Post-Quantum Secure Boot

Vinay B. Y. Kumar*, Naina Gupta†, Anupam Chattopadhyay*, Michael Kasper†, Christoph Krauß‡, and Ruben Niederhagen‡

*Nanyang Technological University {vinay.kumar, anupam}@ntu.edu.sg

†Fraunhofer Singapore {firstname.lastname}@fraunhofer.sg

‡Fraunhofer SIT {firstname.lastname}@sit.fraunhofer.de

***Abstract*—A secure boot protocol is fundamental to ensuring the integrity of the trusted computing base of a secure system. The use of digital signature algorithms (DSAs) based on traditional asymmetric cryptography, particularly for secure boot, leaves such systems vulnerable to the threat of quantum computers. This paper presents the first post-quantum secure boot solution, implemented fully as hardware for reasons of security and performance. In particular, this work uses the eXtended Merkle Signature Scheme (XMSS), a hash-based scheme that has been specified as an IETF RFC. The solution has been integrated into a secure SoC platform around RISC-V cores and evaluated on an FPGA and is shown to be orders of magnitude faster compared to corresponding hardware/software implementations and to compare competitively with a fully hardware elliptic curve DSA based solution.**

***Index Terms*—secure SoC, secure boot, PQC, XMSS, RISC-V**

I. Introduction

While there are high hopes for a game changing impact of quantum computing on fields like weather forecast and drug design, the development of large and powerful quantum computers will have a devastating impact on today's cyber security [1]: cryptographic primitives that are currently in use, e.g., for authentication, key exchange, and digital signatures will no longer be secure. Research into schemes that promise to defend against this prospect has emerged into the field of post-quantum cryptography (PQC) and led to the initiation of a NIST process, in 2016, towards its standardization. Virtually all submissions in the NIST PQC standardization process require more resources in regards to computing power or key, message, or signature sizes compared to "classical" primitives currently in use. This poses a particular challenge for resource-restricted embedded devices used, e.g., for the Internet of things (IoT) and in automotive and industrial applications.

An important security property for such embedded devices is to ensure the integrity of platform hardware/software through a "secure boot" process (see Section II-B). Although secure boot approaches exist which make use of only symmetric cryptography (cf. the automotive SHE [2] module), the use of asymmetric cryptography for signature generation in a secure boot process has several advantages. First, there are less difficulties in managing the integrity of a public key than in managing the secrecy of a symmetric key. The concerns include attacks on both the embedded device as well as the supply chain. Second, an OEM could sign certificates for manufacturing partners to allow them to perform firmware provisioning for the system.

Using asymmetric cryptography in embedded systems, which are often used for a long period of time, makes the threat of quantum computers even more severe. For applications like secure boot, there are also operating and resource constraints that PQC implementations must meet. An automotive ECU connected to the CAN bus, for example, usually has to receive CAN messages within 30 to 100 milliseconds from the time it is powered on or risk missing critical messages [3].

In this paper, in moving away from quantum-unsafe signature schemes, we investigate at what cost and with what performance can a secure boot protocol be implemented in an embedded device when using PQC primitives, specifically the hash-based digital signature scheme XMSS, for checking the authenticity of the boot images. We chose XMSS for the integration with secure boot, because it is well researched and trusted and because it is specified in IETF RFC 8391 [4], which NIST intends to approve, making it one of few PQC schemes that is ready for deployment as of today.

Contributions: A complete post-quantum secure boot solution based on XMSS, implemented as a fully hardware solution for reasons of performance, and security against fault attacks. Integration and validation on FPGA with a RISC-V based SoC platform and compared with other hardware/software implementations of XMSS and also ECDSA.

II. Background

A. XMSS—a post-quantum digital signature scheme

The eXtended Merkle signature scheme (XMSS) belongs to the family of hash-based signature schemes. It uses hash chains and hash trees based on cryptographically secure hash functions. Fig. 1 gives an overview of the tree structure of the scheme. XMSS uses a hash-based one-time signature scheme—a variant of the Winternitz one-time signature scheme (WOTS$^+$ [4], or henceforth, WOTS)—as basic primitive and a Merkle tree on top of several (many) one-time signature schemes at the leafs of the Merkle tree. Consequently, XMSS is stateful: the owner of the private signing key must ensure that each one-time signature is used only once. Furthermore, the number of possible signatures per private/public key pair is well defined and restricted by the

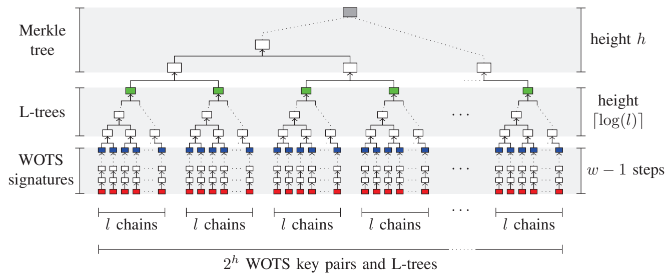

Fig. 1. Merkle tree structure and WOTS chains of XMSS (Figure from [5]).

Merkle tree height. WOTS works by splitting a message digest and a corresponding check sum into an array wd of l words of $\log w$ bits (where w is the "Winternitz parameter") and by computing l hash chains of $w-1$ steps for each of the words. The starting value $v_{i,0}$ (a red box in Figure 1) of each chain is part of the secret key, the last value $v_{i,w-1}$ (a blue box) is part of the public key, and the value v_d for a $\log w$-bit message digest word d (i.e., wd$[i]$) is part of the signature. The l public keys of the WOTS chains are combined into the WOTS public key (a green box) using an unbalanced binary hash tree called L-tree. On top of the WOTS signatures, XMSS uses a Merkle tree of height h for providing up to 2^h signatures, with the root node (the gray box) as the main public XMSS key. More detailed information can be found in [4] and [5]. The most expensive operation in XMSS is the key generation since all WOTS public keys and the entire Merkle tree need to be computed in order to obtain the root node of the Merkle tree as the XMSS public key. For signing too, in general, the entire Merkle tree needs to be recomputed, but the cost can be reduced by various caching techniques. Signature verification is the cheapest operation but still requires the computation of a few thousand hash function calls.

B. Secure boot

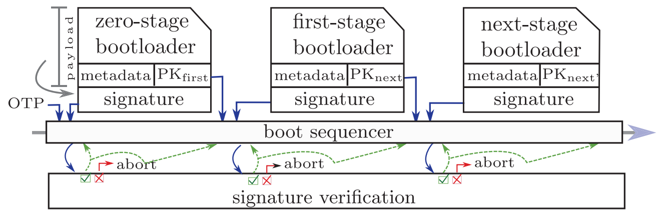

Fig. 2. Secure boot chain-of-trust.

The trusted computing base (TCB) of a secure system is defined to include its secure components (hardware and software), including a trusted execution environment (TEE), that together uphold a set of security properties of the system. In other words, components of TCB are unconditionally trusted. The RoT is the component of the TCB that is most trusted. Note that the RoT ultimately derives its trust from the device secret key(s) (derived, e.g., from an on-device PUF), together with a trusted party certifying them. Beginning at the RoT [the smaller the better], a secure boot protocol establishes a cryptographic chain-of-trust (as sketched in Fig. 2) as it verifies the integrity of the boot process right from power-on, from the zero-stage bootloader (ZSBL), through to enabling the TEE and passing control to an operating system kernel and finally to the user-space. The boot process is aborted if the verification of any stage fails. In one general scheme, the public key (PK) to verify the signature on the current stage comes from the payload of the previous stage (using a common PK being a particular case). For the ZSBL, the PK needs to be signed by a trusted party whose PK is stored in an on-chip one-time-programmable (OTP) memory (or an equivalent). Once the ZSBL is authenticated, the memory map is set up and the subsequent stage BLs can invoke the boot sequencer in the SVU to authenticate specified payloads. In each case, failure to verify would abort RISC-V cores (or an exception could be raised, to be handled in a low-level layer, e.g., OpenSBI).

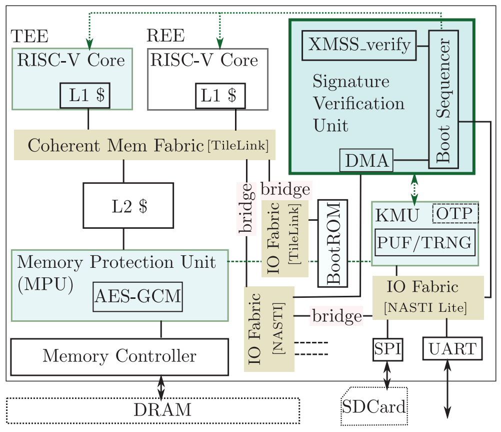

Fig. 3. The secure SoC platform based on RISC-V.

The normal boot sequence in the target platform: This work is implemented in the secure SoC platform outlined in Fig. 3, as the signature verification unit. Overview of the platform can be found in [6]. The first instruction a RISC-V core (say core_0) executes after power-on is from a boot ROM, located at the reset address (32′h0). This zero-stage bootloader (ZSBL) has instructions to jump to the first-stage BL located either in an on-chip BRAM or in external memory. The ZSBL also carries as a payload a device tree structure (here, encoded as a string) that describes the peripheral devices, the memory map, and the cores and their capabilities. The first-stage BL (FSBL) is a small program with limited capabilities and is stored in an on-chip BRAM. It is able to locate a specific file (`boot.bin`) in the first [`vfat` formatted] partition on the SD card—this file contains the second and third stage bootloaders, the Berkeley BL and the linux kernel (`vmlinux`) respectively. The control is transfered to BBL which then transfers control to the linux kernel while it continues executing at the hypervisor level.

A note on implementation security of secure boot: A common and very practical attack on secure boot, often overlooked, are fault based bypass attacks. While the details are omitted in the interest of space, countermeasures involve techniques using redundancy in space, and time (with randomization)—effectively making the attacks hard to target.

We also note that a fully-hardware solution with such measures is more secure than a SW or a HW-SW solution.

C. *Related work*

Sanctum [7] presents a comprehensive secure boot solution together with attestation, implemented as software in a RISC-V platform. This has also been used later in Keystone [8]. The boot ROM in these solutions is defined as the root-of-trust and hence a part of TCB. The work in [9] is also a RISC-V based system and it presents a fully-HW solution for secure boot and uses an elliptic curve based signature scheme.

Two hardware-software implementations of XMSS have been reported: [5], [10]. The co-design in [5] is implemented in a RISC-V system context while [10] implements a variant of XMSS (using Keccak-400 in place of SHA256 in the interest of lower resource usage) in the context of an FPGA–micro-controller HW-SW system

III. Design and Implementation

A. *XMSS signature verification*

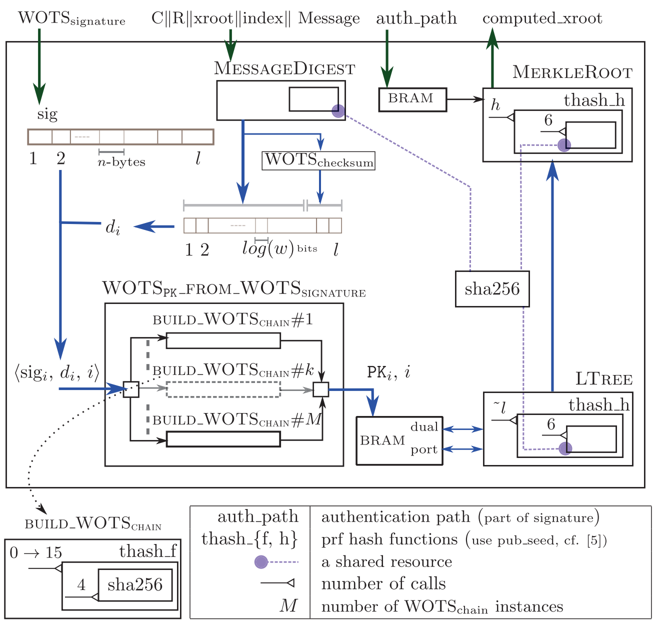

Fig. 4. XMSS verification hardware design outline

While XMSS has been implemented as a parametric design, let us consider the following set of parameters as reference for convenience of discussion: $n = 32$, $w = 16$, $len_1 = 64$, $len_2 = 3$, $l = len_1 + len_2 = 67$. These correspond to the parameter ID `XMSS-SHA2_10_256` in [4]. Notation: x.NB represents `ValueOf`(x) as an array of length `N` bytes and ‖ represents concatenation. The design has been implemented in a combination of Verilog and Bluespec [11]. XMSS verification involves a sequence of four steps. Fig. 4 shows the sketch of the implementation. For secure boot, the inputs and outputs of this component are managed by the Signature Verification Unit (SVU) (Fig. 3) over AXI (DMA) and AXI Lite interfaces (NASTI ('Not A STandard Interface') in Fig. 3 is ≈AXI).

The information available to **step one** is the signed message `sm` and the public key `PK` corresponding to the signature. The `PK` is ⟨ merkle_root.32B ‖ pub_seed.32B ⟩, and `sm` has the form ⟨ sig_leaf_index.4B ‖ R.32B ‖ sig.(67×32)B ‖ auth_path.10B ‖ message ⟩. A sha256 digest is computed on ⟨ $\texttt{C}_1$.32B ‖ R ‖ merkle_root ‖ index.32B ‖ message ⟩. The 32B hash digest together with a computed WOTS checksum is split into $l = 64 + 3$ base-16 4-bit words; let the label `wd` refer to this array. The **step two** involves using the given WOTS signature to compute the WOTS public key. This is the WOTS$_{\texttt{PK}}$_from_WOTS$_{\text{signature}}$ module in Fig. 4. In Fig. 1, the bottom red squares correspond to the WOTS secret keys and the blue squares at the top, reached by computing the 'WOTS hash chain' of length $w - 1$, correspond to the WOTS public keys. A WOTS signature is an array of l n-byte components where component i corresponds to the key i chained $\texttt{wd}[i]$ times. Recovering component i of the public key therefore involves continuing the chain $w - 1 - \texttt{wd}[i]$ more times starting from the signature[i]. The **step three** is computing the L-Tree root: the l WOTS public key components just computed are treated as leaf nodes of an unbalanced binary tree and are pair-wise hashed towards a single root hash using a keyed hash function `thash_h`. The **final step** is computing the Merkle root node. This is computed using the L-Tree root node together with the specified nodes in the authentication path (auth_path.10B in `sm`).

An obvious target for optimization is to compute the independent WOTS chains in parallel. In Fig. 4, M instances of WOTS$_{chain}$ modules are instantiated which compete to 'complete' the l chains. The L-Tree and Merkle root computation stages are less suitable for parallelization, but they share the same hash kernel instance. The randomized hash functions `thash_h` and `thash_f`, used as-is from [5], do take advantage of the fact that hashes are computed on blocks of either 768 or 1024 bit data, reducing the effective number of calls from 6 and 9 to 4 and 6 calls in `thash_f` and `thash_h` respectively. If the key and signature generation components of XMSS were also to be implemented, the multiple WOTS$_{chain}$ instances could be re-used. Some optimizations paths are available for signing and key generation (as noted in [5]).

IV. Evaluation

This evaluation corresponds to the XMSS parameter set with the parameter ID `XMSS-SHA2_10_256` (as described in Section III-A). The following numbers have been measured on a Xilinx KC705 board with an `xc7k325tffg900-2` FPGA. Table I reports resource usage and computation time for different degree of parallelism (in terms of instances of WOTS$_{\text{chains}}$). We verified that the secure boot solution performs correctly on the FPGA for all the cases. In Table I—a '*' is marked to note that the corresponding computation times shown are data dependent and not absolute; and the cycle count with † is only the verification time (excluding the DMA transfer cycles as that depends on the payload). To evaluate the costs and benefits of the PQC signature scheme, we compare our XMSS implementation with HW and SW implementations of the

TABLE I
COMPONENTS OF XMSS SECURE BOOT AND RESOURCE UTILIZATION AND COMPUTATION TIMES

Module	LUTs	FFs	Computation Time (clock cycles)
sha256	1402	1546	41
MESSAGEDIGEST	1667	2077	-
BUILD_WOTS$_{\text{CHAIN}}$ (1 instance)	2780	3567	(depends on $\mathtt{wd}[i]$)
LTREE (without sha256)	1103	2485	26665
MERKLEROOT (without sha256)	592	1290	4753
Parallel WOTS chain computations:			
WOTS$_{\text{PK}}$_FROM_WOTS$_{\text{SIG}}$ (1 chain)	3301	5367	129851*
WOTS$_{\text{PK}}$_FROM_WOTS$_{\text{SIG}}$ (2 chains)	6134	9394	71523*
WOTS$_{\text{PK}}$_FROM_WOTS$_{\text{SIG}}$ (4 chains)	12408	17621	42807*
WOTS$_{\text{PK}}$_FROM_WOTS$_{\text{SIG}}$ (8 chains)	24159	33637	20500*
Complete SVU with parallel WOTS chain computations:			
SVU+XMSS_VERIFY (1 WOTS$_{\text{chains}}$)	12162	16348	161280†
SVU+XMSS_VERIFY (2 WOTS$_{\text{chains}}$)	14992	20373	102952†
SVU+XMSS_VERIFY (4 WOTS$_{\text{chains}}$)	21476	28434	74229†
SVU+XMSS_VERIFY (8 WOTS$_{\text{chains}}$)	35637	45604	51725†

elliptic curve digital signature algorithm (ECDSA) in [9] (as also used in the context of secure boot). We also give a brief comparison with two other recent XMSS implementations. While our design meets the timing constraint for $200\,\mathrm{MHz}$, for convenience, when not specified, we assume a design clock of $100\,\mathrm{MHz}$ when reporting in terms of wall-clock times.

a) ECDSA: The ECDSA implementation in [9] uses a $233\,\mathrm{bit}$ elliptic curve (sec233r1). The hardware implementation for verification consumes about $30,000$ slice LUTs with an execution time of about 5400 clock cycles, which is about $0.05\,\mathrm{ms}$ at $100\,\mathrm{MHz}$. Table I shows that the XMSS implementation with eight chain instances matches this in terms of area. While this case is about $10\times$ slower with $0.5\,\mathrm{ms}$ in terms of execution speed, the performance is acceptable in practice for most secure boot scenarios. The XMSS implementations computing four, two, and one chains in parallel consume $\{1.4, 2.0, \text{and } 2.5\}\times$ lower area, respectively, but are $\{13, 19, \text{and } 30\}\times$ slower. Even the serial case (with one chain), which is $30\times$ slower, is still acceptable for most embedded systems (translating to 1.5 ms in wall-clock time).

b) XMSS: The hardware-software implementation for XMSS verification on the Murax RISC-V SoC in [5] takes $174\,\mathrm{ms}$ (at $100\,\mathrm{MHz}$, with the hardware components consuming 6530 Altera Cyclone V ALMs). Another recent work [10] also implements XMSS as a HW-SW co-design with the WOTS^{+} module in HW (with a scope a little larger than the BUILD_WOTS$_{\text{CHAIN}}$ module in this work). This WOTS^{+} module was reported to consume about 3000 combinational logic cells and about 2300 registers on a Stratix IV FPGA. XMSS verify operation (for the same parameters used for this discussion) is reported to take about 4.8×10^{6} cycles or $48\,\mathrm{ms}$.

c) Compared to qTesla and NCRYSTALS-Dilithium: While this work used XMSS for secure boot, two of NIST PQC round-2 candidates for signature schemes—qTesla and NCRYSTALS-Dilithium—could have been evaluated as well. The following is a preliminary comparison of XMSS (Tab. I) with high-level synthesis (HLS) based hardware designs [12] of those two candidates (across multiple HLS optimizations and security-level parameters). XMSS verification appears to require significantly lower area, however, a definitive/fair comparison can only be made with similar custom design efforts for these candidates.

TABLE II
COMPARING HARDWARE RESOURCE REQUIREMENTS OF SIGNATURE VERIFICATION STEP FOR TWO NIST ROUND-2 CANDIDATES AND XMSS

Module	LUTs	FFs	Time (clock cycles)
qTesla [12]	86-145 K	17-37 K	0.65×10^{6}
NCRYSTALS-Dilithium [12]	65-123 K	15-44 K	$1.1\text{-}3.8\times10^{6}$
XMSS (this work)	12-35 K	16-45 K	$0.5\text{-}1.6\times10^{6}$

V. CONCLUSION

The higher cost of PQC schemes compared to "classical" schemes is a burden for embedded systems with operating constraints and restricted computing, memory, storage, and networking resources. For the particular use case of secure boot, we show through this work that the hash-based signature scheme XMSS, implemented as hardware, performs well on an embedded SoC device and meets strict latency requirements for the application at a low overhead in area but under longer verification times compared to an ECDSA solution. While this work chooses XMSS, other PQC schemes should be evaluated for this application as well.

ACKNOWLEDGMENT

The authors acknowledge the support from the Singapore National Research Foundation (SOCure grant NRF2018NCR-NCR002-0001 – www.green-ic.org/socure).

REFERENCES

[1] 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, Oct. 1997.
[2] H. S. (HIS), "She secure hardware extension version 1.1," 2009.
[3] K. Matheus and T. Königseder, *Automotive Ethernet*, 1st ed. New York, NY, USA: Cambridge University Press, 2015.
[4] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme," RFC 8391, May 2018.
[5] W. Wang, B. Jungk, J. Wlde, S. Deng, N. Gupta, J. Szefer, and R. Niederhagen, "XMSS and embedded systems - XMSS hardware accelerators for RISC-V," Cryptology ePrint Archive, Report 2018/1225.
[6] V. B. Y. Kumar, A. Chattopadhyay, J. Haj-Yahya, and A. Mendelson, "Itus: A secure risc-v system-on-chip," in *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, Sep. 2019.
[7] I. Lebedev, K. Hogan, and S. Devadas, "Invited paper: Secure boot and remote attestation in the Sanctum processor," *Proceedings - IEEE Computer Security Foundations Symposium*, vol. July, pp. 46–60, 2018.
[8] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanovic, "Keystone: An open framework for architecting tees," 2019.
[9] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay, "Lightweight secure-boot architecture for RISC-V system-on-chip," in *20th International Symposium on Quality Electronic Design (ISQED)*, March 2019, pp. 216–223.
[10] S. Ghosh, R. Misoczki, and M. R. Sastry, "Lightweight post-quantum-secure digital signature approach for IoT motes," Cryptology ePrint Archive, Report 2019/122, 2019, https://eprint.iacr.org/2019/122.
[11] Bluespec Inc., Bluespec Inc. http://www.bluespec.com.
[12] D. Soni, K. Basu, M. Nabeel, and R. Karri, "A hardware evaluation study of nist post-quantum cryptographic signature schemes," Second PQC Standardization Conference, Aug 2019.