Enabling PERK on Resource-Constrained Devices

Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Marco Palumbi and Lucas Pandolfo Perin

Technology Innovation Institute, Abu Dhabi, UAE,
{slim.bettaieb,loic.bidoux,alessandro.budroni,marco.palumbi,lucas.perin}@tii.ae

Abstract. PERK is a digital signature scheme submitted to the recent NIST Post-Quantum Cryptography Standardization Process for Additional Digital Signature Schemes. For NIST security level I, PERK features sizes ranging from 6kB to 8.5kB, encompassing both the signature and public key, depending on the parameter set. Given its inherent characteristics, PERK's signing and verification algorithms involve the computation of numerous large objects, resulting in substantial stack-memory consumption ranging from 300kB to 1.5MB for NIST security level I and from 1.1MB to 5.7MB for NIST security level V. In this paper, we present a memory-versus-performance trade-off strategy that significantly reduces PERK's memory consumption to a maximum of approximately 82kB for any security level, enabling PERK to be executed on resource-constrained devices. Additionally, we explore various optimizations tailored to the Cortex M4 and introduce the first implementation of PERK designed for this platform.

Keywords: Post-Quantum Cryptography · PERK · Stack Usage · Cortex M4

1 Introduction

With the recent unveiling of the latest Post-Quantum Cryptography (PQC) standards, the National Institute of Standards and Technology (NIST) has released the preliminary public drafts of FIPS 203, FIPS 204, and FIPS 205, which are founded on cryptographic schemes commonly known as Kyber [ABD+22], Dilithium [DKL+22] and SPHINCS+ [ABB+22] respectively. Additionally, NIST has confirmed its intention to standardize Falcon [PFH+22] in the near future. This milestone marks the culmination of a protracted effort initiated in 2016, spanning three rounds of evaluation and public scrutiny. Concurrently, an ongoing fourth round permits to conduct further assessments and potentially standardize additional Key Encapsulation Mechanism (KEM) algorithms [AAB+22a, BCC+22, AAB+22b].

Furthermore, NIST has expressed a keen interest in investigating alternative general-purpose signature schemes, either those not anchored in structured lattices or those demonstrating superior performance compared to Dilithium and Falcon. This interest materialized through a recent call for proposals, inviting submissions of additional digital signature schemes. As a result, numerous new signature schemes have been submitted, initiating a fresh scrutiny process and fostering research around these new candidates.

A significant area of research revolves around the portability and efficiency of algorithms concerning resource-constrained devices. Typically, post-quantum cryptographic constructs exhibit substantial size, leading to implementations that consume significant amounts of memory. Consequently, the Cortex M4 platform has been regarded as the standard choice for benchmarking Post-Quantum Cryptography (PQC) implementations in scenarios where resource constraints are paramount.

This paper is focused on the recent post-quantum signature candidate PERK [ABB+23] which is based on the Permuted Kernel Problem [Sha90]. PERK is a signature scheme

relying on the Multi-Party Computation in-the-Head (MPCitH) paradigm [IKOS07]. Notably, PERK demonstrates competitiveness in terms of public key and signature sizes within the new NIST signature standardization process. However, a drawback to PERK lies in the necessity to compute large objects resulting in significant memory consumption.

Contributions. This paper studies and shows how to implement PERK on resource-constrained devices. Our contributions are threefold:

1. We present a memory versus performance trade-off strategy, resulting in a substantial reduction in PERK's memory cost from thousands of kilobytes to a maximum of approximately 82kB for the highest security levels. This optimization facilitates the deployment of PERK within the memory limits of the typical STM32F407 discovery board;

2. Additionally, we explore various optimization strategies to further enhance the running-time of the scheme for Cortex M4 devices. Our modifications include a streamlined protocol adhering to the PERK specification, enabling our implementation to successfully pass public Known Answer Tests (KAT) for compatibility;

3. Furthermore, our investigation extends to a different scenario where we deviate from the PERK reference implementation. In this alternative approach, we employ distinct techniques for operating with permutations, achieving superior performance on the M4 device. Although instantiating PERK in this way preserves all the features and guarantees of the scheme, one should note that the resulting implementation does not replicate the KAT of the reference implementation.

To the best of our knowledge, this work constitutes the first implementation of PERK for constrained devices.[1] Table 1 gives an overview of the stack usage of our implementation with respect to the PERK optimized one (one should note that PERK reference and optimized implementations have similar stack usage consumption, see Section 2.2 for more details).

Table 1: Stack usage comparison between PERK optimized implementation and our work

Instance	Implementation	Keygen	Signing	Verification
PERK-I-short3	PERK Ref. [ABB+23]	10 kB	1.49 MB	1.49 MB
	This work	8 kB	28 kB	25 kB
PERK-V-short3	PERK Ref. [ABB+23]	27 kB	5.74 MB	5.74 MB
	This work	26 kB	82 kB	75 kB

Paper Organization. We present the PERK signature scheme and the stack usage of its reference implementation in Section 2. Then, we discuss memory vs performances trade-offs allowing to implement PERK on memory-constrained devices in Section 3. Cortex-M4 specific optimizations are considered in Section 4. We provide an implementation that passes the KAT provided in the PERK submission for compatibility (Section 4.1) as well as a faster implementation that is compatible with the PERK specification but produces different KAT (Section 4.2). Experimental results are presented in Section 4.3.

[1] Our implementation forks `pqm4` [KPR+] and can be accessed publicly at https://anonymous.4open.science/r/perk-on-resource-constrained-devices.

2 Background

This paper follows the notation as in [ABB+23]. We denote by $[n]$ the set of integers $1 \leq i \leq n$, and with $\mathcal{S}_n$ the group of permutations of $[n]$. For a power of a prime q, let $\mathbb{F}_q$ denote the finite field of order q. Vectors are denoted with bold lower-case letters (e.g. $\boldsymbol{v} = (v_j)_{1 \leqslant j \leqslant k} \in \mathbb{F}_q^n$) and matrices with bold upper-case letters. Let X be a finite set, we use the notation $x \xleftarrow{\$} X$ to say that x is chosen uniformly at random from X, and the notation $x \xleftarrow{\$,\theta} X$ to say that x is sampled pseudo-randomly from X using the seed θ.

2.1 The PERK Signature Scheme

PERK is built from a zero-knowledge proof of knowledge for the relaxed Inhomogeneous Permuted Kernel Problem r-IPKP. Informally, given a matrix $\boldsymbol{H} \in \mathbb{F}_q^{m \times n}$ and t pairs of vectors $(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathbb{F}_q^n \times \mathbb{F}_q^m$, the r-IPKP problem asks to find a permutation $\pi \in \mathcal{S}_n$ such that $\boldsymbol{H}\pi(\boldsymbol{x}) = \boldsymbol{y}$ where $\boldsymbol{x} := \sum_i \kappa_i \boldsymbol{x}_i$ (respectively $\boldsymbol{y} := \sum_i \kappa_i \boldsymbol{y}_i$) and $\kappa = (\kappa_1, \ldots, \kappa_t) \in \mathbb{F}_q^t \setminus \mathbf{0}$. We define the relation $\mathcal{R}_{\text{r-IPKP}}$ for this problem as:

$$\mathcal{R}_{\text{r-IPKP}} := \left\{ \left(\left(\boldsymbol{H}, (\boldsymbol{x}_i, \boldsymbol{y}_i)_{i \in [t]} \right) ; \tilde{\pi} \right) \; : \; \begin{array}{l} \boldsymbol{H}\left(\tilde{\pi}\left[\sum_{i \in [1,t]} \kappa_i \cdot \boldsymbol{x}_i \right] \right) = \sum_{i \in [1,t]} \kappa_i \cdot \boldsymbol{y}_i \\ \text{for any } \kappa \in {\mathbb{F}_q}^t \setminus \mathbf{0} \end{array} \right\}.$$

The zero-knowledge proof of knowledge used in PERK is inspired from [BG23, FJR23] and constructed using the MPCitH paradigm [IKOS07]. It is then transformed into a signature scheme using the Fiat-Shamir transform [FS87] within the random oracle model.

For the three security levels specified by the NIST, PERK provides four distinct sets of parameters. Parameters denoted as *short* are designed to optimize the signature's size while parameters denoted as *fast* prioritize the running time of the algorithms. In addition, parameters differs based on the value of t (either 3 or 5) in the underlying r-IPKP problem. As a consequence, PERK instances are referred as PERK-X-Y where X denotes the NIST security level (I, III or V) and Y is either *fast3*, *fast5*, *short3* or *short5*. PERK parameters are given in Table 2.

Table 2: Parameters of the PERK signature scheme [ABB+23]

Parameter Set	λ	q	n	m	t	N	τ	pk size	sk size	σ size
PERK-I-fast3	128	1021	79	35	3	32	30	0.15 kB	16 B	8.36 kB
PERK-I-fast5	128	1021	83	36	5	32	28	0.24 kB	16 B	8.03 kB
PERK-I-short3	128	1021	79	35	3	256	20	0.15 kB	16 B	6.25 kB
PERK-I-short5	128	1021	83	36	5	256	18	0.24 kB	16 B	5.78 kB
PERK-III-fast3	192	1021	112	54	3	32	46	0.23 kB	24 B	18.8 kB
PERK-III-fast5	192	1021	116	55	5	32	43	0.37 kB	24 B	18.0 kB
PERK-III-short3	192	1021	112	54	3	256	31	0.23 kB	24 B	14.3 kB
PERK-III-short5	192	1021	116	55	5	256	28	0.37 kB	24 B	13.2 kB
PERK-V-fast3	256	1021	146	75	3	32	61	0.31 kB	32 B	33.3 kB
PERK-V-fast5	256	1021	150	76	5	32	57	0.51 kB	32 B	31.7 kB
PERK-V-short3	256	1021	146	75	3	256	41	0.31 kB	32 B	25.1 kB
PERK-V-short5	256	1021	150	76	5	256	37	0.51 kB	32 B	23.0 kB

2.1.1 PERK Overview

The Keygen, Sign and Verify algorithms of PERK are described in Figures 1, 2 and 3 respectively. PERK Keygen algorithm consists in generating an r-IPKP instance. PERK Sign algorithm emulates a proof of knowledge for the r-IPKP problem between a Prover and a Verifier. Informally, the Prover generates a sharing of its secret permutation π for N parties and commits to each share $(\pi_i, \boldsymbol{v}_i)_{i\in[1,N]}$ as well as $\boldsymbol{Hv}$ for some mask $\boldsymbol{v}$ (Step 1). The Verifier chooses a random challenge κ (Step 2) then the Prover computes "in-its-head" $\pi[\sum_{j\in[1,t]} \kappa_j \cdot \boldsymbol{x}_j] + \boldsymbol{v}$ and commit to the output of each party within the computation (Step 3). Next, the Verifier picks a random party $\alpha \in [N]$ (Step 4) and the Prover computes the signature accordingly by revealing the input of all parties except the α one along with information allowing to emulate the MPC protocol without knowing α's input (Step 5). PERK verification algorithm starts by parsing the signature in order to retrieve the challenges κ and α (Step 1). Once it is done, the signature is verified by checking that both the input of the received parties and the MPC protocol execution are consistent with the received commitments (Step 2).

1. Sample $\mathsf{sk\_seed} \xleftarrow{\$} \{0,1\}^\lambda$ and $\mathsf{pk\_seed} \xleftarrow{\$} \{0,1\}^\lambda$
2. Sample $\pi \longleftarrow \mathsf{PRG}(\mathsf{sk\_seed})$ from $\mathcal{S}_n$
3. Sample $(\boldsymbol{H}, (\boldsymbol{x}_i)_{i\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
3. For $j \in [1,t]$,
   - ◇ Compute $\boldsymbol{y}_j = \boldsymbol{H}\pi[\boldsymbol{x}_j]$
4. Output $(\mathsf{sk}, \mathsf{pk}) = (\mathsf{sk\_seed}, (\mathsf{pk\_seed}, (\boldsymbol{y}_j)_{j\in[1,t]}))$

Figure 1: PERK - KeyGen algorithm

The efficiency of PERK is heavily dependent on operations involving permutations. In the subsequent section, we delve into the permutation related operations (namely permutation sampling, permutation composition, permutation inversion and permutation compression) that play a crucial role in PERK performances.

2.1.2 Permutation Sampling, Composition, Inversion and Compression

Along with randomness sampling, permutation related operations are one of the most computationally expensive tasks in PERK.

Permutation sampling. Random permutations are sampled from $\mathcal{S}_n$ using a method that includes the use of a sorting algorithm. Specifically, to create a permutation $\pi \in S_n$, one first samples a random vector $\boldsymbol{e} = (e_0, \ldots, e_{n-1}) \in (\mathbb{F}_2^{16})^n$, then constructs $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$, where $p_i = (e_i|i)$. The integer sequence $\boldsymbol{p}$ is then sorted using a constant-time sorting algorithm, and the permutation π is extracted from the lower-order bits of each p_i. If there are any duplicate values in the vector $\boldsymbol{e}$, the process discards it and generates a new one. The official PERK implementation employs the constant-time software library `djbsort` [Ber19] for the sorting algorithm.

Permutating vectors. In order to permute vectors $\boldsymbol{v}$ according to a permutation π in constant time, the PERK official implementation builds a vector $\boldsymbol{p}(\pi_i|v_i)$, sorts it with `djbsort`, and extracts the lower-bits order vector from it.

Permutation composition and inversion. The aforementioned approach is also used to invert, compose and apply permutations to vectors. Let $\pi \in \mathcal{S}_n$ be a permutation. After sorting, the lower-bits order of the vector $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$ where $p_i = (\pi_i|i)$ corresponds to the inverse permutation π^{-1}. Similarly given two permutations $\pi, \tau \in \mathcal{S}_n$, after sorting, the lower-bits order of the vector $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$ where $p_i = (\tau_i|\pi_i)$ corresponds to the

Inputs: The key pair $(\mathsf{sk}, \mathsf{pk}) = (\mathsf{sk\_seed}, (\mathsf{pk\_seed}, (\boldsymbol{y}_j)_{j\in[1,t]}))$ and a message $m \in \{0,1\}^*$

Step 1: Commitment

1. Sample $\pi \longleftarrow \mathsf{PRG}(\mathsf{sk\_seed})$ from $\mathcal{S}_n$
2. Sample $(\boldsymbol{H}, (\boldsymbol{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
3. Sample salt and master seed $(\mathsf{salt}, \mathsf{mseed}) \xleftarrow{\$} \{0,1\}^{2\lambda} \times \{0,1\}^{\lambda}$
4. Sample seeds $(\theta^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \mathsf{mseed})$ from $(\{0,1\}^{\lambda})^{\tau}$
5. For each iteration $e \in [1,\tau]$,
   - $\diamond$ Compute $(\theta_i^{(e)})_{i\in[1,N]} \longleftarrow \mathsf{TreePRG}(\mathsf{salt}, \theta^{(e)})$
   - $\diamond$ For each party $i \in \{N,\dots,1\}$,
     - If $i \neq 1$, sample $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)}) \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_i^{(e)})$ from $\mathcal{S}_n \times \mathbb{F}_q^n$
     - If $i = 1$, sample $\boldsymbol{v}_1^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_1^{(e)})$ from $\mathbb{F}_q^n$
     - If $i \neq 1$, compute $\mathsf{cmt}_{1,i}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, i, \theta_i^{(e)})$
     - If $i = 1$, compute $\pi_1^{(e)} = (\pi_2^{(e)})^{-1} \circ \cdots \circ (\pi_N^{(e)})^{-1} \circ \pi$ and $\mathsf{cmt}_{1,1}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$
   - $\diamond$ Compute $\boldsymbol{v}^{(e)} = \boldsymbol{v}_N^{(e)} + \sum_{i\in[1,N-1]} \pi_N^{(e)} \circ \cdots \circ \pi_{i+1}^{(e)}[\boldsymbol{v}_i^{(e)}]$ and $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \boldsymbol{H}\boldsymbol{v}^{(e)})$

Step 2: First Challenge

6. Compute $h_1 = \mathsf{H}_1(\mathsf{salt}, m, \mathsf{pk}, (\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{e\in[1,\tau], i\in[1,N]})$
7. Sample $(\kappa_j^{(e)})_{e\in[1,\tau], j\in[1,t]} \longleftarrow \mathsf{PRG}(h_1)$ from $(\mathbb{F}_q^t)^{\tau}$

Step 3: First Response

8. For each iteration $e \in [1,\tau]$,
   - $\diamond$ Compute $\boldsymbol{s}_0^{(e)} = \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \boldsymbol{x}_j$
   - $\diamond$ For each party $i \in [1,N]$,
     - Compute $\boldsymbol{s}_i^{(e)} = \pi_i^{(e)}[\boldsymbol{s}_{i-1}^{(e)}] + \boldsymbol{v}_i^{(e)}$

Step 4: Second Challenge

9. Compute $h_2 = \mathsf{H}_2(\mathsf{salt}, m, \mathsf{pk}, h_1, (\boldsymbol{s}_i^{(e)})_{e\in[1,\tau], i\in[1,N]})$
10. Sample $(\alpha^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1,N])^{\tau}$

Step 5: Second Response

11. For each iteration $e \in [1,\tau]$,
    - $\diamond$ Compute $\boldsymbol{z}_1^{(e)} = \boldsymbol{s}_{\alpha}^{(e)}$
    - $\diamond$ If $\alpha^{(e)} \neq 1$, $z_2^{(e)} = (\pi_1^{(e)} \,||\, (\theta_i^{(e)})_{i\in[1,N]\setminus\alpha^{(e)}})$
    - $\diamond$ If $\alpha^{(e)} = 1$, $z_2^{(e)} = (\theta_i^{(e)})_{i\in[1,N]\setminus\alpha^{(e)}}$
    - $\diamond$ Compute $\mathsf{rsp}^{(e)} = (\boldsymbol{z}_1^{(e)}, z_2^{(e)}, \mathsf{cmt}_{1,\alpha^{(e)}}^{(e)})$
12. Compute $\sigma = (\mathsf{salt}, h_1, h_2, (\mathsf{rsp}^{(e)})_{e\in[1,\tau]})$

Figure 2: PERK - Sign algorithm

permutation $\pi \circ \tau^{-1}$. Hence, to compute the composition $\pi \circ \tau$, one should first invert τ, and then compute $\pi \circ (\tau^{-1})^{-1}$ with the procedure just described.

Permutation compression. PERK signatures include sending permutations in clear. To keep the sizes reduced, PERK involves two different approaches to represent permutations respectively for *short* and *fast* parameters sets trading off length and efficiency of computations. More specifically, *short* parameters employ a ranking/unranking algorithm to represent each permutation as a unique integer in the set $\{0,...,n!-1\}$ that is optimal

Inputs: The public key $\mathsf{pk} = (\mathsf{pk\_seed}, (\boldsymbol{y}_j)_{j\in[1,t]})$, a signature σ and a message $m \in \{0,1\}^*$

Step 1: Parse signature

1. Sample $(\boldsymbol{H}, (\boldsymbol{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
2. Parse signature as $\sigma = (\mathsf{salt}, h_1, h_2, (\boldsymbol{z}_1^{(e)}, z_2^{(e)}, \mathsf{cmt}_{1,\alpha^{(e)}}^{(e)})_{e\in[1,\tau]})$
3. Recompute $(\kappa_j^{(e)})_{e\in[1,\tau],j\in[1,t]} \longleftarrow \mathsf{PRG}(h_1)$ from $(\mathbb{F}_q^t \setminus \mathbf{0})^\tau$
4. Recompute $(\alpha^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1,N])^\tau$

Step 2: Verification

5. For each iteration $e \in [1,\tau]$,
   - ◇ Compute $\boldsymbol{s}_0^{(e)} = \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \boldsymbol{x}_j$ and $\boldsymbol{s}_\alpha^{(e)} = \boldsymbol{z}_1^{(e)}$
   - ◇ Compute $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})_{i\in[1,N]\setminus\alpha}$ from $\boldsymbol{z}_2^{(e)}$
   - ◇ For each party $i \in [1,N] \setminus \alpha^{(e)}$,
     - If $i \neq 1$, compute $\mathsf{cmt}_{1,i}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, i, \theta_i^{(e)})$
     - If $i = 1$, compute $\mathsf{cmt}_{1,1}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$
   - ◇ For each party $i \in [1,N] \setminus \alpha^{(e)}$,
     - Compute $\boldsymbol{s}_i^{(e)} = \pi_i^{(e)}[\boldsymbol{s}_{i-1}^{(e)}] + \boldsymbol{v}_i^{(e)}$
   - ◇ Compute $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \boldsymbol{H}\boldsymbol{s}_N^{(e)} - \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \boldsymbol{y}_j)$
6. Compute $\bar{h}_1 = \mathsf{H}_1(\mathsf{salt}, m, \mathsf{pk}, ((\mathsf{cmt}_{1,i}^{(e)})_{i\in[N,1]}, \mathsf{cmt}_1^{(e)})_{e\in[1,\tau]})$
7. Compute $\bar{h}_2 = \mathsf{H}_2(\mathsf{salt}, m, \mathsf{pk}, h_1, (\boldsymbol{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]})$.
8. Output accept if and only if $\bar{h}_1 = h_1$ and $\bar{h}_2 = h_2$.

Figure 3: PERK - Verify algorithm

(i.e. the shortest as possible) but requires making computations with arbitrary-precision arithmetic. To this end, the official PERK implementation makes use of the the `gmp` library [Pro23]. To speed-up the execution time, a look-up table is used to store some pre-computed factorials used in the algorithm. On the other hand, *fast* parameters employ a packing algorithm that is sub-optimal on the size but much lighter in the computation and does not require arbitrary-precision arithmetic.

2.1.3 Object Representation, Randomness and Hash Functions

Object representation. In PERK, elements of $\mathbb{F}_q$ are stored in 16 bit unsigned integers, vectors are represented as arrays of $\mathbb{F}_q$ elements and matrices are represented as a two dimensional arrays of $\mathbb{F}_q$ elements. In addition, permutations are represented as an array of length n of elements in $[0, n-1]$ encoded on 8 bits. Permutations are also represented as string of bits whenever they are compressed.

PRG and Hash functions. In PERK, the PRG function is instantiated using `SHAKE-128` for $\lambda = 128$ and `SHAKE-256` for $\lambda = 192$ or $\lambda = 256$. In addition, the hash functions are instantiated using `SHA3-256` for $\lambda = 128$, `SHA3-384` for $\lambda = 192$ and `SHA3-512` for $\lambda = 256$. Both functions are instantiated using domain separators.

TreePRG. In PERK, the signer must generate sets of N seeds then reveal $N-1$ of these as part of the signature. In order to do so, the signer uses a master seed to compute a binary tree structure whose N leaves correspond to the N seeds to be generated. This allows to reveal $N-1$ of these seeds efficiently by revealing intermediate nodes in the tree.

2.2 Stack Usage in PERK

In the reference and optimized implementations of PERK, the primary factor influencing stack usage during both signing and verification is the storage of variables such as seed trees, permutations, and vectors. Let's explore these variables by examining their computation and usage across the different steps of the protocol in the provided implementations.

We start by the signing algorithm described in Figure 2. In each iteration $e \in [1, \tau]$ of the commitment step (Step 1), the signer generates a seed tree having the seeds $(\theta_i^{(e)})_{i\in[1,N]}$ as leaves, the permutations $(\pi_i^{(e)})_{i\in[1,N]}$, the vectors $(\boldsymbol{v}_i^{(e)})_{i\in[1,N]}$ and a set of commitments $(\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{i\in[1,N]}$. In order to generate the first challenge (Step 2), the signer uses the commitments computed in the previous step to generate h_1 then samples the first challenge. In the first response step (Step 3), in order to compute the vectors $(\boldsymbol{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]}$, the signer must possess the pairs $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})_{e\in[1,\tau],i\in[1,N]}$ generated in the commitment step. Then, the signer uses the $(\boldsymbol{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]}$ to sample the second challenge through h_2 (Step 4). Finally, in order to generate the signature σ (Step 5), the signer needs the seeds $(\theta_i^{(e)})_{e\in[1,\tau],i\in[1,N]}$, the permutations $(\pi_i^{(e)})_{e\in[1,\tau],i\in[1,N]}$ and the commitments $(\mathsf{cmt}_{1,i}^{(e)})_{e\in[1,\tau],i\in[1,N]}$ generated in the Step 1. It also needs the $\boldsymbol{s}_i^{(e)}$ computed in the Step 3. In order to optimize performance and in light of variable reuse across different steps of the scheme, the provided implementations leverage a data structure for storing these variables to enable efficient reuse (see Figure 4). One should note that there is a total of τ instances of the aforementioned data structure allocated in the stack, corresponding to the number of rounds in the algorithm.

```
typedef struct {
    perk_theta_seeds_tree_t theta_tree;
    perm_t pi_i[PARAM_N];
    vect1_t *v_i;
    vect1_t s_i[PARAM_N + 1];
    cmt_t cmt_1_i[PARAM_N];
    cmt_t cmt_1;
} perk_instance_t;
```

Figure 4: Data structure used to store seed trees, permutations π_i's, random vectors $\boldsymbol{v}_i$'s, vectors $\boldsymbol{s}_i$'s and commitments $(\mathsf{cmt}_1, \mathsf{cmt}_{1,i})$ [ABB+23].

Having analyzed the stack usage of the signing algorithm, let us now shift our focus to the verification algorithm where a similar observation can be made. Step 1 in the verification algorithm involves parsing the signature and generating the challenges. During Step 2, the signer generates and stores $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})_{e\in[1,\tau],i\in[1,N]\setminus\alpha^{(e)}}$. To achieve this, they first need to generate their corresponding seeds $(\theta_i^{(e)})_{i\in[1,N]\setminus\alpha^{(e)}}$ from the partial tree, necessitating the storage of seed trees. Notice that the set of pairs $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})_{e\in[1,\tau],i\in[1,N]\setminus\alpha^{(e)}}$ are used later to generate the vectors $(\boldsymbol{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]\setminus\alpha^{(e)}}$. Then, the signer computes the commitments $(\mathsf{cmt}_{1,i}^{(e)})_{e\in[1,\tau],i\in[1,N]\setminus\alpha^{(e)}}$ using the seed trees and computes $(\boldsymbol{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]\setminus\alpha^{(e)}}$ which are used twice: first, for the computation of commitments $(\mathsf{cmt}_1^{(e)})_{e\in[1,\tau]}$ and then subsequently for calculating the hash value $\bar{h}_2$. Finally, the signer uses the stored commitments $(\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{e\in[1,\tau],i\in[1,N]}$ to compute $\bar{h}_1$. Similarly to the signing algorithm, the verify algorithm relies on the data structure depicted in Figure 4 to store these variables.

We provide in Table 3 the stack usage of the official PERK implementations. One can see that the reference and optimized implementations have a very similar memory consumption profile. As a consequence, we won't differentiate them in the remaining of

this paper for simplicity. Hereafter, whenever memory consumption values are given, they correspond to the optimized implementation.

Table 3: Stack usage of the PERK reference and optimized implementations [ABB+23].

Algorithm	**Keygen**		**Signing**		**Verification**	
	Ref.	Opt.	Ref.	Opt.	Ref.	Opt.
PERK-I-fast3	10 kB	10 kB	307 kB	310 kB	307 kB	310 kB
PERK-I-fast5	11 kB	11 kB	299 kB	302 kB	299 kB	302 kB
PERK-I-short3	10 kB	10 kB	1.49 MB	1.49 MB	1.49 MB	1.49 MB
PERK-I-short5	11 kB	11 kB	1.40 MB	1.40 MB	1.40 MB	1.40 MB
PERK-III-fast3	17 kB	17 kB	671 kB	674 kB	671 kB	674 kB
PERK-III-fast5	18 kB	18 kB	647 kB	650 kB	647 kB	650 kB
PERK-III-short3	17 kB	17 kB	3.31 MB	3.31 MB	3.31 MB	3.31 MB
PERK-III-short5	18 kB	18 kB	3.08 MB	3.08 MB	3.08 MB	3.08 MB
PERK-V-fast3	27 kB	27 kB	1.14 MB	1.14 MB	1.14 MB	1.14 MB
PERK-V-fast5	29 kB	29 kB	1.09 MB	1.09 MB	1.09 MB	1.09 MB
PERK-V-short3	27 kB	27 kB	5.73 MB	5.74 MB	5.73 MB	5.74 MB
PERK-V-short5	29 kB	29 kB	5.29 MB	5.29 MB	5.29 MB	5.29 MB

3 PERK on Memory-Constrained Devices

In order to enable PERK on resource-constrained devices, we propose several modifications for the sign and verification algorithms in Section 3.1 and 3.2 respectively. The resulting streamlined algorithms are described in 5 and 6 and constitutes the keystone of the implementation evaluated in Section 4.

3.1 Sign Algorithm

We present several optimizations reducing the memory footprint of the Sign algorithm in Section 3.1.1. Then, we discuss performance related optimizations in Section 3.1.2. Some of these performance related optimizations increase the memory consumption of the algorithm but are nonetheless considered as they constitutes interesting memory vs performances trade-offs.

3.1.1 Memory related optimizations

Streamlining seed trees sampling and usage. As explained in Section 2.2, the seed trees generated in Step 1 are preserved for later use in Step 5. Storing them requires $\tau(2N-1)\lambda/8$ bytes hence inducing a large memory consumption. Alternatively, we opt to generate the seed tree of each iteration, utilize them to derive required variables, and then promptly erase them from memory. In Step 3 and Step 5, we recompute the seed trees for each iteration as needed leveraging the knowledge of the salt and master seed $(\mathsf{salt}, \mathsf{mseed})$. By adopting this approach, we can reduce the amount of stored memory by a factor τ resulting in substantial savings in terms of memory footprint (see Table 4). However, this comes at the cost of needing to recalculate the seed trees twice, specifically in Steps 3 and 5.

Table 4: Stack memory required to store the seed trees in kB.

Algorithm	PERK	This work
PERK-I-fast3	30.2	1
PERK-I-fast5	28.2	1
PERK-I-short3	163	8.1
PERK-I-short5	147	8.1
PERK-III-fast3	69.5	1.5
PERK-III-fast5	65	1.5
PERK-III-short3	380	12.2
PERK-III-short5	343	12.2
PERK-V-fast3	122	2
PERK-V-fast5	114	2
PERK-V-short3	670	16.3
PERK-V-short5	605	16.3

Streamlining $\boldsymbol{\pi_i}$'s and $\boldsymbol{v_i}$'s sampling and usage. The permutations and random vectors $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})_{e\in[1,\tau],i\in[1,N]}$ are generated in Step 1 and preserved for subsequent use in Step 3 and 5. Alternatively, we choose to generate only one pair $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})$ at a time. This strategy leads to significant savings in terms of memory footprint as shown in Table 5. Indeed in the PERK implementation, permutations and vectors are stored using $nN\tau$ and $2nN\tau$ bytes respectively while our implementationonly requires $3n$ bytes. However, this modification implies to recomputate these values in Step 3 and 5.

Table 5: Stack memory required to store permutations and vectors.

Algorithm	PERK	This work
PERK-I-fast3	227 kB	237 B
PERK-I-fast5	223 kB	249 B
PERK-I-short3	1.21 MB	237 B
PERK-I-short5	1.14 MB	249 B
PERK-III-fast3	494 kB	336 B
PERK-III-fast5	478 kB	348 B
PERK-III-short3	2.66 MB	336 B
PERK-III-short5	2.49 MB	348 B
PERK-V-fast3	854 kB	438 B
PERK-V-fast5	820 kB	450 B
PERK-V-short3	4.59 MB	438 B
PERK-V-short5	4.26 MB	450 B

Streamlining commitments and hash values computation. The commitments $(\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{i\in[1,N]}$ are generated in Step 1 and used for computing h_1 in Step 2. Subsequently, only one commitment among $(\mathsf{cmt}_{1,i}^{(e)})_{i\in[1,N]}$ is required for later use in Step 5 where, depending on the challenge $\alpha^{(e)}$, a commitment becomes part of the signature. Storing these commitments requires a significant amount of memory. Alternatively, instead of deferring the absorption of these values until Step 2, we can efficiently absorb them within Step 1 right after their generation. This is feasible for two reasons. Firstly, in Step 2, the commitments $(\mathsf{cmt}_{1,i}^{(e)})_{i\in[1,N]}$ are absorbed before $\mathsf{cmt}_1^{(e)}$, and the computation of the latter does not depend on the values of the former. Secondly, the hash function's state is prepared in order to absorb these values. Hence, there is no need to wait until the generation of all commitments is complete to compute h_1. This results in a reduction of the memory usage from $2\lambda(N+1)\tau$ bits down to 2λ bits (see Table 6) as we reuse the buffer initially used for the computation of $(\mathsf{cmt}_{1,i}^{(e)})_{i\in[1,N]}$ for $\mathsf{cmt}_1^{(e)}$. On the other hand, since we don't save the commitment value, this only comes at the cost of recomputing one commitment $\mathsf{cmt}_{1,\alpha^{(e)}}^{(e)}$, in Step 5 which is computationally negligible. Similarly, we suggest simplifying the computation of h_2, mirroring the approach taken for h_1. Specifically, the absorption of vectors $\boldsymbol{s}_i^{(e)}$ occurs within Step 3.

Streamlining $\boldsymbol{s_i}$ computation. The vectors $\boldsymbol{s}_i^{(e)}$ are calculated during Step 3 and used to compute h_2 in Step 4 as well as in Step 5 where depending on the value of the challenge $\alpha^{(e)}$, a specific $\boldsymbol{s}_{\alpha^{(e)}}^{(e)}$ is selected from the set of stored $\boldsymbol{s}_i^{(e)}$ to be included in the signature. It's worth noting that in the current PERK implementation, $\boldsymbol{s}_i^{(e)}$ and $\boldsymbol{v}_i^{(e)}$ share the same memory address. This approach allows to save memory as freshly computed $\boldsymbol{s}_i^{(e)}$ can be stored at the index of the corresponding $\boldsymbol{v}_i^{(e)}$ used in its computation. Since $\boldsymbol{v}_i^{(e)}$ is not used after Step 3, they can be overwritten in the buffer storing $\boldsymbol{s}_i^{(e)}$. Rather than reusing the buffer for $\boldsymbol{v}_i^{(e)}$, our strategy involves recalculating $\boldsymbol{s}_i^{(e)}$ during Step 5. This choice is based on the fact that, upon computation, each $\boldsymbol{s}_i^{(e)}$ is immediately absorbed by H_2. Consequently, there is no need to store these values and wait until Step 4. Additionally, it's important to highlight that in Step 5, we are not compelled to generate all N pairs; instead, one can halt the generation at index $\alpha^{(e)}$.

Table 6: Stack memory required to store commitments.

Algorithm	PERK	This work
PERK-I-fast3	31.6 kB	32 B
PERK-I-fast5	29.5 kB	32 B
PERK-I-short3	164 kB	32 B
PERK-I-short5	148 kB	32 B
PERK-III-fast3	72.8 kB	48 B
PERK-III-fast5	68.1 kB	48 B
PERK-III-short3	382 kB	48 B
PERK-III-short5	345 kB	48 B
PERK-V-fast3	128 kB	64 B
PERK-V-fast5	120 kB	64 B
PERK-V-short3	674 kB	64 B
PERK-V-short5	608 kB	64 B

3.1.2 Performance related optimizations

Computing π^{-1}. In Step 1, one can calculate $\pi_1^{(e)}$ by first computing the permutation $\pi^{-1} \circ (\pi_N^{(e)}) \circ \ldots \circ (\pi_2^{(e)})$ and then inverting it. This approach enables the avoidance of $N-1$ permutation inverses, reducing the computation load to only two inversions: inverting the secret permutation π and the intermediate result. Since π^{-1} is utilized in each iteration $e \in [1, \tau]$, it is suggested to compute and store this value. By doing so, the need to invert π multiple times in Step 1 is mitigated, incurring only a negligible memory cost of n bytes to store π^{-1}.

Storing permutation π_1. As we no longer retain the permutations $(\pi_i^{(e)})_{i \in [1,N]}$ and instead recalculate them in Step 5, it becomes necessary to recompute the permutation $\pi_1^{(e)}$, derived from the permutation $(\pi_2^{(e)}, \ldots, \pi_N^{(e)})$. This computation involves composing $(N-1)$ permutations making it a resource-intensive task. To mitigate this, we adopt the strategy of preserving the value of $\pi_1^{(e)}$ for each iteration e in Step 1. This approach induces an overhead of $N\tau$ bytes in memory consumption as illustrated in Table 7. This amount is relatively modest hence this constitutes an interesting trade-off considering the substantial performance gains achieved.

Optimizing permutation sampling and composition. In the streamlined version of the PERK sign algorithm, right after sampling the permutation π_i, this gets composed with another permutation $\pi_1 \circ (\pi_i^{(e)})$ in Step 1. We improve these two operations thanks to the following observation. Assume that we want to sample a permutation π and right after we want to compute the composition $\tau \circ \pi$ with another permutation τ. Let $e_0, \ldots, e_{n-1}$ be the random buffer utilized to sample π via `djbsort`. This main observation here is that this buffer inherently represents the inverse of π_i. More precisely, the sorting of the buffer aligns with the one of π_i^{-1}. Exploiting this observation, we optimize computations as follows. Construct $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$, where $p_i = (e_i | \tau_i | i)$. After sorting $\boldsymbol{p}$ using `djbsort`, the lower bits, corresponding to i, encode the permutation π while the center bits corresponding to τ precisely encode $\tau \circ \pi$. The main advantage of this novel approach is that we perform all these operations by making only one call to `djbsort` instead of three (sample, invert, compose). Notice that we can leverage this idea thanks to the fact that `djbsort` works with 32 bits words, and this is enough for using 16 bits for the randomness, 8 bits for i and the remaining bits for τ.

Table 7: Stack memory required to store π_1 in kB.

Algorithm	**This work**
PERK-I-fast3	2.3
PERK-I-fast5	2.3
PERK-I-short3	1.5
PERK-I-short5	1.4
PERK-III-fast3	5.1
PERK-III-fast5	4.9
PERK-III-short3	3.4
PERK-III-short5	3.2
PERK-V-fast3	8.9
PERK-V-fast5	8.5
PERK-V-short3	5.9
PERK-V-short5	5.5

Balancing memory and performance using s_i. Given a targeted memory limit, one can fine tune the implementation using the $\boldsymbol{s}_i$ in order to get the best possible performance with respect to the available memory. By storing the $(\boldsymbol{s}_i^{(e)})_{e\in[1,\tau'],i\in[1,N]}$ values (where $\tau' \leq \tau$) after their computation in Step 3, one only need to recompute them in Step 5 for the remaining $\tau - \tau'$ rounds thus avoiding $\tau' N$ computations of $\boldsymbol{s}_i$ values. In addition, keeping $\boldsymbol{s}_0^{(e)}$ is unnecessary and thus its computation can be omitted once the $\boldsymbol{s}_i^{(e)}$ values are stored. For instance, using PERK-I-fast3 parameters, setting $\tau' = 7$ requires 35.3 kB of memory but saves 224 computations of $\boldsymbol{s}_i$ values. Similarly, using $\tau' = 15$ and $\tau' = 30$ requires 75.8 kB and 151 kB of memory but saves 480 and 960 computations of $\boldsymbol{s}_i$ values.

Inputs: The key pair $(\mathsf{sk}, \mathsf{pk}) = (\mathsf{sk\_seed}, (\mathsf{pk\_seed}, (\boldsymbol{y}_j)_{j\in[1,t]}))$ and a message $m \in \{0,1\}^*$

Step 1: Commitment

1. Sample $\pi \longleftarrow \mathsf{PRG}(\mathsf{sk\_seed})$ from $\mathcal{S}_n$ and compute $\pi_{\mathsf{inverse}} = \pi^{-1}$
2. Sample $(\boldsymbol{H}, (\boldsymbol{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
3. Sample salt and master seed $(\mathsf{salt}, \mathsf{mseed}) \xleftarrow{\$} \{0,1\}^{2\lambda} \times \{0,1\}^{\lambda}$
4. $h_1.\mathsf{state} = \mathsf{H.init}(\mathsf{salt})$
5. $h_1.\mathsf{state} = h_2.\mathsf{state} = \mathsf{H.update}(m, \mathsf{pk})$
6. For each iteration $e \in [1, \tau]$,
   - ◇ Set $\pi_1^{(e)} = \pi_{\mathsf{inverse}}$
   - ◇ Sample seed $\theta^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \mathsf{mseed})$ from $\{0,1\}^{\lambda}$
   - ◇ Compute $(\theta_i^{(e)})_{i\in[1,N]} \longleftarrow \mathsf{TreePRG}(\mathsf{salt}, \theta^{(e)})$
   - ◇ For each party $i \in \{N, \ldots, 2\}$,
     - Sample $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)}) \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_i^{(e)})$ from $\mathcal{S}_n \times \mathbb{F}_q^n$
     - Compute $\mathsf{cmt}_{1,i}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, i, \theta_i^{(e)})$ and $h_1.\mathsf{state} = \mathsf{H.update}(h_1.\mathsf{state}, \mathsf{cmt}_{1,i}^{(e)}))$
     - $\pi_1^{(e)} = \pi_1^{(e)} \circ (\pi_i^{(e)})$,
     - If $i = N$, $\boldsymbol{v}^{(e)} = \boldsymbol{v}_N^{(e)}$ and $\pi_{\mathsf{comp}}^{(e)} = \pi_N^{(e)}$
     - If $i \neq N$ $\boldsymbol{v}^{(e)} = \boldsymbol{v}^{(e)} + \pi_{\mathsf{comp}}^{(e)}(\boldsymbol{v}_i^{(e)})$ and $\pi_{\mathsf{comp}}^{(e)} = \pi_{\mathsf{comp}}^{(e)} \circ \pi_i^{(e)}$
   - ◇ Compute $\pi_1^{(e)} = \pi_1^{(e)}{}_{\mathsf{inverse}}$ and $\mathsf{cmt}_{1,1}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$ // `We save` $\pi_1^{(e)}$.
   - ◇ $h_1.\mathsf{state} = \mathsf{H.update}(h_1.\mathsf{state}, \mathsf{cmt}_{1,1}^{(e)}))$,
   - ◇ Sample $\boldsymbol{v}_1^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_1^{(e)})$ from $\mathbb{F}_q^n$
   - ◇ $\boldsymbol{v}^{(e)} = \boldsymbol{v}^{(e)} + \pi_{\mathsf{comp}}^{(e)}(\boldsymbol{v}_1^{(e)})$
   - ◇ Compute $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \boldsymbol{H}\boldsymbol{v}^{(e)})$
   - ◇ $h_1.\mathsf{state} = \mathsf{H.update}(h_1.\mathsf{state}, \mathsf{cmt}_1^{(e)})$,

Step 2: First Challenge

7. Compute $h_1 = \mathsf{H}_1.\mathsf{final}(h_1.\mathsf{state}, \mathsf{H}_1)$
8. Sample $(\kappa_j^{(e)})_{e\in[1,\tau], j\in[1,t]} \longleftarrow \mathsf{PRG}(h_1)$ from $(\mathbb{F}_q^t \setminus \mathbf{0})^{\tau}$

Step 3: First Response

9. Use $h_2.\mathsf{state}$ from 5 (Step 1).
10. Compute $h_2.\mathsf{state} = \mathsf{H.update}(h_2.\mathsf{state}, h_1)$
11. For each iteration $e \in [1, \tau]$,
    - ◇ Compute $\boldsymbol{s}_0^{(e)} = \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \boldsymbol{x}_j$
    - ◇ Sample seeds $\theta^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \mathsf{mseed})$ from $\{0,1\}^{\lambda}$
    - ◇ Compute $(\theta_i^{(e)})_{i\in[1,N]} \longleftarrow \mathsf{TreePRG}(\mathsf{salt}, \theta^{(e)})$
    - ◇ Sample $\boldsymbol{v}_i^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_1^{(e)})$ from $\mathbb{F}_q^n$
    - ◇ Compute $\boldsymbol{s}_1^{(e)} = \pi_1^{(e)}[\boldsymbol{s}_0^{(e)}] + \boldsymbol{v}_1^{(e)}$ // `We use the saved` $\pi_1^{(e)}$.
    - ◇ Compute $h_2.\mathsf{state} = \mathsf{H.update}(h_2.\mathsf{state}, \boldsymbol{s}_1^{(e)})$.
    - ◇ For each party $i \in [2, N]$,
      - Sample $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)}) \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_i^{(e)})$ from $\mathcal{S}_n \times \mathbb{F}_q^n$
      - Compute $\boldsymbol{s}_i^{(e)} = \pi_i^{(e)}[\boldsymbol{s}_{i-1}^{(e)}] + \boldsymbol{v}_i^{(e)}$
      - Compute $h_2.\mathsf{state} = \mathsf{H.update}(h_2.\mathsf{state}, \boldsymbol{s}_i^{(e)})$.

Step 4: Second Challenge

12. Compute $h_2 = \mathsf{H}_2.\mathsf{final}(h_2.\mathsf{state}, \mathsf{H}_2)$
13. Sample $(\alpha^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1, N])^{\tau}$

Step 5: Second Response

14. For each iteration $e \in [1, \tau]$,
   - $\diamond$ Compute $\boldsymbol{s}_0^{(e)} = \sum_{j \in [1,t]} \boldsymbol{\kappa}_j^{(e)} \cdot \boldsymbol{x}_j$
   - $\diamond$ Sample seeds $\theta^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \mathsf{mseed})$ from $\{0,1\}^\lambda$
   - $\diamond$ Compute $(\theta_i^{(e)})_{i \in [1,N]} \longleftarrow \mathsf{TreePRG}(\mathsf{salt}, \theta^{(e)})$
   - $\diamond$ Sample $\boldsymbol{v}_i^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_1^{(e)})$ from $\mathbb{F}_q^n$
   - $\diamond$ Compute $\boldsymbol{s}_1^{(e)} = \pi_1^{(e)}[\boldsymbol{s}_0^{(e)}] + \boldsymbol{v}_1^{(e)}$ // `We use the saved` $\pi_1^{(e)}$.
   - $\diamond$ If $\alpha^{(e)} > 1$, for each party $i \in [2, \alpha^{(e)}]$,
     - Sample $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)}) \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_i^{(e)})$ from $\mathcal{S}_n \times \mathbb{F}_q^n$
     - Compute $\boldsymbol{s}_i^{(e)} = \pi_i^{(e)}[\boldsymbol{s}_{i-1}^{(e)}] + \boldsymbol{v}_i^{(e)}$
   - $\diamond$ Compute $\boldsymbol{z}_1^{(e)} = \boldsymbol{s}_\alpha^{(e)}$
   - $\diamond$ If $\alpha^{(e)} \neq 1$, $z_2^{(e)} = (\pi_1^{(e)} \,||\, (\theta_i^{(e)})_{i \in [1,N] \setminus \alpha^{(e)}})$, compute $\mathsf{cmt}_{1,\alpha^{(e)}}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \alpha^{(e)}, \theta_{\alpha^{(e)}}^{(e)})$
   - $\diamond$ If $\alpha^{(e)} = 1$, $z_2^{(e)} = (\theta_i^{(e)})_{i \in [1,N] \setminus \alpha^{(e)}}$, compute $\mathsf{cmt}_{1,\alpha^{(e)}}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$
   - $\diamond$ Compute $\mathsf{rsp}^{(e)} = (\boldsymbol{z}_1^{(e)}, z_2^{(e)}, \mathsf{cmt}_{1,\alpha^{(e)}}^{(e)})$
15. Compute $\sigma = (\mathsf{salt}, h_1, h_2, (\mathsf{rsp}^{(e)})_{e \in [1,\tau]})$

Figure 5: Streamlined PERK - Sign algorithm

3.2 Verify Algorithm

In this section, we present several optimizations reducing the memory footprint of the Verify algorithm. Some of these optimizations are presently concisely as they are the counterpart of the optimizations described in Section 3.1.1 in the verification setting.

Streamlining seed trees sampling and usage. The Step 2 of the verification algorithm involves generating and storing seed trees in memory for all iterations. Instead, a streamlined technique akin to the signing algorithm detailed in Section 3.1 is adopted hence seed trees are computed dynamically in each iteration.

Streamlining π_i's and v_i's sampling and usage. As in the signing algorithm, one can streamline the generation and use of permutation and vector pairs $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)})$. Specifically, one generate a single pair for each iteration (e) thus significantly reducing the memory footprint. The pairs are generated immediately before their usage, precisely for computing the vectors $\boldsymbol{s}_i^{(e)}$, and are promptly removed from memory after.

Streamlining commitments and hash values computation. As outlined in Section 2.2, PERK involves the computation and storage of commitments $(\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{i \in [1,N]})$. We consider a comparable strategy to the aforementioned signing algorithm aiming to streamline their computation and their application in the computation of $\bar{h}_1$. For each iteration e and for each party i, and depending on the challenge $\alpha^{(e)}$, the computation of $\mathsf{cmt}_{1,i}^{(e)}$ involves the seed tree leaves along with data extracted from the signature σ. One should note that the commitments are absorbed in a reverse order with respect to the signing algorithm. Once the absorption of $\mathsf{cmt}_{1,i}^{(e)}$ is complete, the computation of $\mathsf{cmt}_1^{(e)}$ follows and is subsequently absorbed.

Computing $\bar{h}_2$. In our proposed approach, the computation of $\bar{h}_1$ and $\bar{h}_2$ occurs concurrently, diverging from the sequential process outlined in the PERK reference implementation. Despite this difference in computation, it's noteworthy that our approach yields the same

result as the reference implementation. In the reference implementation, $\boldsymbol{s}_i^{(e)}$ values are initially computed and stored. Then, $\bar{h}_1$ is computed, and finally, the precomputed $\boldsymbol{s}_i^{(e)}$ values are used in the calculation of $\bar{h}_2$. To facilitate parallel computation and eliminate the necessity of storing $\boldsymbol{s}_i^{(e)}$ values, we suggest initializing the $\bar{h}_2$ state with salt, m, pk, and h_1, all extractable from the signature σ. Moreover, the absorption of these values can be accomplished in two stages. Firstly, leveraging the fact that $\bar{h}_1$ and $\bar{h}_2$ share certain inputs, specifically the values salt, m, and pk, it is prudent to absorb these values initially. Subsequently, the obtained state can be utilized for the concurrent computation of both $\bar{h}_2$ and $\bar{h}_1$. This approach ensures efficient utilization of shared inputs in the calculation process. This modification allows us to absorb $\boldsymbol{s}_i^{(e)}$ values on-the-fly as they are generated, fulfilling our objective of abstaining from storing them.

Inputs: The public key $\mathsf{pk} = (\mathsf{pk\_seed}, (\boldsymbol{y}_j)_{j\in[1,t]})$, a signature σ and a message $m \in \{0,1\}^*$

Step 1: Parse signature

1. Sample $(\boldsymbol{H}, (\boldsymbol{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
2. Parse signature as $\sigma = (\mathsf{salt}, h_1, h_2, (\boldsymbol{z}_1^{(e)}, z_2^{(e)}, \mathsf{cmt}_{1,\alpha^{(e)}}^{(e)})_{e\in[1,\tau]})$
3. Recompute $(\kappa_j^{(e)})_{e\in[1,\tau],j\in[1,t]} \longleftarrow \mathsf{PRG}(h_1)$ from $(\mathbb{F}_q^t \setminus \mathbf{0})^\tau$
4. Recompute $(\alpha^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1,N])^\tau$

Step 2: Verification

5. $\bar{h}_1.\mathsf{state} = \mathsf{H.init}(\mathsf{salt})$
6. $\bar{h}_1.\mathsf{state} = \bar{h}_2.\mathsf{state} = \mathsf{H.update}(m, \mathsf{pk})$
7. Compute $\bar{h}_2.\mathsf{state} = \mathsf{H.update}(\bar{h}_2.\mathsf{state}, h_1)$
8. For each iteration $e \in [1,\tau]$,
   - $\diamond$ $(\theta_i^{(e)})_{i\in[1,N]\setminus\alpha^{(e)}} \leftarrow \mathsf{PartialTreePRG}(z_2)$
   - $\diamond$ For each party $i \in \{N, \cdots, 1\}$
     - If $i \neq 1$ and $i \neq \alpha^{(e)}$, compute $\mathsf{cmt}_{1,i}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, i, \theta_i^{(e)})$ and $\bar{h}_1.\mathsf{state} = \mathsf{H.update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_{1,i}^{(e)}))$
     - If $i = 1$ and $\alpha^{(e)} \neq 1$, compute $\mathsf{cmt}_{1,1}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$ and $\bar{h}_1.\mathsf{state} = \mathsf{H.update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_{1,1}^{(e)}))$
     - If $i = \alpha^{(e)}$, compute $\bar{h}_1.\mathsf{state} = \mathsf{H.update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_{1,\alpha}^{(e)}))$
   - $\diamond$ Compute $\boldsymbol{s}_0^{(e)} = \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \boldsymbol{x}_j$ and $\boldsymbol{s}_\alpha^{(e)} = \boldsymbol{z}_1^{(e)}$
   - $\diamond$ For each party $i \in [1,N]$,
     - If $i \neq \alpha^{(e)}$, Sample $(\pi_i^{(e)}, \boldsymbol{v}_i^{(e)}) \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_i^{(e)})$ from $\mathcal{S}_n \times \mathbb{F}_q^n$
     - If $i \neq \alpha^{(e)}$, compute $\boldsymbol{s}_i^{(e)} = \pi_i^{(e)}[\boldsymbol{s}_{i-1}^{(e)}] + \boldsymbol{v}_i^{(e)}$ and $\bar{h}_2.\mathsf{state} = \mathsf{H.update}(\bar{h}_2.\mathsf{state}, \boldsymbol{s}_i^{(e)})$
     - If $i = \alpha^{(e)}$, compute $\bar{h}_2.\mathsf{state} = \mathsf{H.update}(\bar{h}_2.\mathsf{state}, \boldsymbol{s}_\alpha^{(e)})$
   - $\diamond$ Compute $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \boldsymbol{H}\boldsymbol{s}_N^{(e)} - \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \boldsymbol{y}_j)$
   - $\diamond$ Compute $\bar{h}_1.\mathsf{state} = \mathsf{H.update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_1^{(e)})$,
9. Compute $\bar{h}_1 = \mathsf{H}_1.\mathsf{final}(\bar{h}_1.\mathsf{state}, \mathsf{H}_1)$
10. Compute $\bar{h}_2 = \mathsf{H}_2.\mathsf{final}(\bar{h}_2.\mathsf{state}, \mathsf{H}_2)$
11. Output accept if and only if $\bar{h}_1 = h_1$ and $\bar{h}_2 = h_2$.

Figure 6: Streamlined PERK - Verify algorithm

4 Implementation and Evaluation on Cortex M4

We consider two additional implementations tailored to Arm Cortex M4 devices, building upon the generic memory optimizations discussed in Section 3. In Section 4.1, we describe various optimizations aimed at improving the running time on Cortex M4. This approach is aligned with the PERK specification and successfully verifies the KAT for compatibility. In Section 4.2, we discuss an alternative approach relying on distinct techniques to compute permutation relation operations that achieve superior performances on the M4 device. Although instantiating PERK in this way preserves all the features and guarantees of the scheme, the resulting implementation does not replicate the KAT of the reference implementation. In Section 4.3, we present the experimental results of our implementations associated to Section 3, Section 4.1 and Section 4.2 respectively.

4.1 Implementation Compliant with Specifications

Optimizing permutation operations. While the PERK reference implementation employs `djbsort` for permutation composition, inversion and application to vector as explained in Section 2.1.2, we leverage the cache-less RAM architecture of Cortex M4 devices to enhance performances. Standard iterative algorithms for permutation operations offer greater speed, but their secret-dependent memory accesses introduce vulnerabilities to cache side-channel attacks. However, most Cortex M4 devices, including our target STM32F407, lack of RAM cache hardware [Lim, Sec. 3.6], allowing us to mitigate this threat. We then replaced the `djbsort`-based algorithms with their standard counterparts, achieving a notable speed-up in the Sign algorithm. Note that the standard method to permute a vector was already used in the Verification algorithm as constant-time is not a requirement in that context.

Optimizing `djbsort` for Cortex M4. To obtain a significant speed-up in the operation of sampling permutations at random, we employed a variant implementation of `djbsort` optimized for Cortex M4 devices from the work of [GFSL]. More specifically, this implementation builds upon the portable `djbsort/int32/portable4` implementation and translates the macro `int32_MINMAX` to its assembly equivalent on the M4 architecture while maintaining the original functionality.

Stack only permutation compression. The packing algorithm used for the *fast* parameters of PERK in the reference implementation do not require any modification for running on Cortex M4 devices. On the other hand, the ranking and unranking algorithms used for compressing permutations for the *short* parameters are memory and time consuming. For this reason, we dropped the `gmp` implementation used in the PERK reference implementation for an equivalent stack-only implementation that makes use of the `tiny-bignum-c`[2] library for multiple-precision integer operations. More specifically, `tiny-bignum-c` is a stack-only multiple-precision library characterized by a relatively small code size. Here, we customized the library to make use of the minimum amount of memory for every *short* parameters set of PERK. Furthermore, we enhanced big numbers multiplication by integrating it with the Karatsuba integer multiplication algorithm for big numbers[3]. Similarly to what is done with `gmp` in the PERK reference implementation, we also make use of a look-up table for storing the factorials $0!, 1!, ..., n!$ used in the ranking and unranking algorithms. However, while in the PERK reference implementation the factorial are stored in base-62 representation strings and then converted when reading, we can store the factorials directly in the `tiny-bignum-c` native format, thanks to the fact that in this case, big numbers are represented simply as `uint32_t` arrays.

[2] available at https://github.com/kokke/tiny-bignum-c.
[3] available at https://github.com/umnikos/tiny-bignum-c.

4.2 Faster Variant of PERK

The Fisher-Yates shuffle [FY38, Dur64] stands out as one of the most widely used methods for generating random permutations. However, in a cryptographic context, this algorithm is often dismissed because, in its naive and most efficient implementation, it uses secret-dependant memory accesses thus making cache side-channel attacks possible [BCMP24]. For the reasons explained in Section 4.1, Fisher-Yates is not vulnerable to these kind of attacks when implemented on our target devices. Hence, we propose here a variant of PERK that uses Fisher-Yates to sample random permutations at the place of the `djbsort`-based technique (see Section 2.1.2).

More specifically, we implemented the variant of Fisher-Yates outlined in Algorithm 1. This algorithm comprises two main subroutines: sampling random integers from decreasing-length intervals (Line 2) and shuffling (Line 4). While the latter is straightforward to implement, the former demands careful considerations. Sampling random integer from an interval whose length is not a power of two might introduce unwanted bias to the algorithm if not implemented correctly. To address this, we adopted the approach recommended in [BCMP24] and employed the algorithm proposed by Lemire [Lem19].

Notice that, for a given seed, Algorithm 1 produces a different permutation than the one sampled using `djbsort` as explained in Section 2.1.2. Hence, this variant of PERK will not produce and pass the official KAT of PERK.

Algorithm 1: Fisher-Yates algorithm for sampling random permutations.

Input: A positive integer n
Output: A random permutation π from $\mathcal{S}_n$

1 Initialize lists ℓ, π of length n;
2 **for** $i \leftarrow 0$ **to** $n-1$ **do**
3 $\quad \ell[i] \xleftarrow{\$} \{0, \ldots, i\}$;
4 **for** $i \leftarrow 0$ **to** $n-1$ **do**
5 $\quad \pi[i] = \pi[\ell[i]]$;
6 $\quad \pi[\ell[i]] = i$;
7 **return** π

4.3 Experimental Results

The approaches described in Section 3, Section 4.1 and Section 4.2 have all been implemented on a Cortex M4 device. Hereafter, we provide an evaluation of the resulting implementations by reporting in Tables 8, 9 and 10 their stack consumption and CPU Cyles count during key generation, signing and verification for each NIST security level. We compare our implementations to the PERK reference implementation [ABB+23].

Our benchmark uses the STM32F407 discovery board that is also used by the testing and benchmarking framework `pqm4` [KPR+]. The STM32F407 board is equipped with 192 kB of memory, 1 MB of flash, and can operate at frequencies of up to 168 MHz. Our build and performance evaluation configuration rely on `pqm4`, with benchmarks conducted at a frequency of 24 MHz to minimize flash memory related wait cycles. For each parameter set, the results have been obtained by computing the average from 20 random instances. In particular, all cycle counts were produced with the compiler `arm-none-eabi-gcc` in version 13.2.rel1 with the default optimization flags specified by the `pqm4` build framework.

One can see that our implementations require between 24 kB and 82 kB of stack memory depending on the considered NIST security level. One should note that we have not used

the memory vs performance trade-off that exploits all the available memory (see "Balancing memory and performances using s_i" in Section 3.1.2) as stack consumption was prioritized for this benchmark. Our implementations reduce the amount of required memory by a factor 12 to 65 depending on the considered parameter set. As expected, such a reduction is achieved at the cost of performance. Indeed, our generic implementation from Section 3 is approximately 15 times slower for signing and 9 times slower for verification compared to the PERK reference implementation. Similarly, our implementation compliant with PERK specification and KAT from Section 4.1 is approximately 8 times slower than PERK reference implementation. Interestingly, our faster variant from Section 4.2 can improve the performances by up to 12% depending on the considered instance.

Table 8: Performance benchmarks for PERK-I.

PERK Instance		**Stack Consumption**			**CPU Cycles**		
		Keygen	Sign	Verify	Keygen	Sign	Verify
PERK-I fast3	Ref.[ABB+23]	10 kB	307 kB	307 kB	0.1 M	21 M	8.9 M
	Section 3	7 kB	24 kB	20 kB	0.6 M	319 M	82 M
	Section 4.1	7 kB	24 kB	20 kB	0.5 M	165 M	76 M
	Section 4.2	7 kB	24 kB	20 kB	0.5 M	155 M	72 M
PERK-I fast5	Ref.[ABB+23]	11 kB	299 kB	299 kB	0.1 M	21 M	8.5 M
	Section 3	9 kB	25 kB	21 kB	0.8 M	312 M	80 M
	Section 4.1	9 kB	25 kB	21 kB	0.6 M	158 M	73 M
	Section 4.2	9 kB	25 kB	21 kB	0.6 M	147 M	69 M
PERK-I short3	Ref.[ABB+23]	10 kB	1.49 MB	1.49 MB	0.8 M	112 M	48 M
	Section 3	7 kB	27 kB	25 kB	0.6 M	1 760 M	502 M
	Section 4.1	7 kB	27 kB	25 kB	0.5 M	898 M	433 M
	Section 4.2	7 kB	27 kB	25 kB	0.5 M	848 M	411 M
PERK-I short5	Ref.[ABB+23]	11 kB	1.40 MB	1.40 MB	0.1 M	106 M	44 M
	Section 3	9 kB	28 kB	26 kB	0.8 M	1 668 M	470 M
	Section 4.1	9 kB	28 kB	26 kB	0.6 M	830 M	401 M
	Section 4.2	9 kB	28 kB	26 kB	0.6 M	779 M	377 M

Table 9: Performance benchmarks for PERK-III.

PERK Instance		Stack Consumption			CPU Cycles		
		Keygen	Sign	Verify	Keygen	Sign	Verify
PERK-III fast3	Ref.[ABB+23]	17 kB	671 kB	671 kB	0.2 M	50 M	21 M
	Section 3	14 kB	47 kB	41 kB	1.4 M	760 M	200 M
	Section 4.1	14 kB	47 kB	41 kB	1.3 M	394 M	184 M
	Section 4.2	14 kB	47 kB	41 kB	1.3 M	376 M	176 M
PERK-III fast5	Ref.[ABB+23]	18 kB	647 kB	647 kB	0.2 M	48 M	20 M
	Section 3	16 kB	48 kB	42 kB	1.7 M	739 M	193 M
	Section 4.1	16 kB	48 kB	42 kB	1.5 M	376 M	176 M
	Section 4.2	16 kB	48 kB	42 kB	1.5 M	356 M	168 M
PERK-III short3	Ref.[ABB+23]	17 kB	3.31 MB	3.31 MB	0.2 M	273 M	117 M
	Section 3	14 kB	51 kB	46 kB	1.4 M	4 312 M	1 252 M
	Section 4.1	14 kB	51 kB	46 kB	1.3 M	2 221 M	1 105 M
	Section 4.2	14 kB	51 kB	46 kB	1.3 M	2 141 M	1 067 M
PERK-III short5	Ref.[ABB+23]	18 kB	3.08 MB	3.08 MB	0.2 M	253 M	106 M
	Section 3	16 kB	51 kB	47 kB	1.7 M	4 028 M	1 165 M
	Section 4.1	16 kB	51 kB	47 kB	1.5 M	2 071 M	1 031 M
	Section 4.2	16 kB	51 kB	47 kB	1.5 M	1 960 M	986 M

Table 10: Performance benchmarks for PERK-V.

PERK Instance		Stack Consumption			CPU Cycles		
		Keygen	Sign	Verify	Keygen	Sign	Verify
PERK-V fast3	Ref.[ABB+23]	27 kB	1.14 MB	1.14 MB	0.3 M	100 M	45 M
	Section 3	25 kB	80 kB	69 kB	2.4 M	1 531 M	430 M
	Section 4.1	25 kB	80 kB	69 kB	2.3 M	823 M	394 M
	Section 4.2	25 kB	80 kB	69 kB	2.3 M	735 M	360 M
PERK-V fast5	Ref.[ABB+23]	29 kB	1.09 MB	1.09 MB	0.3 M	97 M	43 M
	Section 3	28 kB	80 kB	70 kB	2.9 M	1 465 M	410 M
	Section 4.1	28 kB	80 kB	70 kB	2.7 M	782 M	376 M
	Section 4.2	28 kB	80 kB	70 kB	2.6 M	694 M	340 M
PERK-V short3	Ref.[ABB+23]	27 kB	5.73 MB	5.73 MB	0.3 M	536 M	238 M
	Section 3	25 kB	82 kB	74 kB	2.5 M	8 646 M	2 691 M
	Section 4.1	25 kB	82 kB	74 kB	2.3 M	4 758 M	2 474 M
	Section 4.2	25 kB	82 kB	74 kB	2.3 M	4 265 M	2 280 M
PERK-V short5	Ref.[ABB+23]	29 kB	5.29 MB	5.29 MB	0.3 M	511 M	230 M
	Section 3	28 kB	82 kB	74 kB	2.9 M	7 996 M	2 488 M
	Section 4.1	28 kB	82 kB	74 kB	2.7 M	4 403 M	2 298 M
	Section 4.2	28 kB	82 kB	74 kB	2.6 M	3 905 M	2 104 M

References

[AAB+22a] Carlos Aguilar-Melchor, Nicolas Aragon, Paulo L. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit Flipping Key Encapsulation. NIST's Post-Quantum Cryptography Standardization Project (Round 4), https://bikesuite.org, 2022.

[AAB+22b] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. Hamming Quasi-Cyclic (HQC). NIST's Post-Quantum Cryptography Standardization Project (Round 4), https://pqc-hqc.org, 2022.

[ABB+22] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneverld, Peter Schwabe, and Bas Westerbaaan. SPHINCS+. NIST's Post-Quantum Cryptography Standardization Project (Selected Algorithms), https://sphincs.org/, 2022.

[ABB+23] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, and Jean-Pierre Tillich. PERK. NIST's Post-Quantum Cryptography Standardization of Additional Digital Signature Schemes Project (Round 1), https://pqc-perk.org/, 2023.

[ABD+22] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-Kyber. NIST's Post-Quantum Cryptography Standardization Project (Selected Algorithms), https://pq-crystals.org/kyber/, 2022.

[BCC+22] Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. NIST's Post-Quantum Cryptography Standardization Project (Round 4), https://classic.mceliece.org/nist.html, 2022.

[BCMP24] Alessandro Budroni, Isaac A. Canales-Martínez, and Lucas Pandolfo Perin. Sok: Methods for sampling random permutations in post-quantum cryptography. Cryptology ePrint Archive, Paper 2024/008, 2024. https://eprint.iacr.org/2024/008.

[Ber19] Daniel J. Bernstein. djbsort. https://sorting.cr.yp.to/, 2019. [Online; accessed 20-June-2023].

[BG23] Loïc Bidoux and Philippe Gaborit. Compact Post-quantum Signatures from Proofs of Knowledge Leveraging Structure for the PKP, SD and RSD Problems. In *Codes, Cryptology and Information Security (C2SI)*, pages 10–42. Springer, 2023.

[DKL+22] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-Dilithium. NIST's Post-Quantum Cryptography Standardization Project (Selected Algorithms), https://pq-crystals.org/dilithium/, 2022.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *ACM*, 7(7):420, 1964.

[FJR23] Thibauld Feneuil, Antoine Joux, and Matthieu Rivain. Shared Permutation for Syndrome Decoding: New Zero-Knowledge Protocol and Code-Based Signature. *Designs, Codes and Cryptography*, 91(2):563–608, 2023.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[FY38] Ronald A. Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver & Boyd, London, 6th ed. edition, 1938.

[GFSL] Décio Luiz Gazzoni Filho, Tomás S. R. Silva, and Julio López. Efficient isochronous fixed-weight sampling with applications to NTRU. To be published.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-Knowledge from Secure Multiparty Computation. In *Proceedings of the 39th annual ACM symposium on Theory of computing (STOC)*, 2007.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Lem19] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1), jan 2019.

[Lim] ARM Limited. ARM Cortex-M Programming Guide to Memory Barrier Instructions. https://developer.arm.com/documentation/dai0321/a/.

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. NIST's Post-Quantum Cryptography Standardization Project (Selected Algorithms), https://falcon-sign.info/, 2022.

[Pro23] The GNU Project. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/, 2023. [version 6.2.1].

[Sha90] Adi Shamir. An Efficient Identification Scheme Based on Permuted Kernels. In *Annual International Cryptology Conference (CRYPTO)*. Springer, 1990.