bit bus through which SW driver sends and receives data to and from the WOTS+ engine. This bus takes four clock cycles to perform each write/read operation to/from the WOTS+ engine which are included into the WOTS+ latency numbers.

Table 3: Latency of WOTS+ operations on software and WOTS+ HW Engine

WOTS+ Operations	**Software Latency [clock cycles]**	**WOTS+ HW Latency [clock cycles]**	**Speed up**
Private key generation	1 042 304	6 325	164x
Public key generation	42 304 000	349 600	121x
WOTS+ Sign	21 160 512	176 944	120x
WOTS+ Verify	21 160 512	176 944	120x

For the XMSS operations, we measured the latency numbers for pure SW execution and the current hybrid approach for both sign (excluding the authentication path computation, which can always be computed offline) and verify processes. The XMSS SW stack configures the WOTS+ HW through simple MMIO operations and do polling from a specific status register to inform completion of the current WOTS+ operation. Upon receiving a zero at WOTS+ busy bit the SW module reads back the result from specific registers by MMIO calls and reconfigure the WOTS+ engine for processing the next chunk of inputs (Sig[i], sk[i] etc). It is demonstrated that the current solution provides 5.35 times speedup for XMSS sign/verify operation compared to the pure SW approach for 32-bit microcontroller based IoT motes. Further speedup can be achieved by offloading the internal hash operations to the WOTS+ HW engine during Merkle-Tree node computations in XMSS sign/verify.

Table 4: Latency of XMSS operations on pure software and our hybrid solution

XMSS Operations	**Software Latency [clock cycles]**	**WOTS+ HW Latency [clock cycles]**	**Speed up**
XMSS Sign	25 797 728	4 814 160	5.35x
XMSS Verify	25 797 728	4 814 160	5.35x

There are few elliptic curve based lightweight public key solutions exist [6,20] which are suitable for IoT motes. Table 5 provides a cost and performance comparison of our proposed XMSS scheme with existing solutions. A lightweight implementation of elliptic curve over 2^{131} filed is described in [6] with 15k gates with 75 thousand clock cycles latency per scalar multiplication. A relatively new work in [20] demonstrates that a NIST p-256 elliptic curve operations can be computed in RFID Tags by executing 34kB codes in more than 15 million clock cycles. [10] describes a 16-bit microcontroller based memory mapped ECC coprocessor for NIST p-256 elliptic curve with 5,933 gates and 256 $\times$ 16-bit RAM on 130nm CMOS technology. In total, it spans out to 11.7k gates as estimated based on RAM size estimation provided in [29]. It takes 6 millions clock cycles for one scalar multiplication. The first one provides only 65 bits security which is not acceptable even in today as per standard security recommendations by NIST [5]. The later two are good for current usages without providing a long term security [25]. In order to have better area measure for silicon, we synthesized our WOTS+ HW engine in 14nm CMOS technology [1, 2] where it spans out to $3013\mu m^2$ which is equivalent to 13 484 logic gates targeting for 125MHz operating clock frequency with 0.75V input voltage. Note that, the ECDSA sign/verify comprised of several other operations than scalar multiplications which will add a code footprint and latency overhead which are not accounted in the existing elliptic curve works. Also, none of the existing solutions are secure against quantum computers. To the best of our knowledge, this work presents a novel approach for post-quantum secure public-key signature for ultra-lightweight IoT

motes. We expect that the proposed solution would provide significant advancement for deploying IoT technology with long term security and so it provides longer life cycle for IoT edge devices for preserving end-to-end security.

Table 5: Cost and latency comparison with existing solutions

	ECC on 2^{131} [6]†‡	NIST p-256 SW [20]‡	NIST p-256 HW-SW [10]‡	**Our PQ-Secure XMSS**
Latency [clock cycles]	75 000	15 584 000	6 000 000	4 814 160
Resources	15K Gates	34KB code + HW Multiplier	11.7K gates	13.5K Gates + 5.22KB code

† Provides 65-bit classic security.
‡ Does not provide security against Quantum computers.

8 Conclusion

We present a novel approach for post-quantum-secure signature suitable for resource constrained IoT motes. We proposed a hardware-software co-design and implementation yielding small footprint (13.5k gates and 5.22kB object code) and low latency (4.8M clock cycles) for computing XMSS sign and verify operations. This public-key signature solution provides 128-bit security against quantum attacks. Secure boot/update and attestation for long-lived IoT would benefit from the proposed post-quantum-secure XMSS scheme. The lightweight digital signature approach presented in this paper serves as a foundation for future end-to-end IoT security.

References

1. et al., N.: A 14nm logic technology featuring 2nd-generation finfet, air-gapped interconnects, self-aligned double patterning and a 0.0588 μm^2 sram cell size. In: IEEE IEDM. pp. 3.7.1 – 3.7.3 (2014)
2. et al., N.: A 14nm logic technology featuring 2nd generation finfet, air-gapped interconnects, self-aligned double patterning and a 0.0588 μm^2 sram cell size. In: IEEE IEDM. pp. 3.7.1 – 3.7.3 (2014)
3. ARM: Amba specification (1999), https://www.arm.com/products/system-ip/amba-specifications
4. Aumasson, J.P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A lightweight hash. Journal of Cryptology **26**(2), 313–339 (Apr 2013). https://doi.org/10.1007/s00145-012-9125-6, https://doi.org/10.1007/s00145-012-9125-6
5. Barker, E.: Recommendation for key management part 1: General (2016), http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4

6. Batina, L., Guajardo, J., Kerins, T., Mentens, N., Tuyls, P., Verbauwhede, I.: An elliptic curve processor suitable for rfid-tags. IACR Cryptology ePrint Archive **2006**, 227 (2006), http://eprint.iacr.org/2006/227
7. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The keccak reference (2011), https://keccak.team/index.html
8. Bohr, M.: Silicon technology leadership for the mobility era. IDF (2012)
9. Bohr, M.: 14nm process technology: Opening new horizons. IDF (2014)
10. Bosmans, J., Roy, S.S., Järvinen, K., Verbauwhede, I.: A tiny coprocessor for elliptic curve cryptography over the 256-bit NIST prime field. In: 29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID 2016, Kolkata, India, January 4-8, 2016. pp. 523–528. IEEE Computer Society (2016). https://doi.org/10.1109/VLSID.2016.82, https://doi.org/10.1109/VLSID.2016.82
11. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS – a Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In: PQCrypto. pp. 117–129. Springer (2011)
12. Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. In: International Workshop on Post-Quantum Cryptography. pp. 63–78. Springer (2008)
13. CAST: SHA-3 secure hash function core (2019), http://www.cast-inc.com/ip-cores/encryption/sha-3/index.html
14. Cerrudo, C.: An emerging us (and world) threat: Cities wide open to cyber attacks. White Paper (2015), https://ioactive.com/pdfs/IOActive_HackingCitiesPaper_CesarCerrudo.pdf
15. Corporation, I.: Intel curie low-power compute module (2016)
16. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of ACM STOC 1996. pp. 212–219. ACM, New York, NY, USA (1996). https://doi.org/10.1145/237814.237866
17. Guo, X., Srivastav, M., Huang, S., Ganta, D., Henry, M.B., Nazhandali, L., Schaumont, P.: ASIC implementations of five SHA-3 finalists. In: 2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012. pp. 1006–1011 (2012). https://doi.org/10.1109/DATE.2012.6176643, https://doi.org/10.1109/DATE.2012.6176643
18. Hülsing, A.: W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes, pp. 173–188. Springer, Berlin, Heidelberg (2013)
19. Kim, M., Ryou, J., Jun, S.: Efficient hardware architecture of SHA-256 algorithm for trusted mobile computing. In: Yung, M., Liu, P., Lin, D. (eds.) Information Security and Cryptology, 4th International Conference, Inscrypt 2008, Beijing, China, December 14-17, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5487, pp. 240–252. Springer (2008). https://doi.org/10.1007/978-3-642-01440-6_19, https://doi.org/10.1007/978-3-642-01440-6_19
20. Mane, D.H., Schaumont, P.: Energy-architecture tuning for ecc-based RFID tags. In: Hutter, M., Schmidt, J. (eds.) Radio Frequency Identification - Security and Privacy Issues 9th International Workshop, RFIDsec 2013, Graz, Austria, July 9-11, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8262, pp. 147–160. Springer (2013). https://doi.org/10.1007/978-3-642-41332-2_10, https://doi.org/10.1007/978-3-642-41332-2_10
21. McGrew, D., Kampanakis, P., Fluhrer, S., Gazdag, S.L., Butin, D., Buchmann, J.: State Management for Hash-Based Signatures, pp. 244–260. Springer International Publishing (2016)

22. Merkle, R.C.: Secrecy, authentication and public key systems / A certified digital signature. Ph.D. thesis, Stanford (1979)
23. Microchip. https://www.microchip.com/, accessed: 2018-12-01
24. Miller, V.S.: Use of elliptic curves in cryptography. In: Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings. pp. 417–426 (1985). https://doi.org/10.1007/3-540-39799-X_31, https://doi.org/10.1007/3-540-39799-X_31
25. NSA/CSS: Commercial national security algorithm suite and quantum computer faq. In: MFQ U/OO/815099-15 (2016), https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm
26. Pessl, P., Hutter, M.: Pushing the limits of SHA-3 hardware implementations to fit on RFID. In: Bertoni, G., Coron, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8086, pp. 126–141. Springer (2013). https://doi.org/10.1007/978-3-642-40349-1_8, https://doi.org/10.1007/978-3-642-40349-1_8
27. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). https://doi.org/10.1145/359340.359342, http://doi.acm.org/10.1145/359340.359342
28. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997). https://doi.org/http://dx.doi.org/10.1137/S0097539795293172
29. Wenger, E.: Hardware architectures for msp430-based wireless sensor nodes performing elliptic curve cryptography. In: Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings. pp. 290–306 (2013). https://doi.org/10.1007/978-3-642-38980-1_18, https://doi.org/10.1007/978-3-642-38980-1_18
30. Zhijie Shi, Chujiao Ma, J.C., Wang, B.: Hardware implementation of hash functions. In: Introduction to Hardware Security and Trust. Springer Science & Business Media (2012)
31. Zimmer, V., Krau, M.: Establishing the root of trust. UEFI (2016), http://www.uefi.org