Post-Quantum Hash-Based Signatures for Secure Boot

Panos Kampanakis, Peter Panburana, Michael Curcio, Chirag Shroff
Security and Trust Organization
Cisco Systems, USA
{pkampana, pepanbur, micurcio, cshroff}@cisco.com

Updates: Initially uploaded to Cryptology ePrint Archive on Dec 18, 2020 with minor changes in Section III-B from the originally submitted SVCC 2020 camera-ready manuscript. Re-uploaded on Dec 21, 2020 with minor updates in Section III-B for accuracy.

***Abstract*—The potential development of large-scale quantum computers is raising concerns among IT and security research professionals due to their ability to solve (elliptic curve) discrete logarithm and integer factorization problems in polynomial time. All currently used, public-key cryptography algorithms would be deemed insecure in a post-quantum setting. In response, the United States National Institute of Standards and Technology has initiated a process to standardize quantum-resistant cryptographic algorithms, focusing primarily on their security guarantees. Additionally, the Internet Engineering Task Force has published two quantum-secure signature schemes and has been looking into adding quantum-resistant algorithms in protocols. In this work, we investigate two post-quantum, hash-based signature schemes published by the Internet Engineering Task Force and submitted to the National Institute of Standards and Technology for use in secure boot. We evaluate various parameter sets for the use-cases in question and we prove that post-quantum signatures would not have material impact on image signing. We also study the hierarchical design of these signatures in different scenarios of hardware secure boot.**

***Index Terms*—HBS signatures, PQ image signing, PQ root of trust, post-quantum secure boot**

I. Introduction

Digital communications have completely penetrated everyday life as enablers of numerous critical services including telemedicine, online banking, massive e-commerce, machine-to-machine automation, mobile and cloud computing. As part of guaranteeing that the software on digital devices is genuine, vendors have implemented several security features that validate the authenticity of software. When starting the booting process, a device's firmware is initially booted from a tamper-resistant ROM or flash memory. Then the boot process is passed onto a bootloader that is responsible for further loading the operating system (OS). Software verification takes place at every step of the process. Before being loaded, a bootloader signature is verified by boot 0 code, namely Unified Extensible Firmware Interface (UEFI) on x86 based systems. The OS signature is verified by the bootloader before loading the OS. This verification process is often referred to as secure boot and ensures that there is a chain of trust that is passed from the very first step until the operating system comes live [1]. In virtual environments, vendors often follow a similar paradigm.

The signatures that are validated in each step of the secure boot process are usually classical RSA signatures. While the security of these signatures cannot effectively be challenged by conventional computer systems, this would not be the case in a post-quantum (PQ) world where a large scale quantum computer is a reality [2]. Shor's quantum algorithm [3], [4], assuming a practical quantum computer (QC) becomes available, would solve elliptic curve discrete logarithm (ECDL) and integer factorization (IF) problems in polynomial time rendering the algorithms insecure. In this scenario a QC-equipped attacker would be able to sign any specially crafted malicious software in the secure trust chain and boot non-genuine, malicious software.

The cryptographic community has been researching quantum-secure public key algorithms for some time in order to address the QC threat, and the US National Institute of Standards and Technology (NIST) has started a public project to standardize quantum-resistant public key encapsulation and digital signature algorithms. At the time of this writing, NIST's evaluation process has moved to Round 3 with 15 PQ algorithms remaining. Similarly, the European Telecommunications Standards Institute (ETSI) has formed a Quantum-Safe Working Group [5] that aims to make assessments and recommendations on the various proposals from industry and academia regarding real-world deployments of quantum-safe cryptography. In addition, the Internet Engineering Task Force (IETF) has seen multiple proposals that attempt to introduce PQ algorithms in protocols like TLS and IKE [6]–[10]. The integration of PQ algorithms in today's technologies presents challenges that pertain to (a) bigger keys, ciphertexts and signatures, (b) slower performance, (c) backwards compatibility, and (d) lack of hardware acceleration. The IETF has also published two PQ signature algorithms in Informational RFCs [11], [12].

Our focus for this work is on post-quantum secure boot signatures. We chose to evaluate two well-established hash-based signature (HBS) schemes, namely LMS [11] and

SPHINCS$^+$ [13], which are based on mature, quantum-secure primitives. We compare their verification time against classical RSA signatures used today. Verification happens every time booting takes place, thus its performance is important. We also study the signing time; even though signing takes place once per image, we still need to maintain relatively fast signing for high-rate signers. We also investigate the signatures and public key sizes which would affect the verification process, especially for resource-constrained verifiers. We finally study the code footprint and stack used at the verifier in order to estimate the impact on resource-limited verifiers.

The **key contributions of our work** are summarized as follows:

(i) We formalize and propose post-quantum HBS signature hierarchies for secure boot software signing.

(ii) We establish parameter sets suitable for different software signing use-cases for two PQ security levels.

(iii) We analyze the impact of two well-studied HBS algorithms (one stateful, one stateless) when they are used for image signing in hardware secure boot and FPGAs, and compare them to classical RSA.

(iv) We show that trusted post-quantum signatures are possible with immaterial impact on the verifier, and an acceptable impact on the signer.

The rest of the paper is organized as follows: Section II summarizes related work. Section III lays out the signature schemes, the proposed hierarchy and parameter sets for our investigation and Section IV presents our experimental results and analysis in various platforms. Section V concludes the paper.

II. Related Work

There has been a large body of research on PQ cryptography [14]–[17]. Recently, more works are exploring NIST's PQ candidate schemes focusing mainly on their security and computational performance.

More closely related to this work which focuses on PQ signatures for secure boot, is the potential for using quantum-secure, HBS schemes on constrained processors that was first demonstrated in [18]. What is more, a variant of a stateful HBS scheme, called XMSS [19], was implemented on a 16-bit smart-card in [20] which showed the practicality of stateful HBS in constrained devices. [21] also demonstrated an efficient implementation of XMSS in constrained IoT motes. [22] integrated XMSS in a SoC platform around RISC-V cores and evaluated on an FPGA to show that it is very efficient. XMSS shares similarities with LMS investigated in this work, but with worse performance and a tighter security proof [23]. What's more, [24] investigated the practicality of stateful HBS (LDWM, a predecessor of LMS studied here) in TPMs without studying real time, performance or memory footprint or stateless HBS schemes and HBS parameters specific to the secure boot use-case.

Boneh and Gueron proposed stateful HBS signatures using various one-way functions for signing Intel SGX enclaves in [25]. They concluded that HBS verification can be faster than RSA3072 and QVRSA. Additionally, Hülsing et al. implemented stateless HBS SPHINCS signatures in an ARM Cortex M3 with only 16KB of memory [26]. They also compared SPHINCS with stateful XMSS in an ARM Cortex M3 and showed that stateless HBS verification is acceptable but its signing is 30 times slower. The authors in [27], [28] conducted a comparison of NIST Round 2 candidate algorithm hardware implementations with a focus on algorithm operations and impact on hardware design. In [29], Kannwischer et al. benchmarked Round 2 algorithms on ARM Cortex-M4 and evaluated the more suitable ones for embedded devices. [30] studied the energy consumption of the NIST algorithm candidates and identified the most expensive ones in terms of energy. The SPHINCS$^+$ variants were found to be one of the most energy-intensive ones compared to their competitors at the same security level.

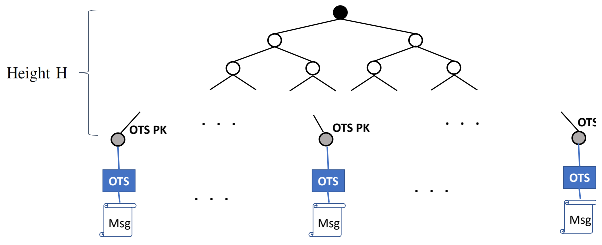
III. HBS for Secure Boot

A. Intro to HBS

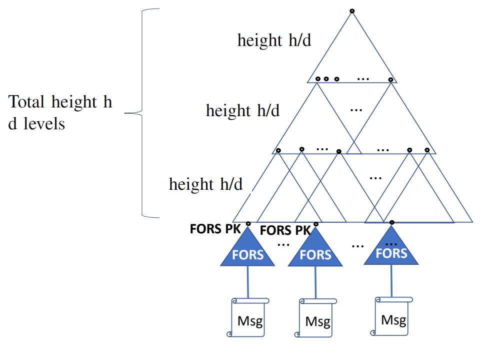
The HBS family of PQ signature algorithms is considered mature, well-understood, and significantly reliable. The first scheme in the family, the Merkle signature scheme (MSS), was presented in the late 1970s [31]. HBS signatures rely on Merkle trees and few or one-time-signature (FTS/OTS) used with secure cryptographic hash functions.

HBS schemes generate keypairs for the FTS/OTS. The FTS/OTS signs a message and the corresponding public key is a leaf in the Merkle tree, whose root is the public key that is expected to be pre-trusted by the verifier. An HBS signature consists of the OTS/FTS signature and the path to the root of the tree. The signature is verified using the public key inside it, and by using the authentication path to construct the root of the hash tree. Currently, the most mature members of the HBS family are the stateful LMS [11] and XMSS [12], and stateless SPHINCS$^+$, one of the NIST signature candidates [13]. Fig. 1a shows an LMS tree. The message is signed by an OTS (LMS-OTS or WOTS+) and the OTS public key forms a binary tree leaf that is aggregated all the way to the Merkle tree root. The parameters of the tree include the hash function, the tree height, and the Winternitz parameter of the OTS. Multi-level tree variants are also available and consist of smaller subtrees that form a big HBS tree. Readers should note that stateless XMSS [19], which was also published by IETF [12], shares similarities with LMS, but with worse performance and a tighter security proof [23].

Fig. 1b shows a SPHINCS$^+$ tree which consists of an FTS, namely a FORS tree, that signs a message, and a multi-level Merkle tree. The FORS tree root is signed by an OTS (i.e., WOTS+). The corresponding WOTS+ public key forms the Merkle tree leaf that is aggregated all the way to the bottom subtree root. That root is signed by WOTS+ and the WOTS+ public key is aggregated to the root of the subtree above. Subtree roots are signed by subtrees above until we reach the top subtree. Going forward we focus only on LMS and SPHINCS$^+$.



(a) LMS tree



(b) SPHINCS$^+$ tree

Fig. 1: HBS trees

Excluding their architectural differences, the most significant difference between a stateful and stateless HBS is state. Stateful schemes (i.e., LMS, XMSS) include a four-byte index value in their signature which represents the state. The state is used when signing a message and should never be reused as that could allow for forgeries. The state management requirement is considered an important disadvantage which has been often brought up in IETF and NIST fora [32], [33]. Stateless HBS (i.e., SPHINCS$^+$), on the other hand, have no state requirement. Hence, messages are signed by an HBS hypertree without having to keep state with every signature. While stateless SPHINCS$^+$ eliminates the need for proper state management, it also leads to a significant increase in signature size and slower performance because of the FORS structure. Section IV shows the performance comparison between LMS and SPHINCS$^+$.

B. Hierarchy

An HBS hierarchy would be needed to provide signatures for multiple platforms or chips and firmware / software versions, and comply with existing secure boot standards. Typically, these hierarchies would include multiple tree levels. Each multi-level tree would serve as a different UEFI signing structure, namely the PK, Firmware Update Key or KEK. The root of each tree would become the key included in the UEFI db/dbx databases pre-trusted by the verifier. Each tree would be responsible for signing firmware, firmware packages, PK updates, or software images. Fig. 2 shows such a hierarchy with a three-level tree architecture. The hypertree structurally consists of smaller trees in order to limit the key generation and signing times.

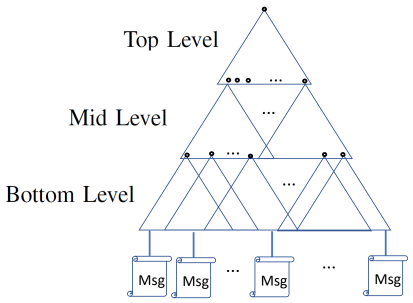


Fig. 2: HBS hierarchy

In such architectures, the height of the trees should be chosen so they can provide enough signatures for the use-case. A typical top tree could sign 2^{15} bottom trees. A bottom tree that can sign one million images (2^{20}) would probably suffice for most use-cases. Most vendors' portfolio would never exceed 2^{35} total images. As an example, in 2020, major IT vendors with thousands of products in their portfolio were signing less than 250 million ($\sim 2^{28}$) images annually which would lead to $\sim 2^{33}$ signatures over a 30 year signing root's lifetime.

Key Revocation is an important concept for HBS multi-level tree hierarchies. Revoking a tree root, a signature or any of the lower level trees is essential to providing a way to eliminate trust in roots that have been compromised, broken or replaced. In secure boot, the trusted certificate/key and revocation list exist in the UEFI db and dbx respectively [34]. Often the UEFI db/dbx is stored in a Trust Anchor module. Updating these lists takes place with messages authenticated by a KEK tree. The KEK HBS tree would be used to sign UEFI db/dbx updates or sign revocation images.

C. Parameters

In the process of evaluating HBS for secure boot and image signing, we had to come up with parameters that would satisfy most use-cases. We experimented with many parameters in order to find the most suitable ones. LMS offers many options up to 2^{25} signed messages which would suffice for the multi-level architecture described in Section III-B. The PQ security level of all LMS parameters is 128 bits. Note that all LMS/HSS parameters could be adjusted to provide 96-bit PQ security by truncating the SHA256 outputs to 192 bits [35]. On the other hand, SPHINCS$^+$, as proposed to NIST, offers three PQ security levels of 64, 96 and 128 bits. For our evaluation we chose to be conservative and use the latter two. The SPHINCS$^+$ variants submitted to NIST can sign up to 2^{64} messages, as required by NIST. We generated new parameters for our maximum signature count which does not

Parameter name	HBS Scheme	Parameters	PQ Security Level
LMS256H5W4	LMS	LMS_SHA256_M32_H5 with LMOTS_SHA256_N32_W4	128-bits
LMS256H5W8	LMS	LMS_SHA256_M32_H5 with LMOTS_SHA256_N32_W8	128-bits
LMS256H10W4	LMS	LMS_SHA256_M32_H10 with LMOTS_SHA256_N32_W4	128-bits
LMS256H10W8	LMS	LMS_SHA256_M32_H10 with LMOTS_SHA256_N32_W8	128-bits
LMS256H15W4	LMS	LMS_SHA256_M32_H15 with LMOTS_SHA256_N32_W4	128-bits
LMS256H15W8	LMS	LMS_SHA256_M32_H15 with LMOTS_SHA256_N32_W8	128-bits
LMS256H20W4	LMS	LMS_SHA256_M32_H20 with LMOTS_SHA256_N32_W4	128-bits
LMS256H20W8	LMS	LMS_SHA256_M32_H20 with LMOTS_SHA256_N32_W8	128-bits
HSS256L2W4	HSS	$L = 2$ with top LMS256H20W4 and bottom LMS256H15W4	128-bits
HSS256L2W8	HSS	$L = 2$ with top LMS256H20W8 and bottom LMS256H15W8	128-bits
HSS256L3oW4	HSS	$L = 3$ with LMS256H15W4, LMS256H10W4 and bottom LMS256H10W4	128-bits
HSS256L3oW8	HSS	$L = 3$ with LMS256H15W8, LMS256H10W8 and bottom LMS256H10W8	128-bits
SPX192H15w16	SPHINCS$^+$	$n = 24$, $H = 15$, $d = 3$, $k = 18$, $a = 13$ $w = 16$	96-bits
SPX192H15w256	SPHINCS$^+$	$n = 24$, $H = 15$, $d = 3$, $k = 18$, $a = 13$ $w = 256$	96-bits
SPX192H20w16	SPHINCS$^+$	$n = 24$, $H = 20$, $d = 4$, $k = 18$, $a = 13$ $w = 16$	96-bits
SPX192H20w256	SPHINCS$^+$	$n = 24$, $H = 20$, $d = 4$, $k = 18$, $a = 13$ $w = 256$	96-bits
SPX192H35w16	SPHINCS$^+$	$n = 24$, $H = 35$, $d = 5$, $k = 20$, $a = 12$ $w = 16$	96-bits
SPX192H35w256	SPHINCS$^+$	$n = 24$, $H = 35$, $d = 5$, $k = 20$, $a = 12$ $w = 256$	96-bits
SPX256H15w16	SPHINCS$^+$	$n = 32$, $H = 15$, $d = 3$, $k = 19$, $a = 16$ $w = 16$	128-bits
SPX256H15w256	SPHINCS$^+$	$n = 32$, $H = 15$, $d = 3$, $k = 19$, $a = 16$ $w = 256$	128-bits
SPX256H20w16	SPHINCS$^+$	$n = 32$, $H = 20$, $d = 4$, $k = 19$, $a = 16$ $w = 16$	128-bits
SPX256H20w256	SPHINCS$^+$	$n = 32$, $H = 20$, $d = 4$, $k = 19$, $a = 16$ $w = 256$	128-bits
SPX256H35w16	SPHINCS$^+$	$n = 32$, $H = 35$, $d = 5$, $k = 21$, $a = 15$ $w = 16$	128-bits
SPX256H35w256	SPHINCS$^+$	$n = 32$, $H = 35$, $d = 5$, $k = 21$, $a = 15$ $w = 256$	128-bits

TABLE I: HBS Parameters

exceed a total of 34 trillion (2^{35}) signatures. In preliminary investigations, we found that the equivalent NIST-submitted SPHINCS$^+$ parameter sets (2^{64} signatures) would perform much slower verification which would lead to significant delays in the booting process. We also chose to only evaluate the 'simple' variants of SPHINCS$^+$ because of their superior performance. What's more, we only used SHA256 SPHINCS$^+$ variants because of its superiority in SPHINCS$^+$ benchmarks and its prevalence in hardware implementations.

Table I summarizes all the LMS and SPHINCS$^+$ parameter sets we used in our evaluation. LMS256H5W4, LMS256H5W8, LMS256H10W4 and LMS256H10W8 are not included as single tree LMS variants in our analysis. We just list them because they are used in multi-level HSS variants. Additionally, LMS256H20W4 and LMS256H20W8 could be tested as an HSS with two levels of height 10, which would speed up key generation and key loading, at the expense of increased signature sizes. Key generation and loading can be considered offline operations for the secure boot use-cases and thus we chose to use height 20 LMS trees. HSS256L3oW4 and HSS256L3oW8 offer a balance between key generation and key load time.

IV. Experiments

A. *RSA vs LMS vs SPHINCS$^+$*

In order to compare our options. we initially experimented and measured the performance of RSA and all the LMS and SPHINCS$^+$ parameters of interest. We ran all these tests in a Google Cloud instance with an Intel Xeon CPU 2.20GHz with 2 cores and 7.68GB RAM. To compile our code we used gcc version 7.4.0. The tests were run 1000 times for each parameter set.

To measure RSA performance, we used OpenSSL 1.1.1c with OPENSSL_BN_ASM_MONT enabled. Our LMS testing code was based on [36], dynamically linked to OpenSSL's SHA256 implementation and properly instrumented with various performance optimizations and memory speed trade-offs. Our SPHINCS$^+$ testing code was based on [37] which is a fork of the original code [38] from the SPHINCS$^+$ NIST submission. The SPHINCS$^+$ verification was dynamically linked to the OpenSSL 1.1.1c library for its SHA256 implementation which proved to provide the best verification performance. For SPHINCS$^+$ signing, we found that the AVX2 optimized code in [38] provided the best results possible and thus we didn't link SPHINCS$^+$ signing to OpenSSL. Multi-threading was disabled for both implementations. We also measured the memory footprint (code and stack) of LMS/HSS and SPHINCS$^+$ verification in order to assess their practicality in constrained verifier chips. Both verifiers used OpenSSL's SHA256 implementation which was not counted against the reported code size. As in [29], we measured the stack usage by writing a random canary to a big chunk of the available stack space, then running the verification, and finally checking which parts of the memory have been overwritten.

Table II shows the results from our testing. We can see that all the private and public keys are of negligible size. It is also clear that the verifier code and stack size of a few KB would not practically impact most secure boot verifiers. The heap

Parameter	Keys (B)		Verifier (KB)		Keygen	Sign (Mcycles)		Verify (Mcycles)	
	Priv	Pub	Code	Stack	(s)	Mean	Stdv	Mean	Stdv
LMS256H15W4	48	60	2.57	1.81	2.519	1.145	0.051	0.370	0.033
LMS256H15W8	48	60	2.15	1.81	13.720	6.237	0.302	2.855	0.290
LMS256H20W4	48	60	2.57	1.81	3.222	1.465	0.037	0.373	0.026
LMS256H20W8	48	60	2.15	1.81	19.373	8.807	0.555	2.857	0.274
HSS256L2W4	48	60	3.15	1.81	350.2	3.945	0.041	0.716	0.026
HSS256L2W8	48	60	2.51	1.81	2712	19.351	0.288	5.771	0.271
HSS256L3oW4	48	60	3.15	1.81	10.86	3.771	0.024	1.050	0.022
HSS256L3oW8	48	60	2.51	1.81	84.5	13.084	0.261	8.187	0.254
SPX192H15w16	96	48	4.46	4.98	0.003	176.049	1.170	1.022	0.020
SPX192H15w256	96	48	4.41	2.87	0.026	332.180	1.860	7.045	0.094
SPX192H20w16	96	48	4.46	4.39	0.003	183.444	1.154	1.382	0.024
SPX192H20w256	96	48	4.41	3.24	0.026	391.554	2.219	10.193	0.189
SPX192H35w16	96	48	4.50	4.97	0.012	212.532	1.374	1.591	0.027
SPX192H35w256	96	48	4.43	2.87	0.103	1,218.257	4.139	11.711	0.151
SPX256H15w16	128	64	4.54	6.14	0.004	1,297.180	5.589	1.385	0.024
SPX256H15w256	128	64	4.47	4.68	0.032	1,491.183	5.405	9.219	0.163
SPX256H20w16	128	64	4.54	6.70	0.004	1,310.393	5.480	1.768	0.046
SPX256H20w256	128	64	4.47	4.25	0.032	1,566.816	4.856	11.599	0.139
SPX256H35w16	128	64	4.53	6.58	0.016	814.356	3.697	2.080	0.031
SPX256H35w256	128	64	4.46	4.52	0.128	2,057.650	6.223	15.341	0.214

TABLE II: HBS Key, Signature Sizes, Memory footprint and Performance

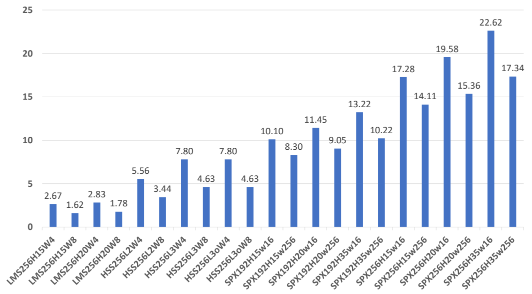


Fig. 3: HBS Signature Size (KB)

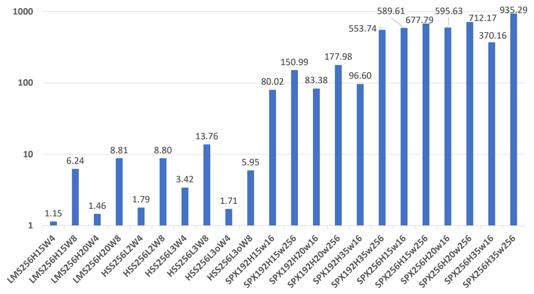


Fig. 4: HBS Signing Time (log-scale in ms)

used by the verifier was always zero. What's more, LMS/HSS signing was less costly in CPU cycles than SPHINCS$^+$. SPHINCS$^+$ verification was worse than LMS/HSS but not as significantly as signing. The standard deviation for both signing and verification CPU cycles was insignificant. Our results are in agreement with the ARM Cortex M3 results in [26]. In terms of key generation, LMS/HSS took much longer mainly because LMS generates trees at every level which is counted against key generation; in SPHINCS$^+$ only the top tree is counted as part of the key generation and all the other tree generations are part of the signature operation. Thus, LMS is optimized by taking longer to generate keys but performs signing faster by using auxiliary data and pre-loaded private keys, whereas SPHINCS$^+$ includes part of this work in the signature generation itself.

Fig. 3 demonstrates the signature sizes of all parameter sets. As expected, parameter sets with $W = 8$ and $w = 256$ have smaller signatures. We also see that LMS/HSS offers signatures that stay below 8KB. Note that all LMS/HSS parameters could be adjusted to provide 96-bit PQ security by truncating the SHA256 outputs to 192 bits [35] which would shrink all reported LMS/HSS signatures by 25%. SPHINCS$^+$ signatures exceed 10KB and grow to almost 23KB for 128-bit PQ security level parameters with $w = 16$. 23KB signatures are small enough for most secure boot use-cases. In rare scenarios where small signatures are required, LMS/HSS would be preferable.

Fig. 4 shows the absolute signing times for all parameter sets measured on our testing platform. We can see that all signatures take less than one second which would suffice for almost all software signing use-cases where signing does not take place live. Signing is performed offline, only once and thus is immaterial to the booting process. The standard deviation was insignificant. LMS/HSS and SPHINCS$^+$ with $W = 8$ and $w = 256$ perform worse than with $W = 4$ and $w = 16$ respectively, but their time is still acceptable. SPHINCS$^+$ performs orders of magnitude worse than LMS/HSS, but still at an acceptable level. Note that signing performance would increase even further if multi-threading was enabled.

Fig. 5 shows the HBS verification times. None of the

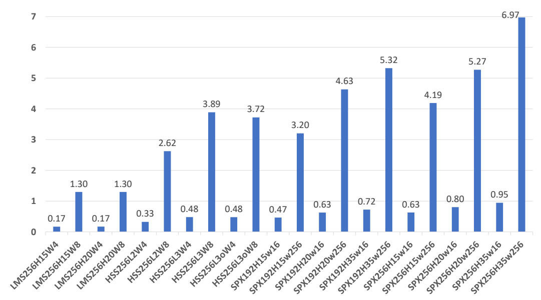


Fig. 5: HBS Verification Time (ms)

proposed parameter sets perform verification slower than 7ms which is satisfactory. Verification times would still be acceptable even at 100 or 200ms with slower processors than in our testbed. The standard deviation was insignificant. Parameters with $W = 4$ (LMS) or $w = 16$ (SPHINCS$^+$) verify signatures significantly faster than with $W = 8$ or $w = 256$ respectively.

To give some perspective, today's status-quo with secure boot and image signing is classical RSA2048. An **RSA2048** signature offers 112 bits of classical security, 0.26KB private, public key and signature, and was measured to take 1.657Mcycles/0.753ms per signature and 0.049Mcycles/0.022ms per verification in our testbed. **RSA4096** offers over 128-bits of classical security, 0.52KB private, public key and signature, and was measured to take 11.241Mcycles/5.11ms per signature and 0.173Mcycles/0.078ms per verification. Thus, RSA has smaller signatures and performs faster than HBS, but within the same order of magnitude which makes HBS appealing. Note that all RSA variants are considered to offer $\sim$0 bits of PQ security.

By combining all of the collected data, we can conclude that the best LMS parameters are with $W = 8$ where verification takes immaterially longer but signature sizes are a few Kilobytes smaller. The best HSS parameter is `HSS256L3oW4` which offers a balance of fast signing and verification with slightly bigger signature sizes. Regarding SPHINCS$^+$, parameters with $w = 256$ seem more favorable as they keep signature sizes smaller with acceptable signing and verification performance. Adopters could of course make different signature, signing and verification performance trade-off decisions and pick different parameters.

B. *FPGA-based signature verification*

Modern embedded systems, use FPGAs to perform variety of functions. While this can be facilitated by custom ASICs, FPGAs can also be used. FPGA configuration bitstreams exist at the lowest level of user-programmable functionality in the system. FPGAs provide greater logic density than circuits composed of discrete gates or most CPLDs, and they do so at a lower development cost than a custom ASIC. Custom ASICs also do not provide the same degree of flexibility and upgradability. An FPGA device often plays a fundamental role in the functioning of the system, capable of implementing basic "glue logic" that manages separate design domains and their components. In more advanced applications, FPGAs may also be customizable hardware acceleration resources. When implementing signature verification functions or system critical functions, it is essential that the system only use authentic FPGA images.

Vendors of FPGAs have included various technologies in their devices to protect the confidentiality of bitstream configuration data and integrity assurances to ensure correct hardware functionality. However, these technologies are not equivalent among all vendors, nor are the same levels of security shared among different device families from the same vendor. Often times, the built-in bitstream integrity functions are not available to user logic. Furthermore, it is not uncommon for an FPGA design to implement user logic that emulates a microcontroller or even a full-fledged "soft" CPU core that itself runs user-defined microcode or firmware. Using post-quantum signatures in upgrade images, microcontroller code or code for other parts of the system would improve the security of the system in a quantum computing future.

To evaluate the practicality of HBS verification in FPGAs, we developed an FPGA core capable of verifying a subset of the LMS parameters. We implemented verifiers for HSS with two (`L=2`) `LMS256H5W8` subtrees. We also implemented single-level LMS `LMS256H20W4` and `LMS256H20W8` verifiers. We tested these parameters through functional simulation. We chose to focus on the LMS algorithms for this evaluation as their smaller signature size relative to SPHINCS$^+$ requires fewer Block RAM resources to store a signature on-chip for verification. This concern is particularly relevant to lower power, lower density, and less costly devices that are more likely to be used in embedded and IoT-type applications. We did not implement SPHINCS$^+$ verification in FPGAs. We expect their required resources to be higher but in-line with the resources observed in our FPGA-based LMS verifier. To support this claim, [27] evaluated SPHINCS$^+$ verification with Xilinx Vivado HLS and Xilinx Virtex-7 FPGA as the target devices and showed that SPHINCS$^+$ verification would perform acceptably even in constrained devices.

To provide a more direct comparison of LMS with an RSA implementation, we implemented our design in an FPGA from the Xilinx UltraScale architecture family. Testing assumed that a short message and its signature were preloaded into a known location of FPGA Block RAM dedicated to the LMS verifier logic. The LMS algorithm would run to completion and provide a pass/fail indication via a top-level interface port. The messages were very short (less than 256 bytes) to ensure time measurements also predominantly represented signature verification time. Table III shows our results. The stateful HBS data points were collected from Xilinx Vivado 2018.2, targeting a mid-speed, industrial temperature grade Kintex UltraScale device (`xcku025-ffva1156-2-i`). Vivado was set to use the default options for the `Flow_PerfOptimized_high` Synthesis Strategy and the `Performance_Explore` Implementation Strategy. Timing constraints defining a 4ns logic clock period were successfully met. The RSA4096 results were kindly provided by Xilinx from an example design that was optimized for low logic area or high clock rate. The RSA2048 results were taken from [39]. The latter design

Parameter	Time (ms)	Clock (MHz)	FFs	LUTs	BRAM	DSP	CLB
RSA2048 [39]	-	$\sim$300	94455	28668	0	128	3428
RSA4096-Verilog [*]	$\sim$60	200	316	818	1	1	-
RSA4096-VHDL [†]	$\sim$10	200	484	346	1	3	-
LMS256H5W8 (HSS L=2) [#]	$\sim$10						
LMS256H20W4[#]	$\sim$0.8	250	468 [#]	811 [#]	1	1	97 [#]
LMS256H20W8 [#]	$\sim$5						

TABLE III: Verification in Xilinx UltraScale chips

[*] Measurements from example Verilog design provided by Xilinx.

[†] Measurements from example VHDL design provided by Xilinx.

[#] SHA256 logic excluded. SHA256 accounted for 1117 FFs, 1852 LUTs and 374 CLBs. The hashing of the image is limited by the Serial Peripheral Interface (SPI) speed when reading the image from flash at just a few MiB/s. Our SHA256 implementation was hashing data $\sim$184MiB/s. The Xilinx Vitis Libraries documentation reports 296 MiB/s respectively (64 bytes per 68 cycles at 330.25 MHz). Thus, the image hash calculation was excluded from our evaluation so that our results are not dominated by the amount of time it takes to read the signature into on-chip memory.

is optimized for speed and compatibility with the new Vitis development platform and therefore has a significantly higher logic footprint. All RSA data points exclude the image hash calculations that would precede an image verification, which would increase the footprint significantly.

Given this approach, we observe that the performance of our LMS implementation, in terms of clock rate, exceeds that of comparative cores implementing RSA4096 signature verification. It does not attain the speed of the Vitis library module (RSA2048) though. Total verification time between the classical and PQ signature algorithms is on the same order of magnitude. In terms of area, LMS, while not nearly as large as the the Vitis design (RSA2048), is larger than the VHDL-optimized RSA4096 module and similar to the RSA4096-Verilog one. Note that, even though it is not included in the RSA rows in Table III, extra SHA256 footprint will still exist in an RSA verification scenario to calculate the digest of the image, which we expect to be similar to the additional LMS SHA256 logic reported in Table III-footnote [#].

It still remains feasible to further optimize an LMS verifier to fit into almost the smallest of recent-generation FPGAs. We expect that the footprint of verifier logic for other LMS/HSS and SPHINCS$^+$ parameters will be higher, but easily supported by most modern FPGAs.

In terms of total impact to the booting process if we assume the boot chain starts from a Xilinx verifier, it is clear that $\sim$10ms will be unnoticeable even in use-cases where booting up takes just a few seconds. In any case, booting from a HBS-signed Xilinx bitstream will not affect the total boot time more than a few milliseconds which is the case for classical signatures as well.

V. Conclusion

In conclusion, in this work we evaluated the impact of using post-quantum hash-based signatures for secure boot and software signing. We proposed parameter sets at different security levels that would be useful for different software signing use-cases and introduced an architecture that would work for most vendors. We experimentally showed that the impact of switching to such signatures will be insignificant for the verifier compared to conventional RSA used today in various use-cases (i.e. hardware secure boot, FPGA). We also showed that the signer will be more impacted, but still at an acceptable level. Finally, we discussed practical issues and concerns of migrating to HBS signatures and their alternatives.

Acknowledgments

We would like to thank Scott Fluhrer for his LMS code, his optimizations and his valuable guidance and feedback. The authors would also like to thank Jason Moore and Jim Wesselkamper from Xilinx for their sample Xilinx design data points included in this work. The LMS FPGA logic measurements were based on code developed by Md Mahbub Alam, a Ph.D. Candidate at the University of Florida at the time. Thanks to Bruno Couillard and Jim Goodman from Crypto4A for the interesting discussions about HBS and their feedback. Finally, we would like to acknowledge Joost Rijneveld for his feedback and comments regarding SPHINCS$^+$ parameters and the SPHINCS$^+$ implementation and Dimitrios Sikeridis for his help with the experiments.

References

[1] Cisco, "Cisco Secure Boot and Trust Anchor Module differentiation solution overview," 2017, https://www.cisco.com/c/en/us/products/collateral/security/cloud-access-security/secure-boot-trust.html.

[2] M. Mosca, "Cybersecurity in an era with quantum computers: will we be ready?" *IEEE Security & Privacy*, vol. 16, no. 5, pp. 38–41, 2018.

[3] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[4] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," *Quantum Info. Comput.*, vol. 3, no. 4, pp. 317–344, Jul. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=2011528.2011531

[5] ETSI, "ETSI TC Cyber Working Group for Quantum-Safe Cryptography," https://portal.etsi.org/TBSiteMap/CYBER/CYBERQSCToR.aspx, 2017, Web page. Accessed 2019-07-25.

[6] S. Fluhrer, D. McGrew, P. Kampanakis, and V. Smyslov, "Postquantum Preshared Keys for IKEv2," Internet Engineering Task Force, Internet-Draft draft-ietf-ipsecme-qr-ikev2-08, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ipsecme-qr-ikev2-08

[7] M. Ounsworth and M. Pala, "Composite Keys and Signatures For Use In Internet PKI," Internet Engineering Task Force, Internet-Draft draft-ounsworth-pq-composite-sigs-01, Jul. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ounsworth-pq-composite-sigs-01

[8] D. Steblia, S. Fluhrer, and S. Gueron, "Design issues for hybrid key exchange in TLS 1.3," Internet Engineering Task Force, Internet-Draft draft-stebila-tls-hybrid-design-01, Jul. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-stebila-tls-hybrid-design-01

[9] C. Tjhai, M. Tomlinson, grbartle@cisco.com, S. Fluhrer, D. V. Geest, O. Garcia-Morchon, and V. Smyslov, "Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2)," Internet Engineering Task Force, Internet-Draft draft-tjhai-ipsecme-hybrid-qske-ikev2-04, Jul. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-tjhai-ipsecme-hybrid-qske-ikev2-04

[10] P. E. Hoffman, "The Transition from Classical to Post-Quantum Cryptography," Internet Engineering Task Force, Internet-Draft draft-hoffman-c2pq-05, May 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-hoffman-c2pq-05

[11] D. McGrew, M. Curcio, and S. Fluhrer, "Leighton-Micali Hash-Based Signatures," RFC 8554, Apr. 2019. [Online]. Available: https://rfc-editor.org/rfc/rfc8554.txt

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

[13] J.-P. Aumasson, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange *et al.*, "SPHINCS+ - Submission to the 2nd round of the NIST post-quantum project," https://sphincs.org/data/sphincs+-round2-specification.pdf, 2019, Specification document (part of the submission package).

[14] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn, "SPHINCS: Practical Stateless Hash-Based Signatures," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2015, pp. 368–397.

[15] A. Hülsing, J. Rijneveld, S. Samardjiska, and P. Schwabe, "From 5-pass MQ-based identification to MQ-based signatures." *IACR Cryptology ePrint Archive*, vol. 2016, p. 708, 2016.

[16] S. Kölbl, M. M. Lauridsen, F. Mendel, and C. Rechberger, "Haraka v2-efficient short-input hashing for post-quantum applications," *IACR Transactions on Symmetric Cryptology*, pp. 1–29, 2016.

[17] Y. Yoo, R. Azarderakhsh, A. Jalali, D. Jao, and V. Soukharev, "A Post-Quantum Digital Signature Scheme Based on Supersingular Isogenies," Cryptology ePrint Archive, Report 2017/186, 2017, http://eprint.iacr.org/2017/186.

[18] S. Rohde, T. Eisenbarth, E. Dahmen, J. Buchmann, and C. Paar, "Fast hash-based signatures on constrained devices," in *Smart Card Research and Advanced Applications*, G. Grimaud and F.-X. Standaert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 104–117.

[19] J. Buchmann, E. Dahmen, and A. Hülsing, *XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 117–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25405-5_8

[20] A. Hülsing, C. Busold, and J. Buchmann, "Forward secure signatures on smart cards," in *Selected Areas in Cryptography*, L. R. Knudsen and H. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 66–80.

[21] 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.

[22] V. B. Y. Kumar, N. Gupta, A. Chattopadhyay, M. Kaspert, C. Krauß, and R. Niederhagen, "Post-quantum secure boot," in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, ser. DATE '20. San Jose, CA, USA: EDA Consortium, 2020, p. 15821585.

[23] S. F. Panos Kampanakis, "LMS vs XMSS: A comparison of the Stateful Hash-Based Signature Proposed Standards," Cryptology ePrint Archive, Report 2017/349, 2017, http://eprint.iacr.org/2017/349.

[24] M. Ando, J. D. Guttman, A. R. Papaleo, and J. Scire, "Hash-based tpm signatures for the quantum world," in *Applied Cryptography and Network Security*, M. Manulis, A.-R. Sadeghi, and S. Schneider, Eds. Cham: Springer International Publishing, 2016, pp. 77–94.

[25] D. Boneh and S. Gueron, "Surnaming schemes, fast verification, and applications to SGX technology," in *Topics in Cryptology – CT-RSA 2017*, H. Handschuh, Ed. Cham: Springer International Publishing, 2017, pp. 149–164.

[26] A. Hlsing, J. Rijneveld, and P. Schwabe, "Armed sphincs – computing a 41kb signature in 16kb of ram," Cryptology ePrint Archive, Report 2015/1042, 2015, https://eprint.iacr.org/2015/1042.

[27] K. Basu, D. Soni, M. Nabeel, and R. Karri, "Nist post-quantum cryptography- a hardware evaluation study," Cryptology ePrint Archive, Report 2019/047, 2019, https://eprint.iacr.org/2019/047.

[28] D. Soni, K. Basu, M. Nabeel, and R. Karri1, "A Hardware Evaluation Study of NIST Post-Quantum Cryptographic Signature schemes," *Second PQC Standardization Conference*, Aug 2019.

[29] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4," Cryptology ePrint Archive, Report 2019/844, 2019, https://eprint.iacr.org/2019/844.

[30] C. Roma, C.-E. A. Tai, and M. A. Hasan, "Energy Consumption of Round 2 submissions for NIST PQC Standards," *Second PQC Standardization Conference*, Aug 2019.

[31] J. A. Buchmann, D. Butin, F. Göpfert, and A. Petzoldt, "Post-quantum cryptography: state of the art," in *The New Codebreakers*. Springer, 2016, pp. 88–108.

[32] A. Langley, "Email thread: Proposed addition of hash-based signature algorithms for certificates to the LAMPS charter," 2018, https://mailarchive.ietf.org/arch/msg/spasm/PgzLjPcg-jfywQFQs9gMLFcgRd8.

[33] "Stateful Hash-Based Signatures - public comments on misuse resistance," 2019, https://csrc.nist.gov/CSRC/media/Projects/Stateful-Hash-Based-Signatures/documents/stateful-HBS-misuse-resistance-public-comments-April2019.pdf.

[34] Microsoft, "Windows Secure Boot Key Creation and Management Guidance," 2017, https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance.

[35] S. Fluhrer and Q. Dang, "Additional Parameter sets for LMS Hash-Based Signatures," Internet Engineering Task Force, Internet-Draft draft-fluhrer-lms-more-parm-sets-00, Sep. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-fluhrer-lms-more-parm-sets-00

[36] S. Fluhrer, "Lms hash based signature open-source implementation," 2019, https://github.com/cisco/hash-sigs.

[37] P. Kampanakis, "Slim SPHINCS$^+$ open-source implementation," 2019, https://github.com/csosto-pk/slim_sphincsplus/tree/master/ref.

[38] SPHINCS$^+$ team, "SPHINCS$^+$ open-source implementation," 2019, https://github.com/sphincs/sphincsplus.

[39] Xilinx, "Vitis Security Library," 2019, https://github.com/Xilinx/Vitis_Libraries/blob/8ee9037aeb2bdf44096c256ec6779973387e0c0f/security/docs/guide_L1/internals/rsa.rst.