PRUNE-HORST

v1, September 13, 2017

Inventor, submitter, main contact:

Jean-Philippe Aumasson, Kudelski Security

Route de Genève 22, 1033 Cheseaux-sur-Lausanne, Switzerland

jeanphilippe.aumasson@gmail.com, +41 79 726 05 08

Inventor, submitter, backup contact:

Guillaume Endignoux, TBD

TBD

guillaume.endignoux@gmail.com, TBD

Contents

1 Introduction

PRUNE-HORST is a hash-based *few-time signature scheme*, meaning that it's only secure for a limited number of messages—a dozen, hundreds, or thousands, depending on the version.

This limitation allows PRUNE-HORST to be relatively simple and efficient. It's also *stateless*, as required by NIST.

PRUNE-HORST is based on Hash to Obtain Random Subset, or HORS, a stateless few-time signature scheme proposed by Reyzin and Reyzin in 2002 [5]. Like HORST—a variant of HORS introduced for the SPHINCS construction by Bernstein et al. in 2015 [1]—PRUNE-HORST reduces HORS' public key size thanks to a Merkle tree. The prefix PRUNE summarizes additional tricks to optimize HORS' security and efficiency:

- A *pseudo-random* generator is used rather than just a hash function in order to obtain a random subset of a set of integers.
- The pseudo-random generator allows to ensure that the subset elements are *unique*, which increases the security of the scheme [2, §4].
- The Merkle tree is *pruned* in order to trade signature size for public key size, by eliminating redundant key nodes, similarly to what SPHINCS does to minimize the signature size.

1.1 Advantages and Limitations

Advantages:

- **Simple**: PRUNE-HORST is much simpler than hash-based signature schemes such as SPHINCS or XMSS, and than signatures schemes from other post-quantum families. The logic of the scheme fits in about 200 lines of C code.
- **High-assurance**: Security essentially depends on the collision resistance of hash functions, an assumption unlikely to fail for the proposed functions. PRUNE-HORST also leverages our detailed analysis and bounds detailed in [2, §4].
- **Speed/security trade-offs** are easily done by varying the parameters T and K. For a given choice of T and K, an additional parameter C allows to reduce the signature size at the cost of a larger public key.
- **Forgeries detection**: If the limit of messages to be signed is exceeded, thereby making forgeries easier, the legitimate signer can detect such forgeries. Obviously, a signature forged by stealing the secret key couldn't be detected.

- **Almost optimal**: the signature size can be further reduced by eliminating redundancies, as is our recent Octopus technique [2, §5], however by default PRUNE-HORST avoids the extra complexity and uses suboptimal-length signatures.

Limitations:

- **Few-time**: Only a limited number of messages can be signed while retaining the highest security guarantees: about 100 or about 1000, depending on the instance. If more messages than the limit are signed, then security slowly degrades.
- **Signature size**: Signatures aren't small, but they can be made smaller by using larger public keys. This trade-off has no security impact, and makes signing faster as the public key size grows.

1.2 Motivations

A few-time signature scheme allowing only for hundreds or thousands of distinct messages to be signed is sufficient in a number of major applications, such as: root CAs signing intermediate CA certificates; signing of firmware or bootloader images for long-lived devices; protocols using ephemeral signing keys (as mpOTR did); identify management for device provisioning, for example in messaging applications.

Another class of application would use few-time signatures to enforce a bound in the number of authorized signatures. For example, a bank might issue a digital checkbook limited to 100 transactions. Would a client issues more than the authorized 100 signatures, they would reduce their own security and thereby allow an attacker to forge signatures on their behalf.

We chose HORS(T) as a basis because it's the most efficient few-time signature construction, and because of its simplicity.

Primitives in PRUNE-HORST are SHA-256, AES-256-CTR, and Haraka v2 [4]. The latter is not a NIST standard, because we needed a fast, short-input hash function and NIST doesn't provide such a primitive. Haraka v2 hashes 32- or 64-byte input values and produces a 32-byte hash value. It is based on the AES round function and optimized implementations use AES-NIs. We chose Haraka v2 with 6 rounds, rather than the default 5 round, for a greater assurance against collisions.

2 Specification

Below "log" means "$\log_2$", "$\|$" is string concatenation, all hash values are 32-byte. The type of tree used is a complete binary hash tree (i.e., a complete Merkle tree).

2.1 Primitives

PRUNE-HORST requires three hash functions, which all return 32-byte hash values:

- Hash takes input values of any length. We use **SHA-256** as Hash.
- Hash32 takes input values of 32 bytes. We use **Haraka-256** with 6 rounds as Hash32.
- Hash64 takes input values of 64 bytes. We use **Haraka-512** with 6 rounds as Hash64.

The Haraka versions are the v2 [4].

PRUNE-HORST also uses **AES-256-CTR** as a deterministic generator of pseudorandom bytes.

2.2 Parameters

PRUNE-HORST signature schemes are defined by three parameters:

- The **set size** T, which must be a power of two.
- The **subset size** K, which must be lower than T.
- The **number of subtrees** C, which must be a power of two and strictly lower than T.

These parameters determine:

- The **public key size**, equal to $32C$ bytes.
- The **signature size**, equal to $32 \times (K + K(\log T - \log C) + 1)$ bytes.

T and K together determine the security and signature length, while C determines the public key and signature lengths but not security.

The choice of T and K depends on the target security security level and on the maximum number of messages to be signed. The security level for a given (T, K) is discussed in §3.2.

The choice of C is mostly a trade-off between the public key size (higher with a higher C) and the signature size (smaller with a higher C). The optimal value of C for a given (T, K) is discussed in §4.2.

2.3 Key Generation

A secret key sk is a random 64-byte value, viewed as two 32-byte values $\mathsf{sk}_1 \| \mathsf{sk}_2 = \mathsf{sk}$.

A public key pk is derived from a secret key's sk_1 as follows:

1. Expand sk into T 32-byte *subkeys* $\mathsf{ek}_0, \mathsf{ek}_1, \ldots, \mathsf{ek}_{T-1}$, by taking the first $32T$ bytes generated with AES-256-CTR keyed with sk_1 and with a 16-byte counter initialized to $0000 \ldots 0000$.
2. Hash each subkey to obtain T values $L_i = \mathrm{Hash32}(\mathsf{ek}_i)$, $i = 0, \ldots, T-1$, which will be the leaves of the tree.
3. Compute $\mathsf{pk} = \mathsf{pk}_0, \cdots, \mathsf{pk}_C$ as the C nodes on level $\log C$ of the binary hash tree with leaves $L_0, \ldots, L_{T-1}$, computing $\mathrm{Hash64}(L_0 \| L_1)$, $\mathrm{Hash64}(L_2 \| L_3)$, and so on.

Step 1 requires $32T$ bytes from AES-256-CTR, step 2 requires T calls to Hash32, and step 3 requires $T - C$ calls to Hash64.

The value $\log C$ can be seen as the "cut-off" level of the tree.

The sk_2 component of the secret key is not used in key generation, but only when signing a message.

2.4 Subset Generation

The core component of PRUNE-HORST is its *subset generation function*, which picks K distinct values in $0, 1, \ldots, T-1$ given a 32-byte *signature seed* S and $H = \mathrm{Hash}(M)$, the 32-byte hash of the message.

Subset generation works like this:

- Compute the *subset seed* $D = \mathrm{Hash64}(S \| H)$.

- Generate pseudorandom bits from AES-256-CTR keyed with D and with a 16-byte counter initialized to `0000...0000`.
- Parse the pseudorandom stream as a sequence of 4-byte big-endian unsigned integers $N_0, N_1, N_2, \ldots$, where each integer is reduced modulo T (since T is a power of two, the distribution remains uniform)
- Determine the K distinct values $V_0, \ldots, V_{K-1}$ as follows: $V_0 = N_0$; V_1 is the first $N_i, i > 0$ such that $N_i \neq V_0$; third value is the first subsequent N_i distinct from both V_0 and V_1, and so on.

2.5 Signing a Message

Given a secret key $\mathsf{sk} = \mathsf{sk}_1 \| \mathsf{sk}_2$ and a message M, PRUNE-HORST computes a signature sig as follows:

1. Generate a subset of indices $V_0, V_1, \ldots, V_{K-1}$ as per §2.4, using $S = \text{Hash64}(\mathsf{sk}_2 \| \text{Hash}(M))$ as signature seed. Initialize the signature with the 32-byte S.
2. Expand sk_1 into T 32-byte *subkeys* $\mathsf{ek}_0, \mathsf{ek}_1, \ldots, \mathsf{ek}_{T-1}$ in the same manner as for public key generation.
3. Append K subkeys to the signature in order to have $\mathsf{sig} = S \| \mathsf{ek}_{V_0} \| \mathsf{ek}_{V_1} \| \cdots \| \mathsf{ek}_{V_{K-1}}$.
4. Compute the binary hash tree up to level $1 + \log C$ like for public key generation, but recording the *authentication paths* for each of the K leaves from the subset, as follows:
   (a) Hash each subkey to obtain leaves hashes $L_i = \text{Hash}(\mathsf{ek}_i)$, $i = 0, \ldots, T-1$.
   (b) Append to S the leaf value to be hashed together with L_{V_0}, then the leaf value to be hashed together with L_{V_1}, and so on until $L_{V_{K-1}}$.
   (c) Compute the parent node of each consecutive pair of leaves by doing $\text{Hash64}(L_0 \| L_1)$, $\text{Hash64}(L_2 \| L_3)$, and so on. Append to S the sibling hash value for each of the K parents of a subkey, from V_0 to V_{K-1}; K hashes are thus appended to S.
   (d) Iterate step (c) for upper levels of the tree, appending sibling nodes to S to form K authentication paths, until level $1 + \log C$ (that is, one level deeper than the level of the subtrees' roots in the public key).

The signature sig eventually contains, in this order: the signature seed, the K hashes from step 3, and the $K(\log T - \log C)$ hashes from step 4.

Step 1 requires one call to Hash, two calls to Hash64, and at least $4K$ bytes from AES-256-CTR; step 2 requires $32T$ bytes from AES-256-CTR; step 4.a requires T calls to Hash32, then iterating step 4.b-d requires $T - C$ calls to Hash64.

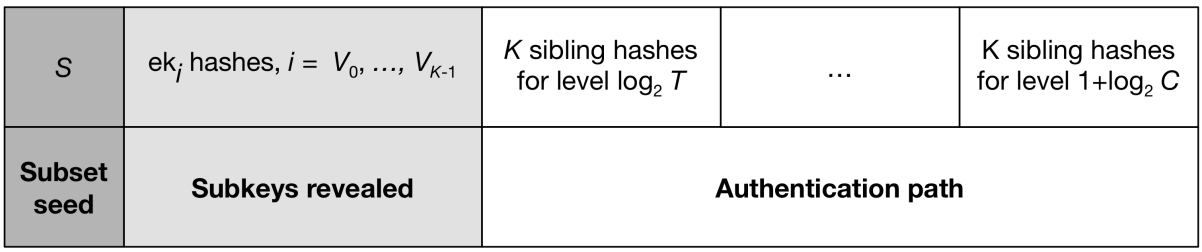

Figure 2.1: Content of a signature.

2.6 Verifying a Signature

Given a public key pk and a message M, PRUNE-HORST verifies a signature sig as follows:

1. Generate a subset of indices $V_0, V_1, \ldots, V_{K-1}$ as per §2.4, using the first 32 bytes of sig as a signature seed.
2. For each of the K indices:
   (a) Retrieve the subkey from the signature, where the subkey of index V_i is the ith 32-byte string in the signature.
   (b) Using the authentication path in the signature (every other K hash after the signature seed), compute parent nodes up to level $\log C$.
   (c) Check if the subtree root found at level $\log C$ is the same as in the public key.
3. Verification of sig is successful only and only if all K verifications succeeded.

Step 1 requires one call to Hash and one call to Hash64, and at least $4K$ bytes from AES-256-CTR; step 2 requires K calls to Hash32 and $K(\log T - \log C)$ calls to Hash64.

2.7 Proposed Instances

See Table 2.1.

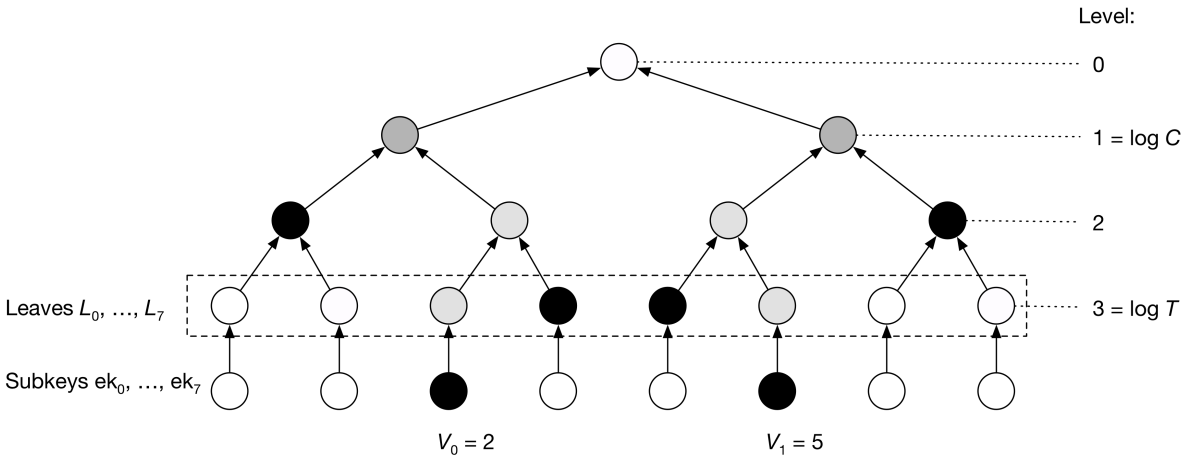

Figure 2.2: Binary hash tree of a signature, with a set of $T = 8$ hashes (thus a tree depth of $\log T = 3$), a subset of $K = 2$ hashes, $C = 2$ subtrees (thus their two respective roots in the public key), and subset of indexes $V_0 = 2, V_1 = 5$. The nodes in black are part of the signature, the nodes in pale grey are computed during the verification, and the nodes in dark grey are part of the public key.

ID	Parameters			Messages	Byte length		
	T	K	C	limit	Sig	Pub	Priv
S	2^{17}	54	2^6	100	20 768	2048	64
M	2^{18}	62	2^7	300	23 840	4096	64
L	2^{19}	64	2^7	600	26 656	4096	64

Table 2.1: Proposed PRUNE-HORST instances, expected to provide 128-bit pre- and post-quantum security.

3 Security

3.1 Expected Strength

Non-repudiation and unforgeability can be broken by finding any SHA-256 collision, since messages are SHA-256-hashed before being signed. PRUNE-HORST may also be broken if Hash and Hash32 are not collision resistant. This puts PRUNE-HORST in NIST's security category 2: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 256-bit hash function (e.g. SHA256/ SHA3-256)".

Attacks that would break PRUNE-HORST by searching for a subset of known subkeys fall into NIST's category 5, however: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES 256)".

Below we detail the later type of attacks.

3.2 Known Attacks

The main attack on PRUNE-HORST works as follows:

1. Observe N signatures in order to collect at most $N \times K$ subkeys, and on average $(T^{NK} - (T-1)^{NK})/T^{NK-1}$. Call the corresponding subset of indices $\mathcal{I}$.
2. Search for a message M and a signature seed S such that the K indices obtained by subset generation belong to $\mathcal{I}$, which happens with probability $(|\mathcal{I}|/T)^K$.
3. Once a pair (M, S) is found at step 2, construct the signature using the subkeys collected in step 1. The signature will be valid even though the signature seed was not derived from the secret key.

Classically, the expected number of attempts before forging a signature is therefore

$$\left(\frac{T}{|\mathcal{I}|}\right)^K = \frac{T^{NK^2}}{(T^{NK} - (T-1)^{NK})^K}$$

where each attempt involves approximately $K/2$ AES calls in order to generate K indices (since a 16-byte blocks provides at most 4 indices).

The classical security level against this attack is therefore, in bits:

$$NK^2 \log T - K \log(T^{NK} - (T-1)^{NK}) + \log K - 2$$

where N is the number of messages signed.

A quantum attack using Grover's algorithm would require quadratically fewer attempts (approximately, and using a quantum circuit), thus a lower bound is

$$(NK^2 \log T - K \log(T^{NK} - (T-1)^{NK}))/2 + \log K - 2$$

For PRUNE-HORST instances in Table 2.1, with N set to the defined message limit, instances S, M, L respectively have

- Classical security levels 243.76, 253.82, 248.72
- Quantum security bounds 123.86, 128.79, 126.36

Figure 3.1 shows the degradation of security as the number of signatures grows.

3.2.1 Other Attacks

Fault attacks or side-channel attacks could recover sk_1 in the subset generation if the AES implementation is not protected. Likewise sk_2 could be recovered during the computation of $S = \text{Hash64}(\mathsf{sk}_2 \| \text{Hash}(M))$ if Hash64 is not protected.

Fault attacks could also force subset generation to return indices of specific subkeys.

Timing attacks are unlikely if the implementations of functions processing sk_1 and sk_2 are time-constant (AES, Haraka-v2).

3.3 Forgery Detection

In the attack described in §3.2 the attacker chooses the signature seed S but can't pick $S = \text{Hash64}(\mathsf{sk}_2 \| \text{Hash}(M))$, as prescribed by the signing procedure. Yet the signature will be verified, and only the legitimate signer can determine that S wasn't properly chosen using sk_2. EdDSA (and Ed25519) have a similar property of invalid-yet-valid signatures.

This property allows a signer to detect signatures forged after exploiting a (too) high number of issued signatures, compared with legitimately computed signatures. But it can't be used to prove the existence of forgeries to a third-party, since the signer could have themselves used an invalid signature seed in order to break non-repudiation. If the signer is trusted, however, then they can reveal sk_2 privately in order to show evidence of a forgery.

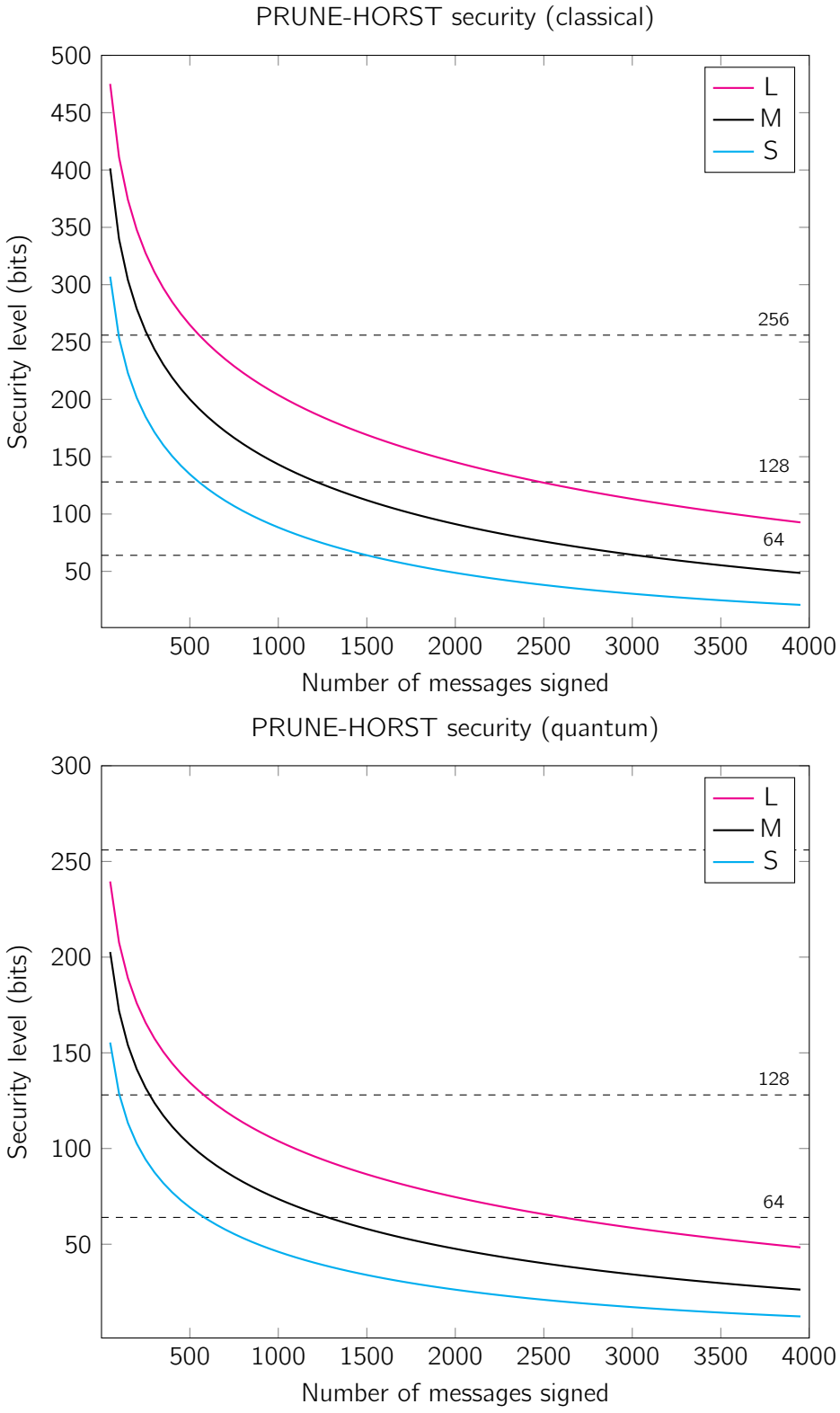

Figure 3.1: Security level in bits of PRUNE-HORST instances against classical and quantum attacks exploiting the subkeys revealed in previous signatures. Quantum attacks use Grover search to get a quadratic speed-up over classical attacks.

4 Performance

4.1 Complexity

Most of operations in PRUNE-HORST are AES rounds, through AES-256-CTR, Haraka-256, or Haraka-512. AES-256-CTR does 14 AES rounds per 16-byte block generated. Haraka-256 does 24 AES rounds per 32-byte value hashed, and Haraka-512 does 48 rounds per 64-byte value hashed.

Table 4.1 gives the count of these operations for each PRUNE-HORST instance, where

- Key generation needs $100T - 48C$ AES rounds
- Signature needs $100T - 48C + 7K + 96$ AES rounds, where we count $8K$ bytes for subset generation, as in our reference code (thus $K/2$ AES calls, or $7K$ rounds)
- Verification needs $K(24 + 48(\log T - \log C) + 7K + 48$ AES rounds

We ignore the cost of hashing the message with Hash, and we also ignore the potential speed-up of pipelined or parallel computation of AES rounds.

Verification is orders of magnitude faster than signature generation, which is ideal for the PRUNE-HORST use cases: typically few signatures would be generated on powerful machines with no speed constraint, whereas verification may be performed multiple times on less powerful machines (for example, for firmware signatures or CA-issued certificates).

4.2 Choosing a Number of Subtrees

The number of substrees C does not affect the security of the scheme, but allows to reduce the signature size by increasing the public key size: instead of including only the root of the tree (case $C = 1$), the public key can include all the roots of subtrees at a lower level.

A larger C also makes signature and verification slightly faster.

For the PRUNE-HORST instances proposed, we choose the value of C that minimizes the total length signature + public key. Other values of C may be used to reduce the signature size or speed-up computations.

Figure 4.1 shows the different trade-offs for the proposed PRUNE-HORST instances.

Operation	ID	AES-256	Haraka-256	Haraka-512	AES rounds
Key generation	S	262 144	131 072	131 008	13 104 128
	M	524 288	262 144	262 016	26 208 256
	L	1 048 576	524 288	524 160	52 422 656
Signature	S	262 171	131 072	131 010	13 104 602
	M	524 319	262 144	262 018	26 208 786
	L	1 048 608	524 288	524 162	52 423 200
Verification	S	27	54	595	30 234
	M	31	62	683	34 706
	L	32	64	769	38 896

Table 4.1: Complexity of PRUNE-HORST instances, where AES-256 does 14 rounds per block, Haraka-256 does 24 AES rounds, and Haraka-512 does 48. The count of AES-256 blocks for subset generation is an upper bound.

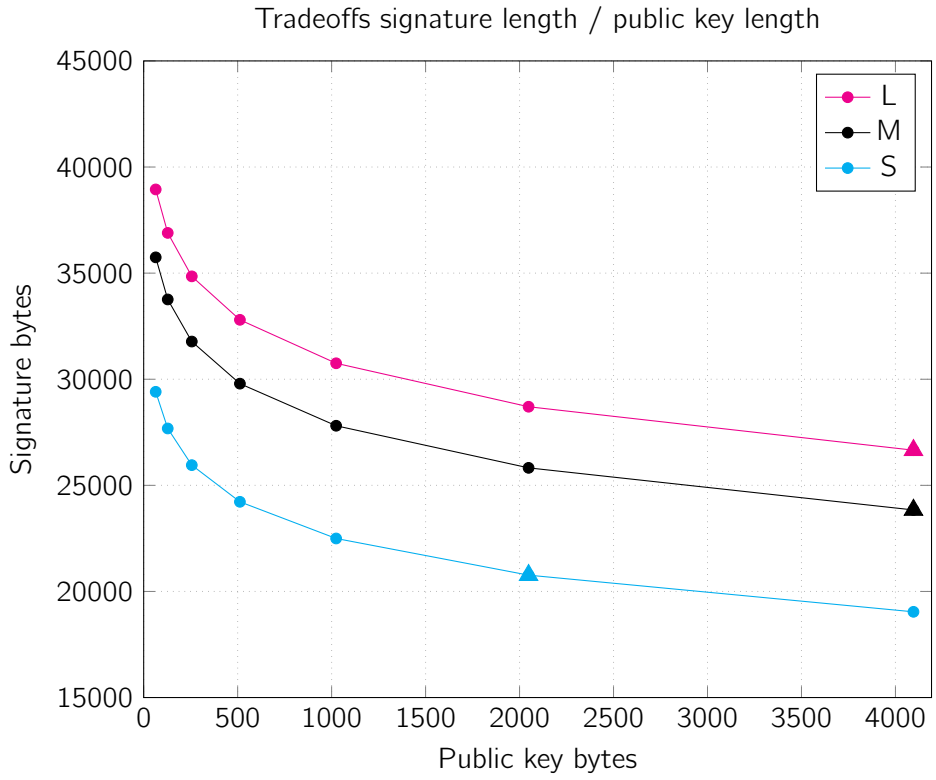

Figure 4.1: Trade-offs between signature length and public-key length, for C in $\{2^1, 2^2, \ldots, 2^7\}$. The triangles indicate the optimal trade-offs, chosen for the instances proposed.

4.3 Software Performance

The AES rounds that constitute most of PRUNE-HORST computations and the largest part of the execution time can be implemented with native AES instructions (AES-NI).

On Skylake CPUs, the AES round instruction AESENC has a latency of four cycles and a reciprocal throughput of one. In PRUNE-HORST, many AES rounds can be pipelined in order to maximize the effective throughput and return one AESENC result per cycle: Haraka-512 computes up to four independent AES rounds, and rounds within four AES-256-CTR instances can be interleaved.

These independent AES round instances can't really be parallelized on current microarchitectures that have a single AES unit. But AMD's new Ryzen CPUs have two AES units, and Intel is expect to follow in future microarchitectures versions and include two AES units as well.

Our optimized implementation thus includes 4-way and 8-way interleaved versions of Haraka for computing the trees and its leaves[1], as well as 4-way interleaved AES-256-CTR.

Signature verification requires only negligible memory: our implementation only allocates $140 + K$ bytes on the stack, which is probably suboptimal.

Our implementations of key generation and signing, however, allocate $32T$ heap bytes to store the subkeys and compute the tree. This is respectively 4MiB, 8MiB, and 16MiB for instances S, M, and L. But key generation and signing can be implemented to use much less memory, for example using techniques described in [3].

Below we report example results from running our benchmark program (`make bench`) on a Intel Core i7-6700 CPU @ 3.40GHz (Turbo Boost disabled), where the wall time is given in microseconds, and `crypto_sign_cached` implements the trick described in §§4.5.1:

Version S ($T = 2^{17}, C = 2^6, K = 54$):

```
# crypto_sign_keypair
median cycles count:     14054574
average wall time:       4124.561 usec

# crypto_sign
median cycles count:     19551800
average wall time:       5747.488 usec

# crypto_sign_cached
median cycles count:     10418326
average wall time:       3063.229 usec

# crypto_sign_open
median cycles count:     72908
average wall time:       21.443 usec
```

[1] Based on Haraka's authors code at https://github.com/kste/haraka/, with a few optimizations and bug fixes.

Version M ($T = 2^{18}, C = 2^7, K = 62$):

```
# crypto_sign_keypair
median cycles count:     28193440
average wall time:       8276.869 usec

# crypto_sign
median cycles count:     40179136
average wall time:       11813.293 usec

# crypto_sign_cached
median cycles count:     20982546
average wall time:       6169.703 usec

# crypto_sign_open
median cycles count:     83650
average wall time:       24.598 usec
```

Version L ($T = 2^{19}, C = 2^7, K = 64$):

```
# crypto_sign_keypair
median cycles count:     58782818
average wall time:       17234.963 usec

# crypto_sign
median cycles count:     80689918
average wall time:       23750.260 usec

# crypto_sign_cached
median cycles count:     42442734
average wall time:       12473.613 usec

# crypto_sign_open
median cycles count:     93702
average wall time:       27.562 usec
```

Note the low verification time for all instances (`crypto_sign_open`).

4.4 Hardware Performance

Hardware architecture may trivially parallelize AES-256 calls within AES-256-CTR, as well as Haraka instances (and AES rounds within Haraka). Many speed/area trade-offs are therefore possible.

ID	Default		Octopus	
	Length	Hashes	Hashes	Length
S	20 768	648	614.03	19 681
M	23 840	744	754.47	24 143
L	26 656	832	839.81	26 906

Table 4.2: Length of a signature without and with the Octopus trick to eliminate redundancies in a signature and use a 32-byte public key. We show the signature byte length (including the signature seed), as well as the number of 32-byte hash values forming the authentication path, including the K leaves, but excluding the signature seed.

4.5 Possible Optimizations

4.5.1 Faster Signing with Key Caching

Instead of taking a 64-byte secret key and expanding it to a $32T$-byte set of subkeys for each new signature, you can compute the subkeys once and cache it for future signatures. This saves $2T$ calls to AES-256, or $28T$ AES rounds, or 28% of the total AES rounds done when signing.

4.5.2 32-Byte Public Keys with Octopus

Authentication paths in a signature may contain several times the same value, if paths from two leaves merge and therefore have the same authentication paths from this point. To avoid these redundancies and offer optimally short signatures, we analyzed this trick—called "Octopus"–in [2, Ch.5], and developed tools to compute the expected number of hashes required in an authentication path. Indeed, the amount of redundancy depends on the choice of the K indices, and therefore a signature's length will depend on the message hashed.

Since most redundancies will occur in the higher levels of the tree, Octopus can be used to create optimally short signatures while having a single subtree root in the public key ($C = 1$), and therefore a 32-byte public key.

Table 4.2 shows the expected length of a signature if Octopus is used, comparing with the default length of $K(\log T - \log C) + K$ hashes plus the 32-byte signature seed.

Bibliography

[1] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *EUROCRYPT*, 2015.

[2] Guillaume Endignoux. Design and implementation of a post-quantum hash-based cryptographic signature scheme. Master's thesis, EPFL, 2017.

[3] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS – computing a 41KB signature in 16KB of RAM. Cryptology ePrint Archive, Report 2015/1042, 2015.

[4] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 - efficient short-input hashing for post-quantum applications. Cryptology ePrint Archive, Report 2016/098, 2016.

[5] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *ACISP*, 2002.