Post-Quantum LMS and SPHINCS$^+$ Hash-Based Signatures for UEFI Secure Boot

PANOS KAMPANAKIS, Security & Trust Organization, Cisco Systems, USA
PETER PANBURANA, Security & Trust Organization, Cisco Systems, USA
MICHAEL CURCIO, Security & Trust Organization, Cisco Systems, USA
CHIRAG SHROFF, Security & Trust Organization, Cisco Systems, USA
MD MAHBUB ALAM, University of Florida, USA

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. This would jeopardize IT security as we know it. In this work, we investigate two quantum-safe, 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-case in question and we prove that post-quantum signatures with less than one second signing and less than 10*ms* verification would not have material impact (less than 1‰) on secure boot. We evaluate the hierarchical design of these signatures in hardware-based and virtual secure boot. In addition, we develop Hardware Description Language code and show that the code footprint is just a few kilobytes in size which would fit easily in almost all modern FPGAs. We also analyze and evaluate potential challenges for integration in existing technologies and we discuss considerations for vendors embarking on a journey of image signing with hash-based signatures.

Additional Key Words and Phrases: HBS signatures, PQ image signing, PQ root of trust, post-quantum secure boot

Updates: Initially uploaded to Cryptology ePrint Archive on Jan 11, 2021. Re-uploaded on June 4, 2021 with two sentences stating that the customized parameter sets perform significantly faster signing than the original SPHINCS$^+$ parameter sets corrected.

1 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 operating system and software applications on digital devices is genuine, vendors have implemented several security features (e.g., secure boot) which validate authenticity by using digital signatures. Without such protections, rogue code running on a system could lead to sensitive information leakage, software or hardware piracy, revenue loss and more. [33] serves as an example where FPGA hardware compromises the secure boot process to boot from a malicious network source instead of an authentic signed kernel image.

Secure boot is a multi-step process that involves verifying signatures on software before booting it. The signatures 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 [50]. Shor's quantum algorithm [58, 63], assuming a practical quantum computer (QC) becomes available, would solve (elliptic curve) discrete logarithm ((EC)DL)

Authors' addresses: Panos Kampanakis, Security & Trust Organization, Cisco Systems, USA, panosk@cisco.com; Peter Panburana, Security & Trust Organization, Cisco Systems, USA, pepanbur@cisco.com; Michael Curcio, Security & Trust Organization, Cisco Systems, USA, micurcio@cisco.com; Chirag Shroff, Security & Trust Organization, Cisco Systems, USA, cshroff@cisco.com; Md Mahbub Alam, University of Florida, USA, mahbub.alam@ufl.edu.

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 code in the secure trust chain and boot non-genuine, malicious software [33].

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 exchange, public key encryption and digital signature algorithms. At the time of this writing, NIST's evaluation process is in Round 2 with 26 PQ algorithms remaining. Similarly, the European Telecommunications Standards Institute (ETSI) has formed a Quantum-Safe Working Group [17] 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 IKEv2 [20, 25, 54, 68, 70] and has published two PQ signature algorithm specifications as Informational RFCs [29, 47]. 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.

Our focus in this work is on post-quantum secure boot signatures. Taking the long lifecycles of many of our systems today and the fact that secure boot is ultimately anchored in hardware, it is important to investigate the viability of post-quantum secure boot signatures in order to have a solution in place to protect against a QC forging software in the future. We chose to evaluate two well-established hash-based signature (HBS) schemes, namely LMS [47] and SPHINCS$^+$ [3], which are based on quantum-secure, proven primitives. These primitives are considered more well-trusted and mature than most other Round 2 NIST PQ signature candidate schemes. We compare their verification time against classical RSA signatures used today. Verification happens a few times with every system boot, 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. To our knowledge, this is the first work that investigates post-quantum UEFI Secure Boot. Prior works measure post-quantum signature performance regardless of application and do not focus on the UEFI use-case and its challenges.

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

(iii) We use instrumented HBS implementations that minimize code size and optimize performance and additionally implement HBS in a Xilinx UltraScale FPGA.

(iv) We analyze the impact of two well-studied HBS algorithms when they are used for image signing in hardware, virtual secure boot and FPGAs, and compare them to classical RSA. We show that trusted post-quantum signatures are possible with immaterial impact on the verifier, and an acceptable impact on the signer.

(v) We identify an important implementation incompatibility with the one-shot calculation of an HBS signature against how the OpenSSL CMS implementation calculates classical signatures.

Note that we do not attempt to benchmark HBS schemes on various CPUs as that has been studied by others. Our goal is to study the viability of HBS signatures in UEFI Secure Boot in systems today.

The rest of the paper is organized as follows: Section 2 summarizes related work. Section 3 gives background details on UEFI Secure Boot and HBS signature details. Section 4 lays out the proposed hierarchy and parameter sets for our investigation and Section 5 presents our experimental results in various platforms. Section 6 discusses concerns and open issues for integrating these signatures in secure boot technologies used today. Section 7 concludes the paper.

2 RELATED WORK

There has been a large body of research on PQ cryptography [5, 31, 39, 74]. Recently, more works are exploring NIST's PQ candidate schemes focusing mainly on their security and computational performance.

Research teams from academia and industry are investigating the performance of PQ candidate algorithms in Internet protocols. The authors in [62] integrated NTRU in DTLS and studied its performance. In [8], Bos et al. introduced RLWE for key exchange in TLS 1.2 and studied its impact. Google, Amazon AWS and Cloudflare have also been working on finding suitable PQ algorithms for TLS key exchange [11, 28, 41, 41–43, 45]. Moreover, Campagna discussed hybrid key exchange and double tunnelling for TLS and presented the significant slowdown introduced to TLS by hybrid key exchange [11]. [37, 64], on the other hand, discussed the impact of PQ signature schemes on protocols that utilize X.509 certificates. Earlier work by Bindel et al. emulated large hybrid PQ certificates and studied their impact on TLS libraries and browsers [6]. [16] presented the challenges of implementing NIST's PQ key exchange and authentication algorithms in TLS and SSH, with a focus on hybrid schemes.

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 were first demonstrated in [60] which measured memory and time performance in 8-bit Atmel AVR microcontrollers. What is more, a variant of a stateful HBS scheme, called XMSS [9], was implemented on a 16-bit Infineon SLE78 smart-card in [30] with acceptable time performance which showed the practicality of stateful HBS in constrained devices. [22] also demonstrated an efficient implementation of XMSS in constrained IoT motes. [40] 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 [12, 55]. What's more, [2] 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 [7]. They proposed signing a limited set of SGX enclaves with stateful HBS and 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 [32]. They also compared SPHINCS with stateful XMSS, evaluated their performance and showed that stateless HBS verification is acceptable but its signing is 30 times slower. The authors in [4, 65] conducted a comparison of NIST Round 2 candidate algorithm hardware implementations with a focus on hardware footprint, latency and impact on hardware design. In [38], Kannwischer et al. benchmarked Round 2 algorithms on ARM Cortex-M4 and evaluated the more suitable ones for embedded devices by studying speed, code size and memory. [61] 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. The independent performance results in the works above were collected out of the UEFI Secure Boot context but can strengthen our argument that these signatures can perform satisfactorily in a secure boot scenario even in constrained devices or chips.

Our work goes beyond plain performance evaluation of HBS schemes as it brings HBS to secure boot and addresses potential concerns and challenges for their application.

Other post-quantum schemes also showed promising results on embedded systems. A lattice-based signature was shown to be faster than RSA in a Xilinx Spartan-6 FPGA in [24]. Additionally, [53] showed that lattice-based signatures can be fast on an ARM Cortex M4 with slightly higher memory usage compared to HBS. Multivariate-quadratic schemes were also proven to be practical on constrained devices as well as ASICs in [73]. These signature schemes are promising but do not come with the maturity and the well-trusted security guarantees of HBS signatures.

3 BACKGROUND

3.1 Secure Boot Overview

When starting the secure boot process, a device's firmware is initially booted from a tamper-resistant ROM, a Secure FPGA or a Trust Anchor Module. Usually the first firmware / boot 0 instructions run are stored in tamper resistant hardware, which establishes a chain of trust. Then the boot process is passed on to a bootloader, namely UEFI on x86 based systems, 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 by the Original Equipment Manufacturer (OEM) is verified by boot 0 code. The keys or certificates used to verify the bootloader are often stored inside a Trust Anchor Module. Onward, the OS signature is verified by the bootloader before loading the OS. While keys/certificates used to validate OS signatures are often attached to the image itself, the root certificate/key is stored in dedicated signature databases inside trust anchors or in BIOS flash. Fig. 1 provides an overview of this process which 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 [14].

UEFI Secure Boot leverages multiple types of keys and inventories: Platform Key (PK), Key Exchange Keys (KEK), allowed signatures database (db), forbidden signatures database (dbx) and Firmware Update Keys. The PK defines the trust between the platform and the firmware. The KEK defines the trust between the OS and the firmware. A Firmware Update Key signs firmware updates. db contains public keys and certificates used to verify firmware and OS signatures [49]. dbx contains hashes of untrusted components and revoked public keys and certificates. Often the UEFI db/dbx is stored in a Trust Anchor module. Adding new image verification keys to db and revoking bad images by adding hashes to dbx takes place with messages authenticated by the KEK, which itself can be managed by the PK.

In a virtual context, vendors often follow a similar paradigm. Instead of running from ROM or flash storage, UEFI firmware is stored as an Open Virtual Machine Format (OVMF) file on the host's file system for use by a hypervisor (e.g., QEMU-KVM). The public keys used to verify signatures on UEFI binaries are stored in UEFI' s db which is represented on disk as a file called `OVMF_VARS`.

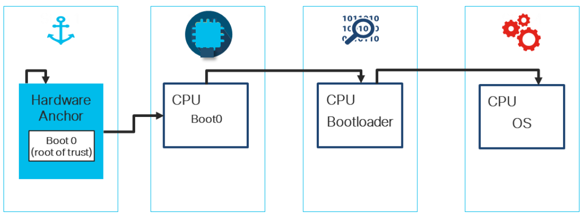


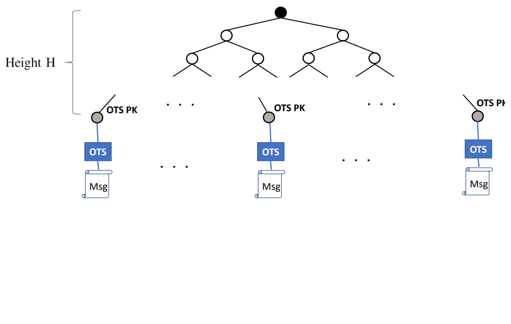
Fig. 1. Secure Boot Overview

Virtual UEFI Secure Boot involves three stages [23, 71]: (1) The UEFI firmware running in the hypervisor verifies the signature on a first stage bootloader called the SHIM [59] and, if successful, boots the bootloader. The SHIM is signed by a private key corresponding to a certificate contained in UEFI's db authenticated variable. Microsoft (or other SHIM signing vendor) certificates are found in the UEFI db of most commercially available personal computers today, and they are used to verify Windows installs and most major Linux distributions' SHIMs before loading them. These certificates use 2048-bit RSA keys. (2) A second stage bootloader (SSBL), most commonly GRUB in Linux, is loaded by the SHIM after verifying the SSBL's signature using a trusted certificate built into the SHIM. (3) Finally, the kernel itself is loaded by GRUB after verifying its signature by leveraging the UEFI db and SHIM certificates. In other words, each stage includes a signature of the software (i.e., bootloader or OS) which is verified by the software at the previous stage. Booting a guest OS includes three signature verifications. If all signatures on these EFI binaries can be successfully validated, then the host OS is booted. A failure results in an EFI error and booting is stopped. All software signatures are in PKCS#7 [35] format.

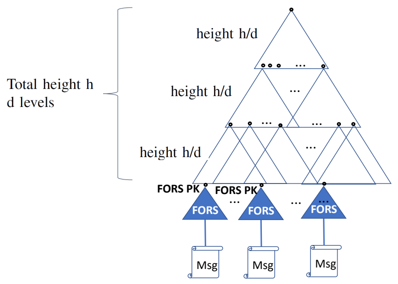
UEFI Secure Boot uses Microsoft Authenticode technology to sign executables. Authenticode is a digital signature format used to determine the authenticity of software binaries. Authenticode is based on the Public-Key Cryptography Standards (PKCS#7) standard and uses X.509 v3 certificates to bind a signed file to a software publisher. Authenticode signatures are used to digitally sign portable executable (PE) files. The signature is embedded in the PE file in a location specified by the `Certificate Table` entry in the Optional Header Data Directories. The signature stored in the location is in a `bCertificate` binary array which includes the PKCS#7 `SignedData` with the signature.

In addition, 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 Complex Programmable Logic Devices (CPLD), and they do so at a lower development cost than custom ASICs. 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 uses authentic FPGA images. FPGA vendors have included various technologies in their devices to protect the confidentiality of bitstream configuration data and integrity assurances to ensure correct hardware functionality. Over time, improvements to built-in features have ranged from obfuscation of the proprietary bitstream image formats and CRC verification, through encryption using the AES algorithm, to authentication using either symmetric (i.e., AES-GCM, HMAC-SHA256) or asymmetric (i.e., RSA, ECDSA) key algorithms. 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. Including a post-quantum authentication algorithm to verify signatures of upgrade images, code for microcontrollers or code for other parts of the system, would extend additional security properties into the hardware of the system.



(a) LMS tree



(b) SPHINCS$^+$ tree

Fig. 2. HBS trees

3.2 HBS Overview

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 [10]. 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 [47] and XMSS [29], and stateless SPHINCS$^+$, one of the NIST signature candidates [3]. Fig. 2a 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 stateful XMSS [9], which was also published by IETF [29], shares similarities with LMS with a tighter security proof, but with worse performance [55]. Fig. 2b 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 keys form the Merkle tree leaves that are 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$^+$.

3.3 Stateful vs Stateless HBS

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 signature forgeries. The state management requirement is considered an important disadvantage which has been often brought up in the IETF and NIST fora [44, 67]. NIST has announced, at the time of this writing, that it is authoring a Special Publication [15] that will make recommendations for stateful HBS and identify use-cases where stateful HBS are more appropriate. In general, HBS should be used in applications where state management is not a concern, signing rates are slow enough to ensure state can be protected, and other PQ signatures are not appropriate.

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.

4 HBS FOR SECURE BOOT

4.1 Hierarchy

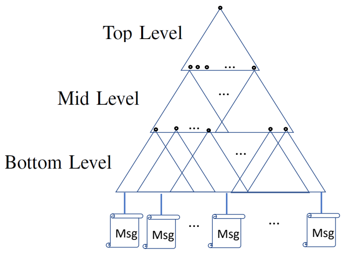


Fig. 3. HBS hierarchy

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. 3 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.

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^{20} bottom trees. A bottom tree that can sign 2^{15} images would probably suffice for most use-cases. Most vendors' portfolio would never exceed 2^{35} total images per signing root. 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. Thus 2^{35} total signatures would suffice for such vendors. In reality, there would be more than one root signing a vendor's software, which means that although 2^{35} is our conservative choice, smaller trees would probably suffice for most vendors.

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, signature and revocation list exist in the UEFI db and dbx respectively [49]. PK revocation and refresh takes place by entering Setup mode or using a message signed by the PK tree that is being revoked. HBS Key revocation would take place by using a revocation, firmware update image, or a signed (by the PK, or KEK tree) message which would add the corresponding HBS root in the dbx database. Fig. 4 shows an example of a revoked KEK being added to the UEFI dbx which automatically invalidates all signed images by this KEK.

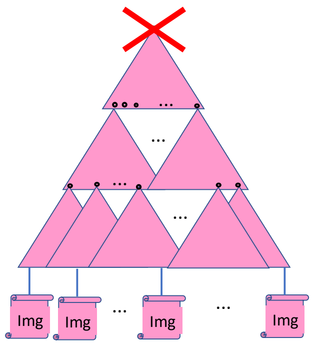


Fig. 4. HBS Root Revocation in UEFI Secure Boot

Some vendors [69] use special images signed by rollover keys in order to revoke PKs. This concept is very similar to UEFI's Firmware Update Keys. A rollover key could be another HBS tree used to sign revocation images which include a new tree root to be loaded on the platform.

4.2 Parameter Sets

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 multiple 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 4.1. The PQ security level of all LMS parameters is 128 bits. These parameters could be adjusted to provide 96-bit PQ security by truncating the SHA256 outputs to 192 bits [19]. 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 exceed a total of 34 trillion (2^{35}) signatures. The equivalent NIST-submitted SPHINCS^{+} parameter sets (2^{64} total signatures) with bigger signatures (~5-25KB) and slower verification ($\sim \times$2-4). 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 1 summarizes all the LMS and SPHINCS^{+} parameter sets we used in our evaluation and their respective post-quantum security level in bits. `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. Regarding `HSS256L3W4` and `HSS256L3W8`, their key generation and key load times are high, but we included them for reference. A more balanced option exists in the `HSS256L3oW4` and `HSS256L3oW8` parameter sets.

When choosing parameter sets, we also considered the type of tree. LMS and SPHINCS^{+} use binary trees, but **2^{y}-ary trees** (e.g., 4-ary trees with four children per node) have also been proposed. We evaluated the benefit of introducing 2^{y}-ary trees instead of binary ones. An x'-height 2^{y}-ary tree can offer as many signatures as an x-height binary tree when $x' = x/y$. The path to the root is x nodes for a binary tree and $(2^{y} - 1)x'$ nodes for the 2^{y}-ary tree. For a 4-ary tree that becomes $3x/2$ nodes, which is $x/2$ more nodes than the corresponding binary tree. For a typical $x = 20$, that would be 10 more nodes, or ~0.3KB bigger signature. For a 8-ary tree, the corresponding signature would be ~0.85KB smaller than a binary tree's. Such small signature size improvements are considered insignificant. In terms of storage size, a binary tree has $2^{x+1} - 1$ total nodes and a 2^{y}-ary one has $(2^{y(x'+1)} - 1)/(2^{y} - 1)$. For a typical $x = 20$ binary tree of 2 million nodes, the corresponding 4-ary tree would have ~1.4 million nodes and the 8-ary one ~1.2 million nodes. So, even if an implementation was storing all the nodes of the tree, it would save 0.6 and 0.8 million nodes by using a 4-ary or 8-ary tree which would amount to 20MB or 25MB of memory respectively. Storing the whole tree is optional as there are techniques to keep auxiliary data to avoid rebuilding the tree with every signature, but 20-25MB of memory seems inexpensive for a typical signer anyway. Regarding performance, 2^{y}-ary tree performance will be faster because it includes less node calculations, but since most HBS signing and verification time is spent in the OTS/FTS hash calculations [55] we do not expect a significant speedup over binary trees. Additionally the LMS and SPHINCS^{+} "simple" parameter sets we experimented with were optimized to only use one hash compression per SHA256 hash calculation. Internal node calculations for a 2^{y}-ary tree would require three hash compressions. Thus, overall we do not expect that 4-ary or 8-ary trees would significantly benefit our scenarios over binary trees, so all our experiments use binary trees.

Parameter name	HBS Scheme	Parameters	PQ Sec Level
LMS256H5W4	LMS	LMS_SHA256_M32_H5 with LMOTS_SHA256_N32_W4	128
LMS256H5W8	LMS	LMS_SHA256_M32_H5 with LMOTS_SHA256_N32_W8	128
LMS256H10W4	LMS	LMS_SHA256_M32_H10 with LMOTS_SHA256_N32_W4	128
LMS256H10W8	LMS	LMS_SHA256_M32_H10 with LMOTS_SHA256_N32_W8	128
LMS256H15W4	LMS	LMS_SHA256_M32_H15 with LMOTS_SHA256_N32_W4	128
LMS256H15W8	LMS	LMS_SHA256_M32_H15 with LMOTS_SHA256_N32_W8	128
LMS256H20W4	LMS	LMS_SHA256_M32_H20 with LMOTS_SHA256_N32_W4	128
LMS256H20W8	LMS	LMS_SHA256_M32_H20 with LMOTS_SHA256_N32_W8	128
HSS256L2W4	HSS	$L = 2$ with top LMS256H20W4 and bottom LMS256H15W4	128
HSS256L2W8	HSS	$L = 2$ with top LMS256H20W8 and bottom LMS256H15W8	128
HSS256L3W4	HSS	$L = 3$ with LMS256H5W4, LMS256H10W4, LMS256H20W4	128
HSS256L3W8	HSS	$L = 3$ with LMS256H5W8, LMS256H10W8, LMS256H20W8	128
HSS256L3oW4	HSS	$L = 3$ with LMS256H15W4, LMS256H10W4, LMS256H10W4	128
HSS256L3oW8	HSS	$L = 3$ with LMS256H15W8, LMS256H10W8, LMS256H10W8	128
SPX192H15w16	SPHINCS$^+$	$n = 24, H = 15, d = 3, k = 18, a = 13$ $w = 16$	96
SPX192H15w256	SPHINCS$^+$	$n = 24, H = 15, d = 3, k = 18, a = 13$ $w = 256$	96
SPX192H20w16	SPHINCS$^+$	$n = 24, H = 20, d = 4, k = 18, a = 13$ $w = 16$	96
SPX192H20w256	SPHINCS$^+$	$n = 24, H = 20, d = 4, k = 18, a = 13$ $w = 256$	96
SPX192H35w16	SPHINCS$^+$	$n = 24, H = 35, d = 5, k = 20, a = 12$ $w = 16$	96
SPX192H35w256	SPHINCS$^+$	$n = 24, H = 35, d = 5, k = 20, a = 12$ $w = 256$	96
SPX256H15w16	SPHINCS$^+$	$n = 32, H = 15, d = 3, k = 19, a = 16$ $w = 16$	128
SPX256H15w256	SPHINCS$^+$	$n = 32, H = 15, d = 3, k = 19, a = 16$ $w = 256$	128
SPX256H20w16	SPHINCS$^+$	$n = 32, H = 20, d = 4, k = 19, a = 16$ $w = 16$	128
SPX256H20w256	SPHINCS$^+$	$n = 32, H = 20, d = 4, k = 19, a = 16$ $w = 256$	128
SPX256H35w16	SPHINCS$^+$	$n = 32, H = 35, d = 5, k = 21, a = 15$ $w = 16$	128
SPX256H35w256	SPHINCS$^+$	$n = 32, H = 35, d = 5, k = 21, a = 15$ $w = 256$	128

Table 1. HBS Parameters

5 EXPERIMENTAL RESULTS

5.1 Algorithm Performance

To evaluate algorithm performance in secure boot we measured key generation, signature generation and verification. Signing takes place once per signed object or image in an offline fashion, thus its performance is not critical but high-rate signers would still want to maintain relatively fast signing. Key generation takes place only when initializing a signer, so key generation performance is not as important. Verification takes place by different verifiers every time booting takes place, thus verification performance is the most important for the use-case. Another important factor for a secure boot verifier is the size of the public key because often keys/certificates are physically stored inside a tamper proof chip with limited storage capacity. The private key size is not as significant as it only affects the signer who is usually not resource-constrained. Additionally, verifiers sometimes have limited memory which would be affected by high code footprint verification, thus we were interested in code size and stack use for HBS verification code.

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 [18], dynamically linked to OpenSSL's SHA256 implementation and properly instrumented with various performance optimizations and memory-speed trade-offs. In HSS, we used auxiliary data that consist of part of the top tree. The auxiliary data was designed to speed up the key load time and can vary in size up to 10KB to balance between speed and memory size. We also instrumented extra data used for signing. This data speeds up signing and has variable length up to 100KB in order trade off between signing speed and memory size. The extra data is recreated from the private key and the auxiliary data. Our LMS implementation can also take advantage of multithreading for HSS but we kept it disabled for our experiments. Our SPHINCS$^+$ testing code [36] is a fork of the original code [66] from the SPHINCS$^+$ NIST submission tweaked for our goals. 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 [66] 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 [38], 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 had been overwritten.

Table 2 shows the results from our testing. We can see that all the private and **public key sizes** are of negligible size. It is also clear that a verifier code and stack size of a few KB would not practically impact most secure boot verifiers. The heap 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 [32]. 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, our LMS implementation was optimized by taking longer to generate keys but performs signing faster by using auxiliary data and pre-loaded private keys, whereas our SPHINCS$^+$ implementation included part of this work in the signature generation itself.

Fig. 5 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 smaller 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 [19] which would shrink all reported LMS/HSS signatures by more than 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.

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
HSS256L3W4	48	60	3.15	1.81	339.6	7.512	0.056	1.054	0.028
HSS256L3W8	48	60	2.51	1.81	2659.7	30.266	0.319	8.553	0.263
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 2. HBS Keys, Memory footprint and Performance

Fig. 6a 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. The standard deviation was insignificant. LMS/HSS and SPHINCS$^+$ with W=8 and w=256 respectively perform worse than with W=4 and w=16, 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. 6b shows the HBS **verification times**. None of the proposed parameter sets perform verification slower than 7ms, which is satisfactory. 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, a classical **RSA2048** signature offers 112 bits of classical security, 0.26KB private, public key and signature. We measured it in our testbed to take 1.657Mcycles/0.753ms per signature and 0.049Mcycles/0.022ms per verification. **RSA4096** offers over 128-bits of classical security, 0.52KB private, public key and signature, and was measured to take 11.241Mcycles/5.11ms

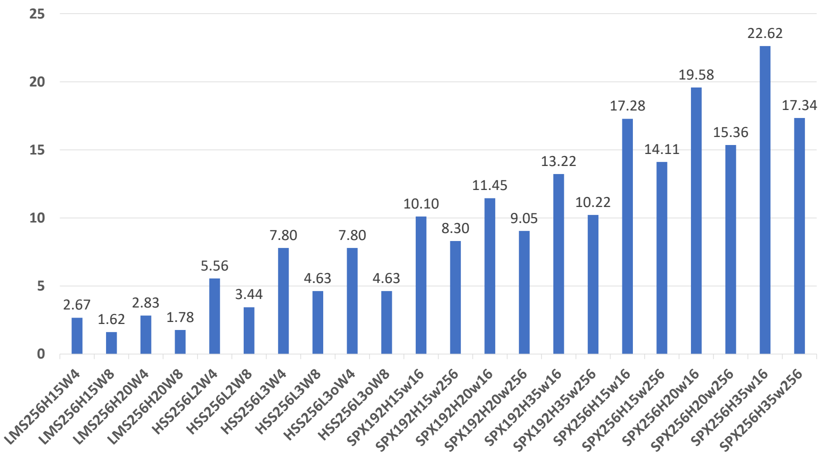


Fig. 5. HBS Signature Size (KB)

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 ~0 bits of PQ security.

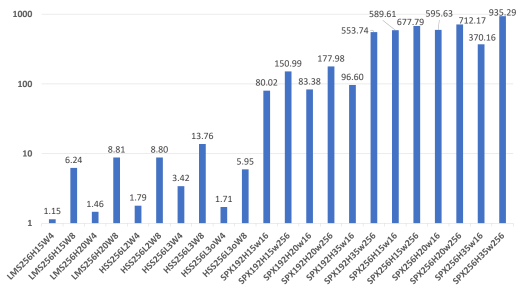
By combining all of the collected data, we can conclude that the best LMS parameters are with W=8 and the best HSS parameter is `HSS256L3oW8`. Regarding SPHINCS$^+$, parameters with w=256 seem more favorable as they keep signature sizes smaller with acceptable signing and verification performance. Adopters could of course make different signature, signing and verification performance trade-off decisions and pick different parameters.

5.2 Hardware Secure Boot Performance

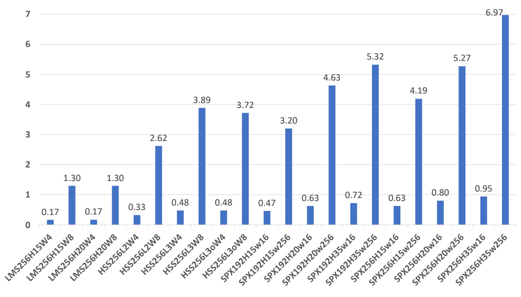
Parameter	Total Boot Time (s)
RSA2048	95

Table 3. Cisco Firepower 2110 Total Secure Boot Time (with RSA2048)

To assess the potential impact of HBS signatures on a hardware secure boot environment, we measured the total boot time of a Cisco Firepower 2110 appliance. Cisco Systems is a big IT vendor offering various platforms with different resource constraints, performance and cost. Appliances like the Firepower 2110 are small next-generation firewalls with average processing resources of an 1.8GHz Intel x86 4-core processor and 16GB DDR4 RAM. Table 3 shows the total boot time we measured with Firepower 2110s. Readers should note that the total time includes two RSA2048 verifications for the firmware and OS



(a) Signing (logarithmic scale)



(b) Verification

Fig. 6. HBS Time (ms)

images and two more verifications for OS packages. It was almost one minute from booting the BIOS until the OS image signature was verified and the OS started loading. The OS image total verification time was measured approximately to be 15s most of which were spent calculating the hash of the image. Most other Cisco platforms we tested manifest similar secure boot times (leveraging RSA2048 image signatures) of several seconds.

We can see that in Cisco platforms, boot times (using RSA2048) are orders of magnitude longer (tens of seconds) than the HBS verification times (few milliseconds) we measured in Table 2. Thus, we believe the HBS signatures have immaterial impact (<0.5‰) on Cisco secure boot and the vast majority of use-cases. Rare cases where HBS signatures may be deemed unfavorable would include constrained or time-sensitive applications. Even though we consider such cases rare, we believe that an optimized LMS verification could still offer the performance needed for these environments.

5.3 FPGA-based Signature Verification Performance

To evaluate the practicality of HBS verification in FPGAs, we developed an FPGA core capable of verifying the LMS parameters. We chose to focus on the LMS algorithms 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. Even though we did not implement SPHINCS$^+$ verification in FPGAs, we expect their required resources to be higher than, but in-line with, the resources observed in our FPGA-based LMS verifier. To support this claim, [4] 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.

We chose to focus on parameters with W=8 since these gave us smaller signature sizes at acceptable levels of performance. We also did not test HSS256L3W8 as HSS256L3oW8 performed better (Section 5.1). We tested these parameters through functional simulation. The Hardware Description Language (HDL) representation of our LMS verifier was developed with the goal of minimizing area use while achieving vendor-agnostic support for different FPGA architectures. In general, optimizing for time leads to an increase in the area footprint and vice versa. Since LMS verification is relatively fast and verifiers can sometimes be constrained, we considered area optimizations more important. Moreover, modern FPGAs usually offer Block RAMs suitable for fast, large, and flexible data storage and distributed RAMs useful for storing coefficients, state machines, and small buffers [1]. We took advantage of these embedded memory options to optimize our LMS verifier.

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 4 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 [72]. The latter design was optimized for speed and compatibility with the new Vitis development platform and therefore has a significantly higher logic footprint. All data points

Parameter	Time (ms)	Clock (MHz)	FFs	LUTs	BRAM	DSP	CLB
RSA2048 [72]	-	~300	94455	28668	0	128	3428
RSA4096-Verilog *	~60	200	316	818	1	1	-
RSA4096-VHDL †	~10	200	484	346	1	3	-
LMS256H10W8	~5.4	250	517 #	968 #	2	1	108 #
LMS256H15W8	~6.3						
LMS256H20W8	~5.1						
HSS256L2W8	~9.8						
HSS256L3oW8	~16.1						

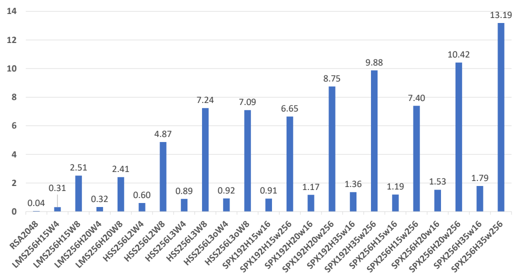
Table 4. Verification in Xilinx UltraScale chips

* Measurements from example Verilog design provided by Xilinx.
† Measurements from example VHDL design provided by Xilinx.
# SHA256 logic excluded. SHA256 accounted for 1117 FFs, 1865 LUTs and 369 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 ~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.

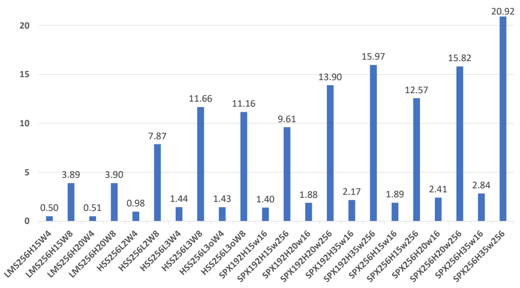
in Table 4 exclude the image hash calculations which would precede an image verification and increase the footprint significantly.

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 4, extra SHA256 logic 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 4-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.

With the desire to make the design of the LMS verification module as efficient as possible, it is worth noting the trade-off between logic area and performance. We discovered that the SHA256 hash engine was the largest component of the design and the primary limiting factor of the maximum supported clock rate. For this study, our hardware SHA256 module took a straightforward, performance-oriented approach towards usage of sequential and combinatorial resources. Different approaches to the hash implementation could have significant effects on resource utilization when mapped into an FPGA.



(a) Lenovo ThinkCentre, Intel i5-8500, 16GB RAM



(b) GCP instance, Intel Xeon @ 2.20GHz, 7.68GB RAM

Fig. 7. ×3 Signature Verification Time for virtual secure boot (ms)

5.4 Virtual Secure Boot Performance

To study Virtual secure boot performance, we used OQS OpenSSL [57] and SHIM code in [59]. QEMU was running on a Lenovo ThinkCentre M920 with an Intel Core i5-8500, 16GB RAM and a 7200rpm spinning hard drive. We first evaluated booting an Ubuntu virtual machine. The Ubuntu guest host was tested with two and four virtual cores without significant difference because UEFI binaries are not optimized for multi-core use. We measured the time from the invocation of QEMU-KVM to the Ubuntu login screen. The average boot time was 14.75s with a standard deviation of 0.16s. This time includes three separate verifications of the three EFI binaries in the UEFI boot process, all of which used RSA2048. The three RSA signature verifications account for a minuscule fraction (0.04ms measured with OpenSSL 1.1.1) of the total boot time. There would be a noticeable speed gain if the tests were running on a solid state drive, but we expect the total boot time to still take several seconds.

Subsequently, to assess the impact of HBS signatures in our Lenovo testbed, we measured the total verification time (three signatures) for the parameter sets under study with multi-threading disabled. We used OQS OpenSSL [57] based on OpenSSL 1.1.1c. The average times for three signature verifications for RSA2048 and the HBS parameters are shown in Fig. 7a. Fig. 7b shows the total HBS verification times in the Google Cloud instance. We can see that the times across our entire set of HBS parameters are longer than for RSA2048, but they all remain well below 21ms. Thus, in a virtual UEFI boot time of several seconds, using post-quantum HBS signatures instead of conventional ones would be of immaterial impact even for much slower verifiers.

6 DISCUSSION

Section 5 showed that HBS are heavier than RSA, but their performance would have minimal impact on today's secure boot technologies. Even though SPHINCS$^+$ is probably the better option because it does not require state management, LMS would be the preferred choice for constrained verifiers where memory and performance are important concerns. Signers with common hardware could sign in less than one second for either scheme. Verification would take just a few milliseconds, which is fast for secure boot and software signing uses. As we explained, a secure boot chain involves two or three signature verifications before the OS is booted. Three HBS verifications will still take minuscule time compared to the total boot time.

Regarding HBS hierarchies, we envision that two or three-level tree architectures would be more practical for most scenarios as they offer a balance between signature generation performance and size. The root of each tree becomes the PK, KEK, or Firmware Update Key included in the UEFI db/dbx databases. As discussed in Section 4, key revocation takes place by placing a HBS public key or signature in the dbx database by signing a database update message with corresponding the KEK tree.

The height of the trees (10, 15, or 20) at each level will depend on the use-case. In terms of Winternitz parameters, we expect W=8 (LMS) and w=256 (SPHINCS$^+$) to be the best options of choice (Table 1). We consider that 96-bit PQ security for SPHINCS$^+$ will suffice for a very long time. A single HSS tree with `HSS192L3oW8` or SPHINCS$^+$ with `SPX192H35w256` are expected to be used in limited scenarios because of revocation challenges. Adopters could of course make different signature, signing and verification performance trade-off decisions and pick different parameters.

One more consideration for HBS signers are fault injection attacks. Benign or malicious fault injections were introduced in [13, 21] which focus on stateless SPHINCS$^+$, but can be extended to HSS and XMSS^{MT}. A fault injection attack can take place because a lower level subtree of an HBS hypertree is usually regenerated to create new signatures. A malicious attacker could take advantage of that at the signer and force a fault in the subtree root that gets signed by the same OTS. That can ultimately lead to a signature forgery with higher probability or full OTS private key recovery if enough faulty signatures were generated. In a software signing scenario, the signer is usually assumed to be trustworthy and fault injections are unlikely. Additionally, signers in these use-cases usually leverage hardware security modules (HSMs) that are assumed fault-resilient. Regardless of their likelihood, such attacks can be prevented. [51] proposed Recomputing with Swapped Nodes (RESN), which recomputes subtrees with swapped nodes and uses an enhanced hash function to protect against faults. Such protections could work with stateful schemes like LMS, but would be more challenging for SPHINCS$^+$ [21]. [21] also proposed caching OTS calculated values and making sure new computations do not deviate. Additonally, it proposed recomputing the OTS in different hardware modules, which would make it impossible to introduce the same faults. That way, a majority of the computed values could be picked for the signature. We consider fault injections to be unlikely for UEFI Secure Boot and their prevention to be relatively straightforward at the signer.

On the other hand, migrating secure boot and image signing to post-quantum HBS comes with a set of **considerations**:

(i) State management is a valid concern raised in various occasions [44, 67]. [48] describes some techniques to protect state that include bulk key and state generation and volatile and non-volatile memory storage. Additionally, NIST published a Special Publication [15] to describe the characteristics of use-cases for stateful HBS. Software signing is believed to be one of these use-cases. The document also makes recommendations for protecting state, such as using hardware security modules to store and update the state before releasing a signature. Even though state can be properly managed for secure boot use-cases, it would not be trivial for crypto policy authors and testers to ensure that the right precautions are taken by a signer. NIST may add stateful HBS in their list of FIPS-approved algorithms for certain uses, but it is not clear how testing labs would verify that state is properly protected. If a testing lab cannot verify proper state management, then another concern could be raised about how a verifier could trust that none of the signatures used the same state. One way to confirm this would be to have a vendor publish all generated signatures so that third parties can confirm state was not re-used, but this is a non-trivial task. Thus, the question of confirming that a signer has taken proper measures to ensure state is not re-used remains open.

(ii) A smooth transition to using PQ signatures is also an important consideration. A vendor that switches to a new signature scheme will still have a lot of deployed devices that use classical RSA signatures. Thankfully, the signing formats used to sign EFI binaries, PKCS#7 [35] and Authenticode Portable Executable (PE), allow for the inclusion of multiple signatures.

PKCS#7 includes the signatures in an ASN.1 type `SignedData`:

```
SignedData ::= SEQUENCE {
  version Version,
    digestAlgorithms DigestAlgorithmIdentifiers,
    contentInfo ContentInfo,
    certificates [0] IMPLICIT ExtendedCertificatesAndCertificates
         OPTIONAL,
    crls [1] IMPLICIT CertificateRevocationLists OPTIONAL,
    signerInfos SignerInfos }

  DigestAlgorithmIdentifiers ::= SET OF DigestAlgorithmIdentifier

  SignerInfos ::= SET OF SignerInfo
```

The `digestAlgorithms` is a collection of message-digest algorithm identifiers under which the content is digested for some signer. The `certificates` includes certificate chains to verify signer information. The `contentInfo` is the content that is signed. There can be many signers (`SignerInfos`) in the `SignedData`. The ASN.1 type `SignerInfo` is defined as

```
  SignerInfo ::= SEQUENCE {
    version Version,
    issuerAndSerialNumber IssuerAndSerialNumber,
    digestAlgorithm DigestAlgorithmIdentifier,
    authenticatedAttributes [0] IMPLICIT Attributes OPTIONAL,
    digestEncryptionAlgorithm
      DigestEncryptionAlgorithmIdentifier,
    encryptedDigest EncryptedDigest,
    unauthenticatedAttributes [1] IMPLICIT Attributes OPTIONAL }
```

The `digestEncryptionAlgorithm` and `encryptedDigest` include the digest and signature identifier and the actual signature. RFC4853 [26] clarifies how verification takes place when there are multiple signatures in the `SignedData`. It considers that the content is verifiably signed by a given signer, if any of the digital signatures from that signer are valid.

On the other hand, the Windows Portable Executable (PE) format used to sign EFI binaries allows for multiple signatures without including multiple `SignerInfos` in the PKCS#7 `SignedData`. The PE section containing the certificate is omitted from the hash calculation of the image and the signature is embedded in the PE file in a location specified by the `Certificate Table` entry in the Optional Header Data Directories. The signature stored in the location is in a `bCertificate` binary array which includes the PKCS#7 `SignedData`. Multiple signatures can be embedded by including multiple `Certificate Table` entries that point to different `bCertificates`. It is common for PE images to have dual/multiple signatures, a feature that is supported by many code signing vendors. Micosoft's Authenticode code-signing technology and Tianocore, the UEFI open-source implementation, support verifying multiple PE signatures as well. We also confirmed that commonly used Linux EFI signing tools like `sbverify` support dual signing and verification of EFI executables.

Thus, to ensure a smooth PQ transition, signers that add HBS signature support could include these signatures along with classical RSA ones in their PE files. A verifier that does not support HBS would still verify the RSA signature, and HBS-capable verifiers could verify the PQ signature. Although the industry has made sure such transition methodologies would work, vendors would need be careful and confirm that existing verifiers will be able to verify the dual signature and proceed with booting.

The post-quantum part of this dual signature would not be FIPS-approved unless HBS were approved by NIST. According to NIST's recent announcement [52], this dual signature approach will still be considered FIPS-140 approved if the classical part of the signature is FIPS-approved and gets verified. Thus, new capabilities could be introduced in the PQ verifier to apply an `AND` operation for verification of the classical and HBS signature. The classical and PQ keys used to generate such signatures would in this case be tied to the dual signature function. It would be up to the signer to decide the signing policy, but in most cases it would seem natural to consider the two keys bound and used together to sign software; using only one of these keys could be forbidden. Such a policy would require changes in the verifier's trust-store / UEFI db to bind these public keys together so they can only be used in conjunction and not independently.

(iii) As mentioned, FIPS approval of HBS is an open question. The future will show if HBS signatures (stateful or stateless) will be added to the FIPS-approved list. Even if these algorithms were FIPS-approved, we will need to see if the new SPHINCS$^+$ parameters introduced in this work will be approved. We introduced new parameters because we found that the NIST SPHINCS$^+$ submitted parameter sets would perform ~ ×2-4 times slower verification with ~5-25KB bigger signatures.

(iv) Tianocore/EDK II (the codebase behind UEFI) and the SHIM use OpenSSL for all RSA and X.509/PKCS#7-related cryptographic operations. The current state of PQ algorithm support is dependent mainly on the OQS OpenSSL [57] project, which relies on liboqs [56] for the implementation of various PQ algorithms. The main focus of the OQS OpenSSL project has been TLS, leaving other facets of OpenSSL such as PCKS#7 relatively untouched. HBS signatures do not operate in a classical digest-and-sign fashion where a message is initially hashed and then the digest is signed; HBS signatures include the message digest as part of the scheme itself. EdDSA [34] is an example of a classical signature scheme that operates in this fashion. At the time of this writing, new PKCS#7 / CMS algorithm identifiers were standardized in the IETF [27] only for a few of the proposed HBS parameters which would be necessary for HBS signatures to be included in software images. Additionally, in order for HBS signature verification to work in UEFI Secure Boot, support for one-shot (without digest) signature algorithms in OpenSSL's PCKS#7 stack would need to be implemented. At the time of this writing, OpenSSL had no plans for adding one-shot signature support for PKCS#7 leaving this a challenge for the uptake of PQ algorithms in UEFI Secure Boot [46].

There are eight more PQ signature candidates in NIST's Round 2 which could serve as **alternatives** to HBS. Most of them offer lengthy signatures and public keys at various speeds. The best options in terms of size are Falcon 512 with 0.69KB signatures at 103 bits of PQ security and Dilithium II with ~2KB signatures at 91 PQ bit security. Falcon 1024 at 230-bit PQ security level has 1.33KB signatures and ~1KB public key and Dilithium IV offers 3.37KB signatures and 1.7KB public key at 158-bit PQ security. Their signing performance is a few milliseconds. Verification performance is less than 1ms for both. There are also three multivariate schemes from the NIST PQC standardization effort (Rainbow, GeMSS and LUOV) which offer signatures smaller than 0.25KB with 58, 352 and 11.5KB public keys respectively at the lowest security level. Excessively large public keys would add extra burden on constrained verifiers that would need to store the public keys in their trust-stores which makes multivariate signatures less appealing. Another scheme, qTesla, has a public key of ~15KB and a 2.5KB signature at the lowest security level. The corresponding verification performance is less than 1ms. MQDSS and Picnic also offer small public keys and 21 and 34KB signatures at the lowest security level. Falcon, Dilithium and some of the qTesla, LUOV, MQDSS and Picnic variants could offer PQ signature alternatives to HBS. However, given that HBS are the most mature, conservative, and well-analyzed options, and based on the analysis in this work, we think that HBS is best suited for software signing and secure boot use-cases.

7 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 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, virtual 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. Thanks to Bruno Couillard and Jim Goodman from Crypto4A for the interesting discussions about HBS and their feedback. Quynh Dang kindly caught our inaccurate overstating of the signing performance of our customized parameter sets. 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] M. M. Alam, S. Tajik, F. Ganji, M. Tehranipoor, and D. Forte. 2019. RAM-Jam: Remote Temperature and Voltage Fault Attack on FPGAs using Memory Collisions. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 48–55.

[2] Megumi Ando, Joshua D. Guttman, Alberto R. Papaleo, and John Scire. 2016. Hash-Based TPM Signatures for the Quantum World. In *Applied Cryptography and Network Security*, Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider (Eds.). Springer International Publishing, Cham, 77–94.

[3] Jean-Philippe Aumasson, Daniel J Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, et al. 2019. SPHINCS+ - Submission to the 2nd round of the NIST post-quantum project. https://sphincs.org/data/sphincs+-round2-specification.pdf. Specification document (part of the submission package).

[4] Kanad Basu, Deepraj Soni, Mohammed Nabeel, and Ramesh Karri. 2019. NIST Post-Quantum Cryptography- A Hardware Evaluation Study. Cryptology ePrint Archive, Report 2019/047. https://eprint.iacr.org/2019/047.

[5] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. 2015. SPHINCS: Practical Stateless Hash-Based Signatures. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 368–397. https://doi.org/10.1007/978-3-662-46800-5_15

[6] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. 2017. Transitioning to a quantum-resistant public key infrastructure. In *Proc. 8th International Conference on Post-Quantum Cryptography (PQCrypto) 2017 (LNCS)*, Tanja Lange and Tsuyoshi Takagi (Eds.). Springer. To appear.

[7] Dan Boneh and Shay Gueron. 2017. Surnaming Schemes, Fast Verification, and Applications to SGX Technology. In *Topics in Cryptology – CT-RSA 2017*, Helena Handschuh (Ed.). Springer International Publishing, Cham, 149–164.

[8] Joppe W Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. 2015. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 553–570.

[9] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. 2011. *XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 117–129. https://doi.org/10.1007/978-3-642-25405-5_8

[10] Johannes A Buchmann, Denis Butin, Florian Göpfert, and Albrecht Petzoldt. 2016. Post-quantum cryptography: state of the art. In *The New Codebreakers*. Springer, 88–108.

[11] Matt Campagna. 2019. Hybrid-Key Exchanges as an Interim-to-Permanent Solution to Cryptographic Agility. https://docbox.etsi.org/Workshop/2019/.

[12] Fabio Campos, Tim Kohlstadt, Steffen Reith, and Marc Stöttinger. 2020. LMS vs XMSS: Comparison of Stateful Hash-Based Signature Schemes on ARM Cortex-M4. In *Progress in Cryptology - AFRICACRYPT 2020*, Abderrahmane Nitaj and Amr Youssef (Eds.). Springer International Publishing, Cham, 258–277.

[13] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. 2018. Grafting Trees: A Fault Attack Against the SPHINCS Framework. In *Post-Quantum Cryptography*, Tanja Lange and Rainer Steinwandt (Eds.). Springer International Publishing, Cham, 165–184.

[14] Cisco. 2017. Cisco Secure Boot and Trust Anchor Module Differentiation Solution Overview. https://www.cisco.com/c/en/us/products/collateral/security/cloud-access-security/secure-boot-trust.html.

[15] David Cooper, Daniel Apon, Quynh Dang, Michael Davidson, Morris Dworkin, and Carl Miller. 2020. Recommendation for Stateful Hash-Based Signature Schemes. https://csrc.nist.gov/publications/detail/sp/800-208/final.

[16] Eric Crockett, Christian Paquin, and Douglas Stebila. 2019. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. Cryptology ePrint Archive, Report 2019/858. https://eprint.iacr.org/2019/858.

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

[18] Scott Fluhrer. 2019. LMS Hash Based Signature Open-source Implementation. https://github.com/cisco/hash-sigs.

[19] Scott Fluhrer and Quynh Dang. 2019. *Additional Parameter sets for LMS Hash-Based Signatures*. Internet-Draft draft-fluhrer-lms-more-parm-sets-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-fluhrer-lms-more-parm-sets-00 Work in Progress.

[20] Scott Fluhrer, David McGrew, Panos Kampanakis, and Valery Smyslov. 2019. *Postquantum Preshared Keys for IKEv2*. Internet-Draft draft-ietf-ipsecme-qr-ikev2-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-ipsecme-qr-ikev2-08 Work in Progress.

[21] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. 2018. Practical Fault Injection Attacks on SPHINCS. Cryptology ePrint Archive, Report 2018/674. https://eprint.iacr.org/2018/674.

[22] Santosh Ghosh, Rafael Misoczki, and Manoj R. Sastry. 2019. Lightweight Post-Quantum-Secure Digital Signature Approach for IoT Motes. Cryptology ePrint Archive, Report 2019/122. https://eprint.iacr.org/2019/122.

[23] Google. 2019. Shielded VM Documentation. https://cloud.google.com/security/shielded-cloud/shielded-vm.

[24] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. 2012. Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, Emmanuel Prouff and Patrick Schaumont (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 530–547.

[25] Paul E. Hoffman. 2019. *The Transition from Classical to Post-Quantum Cryptography*. Internet-Draft draft-hoffman-c2pq-05. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-hoffman-c2pq-05 Work in Progress.

[26] Russ Housley. 2007. Cryptographic Message Syntax (CMS) Multiple Signer Clarification. RFC 4853. https://doi.org/10.17487/RFC4853

[27] Russ Housley. 2020. Use of the HSS/LMS Hash-Based Signature Algorithm in the Cryptographic Message Syntax (CMS). RFC 8708. https://doi.org/10.17487/RFC8708

[28] http archive. [n.d.]. Measuring TLS key exchange with post-quantum KEM. https://www.shodan.io/report/mNs9fa3I.

[29] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. 2018. XMSS: eXtended Merkle Signature Scheme. RFC 8391. https://doi.org/10.17487/RFC8391

[30] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. 2013. Forward Secure Signatures on Smart Cards. In *Selected Areas in Cryptography*, Lars R. Knudsen and Huapeng Wu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 66–80.

[31] Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. 2016. From 5-pass MQ-based identification to MQ-based signatures. *IACR Cryptology ePrint Archive* 2016 (2016), 708.

[32] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. 2015. ARMed SPHINCS – Computing a 41KB signature in 16KB of RAM. Cryptology ePrint Archive, Report 2015/1042. https://eprint.iacr.org/2015/1042.

[33] Nisha Jacob, Johann Heyszl, Andreas Zankl, Carsten Rolfes, and Georg Sigl. 2017. How to Break Secure Boot on FPGA SoCs Through Malicious Hardware. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, Wieland Fischer and Naofumi Homma (Eds.). Springer International Publishing, Cham, 425–442.

[34] Simon Josefsson and Ilari Liusvaara. 2017. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032. https://doi.org/10.17487/RFC8032

[35] Burt Kaliski. 1998. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315. https://doi.org/10.17487/RFC2315

[36] Panos Kampanakis. 2019. Slim SPHINCS$^+$ Open-source Implementation. https://github.com/csosto-pk/slim_sphincsplus/tree/master/ref.

[37] Panos Kampanakis, Peter Panburana, Ellie Daw, and Daniel Van Geest. 2018. The Viability of Post-quantum X.509 Certificates. *IACR Cryptology ePrint Archive* 2018 (2018), 63.

[38] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844. https://eprint.iacr.org/2019/844.

[39] Stefan Kölbl, Martin M Lauridsen, Florian Mendel, and Christian Rechberger. 2016. Haraka v2-efficient short-input hashing for post-quantum applications. *IACR Transactions on Symmetric Cryptology* (2016), 1–29.

[40] Vinay B. Y. Kumar, Naina Gupta, Anupam Chattopadhyay, Michael Kaspert, Christoph Krauß, and Ruben Niederhagen. 2020. Post-Quantum Secure Boot. In *Proceedings of the 23rd Conference on Design, Automation and Test in Europe* (Grenoble, France) *(DATE '20)*. EDA Consortium, San Jose, CA, USA, 1582–1585.

[41] Kris Kwiatkowski. 2019. Towards Post-Quantum Cryptography in TLS. https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/

[42] Adam Langley. 2016. CECPQ1 results. https://www.imperialviolet.org/2016/11/28/cecpq1.html

[43] Adam Langley. 2018. CECPQ2. https://www.imperialviolet.org/2018/12/12/cecpq2.html

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

[45] Adam Langley. 2018. Post-quantum confidentiality for TLS. https://www.imperialviolet.org/2018/04/11/pqconftls.html

[46] OpenSSL Maintainers. 2018. Pull request: Make Ed25519/Ed448 usable from the command line. https://github.com/openssl/openssl/pull/5880.

[47] David McGrew, Michael Curcio, and Scott Fluhrer. 2019. Leighton-Micali Hash-Based Signatures. RFC 8554. https://doi.org/10.17487/RFC8554

[48] David McGrew, Panos Kampanakis, Scott Fluhrer, Stefan Gazdag, Dennis Butin, and Johannes Buchmann. 2016. State Management for Hash-Based Signatures. In *Security Standardisation Research (SSR) 2016 - Lecture Notes in Computer Science*, Vol. 10074. Springer. https://doi.org/10.1007/978-3-319-49100-4_11

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

[50] Michele Mosca. 2018. Cybersecurity in an era with quantum computers: will we be ready? *IEEE Security & Privacy* 16, 5 (2018), 38–41.

[51] Mehran Mozaffari-Kermani, Reza Azarderakhsh, and Anita Aghaie. 2016. Fault Detection Architectures for Post-Quantum Cryptographic Stateless Hash-Based Secure Signatures Benchmarked on ASIC. *ACM Trans. Embed. Comput. Syst.* 16, 2, Article 59 (Dec. 2016), 19 pages. https://doi.org/10.1145/2930664

[52] NIST. 2019. Revising FAQ questions on hybrid modes e-mail thread. https://groups.google.com/a/list.nist.gov/forum/?utm_medium=email&utm_source=footer#!msg/pqc-forum/qRP63ucWIgs/rY5Sr_52AAAJ.

[53] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. 2014. Beyond ECDSA and RSA: Lattice-based digital signatures on constrained devices. In *In DAC '14 Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*. 1–6. https://doi.org/10.1145/2593069.2593098

[54] Mike Ounsworth and Massimiliano Pala. 2019. *Composite Keys and Signatures For Use In Internet PKI*. Internet-Draft draft-ounsworth-pq-composite-sigs-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ounsworth-pq-composite-sigs-01 Work in Progress.

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

[56] Open Quantum Safe Project. 2019. liboqs. https://github.com/open-quantum-safe/liboqs. Web page. Accessed 2019-02-08.

[57] Open Quantum Safe Project. 2019. OQS OpenSSL. https://github.com/open-quantum-safe/openssl. Web page. Accessed 2019-02-08.

[58] John Proos and Christof Zalka. 2003. Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves. *Quantum Info. Comput.* 3, 4 (July 2003), 317–344. http://dl.acm.org/citation.cfm?id=2011528.2011531

[59] RedHat. 2019. SHIM (in github). https://github.com/rhboot/shim.

[60] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. 2008. Fast Hash-Based Signatures on Constrained Devices. In *Smart Card Research and Advanced Applications*, Gilles Grimaud and François-Xavier Standaert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–117.

[61] Crystal Roma, Chi-En Amy Tai, and M Anwar Hasan. 2019. Energy Consumption of Round 2 Submissions for NIST PQC Standards. *Second PQC Standardization Conference* (Aug 2019).

[62] Johanna Sepúlveda, Shiyang Liu, and Jose M Bermudo Mera. 2019. Post-quantum enabled cyber physical systems. *IEEE Embedded Systems Letters* (2019).

[63] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. on Computing* 26, 5 (1997), 1484–1509.

[64] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. 2020. Post-Quantum Authentication in TLS 1.3: A Performance Study. In *NDSS*.

[65] Deepraj Soni, Kanad Basu, Mohammed Nabeel, and Ramesh Karri1. 2019. A Hardware Evaluation Study of NIST Post-Quantum Cryptographic Signature schemes. *Second PQC Standardization Conference* (Aug 2019).

[66] SPHINCS$^+$ team. 2019. SPHINCS$^+$ Open-source Implementation. https://github.com/sphincs/sphincsplus.

[67] stateful-hbs-comments-nist 2019. Stateful Hash-Based Signatures - Public Comments on Misuse Resistance. https://csrc.nist.gov/CSRC/media/Projects/Stateful-Hash-Based-Signatures/documents/stateful-HBS-misuse-resistance-public-comments-April2019.pdf.

[68] Douglas Steblia, Scott Fluhrer, and Shay Gueron. 2019. *Design issues for hybrid key exchange in TLS 1.3*. Internet-Draft draft-stebila-tls-hybrid-design-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-stebila-tls-hybrid-design-01 Work in Progress.

[69] Cisco Systems. 2010. Digitally Signed Cisco Software. https://www.cisco.com/c/en/us/td/docs/ios/fundamentals/configuration/guide/TIPs_Conversion/lmsi_15_1s_book/cf_dgtly_sgnd_sw.html.

[70] C. Tjhai, M. Tomlinson, grbartle@cisco.com, Scott Fluhrer, Daniel Van Geest, Oscar Garcia-Morchon, and Valery Smyslov. 2019. *Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2).* Internet-Draft draft-tjhai-ipsecme-hybrid-qske-ikev2-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-tjhai-ipsecme-hybrid-qske-ikev2-04 Work in Progress.

[71] VMware. 2019. Enable or Disable UEFI Secure Boot for a Virtual Machine. https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.security.doc/GUID-898217D4-689D-4EB5-866C-888353FE241C.html.

[72] Xilinx. 2019. Vitis Security Library. https://xilinx.github.io/Vitis_Libraries/security/guide_L1/internals.html.

[73] Bo-Yin Yang, Chen-Mou Cheng, Bor-Rong Chen, and Jiun-Ming Chen. 2006. Implementing Minimized Multivariate PKC on Low-resource Embedded Systems. In *Proceedings of the Third International Conference on Security in Pervasive Computing* (York, UK) *(SPC'06)*. Springer-Verlag, Berlin, Heidelberg, 73–88. http://dx.doi.org/10.1007/11734666_7

[74] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. 2017. A Post-Quantum Digital Signature Scheme Based on Supersingular Isogenies. Cryptology ePrint Archive, Report 2017/186. http://eprint.iacr.org/2017/186.