Enabling PERK and other MPC-in-the-Head Signatures 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. One category of the digital signatures submitted to the NIST Post-Quantum Cryptography Standardization Process for Additional Digital Signature Schemes comprises proposals constructed leveraging the MPC-in-the-Head (MPCitH) paradigm. Typically, this framework is characterized by the computation and storage in sequence of large data structures both in signing and verification algorithms, resulting in heavy memory consumption. While some research on the efficiency of these schemes on high-performance machines has been done, studying their performance and optimization on resource-constrained ones still needs to be explored. In this work, we aim to address this gap by (1) introducing a general method to reduce the memory footprint of MPCitH schemes and analyzing its application to several MPCitH proposed schemes in the NIST Standardization Process. Additionally, (2) we conduct a detailed examination of potential memory optimizations in PERK, resulting in a streamlined version of the signing and verification algorithms with a reduced memory footprint ranging from 22 to 85 KB, down from the original 0.3 to 6 MB. Finally, (3) we introduce the first implementation of PERK tailored for Arm Cortex M4 alongside extensive experiments and comparisons against reference implementations.

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 [NIS23]. Consequently, a multitude of new signature schemes have been submitted, initiating a fresh scrutiny process and fostering research in this area.

Among these new candidates, a notable subset utilizes the MPC-in-the-Head (MPCitH) paradigm [IKOS07], a framework facilitating the creation of zero-knowledge proofs via

Multiparty Computation (MPC) protocols. Such a subset is constituted by Biscuit [BKPV24], MIRA [ABB+23d], MiRitH [ARZV+23], MQOM [FR23a], PERK [ABB+23a], RYDE [ABB+23c], and SDitH [MFG+23]. Such a framework was already used by the digital signature Picnic [ZCD+20] from the previous NIST competition.

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 PQC implementations in scenarios where resource constraints are paramount.

Contribution This paper studies the methods to implement MPCitH signatures schemes on resource-constrained devices. Our contributions are threefold.

1. We present a general approach to reduce the memory footprint of MPCitH digital signatures significantly. We apply it analytically to Biscuit, MIRA, MiRitH, MQOM, and RYDE and analyze the impact on the stack usage of each of these schemes, paving the way for them to be suitable for memory-constrained devices.

2. For the digital signature PERK, we go through a deep investigation on the techniques to reduce the memory footprint of the protocol. We obtain and describe a streamlined version of PERK compliant with the official reference protocol. Our strategies significantly diminish PERK's memory requirements, reducing them from thousands of kilobytes to a maximum of approximately 85 KB for the highest security levels. Additionally, we introduce a memory-performance trade-off strategy, striking a balance between the two factors.

3. To the best of our knowledge, we produce the first constant-time implementation[1] of PERK tailored for resource-constrained devices, ensuring compatibility within the memory constraints of the standard STM32F407 discovery board. This achievement represents the first dedicated implementation of an MPCitH signature scheme for Cortex M4 devices that covers all security levels. We also provide extensive information to assess its efficiency and make comparisons against other signatures.

We remark that the streamlined version of PERK that we present is compliant with the original specifications, i.e., the scheme passes the official known-answer-tests (KATs). Table 1 gives an overview of the stack reduction of our implementation with respect to the PERK reference implementation for two parameter sets, see Section 4.2 for more details.

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

Algorithm	Implementation	Signing	Verification
`PERK-I-short3`	PERK Ref. [ABB+23a]	1.56 MB	1.56 MB
	This work	30.2 KB	26.1 KB
`PERK-V-short3`	PERK Ref. [ABB+23a]	6.01 MB	6.01 MB
	This work	85.6 KB	75.7 KB

Organization Section 2 introduces some helpful background to understand the manuscript. In Section 3, we present a general approach to streamline MPCitH signatures schemes and apply it to most of the MPCitH signatures from the NIST competition. Section 4 introduces our implementation of PERK for resource-constrained devices and the results

---

[1]Our implementation forks `pqm4` [KPR+] and can be accessed publicly at
https://github.com/Crypto-TII/perk-on-resource-constrained-devices.

of extensive experiments. Finally, we compare PERK against other protocols and give future research directions in Section 5.

2 Background

The sections in this manuscript related to the digital signature PERK follow the notation from [ABB$^+$23a]. In particular, for integers $a < b$, we use the notation $[a, b]$ to indicate the set $\{a, a+1, \ldots, b-1, b\}$, and $[a] := [0, a]$. Vectors are denoted with bold lower-case letters (e.g. $\boldsymbol{v}$), and matrices with bold upper-case letters (e.g. $\boldsymbol{M}$). We denote by $\mathcal{S}_n$ the set of all permutations of $[n]$, for a positive $n \in \mathbb{Z}$. For a power of a prime q, let $\mathbb{F}_q$ denote the finite field of order q. 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.

We present memory-related data in bytes (B), kilobytes (KB) and megabytes (MB). We present performance related data in thousands (K) or million (M) of CPU cycles.

## 2.1 Zero-knowledge Σ-protocols, MPC-in-the-Head and Hypercube

MPC-in-the-Head (MPCitH) is a framework introduced by Ishai et al. [IKOS07] that leverages techniques from multi-party computation (MPC) to construct zero-knowledge proofs. Since its introduction, numerous post-quantum digital signature schemes have been proposed, with seven candidates selected for the first round of the latest NIST call for additional post-quantum signatures [NIS23] being based on the MPCitH paradigm. In this section, we recall the necessary definitions regarding this framework. We refer the reader to [GMR85, GMW86, AF22] and [Lin20, EKR$^+$18] for more detailed sources of information on the topic.

Commitments. Commitments schemes are essential for zero-knowledge as they allow a prover P to commit a value without revealing it. Later, when they choose to disclose the value, the verifier V can trust that it has not been altered, preventing any changes of mind. Informally, a commitment scheme Com takes as input a message $m \in M$ and a randomness $\rho \in R$, where M and R are the message space and the randomness space, respectively. It outputs a commitment $c = Com(m, \rho)$. Revealing both ρ and m enables the opening of the commitment, which can be verified by confirming that $c = Com(m, \rho)$. There are two fundamental security properties for commitment schemes: hiding and binding. Hiding means that an adversary cannot view the message. Binding ensures that once the prover commits to a message and sends the commitment to the verifier, the prover cannot alter the commitment to a different message.

Zero-knowledge Σ-protocols. A zero-knowledge (ZK) protocol allows a prover P to convince a verifier V of the veracity of a public statement, without revealing any more information. A Σ-protocol, whose communication diagram is depicted in Figure 1, is an interactive zero-knowledge protocol where P proves to V the knowledge of a witness x for a public statement h with a relation R, and which satisfies the following properties.

- *Correctness*: if P is honest, then an honest verifier will always accept.

- *t-special soundness*: given t transcripts of the protocol relative to the same commitment, a witness x' with $R(h, x')$ can be extracted.

- *Special honest-verifier zero-knowledge*: assuming that V is honest, there exists a simulator S that simulates transcripts that are indistinguishable from the real ones.

The Fiat-Shamir transform [FS87] allows deriving a digital signature from a Σ-protocol by replacing the random challenge with the output of a hash function, making it non-interactive.
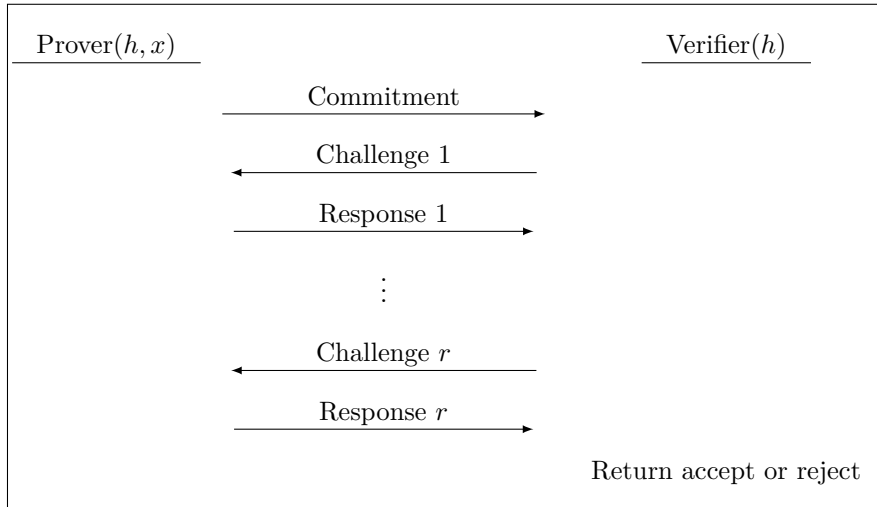
Figure 1: Interactive $(2r + 1)$-round Σ-protocol.

MPC. We recall some basic notions on Multiparty Computation (MPC). A secure MPC protocol allows N mutually distrusting parties $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_N$ to compute a public function f (usually represented as an arithmetic circuit) on their respective private input $x_1, \ldots, x_N$.

In this work, we require MPC protocols to be secure in the semi-honest model (or honest-but-curious), that is, the parties follow the protocol specifications but might try to derive additional information from the messages available to them. To be secure under such a model, an MPC protocol should satisfy:

- *Correctness*: the parties have learned the correct output at the end of the protocol.

- *Privacy*: the parties do not learn anything about the inputs of honest parties beyond what $f(x_1, \ldots, x_N)$ reveals.

The MPC-in-the-Head paradigm. We give here the general approach for obtaining a zero-knowledge proof via an MPC protocol. Let f be a public function and y be a public value. Assume that we want to prove the knowledge of a witness x such that $f(x) = y$ in zero-knowledge. Let $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_N$ be virtual parties. The prover P computes N shares $[\![x]\!]_1, [\![x]\!]_2, \ldots, [\![x]\!]_N$ of the witness x, and gives each $[\![x]\!]_i$ to the virtual party $\mathcal{P}_i$, for $i = 1, \ldots, N$.

The prover P then runs an MPC protocol "in the head" for evaluating f on the shares $[\![x]\!]_i$, resulting in one protocol transcript for each party. Then, P commits to all these transcripts and sends the commitments to the verifier V. With an interactive engagement, a subset of the committed transcripts gets selected and revealed by P. Finally, V checks the consistency and the validity of the revealed transcripts. The soundness error in the MPCitH paradigm is around $1/N$. Therefore, to achieve the target soundness error $2^{-\lambda}$, where λ is the security parameter, the prover P executes τ parallel repetitions of the protocol.

Hypercube. Hypercube is a technique introduced in [AGH$^+$23] to amplify the soundness of an MPC protocol that uses additive secret sharing. The main idea is to generate a N^D sharing of the initial witness and then use them to create D instances of the MPC protocol with N parties each. Within each instance, the N^D shares are partitioned into N subsets of D shares, and the shares within each subset are combined by summation. This process results in N secret shares representing the initial states of N parties, akin to a hypercube arrangement. The prover P commits to all N^D initial shares independently,

and the verifier V selects one index i from 1 to N^D. P then reveals all secret shares except for share i. This method ensures that one party's initial state remains undisclosed in each protocol instance due to the lack of a share for partial recombination. Overall, the hypercube approach achieves a soundness error of N^{-D} at the computational and communication cost of D parallel repetitions of the N party protocol, offering the benefits of virtually increasing the protocol parties without additional auxiliary states.

2.2 MPCitH digital signature schemes in the NIST competition

We list in Table 2 the MPCitH digital signatures candidates to the round 1 of the NIST post-quantum competition for additional signatures [NIS23] (excluding AIMer [KCC+23] as it relies on symmetric primitives). Given that this manuscript mainly focuses on the digital signature PERK, we give below an overview of it. We refer the reader to the relative specifications for details about the other signatures.

Table 2: List of the MPCitH digital signature schemes submitted to the round 1 of the NIST post-quantum competition for additional signatures. The third column says whether the signature scheme uses the hypercube framework or includes a hypercube variant. The fourth column gives the range of the signature sizes considering all proposed variants for all parameter sets.

Scheme	Underlying assumption	Hypercube	Signature size
Biscuit [BKPV24]	PowAff2	✗	4.7–27.3 KB
MIRA [ABB+23d]	MinRank	✓	5.6–27.6 KB
MiRitH [ARZV+23]	MinRank	✓	3.9–34.0 KB
MQOM [FR23a]	MQ	✓	6.3–29.9 KB
PERK [ABB+23b]	r-IPKP	✗	5.7–33.3 KB
RYDE [ABB+23c]	RSD	✓	5.9–29.1 KB
SDitH [MFG+23]	SD	✓	8.2–45.1 KB

2.2.1 PERK overview

PERK is built from a ZK 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 $\kappa = (\kappa_1, \ldots, \kappa_t) \in \mathbb{F}_q^t \setminus \boldsymbol{0}$.

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

For the three security levels specified by 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`.

3 Streamlining MPCitH Digital Signatures and Time/Memory Trade-off

The digital signature schemes submitted to the NIST competition for additional signatures [NIS23] constructed according to the MPCitH paradigm are Biscuit [BKPV24],

MIRA [ABB+23d], MiRitH [ARZV+23], MQOM [FR23a], PERK [ABB+23a], RYDE [ABB+23c], and SDitH [MFG+23].

This section outlines first our approach to reducing the memory footprint of MPCitH signatures in general. Then, we deepen into the application of our method to PERK, which is the principal signature scheme considered in this work and for which an implementation for resource-constrained devices is presented in Section 4. Finally, we detail how our method applies to some of the other MPCitH schemes of the NIST competition. We outline the steps for producing an implementation of these protocols compatible with resource-constrained devices.

General method. Typically, MPCitH digital signatures require the computation of $\tau \cdot N$ commitments in the early steps of the signing procedure. These are usually stored in memory to compute the first challenge. One can save memory by absorbing the inputs to the hash function for the first challenge *on-the-fly* while the commitments are computed. We refer to this method as "streamlining" the commitments and note that the reference implementations of MIRA and RYDE have already started implementing this idea partially by utilizing an incremental hashing. However, to generate the response, one of the N commitments for each of the τ rounds needs to be recomputed. For this reason, all MPCitH digital signature implementations we analyzed keep all the commitments in memory. Indeed, it is unknown until the last step of the signing procedure which commitment will be used. Our method consists of recomputing the required commitment for each round, allowing us not to store all of them and hence significantly decrease the memory footprint. Typically, for all N commitments but one (e.g., the 1-st in PERK or the N-th in MIRA) in each round, such re-computation comes with a relatively small overhead in the time complexity of the whole algorithm. However, one of the commitments is usually more expensive to recompute as the values from all other commitments are necessary for its computation. Hence, only in this case such commitment can be saved in memory to avoid a significant overload on the time complexity for each round, for a total of τ commitments. Overall, this approach reduces the memory footprint of the commitments by a factor of N, at the cost of a relatively small increase in the time complexity.

More optimizations tailored to each protocol are possible and will be discussed separately in the following subsections. We remark that all the following modifications to the original signing and verification algorithms are compatible with the original versions, i.e., the resulting protocols would produce and verify the provided official KATs.

3.1   Reducing the memory footprint of PERK

3.1.1   Stack usage in PERK

In the reference implementation 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 outlined in [ABB+23a, Figure 4]. 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 implementation leverage a data structure for storing these variables to enable efficient reuse. 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.

In verification, similar observations can be made. Step 1 the verification algorithm [ABB^{+}23a, Figure 5] 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 same data structure to store these variables.

We provide in Table 11 the stack usage of the official PERK reference implementations.

3.1.2 Reducing memory in signing

Streamlining seed trees sampling and usage. As explained in Section 3.1.1, the seed trees generated in Step 1 are preserved for later use in Step 5. Storing them requires $\tau \cdot (2N-1)\lambda/8$ 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 (salt, 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 3). However, this comes at the cost of needing to recalculate the seed trees twice, specifically in Steps 3 and 5.

Streamlining π_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 3. Indeed in the PERK implementation, permutations and vectors are stored using $\tau \cdot nN$ and $\tau \cdot 2nN$ bytes respectively while our implementationonly requires $3n$ bytes. However, this modification requires recomputation these values in Step 3 and 5.

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

Table 3: Stack memory required to store seed trees, permutations and vectors

Algorithm	Seed Trees		Permutations and Vectors	
	PERK	**This work**	**PERK**	**This work**
PERK-I-fast3	30.2 KB	1 KB	227 KB	237 B
PERK-I-fast5	28.2 KB	1 KB	223 KB	249 B
PERK-I-short3	163 KB	8.1 KB	1.21 MB	237 B
PERK-I-short5	147 KB	8.1 KB	1.14 MB	249 B
PERK-III-fast3	69.5 KB	1.5 KB	494 KB	336 B
PERK-III-fast5	65 KB	1.5 KB	478 KB	348 B
PERK-III-short3	380 KB	12.2 KB	2.66 MB	336 B
PERK-III-short5	343 KB	12.2 KB	2.49 MB	348 B
PERK-V-fast3	122 KB	2 KB	854 KB	438 B
PERK-V-fast5	114 KB	2 KB	820 KB	450 B
PERK-V-short3	670 KB	16.3 KB	4.59 MB	438 B
PERK-V-short5	605 KB	16.3 KB	4.26 MB	450 B

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 $\tau \cdot 2\lambda(N+1)$ bits down to 2λ bits (see Table 4) 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.

Table 4: 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

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 is important to highlight that in Step 5, we are not required to generate all N pairs; instead, one can halt the generation at index $\alpha^{(e)}$.

3.1.3 Reducing memory in verification

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 3.1.2 is adopted; hence, seed trees are computed dynamically in each iteration.

Streamlining π_i's and $\boldsymbol{v}_i$'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 generates 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 3.1.1, 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 generation 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 reference PERK. 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.

3.2 Reducing the memory footprint of other MPCitH digital signatures candidates

This section outlines the strategies to reduce memory usage for some of the MPCitH digital signatures submitted to the NIST competition. Alongside PERK, we study Biscuit, MIRA, RYDE, MQOM, and MiRitH. Note that MIRA and RYDE also use heap-memory allocations; however, we were able to isolate and reduce some stack-memory usage in these cases, obtaining a partial analysis. For the case of MiRitH, we discuss the potential

improvements given by our method over its already existing streamlined implementation optimized for Cortex M4 [ABB+24]. While SDitH was not directly addressed in our study because of its larger signature size exceeding 8 KB and heavy reliance on heap-memory allocations, we believe similar optimizations could be applied with further investigation. We also exclude from our analysis Picnic [ZCD+20] (from the previous NIST competition) and AIMer [KCC+23], both MPCitH based, as they rely on symmetric primitives.

In the following sections, we use the same notations and namings as in the corresponding reference specification submitted to the NIST competition or as in their updated version document for each protocol analyzed.

3.2.1 Reducing the memory footprint of Biscuit

Biscuit [BKPV24] relies on a structured variant of the multivariate quadratic equations problem for its security. It is designed with a five-round Zero-Knowledge Proof-of-Knowledge (ZKPoK), utilizing the MPCitH paradigm. Its signing algorithm has a much larger memory consumption than its verification algorithm. Hence, we focus our analysis on the signing algorithm.

Stack usage in Biscuit.
The Biscuit reference implementation has been integrated into the `pqm4` [KKPY24], with memory consumption benchmarks for the signing algorithm summarized in Table 5. Among the six parameter sets, three (`biscuit128f`, `biscuit192f`, `biscuit256f`) conform to the constraints set by the `pqm4` project's evaluation board. Upon scrutinizing the available source code in parallel to the diagram [BKPV24, Algorithm 12], it has come to our attention that in the signing algorithm, despite the efficient computation of h_1, where commitments are immediately absorbed post-calculation, all commitments of every round $(\mathsf{com}^{(e,i)})_{e \in [\tau], i \in [N]}$ are stored for later use in Phase 5. During Phase 5, depending on the value of $\bar{i}_e$, the signer integrates $\mathsf{com}^{(e,\bar{i}_e)}$ from the stored commitments into the signature.

Streamlining Biscuit. We propose not storing them in Phase 1 and instead recomputing the one corresponding to the $\bar{i}_e$ value in Phase 5. Additionally, we observe that for all parties $i \in [N]$, the commitment does not depend on auxiliary values; thus, recomputing the commitment in Step 5 does not pose a heavy computational burden. In Table 5, we outline the cost of saving commitments for each parameter set and the corresponding improvement achieved by applying our technique to the current implementation. Our analysis reveals that for the fast parameters, our strategy reduces stack usage by approximately 13–16%, while for short parameters, the reduction ranges from 14–17%. Notice that the improvement in overall stack usage achieved through commitment re-computation for Biscuit is slightly better than the savings we obtain for PERK. This difference primarily stems from PERK's higher overall stack usage than Biscuit, and Biscuit requires fewer commitments than PERK in the initial step of the signing algorithm.

3.2.2 Reducing the memory footprint of MQOM

MQOM is based on the hardness of the unstructured multivariate quadratic problem on a finite field. The reference implementation of MQOM makes use of heap-memory allocations, posing a challenge to our analysis of the impact of streamlining signing and verification algorithms. Nevertheless, a stack-memory-only variant implementation was created in [KKPY24], and it is available in a submodule of [KPR+]. Therefore, we base our analysis on this variant instead of the reference implementation. It's noteworthy to highlight that out of the 12 available parameter sets, only `MQOM-L1-gf31-fast` and `MQOM-L1-gf251-fast` are compatible with the evaluation board utilized in the `pqm4` project.

Table 5: The second column reports the stack usage of the Biscuit singing algorithm in the reference implementation. The third column reports the size of the commitments in the signing algorithm and the memory percentage saved when streamlining are reported in the last column.

Algorithm	Stack Usage	Commitments	Improvement
biscuit128f	138 KB	17.5 KB	12.7 %
biscuit128s	1.10 MB	148 KB	13.5 %
biscuit192f	266 KB	41.5 KB	15.6 %
biscuit192s	2.25 MB	369 KB	16.5 %
biscuit256f	478 KB	74.8 KB	15.7 %
biscuit256s	3.99 MB	656 KB	16.5 %

Stack usage in MQOM. Table 6^2 presents memory consumption benchmarks. Upon reviewing the available source code, we discovered that the storage of commitments significantly influences stack usage during signing and verification procedures. In Phase 1 of the signing algorithm (outlined in [FR23a, Algorithms 8 and 9]), the signer computes a sharing of the witness and proceeds to commit to each share. Specifically, in each iteration $e \in [1 : \tau]$, and for each party $i \in [1 : N]$, the tree PRG generates seeds seed[e][i]. The signer then commits to these shares by computing com[e][i] = Commit(salt, e, i, seed[e][i]) for every $i \in [1 : N-1]$, and com[e][N] = Commit(salt, e, N, seed[e][N] ∥ x_aux[e]), where each commitment is of size 2λ bits and x_aux[e] $\in \mathbb{F}_q^n$ represents the auxiliary value associated with the input sharing. Subsequently, the obtained commitments are used to compute the hash value h_1 in Phase 2. Then, in Phase 8, depending on the view opening ($i^\star[e] \in [1 : N]$), the signer selects $\{$com[e][$i^\star[e]$]$\}_{e \in [1:\tau]}$ from the saved commitments. It's worth noting that a data structure is employed to store a total of $\tau \cdot N$ commitments to facilitate their reuse and enhance performance. We are now focusing on the verification algorithm (outlined in Algorithms 10 and 11 [FR23a]), where similar observations can be made. Indeed, in Phase 1, the verifier recomputes $\tau \cdot (N-1)$ commitments and stores them for later use. Then, in Phase 3, these commitments are utilized to recompute the hash value h_1'. Similarly to the signing algorithm, the verify algorithm depends on a similar data structure for storing these commitments.

Streamlining the commitments. As discussed above, the storage of commitments constitutes the primary component consuming stack resources. Rather than storing these commitments, an alternative approach could involve absorbing them immediately after their generation, within Phase 1, instead of deferring their absorption until Phase 2. This strategy allows for efficient management of resources, as the commitments can be recomputed as needed in Phase 8. This approach is viable because the computation of h_1 within the hash function initiates by absorbing the message m, followed by the salt, and then the commitments. Therefore, streamlining the absorption process is feasible.

In Phase 8, the signer recomputes the commitments similarly to Phase 1, with two distinct scenarios. Firstly, if $i^\star[e] \neq N$, the signer directly computes the commitment using the corresponding seed without recalculating the secret shares. This step is omitted because these shares are unnecessary for the commitment computation and aren't needed afterward. Conversely, in the second scenario where $i^\star[e] = N$, it's possible to store the values of x_aux[e] from the initial Phase 1 execution. By preserving these values, one can regenerate the commitment for party N. Alternatively, if $i^\star[e] = N$ but the

---

[2]In [KPR$^+$], the QEMU simulator was used to measure memory performance up to 4 MiB. It is worth noting that parameters MQOM-L5-gf31-short and MQOM-L5-gf31-fast are omitted from the table due to their memory footprint exceeding the simulator's capacity.

x_aux[e] values aren't stored, the signer must rerun Phase 1 entirely to derive the required commitment. Approximating 1–6KB, depending on the security levels, is the total memory required to store these values, calculated as $\tau \cdot n \log(q)$ bits. This tradeoff reduces computational overhead during Phase 8 at the expense of increased memory utilization.

In Table 6, we provide the stack used to store the commitments for every parameter set of the scheme and the improvement achieved by implementing our streamlining strategy. We observe that our strategy improves the stack footprint by 6–14% for the parameters `MQOM-L1-gf31-fast` and `MQOM-L1-gf251-fast`, which are included in the evaluation platform of the `pqm4` project. Improvements for short parameters for the security level 1 are around 19–48%. This improvement paves the way for these parameters to fit in the evaluation board of the `pqm4` project.

Table 6: Second and third columns give the stack usage of the signing and verification algorithms from the reference implementation of MQOM. The fourth column gives the size of the commitments. The last two columns give the improvement in percentage over the whole stack consumption when streamlining.

Algorithm	Stack Usage		Commitments	Improvement	
	Signing	Verification		Signing	Verification
`MQOM-L1-gf31-short`	869 KB	555 KB	164 KB	18.9 %	29.6 %
`MQOM-L1-gf251-short`	666 KB	380 KB	181 KB	27.2 %	47.7 %
`MQOM-L1-gf31-fast`	613 KB	422 KB	35.9 KB	5.9 %	8.6 %
`MQOM-L1-gf251-fast`	400 KB	253 KB	34.9 KB	8.8 %	13.8 %
`MQOM-L3-gf31-short`	2.27 MB	1.78 MB	369 KB	16.3 %	20.8 %
`MQOM-L3-gf251-short`	1.89 MB	1.15 MB	369 KB	19.6 %	32.1 %
`MQOM-L3-gf31-fast`	2.15 MB	1.54 MB	78.4 KB	3.7 %	5.1 %
`MQOM-L3-gf251-fast`	1.29 MB	823 KB	79.9 KB	6.2 %	9.8 %
`MQOM-L5-gf251-short`	4.12 MB	2.54 MB	672 KB	16.4 %	26.5 %
`MQOM-L5-gf251-fast`	3.23 MB	2.17 MB	136 KB	4.3 %	6.3 %

3.2.3   Comparisons with the streamlined implementation of MiRitH

The authors of MiRitH implemented only two parameter sets optimized for Cortex M4 (`mirith_hypercube_Ia_fast` and `mirith_hypercube_Ib_fast`) [ABB+24]. However, their paper lacks comprehensive details on the streamlining techniques employed. Upon code inspection, it is evident that they partially integrate some concepts we propose, such as the streamlined computation of commitments $\mathsf{com}_i^{(\ell)}$ during the initial phase without persistent storage. However, the absorption of commitments $\mathsf{com}^{(\ell)}$ in step 15 in [ARZV+23, Figure 7] occurs separately, necessitating their storage in memory. Our analysis suggests that integrating these computations into Phase 1 could save memory — 1.3 KB for `mirith_hypercube_Ia_fast` and up to 4.6 KB for `mirith_hypercube_Va_fast` — indicating room for further improvement.

Moreover, MiRitH does not adopt our secondary technique of caching computationally expensive commitments to avoid redundant heavy calculations. For instance, in their approach, hashing data to derive the commitment com_{i^*}, for $i^* = N^D$, consumes approximately 3.7 KB for `mirith_hypercube_Ia_fast` and up to 15.7 KB for `mirith_hypercube_Va_fast`, introducing notable overhead.

Another divergence lies in MiRitH's Phase 1 ([ARZV+23, Figure 7]), which involves computing and storing seed trees (20 KB for `mirith_hypercube_Ia_fast` and up to 71 KB for `mirith_hypercube_Va_fast`). These trees are utilized in Phase 5 but could benefit

from our optimized methods for managing seed usage across different signing algorithm phases.

3.2.4 Reducing the memory footprint of MIRA and RYDE

MIRA [ABB+23d] is based on the minrank problem, while RYDE [ABB+23c] is based on syndrome decoding in the rank metric setting. Both schemes are instantiated using the same MPCitH framework based on a 5-round proof of knowledge with hypercube optimization. We will analyze them together because they share the same implementation structure. As already noticed in [KKPY24], both protocol implementations make use of heap-memory allocations, making it difficult for us to evaluate the impact of streamlining on the total memory footprint. For this reason, we only study the impact on the stack-memory occupied by the seeds and the commitments in signing and verification, believing that streamlining brings a similar impact to the heap-memory allocations relative to the commitments computation.

Stack usage in MIRA and RYDE. Upon analyzing the source code of the reference implementation of both schemes, it becomes evident that the storage of seeds and commitments heavily impacts stack usage during signing and verification. In Step 1 of the signing algorithm (outlined in [ABB+23d, Algorithm 21] and in [ABB+23c, Algorithm 20]), the signer generates a seed $\mathsf{seed}_i^{(e)}$ for each party $i \in [N]$, which is used for secret sharing the witness. Subsequently, the signer commits to these seeds for $i \in [N-1]$, and, for the party indexed by N, additional auxiliary information is committed along with its corresponding seed. The seeds $(\mathsf{seed}_i^{(e)})_{i \in [N]}$ and commitments $(\mathsf{cmt}_i^{(e)})_{i \in [N]}$ of each round are stored upon generation in Step 1 and later utilized in Step 5. Indeed, in Step 5, the signer uses the stored seeds during the reconstruction of secret sharing for the party $i^{\star(e)} \in [N-1]$. Notably, if $i^{\star(e)} = N$, the shares are not recomputed as they are previously stored. Additionally, depending on the value of $i^{\star(e)} \in [N]$, the signer incorporates the corresponding saved commitment $\mathsf{cmt}_{i^{\star(e)}}^{(e)}$ in the second response.

The verification algorithms are outlined in [ABB+23d, Algorithm 22] and [ABB+23c, Algorithm 21]. After parsing the public key and challenges in Steps 0 and 1, respectively, in Step 2, the verifier calculates the commitments $(\mathsf{cmt}_i^{(e)})_{i \in [N]}$ and uses them to compute $\bar{h}_1$. Despite the streamlined computation of $\bar{h}_1$, where the commitments are absorbed immediately after calculation, all these values are stored, resulting in significant stack consumption. Nevertheless, this issue of storing the commitments can be easily mitigated by refraining from storing these values, as they are not utilized in the subsequent steps of the verification algorithm.

In Table 7, we outline the stack utilization for storing seeds and commitments in both the signing and verification algorithms of MIRA and RYDE, along with the stack usage improvements achieved through the techniques described below.

Recomputing the commitments. In the signing algorithms of MIRA and RYDE, commitments are initially generated in Step 1 and utilized to compute h_1, after which they are stored for subsequent use in Step 5. Notably, these commitments are streamed during Step 1 to compute h_1 in the current implementations. Therefore, one can maintain the current procedure in Step 1 without storing the commitments. To reduce memory usage, commitments can be regenerated in Step 5 depending on the value of $i^{\star(e)} \in [N]$, except for the commitment corresponding to party N. This commitment is kept in Step 1 because its computation does not depend solely on a seed. Following this method effectively reduces the memory footprint, with the total storage cost for these commitments estimated at $2\lambda \cdot \tau$ bits, reducing it by a factor of N compared to the reference implementation.

Recomputing the seeds. In the signing algorithms of MIRA and RYDE, seeds are initially generated in Step 1 and utilized in Step 5 to reconstruct the secret sharing of

Table 7: The second and third columns indicate the stack usage for seeds and commitments in the signing algorithms of MIRA and RYDE, respectively. The fourth column shows the size of commitments during verification. To highlight the improvements of the optimization discussed in this section, the fifth and sixth columns detail the total necessary stack space when streamlining for storing seeds and commitments in the signing and verification algorithms, respectively.

Algorithm	Signing		Verification	This work	
	Seeds	Commitments	Commitments	Signing	Verification
MIRA-128f	14.4 KB	28.7 KB	28.7 KB	1.42 KB	0.9 KB
MIRA-128s	73.8 KB	148 KB	148 KB	4.68 KB	0.58 KB
MIRA-192f	31.5 KB	63 KB	63 KB	2.77 KB	2 KB
MIRA-192s	160 KB	320 KB	320 KB	7.5 KB	1.3 KB
MIRA-256f	55.3 KB	111 KB	111 KB	4.6 KB	3.5 KB
MIRA-256s	279 KB	558 KB	558 KB	10.4 KB	2.2 KB
RYDE-128f	15.4 KB	30.8 KB	30.8 KB	1.48 KB	0.96 KB
RYDE-128s	82 KB	164 KB	164 KB	4.74 KB	0.64 KB
RYDE-192f	33.8 KB	67.6 KB	67.6 KB	2.97 KB	2.2 KB
RYDE-192s	179 KB	357 KB	357 KB	7.6 KB	1.4 KB
RYDE-256f	60 KB	119 KB	119 KB	4.9 KB	3.8 KB
RYDE-256s	312 KB	623 KB	623 KB	10.7 KB	2.5 KB

party $i^{\star(e)} \in [N]$. To further reduce the memory usage, seeds can be discarded in Step 1 and then regenerated in Step 5 depending on the value of $i^{\star(e)} \in [N]$ and using the corresponding seed tree root ($\mathsf{seed}^{(e)}$).

4   Enabling PERK on memory-constrained devices

Building upon the memory optimization discussed in Section 3.1, we present the first implementation of streamlined PERK.[3] The optimization and techniques detailed in Section 4.1 are portable to any resource-constrained platform. However, we use some symbolic machine code specific to Arm Cortex M4, one of the chosen testing platforms, to optimize permutation-related operations. In Section 4.2, we perform and present extensive experiments to assess the impact of our modifications to the signing and verification algorithms.

4.1   Streamlined implementation Compliant with Specifications

Below, we detail some performance optimizations that we introduced to our implementation of streamlined PERK. The corresponding diagram specifications for the signing and verification algorithms are reported in Figure 2 and Figure 3 in the Appendix, respectively.

The implementation here introduced is compliant with PERK specifications version 1.1 [ABB+23b] and passes the KATs.

---

[3]The implementation is available at
https://github.com/Crypto-TII/perk-on-resource-constrained-devices.

4.1.1 Performance optimizations

Computing π^{-1}. In Step 1, we 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 $e \in [1, \tau]$, we compute and store this value. Doing so mitigates the need to invert π multiple times in Step 1, 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 8. Such an amount is relatively modest, constituting an interesting trade-off considering the substantial performance gains achieved.

Table 8: Stack memory required for storing π_1.

PERK-I-	Overhead	PERK-III-	Overhead	PERK-V-	Overhead
fast3	2.3 KB	fast3	5.1 KB	fast3	8.9 KB
fast5	2.3 KB	fast5	4.9 KB	fast5	8.5 KB
short3	1.5 KB	short3	3.4 KB	short3	5.9 KB
short5	1.4 KB	short5	3.2 KB	short5	5.5 KB

Optimizing permutation sampling and composition. In our 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 the constant-time software library `djbsort` [Ber19]. The 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 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.

4.1.2 Code optimizations tailored for Arm Cortex M4

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 [FSL24]. 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. The impact of such an optimization is significant and can be seen by looking at second and third row-block in Table 10. For instance, one

can see that the signing algorithm in the reference implementation benefits of around $1.45\times$ speed-up for `PERK-I-fast3` and `PERK-I-fast5` from the addition of this optimization.

Stack only permutation compression. The packing algorithm used for the `fast` parameters of PERK in the reference implementation does 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` [Pro23] implementation used in the PERK reference implementation for an equivalent stack-only implementation that makes use of the `tiny-bignum-c`[4] 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[5]. 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.

4.2   Experimental results

We present in this section the results of our experiments on the streamlined PERK implementation introduced in Section 4.1.

TARGET PLATFORM 1. Following the choice in the `pqm4` project [KPR⁺, KKPY24], we opted to utilize the Nucleo-L4R5ZI evaluation board as our resource-constrained testing and benchmarking platform. This board is equipped with an STM32L4R5ZI microcontroller with 2 MiB flash, 640 KiB SRAM, and a core clock frequency up to 120 MHz. Our build and performance evaluation configuration rely on `pqm4`, with benchmarks conducted at a frequency of 20 MHz to minimize flash memory related wait cycles. For each parameter set, the results have been obtained by computing the average from 10 random instances. In particular, all cycle counts were produced with the compiler `arm-none-eabi-gcc` version 13.2.rel1 with the default optimization flags specified by the `pqm4` build framework.

TARGET PLATFORM 2. To compare our implementation against PERK reference implementation, we utilized a machine with enough memory available to run the reference implementation for all parameter sets of PERK. We chose a machine running Ubuntu 22.04.2 LTS that has 64 GB of memory and an Intel® Core™ i9-13900K @ 3.00 GHz for which the Hyper-Threading and Turbo Boost features were disabled. The code has been compiled with `gcc` (version `11.4.0`) with `-funroll-loops` and `-march=native` optimization flags.

We begin reporting the performances in CPU cycles and the stack consumption of our implementation in Table 10, performed on TARGET PLATFORM 1. In addition, to evaluate our contribution, we report the CPU cycles and the stack usage of the reference implementation running on the same device for the parameter sets `PERK-I-fast3` and `PERK-I-fast5`, which are the only ones that we were able to run on such resource-constrained device. The measurements from a variant of the reference implementation

---

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

featuring the optimized `djbsort` algorithm for Arm Cortex M4 explained in Section 4.1.2 are also reported. This variant is considered when comparing the reference vs. the streamlined to ensure a fairer comparison. In all cases, we use the symmetric function provided by the `pqm4` framework. In addition, we run the same benchmarks on TARGET PLATFORM 2 to cover all parameter sets in the reference implementation and report the results in Table 11. This time, we used `gmp` to compress permutations for `short` parameter sets both in the reference and streamlined implementations to exclude a bias given by using two different libraries for compression.

Looking at Tables 10 and 11, one can see that the experiments give similar results on both testing platforms in terms of stack usage reduction and performance degradation when comparing the reference and the streamlined implementation. Streamlining the signing algorithm degrades the performance by a factor of around 1.6. On the other hand, the streamlined verification algorithm gives similar results to the reference one because of the optimization that we introduced for the computation of $\bar{h}_2$ (see Section 3.1.3). Note that such an optimization should apply to the reference implementation, too. However, improving this one is outside the scope of this paper.

Regarding the stack usage, the reduction obtained by our streamlined implementation is significant. In signing, the memory consumption is reduced between 11 and 14 times for `fast` parameters, and between 47 and 70 times for `short` parameters. In verification, the reduction is between 13 and 16 times for `fast` parameters, and between 44 and 80 times for the `short` parameters. The key-generation algorithm does not get modified when streamlining; hence, we do not measure any variantion between the two implementations.

Lastly, we report the code size of our implementation for TARGET PLATFORM 1 in Table 9, for completeness. We do not include the code size of the reference implementation as it is not able to run for TARGET PLATFORM 1 with all parameters. However, we can remark that the code size for our implementation is slightly increased, since we need to accommodate our optimizations.

Table 9: Code size in Bytes for different PERK parameters, measured using the `pqm4` framework on TARGET PLATFORM 1, excluding hashing and standard library functions.

Algorithm	Size	Algorithm	Size	Algorithm	Size
PERK-I-fast3	11717	PERK-III-fast3	12077	PERK-V-fast3	12129
PERK-I-fast5	11709	PERK-III-fast5	12017	PERK-V-fast5	12041
PERK-I-short3	24605	PERK-III-short3	24009	PERK-V-short3	31697
PERK-I-short5	24673	PERK-III-short5	24649	PERK-V-short5	32693

4.2.1 Time/memory trade-off by storing s_i.

Given a targeted memory limit, one can fine tune the implementation using the s_i in order to get the best possible performance with respect to the available memory. By storing the $(s_i^{(e)})_{e \in [1, \tau'], i \in [1, N]}$ values (where $\tau' \leq \tau$) after their computation in Step 3, one only needs to recompute them in Step 5 for the remaining $\tau - \tau'$ rounds thus avoiding $\tau' N$ computations of s_i values. In addition, keeping $s_0^{(e)}$ is unnecessary and thus its computation can be omitted once the $s_i^{(e)}$ values are stored. Table 13 shows the obtained trade-off for two parameter sets of PERK run on TARGET PLATFORM 1. For `PERK-I-fast3`, we tested $\tau' = 16$ to reflect the case of devices with 128 KB of RAM, and $\tau' = 30$ is the maximum achievable. For `PERK-I-short3`, we tested $\tau' = 5$ to reflect the case of devices with 256 KB of RAM, and $\tau' = 15$ is the maximum allowed on our TARGET PLATFORM 1. The cases $\tau' = 0$ stand for when no memory has been traded for efficiency and correspond to the

Table 10: Performance and memory benchmarks for streamlined PERK in millions (M) of CPU cycles and kilobytes (KB). The reported values are obtained by averaging the results from 10 runs on TARGET PLATFORM 1. The third row block refers to the reference implementation of PERK running a version of `djbsort` optimized for Arm Cortex M4.

	Algorithm	Stack Consumption			CPU Cycles		
		Keygen	Sign	Verify	Keygen	Sign	Verify
This work	PERK-I-fast3	7.70 KB	24.0 KB	20.7 KB	0.64 M	244 M	81.0 M
	PERK-I-fast5	9.03 KB	25.2 KB	21.8 KB	0.82 M	240 M	78.1 M
	PERK-I-short3	7.70 KB	27.8 KB	25.2 KB	0.65 M	1336 M	460 M
	PERK-I-short5	9.03 KB	28.6 KB	26.1 KB	0.82 M	1250 M	428 M
Ref. [ABB+23a]	PERK-I-fast3	7.73 KB	313 KB	313 KB	0.70 M	218 M	96.6 M
	PERK-I-fast5	9.03 KB	306 KB	305 KB	0.91 M	215 M	93.7 M
Ref. [ABB+23a] + djbsort [FSL24]	PERK-I-fast3	7.70 KB	313 KB	313 KB	0.65 M	150 M	82.6 M
	PERK-I-fast5	9.03 KB	306 KB	305 KB	0.82 M	147 M	79.8 M
This work	PERK-III-fast3	15.0 KB	47.7 KB	41.4 KB	1.50 M	581 M	195 M
	PERK-III-fast5	16.9 KB	48.8 KB	42.4 KB	1.81 M	558 M	187 M
	PERK-III-short3	15.0 KB	51.3 KB	46.7 KB	1.50 M	3265 M	1177 M
	PERK-III-short5	16.9 KB	51.9 KB	47.3 KB	1.82 M	3031 M	1099 M
This work	PERK-V-fast3	25.5 KB	80.3 KB	69.9 KB	2.60 M	1188 M	419 M
	PERK-V-fast5	28.1 KB	80.9 KB	70.6 KB	3.09 M	1134 M	398 M
	PERK-V-short3	25.5 KB	82.3 KB	74.8 KB	2.59 M	6746 M	2641 M
	PERK-V-short5	28.1 KB	82.1 KB	74.8 KB	3.07 M	6285 M	2457 M

Table 11: Performance and memory benchmarks for streamlined PERK in thousands (K) or millions (M) of CPU cycles and kilobytes (KB) or megabytes (MB). The reported values are obtained by averaging the results from 1000 runs on TARGET PLATFORM 2.

	Algorithm	Stack Consumption			CPU Cycles		
		Keygen	Sign	Verify	Keygen	Sign	Verify
This work	PERK-I-fast3	10.0 KB	26.5 KB	21.6 KB	80.2 K	36.2 M	10.9 M
	PERK-I-fast5	11.1 KB	27.4 KB	22.7 KB	98.6 K	35.3 M	10.6 M
	PERK-I-short3	10.0 KB	30.2 KB	26.1 KB	81.7 K	195.4 M	61.3 M
	PERK-I-short5	11.1 KB	30.8 KB	26.9 KB	99.2 K	184.1 M	57.2 M
Ref. [ABB+23a]	PERK-I-fast3	10.0 KB	314 KB	314 KB	80.8 K	22.0 M	10.8 M
	PERK-I-fast5	11.1 KB	306 KB	306 KB	99.4 K	21.7 M	10.4 M
	PERK-I-short3	10.0 KB	1.56 MB	1.56 MB	82.9 K	118 M	59.1 M
	PERK-I-short5	11.1 KB	1.46 MB	1.46 MB	99.6 K	113 M	55.5 M
This work	PERK-III-fast3	17.1 KB	50.1 KB	42.3 KB	183 K	86.0 M	27.1 M
	PERK-III-fast5	18.7 KB	50.8 KB	43.3 KB	210 K	83.0 M	25.3 M
	PERK-III-short3	17.1 KB	53.7 KB	47.6 KB	183 K	469 M	147 M
	PERK-III-short5	18.7 KB	53.9 KB	48.2 KB	210 K	442 M	138 M
Ref. [ABB+23a]	PERK-III-fast3	17.1 KB	687 KB	687 KB	183 K	51.9 M	25.5 M
	PERK-III-fast5	18.7 KB	663 KB	663 KB	211 K	50.9 M	24.6 M
	PERK-III-short3	17.1 KB	3.47 MB	3.47 MB	193 K	281 M	140 M
	PERK-III-short5	18.7 KB	3.22 MB	3.22 MB	226 K	265 M	130 M
This work	PERK-V-fast3	27.3 KB	82.6 KB	70.8 KB	315 K	176 M	56.7 M
	PERK-V-fast5	29.3 KB	82.1 KB	71.5 KB	349 K	171 M	54.2 M
	PERK-V-short3	27.3 KB	84.6 KB	75.7 KB	316 K	977 M	317 M
	PERK-V-short5	29.3 KB	83.3 KB	75.7 KB	353 K	900 M	292 M
Ref. [ABB+23a]	PERK-V-fast3	27.3 KB	1.19 MB	1.19 MB	326 K	109 M	56.7 M
	PERK-V-fast5	29.3 KB	1.14 MB	1.14 MB	370 K	105 M	53.8 M
	PERK-V-short3	27.3 KB	6.01 MB	6.01 MB	330 K	588 M	306 M
	PERK-V-short5	29.3 KB	5.55 MB	5.55 MB	375 K	549 M	284 M

values reported in Table 10. We do not report results for other parameter sets of PERK because, in these cases, considering the memory available on the TARGET PLATFORM 1, the allowed trade-off brings only a negligible improvement.

5   Comparisons and Future Directions

This section compares PERK against other MPCitH digital signatures from the NIST competition. We take the `pqm4` project as a reference for the performance of some protocol implementations on resource-constrained devices. In particular, given that the experiments in [KKPY24] have been performed on a platform identical to TARGET PLATFORM 1 with analogous settings, we directly compare our numbers with the numbers reported in [KKPY24, Tables 2 and 3].

PERK vs. MiRitH. Beside our work, MiRitH is the only signature among the MPCitH NIST candidates for which an implementation optimized for Arm Cortex M4 was produced [ABB+24], even if for two parameter sets only: `mirith_hypercube_Ia_fast` and `mirith_hypercube_Ib_fast`. Hence, it is the only protocol that allows a fair comparison against our implementation of PERK. We report in Table 12 the stack consumption and the CPU cycles for these parameter sets, together with the results from Table 10 of the corresponding `fast` PERK instances. While the comparison on performance favors MiRitH in signing and partially in verification, PERK is faster in key generation and has a smaller memory footprint overall.

PERK vs. Biscuit/MQOM. Bisquit and MQOM have been tested on Arm Cortex M4 devices in [KKPY24]. However, their implementations are not optimized for such memory-constrained devices. Indeed, for both protocols, only a selection of the parameter sets fits the requirement of the resource-constrained test platform, similar to the reference implementation of PERK. Hence, comparing their performances against the ones from this work would not produce a fair comparison. For this reason, we do not report the numeric values in a table to prevent the reader from drawing incorrect conclusions. Nevertheless, looking at the values in [KKPY24, Table 2 and 3], we can infer the following preliminary information. Even if our implementation of PERK is streamlined, and hence expected to be slower by design, it is still significantly faster in verification than reference MQOM. This might change if some Arm Cortex M4-specific optimizations are introduced to MQOM. On the other hand, MQOM is substantially faster in signing. However, such a gap might be reduced when streamlining MQOM. As expected, streamlined PERK compares favorably against reference MQOM in terms of memory footprint. Regarding the reference implementation of Biscuit, our PERK implementation is faster and lighter. We leave as a future research direction to produce streamlined implementations of Biscuit and MQOM (following the directives from Section 3) and make a fair comparison against PERK.

Other MPCitH schemes. A comparison against MIRA, RYDE, and SDitH is not possible at the moment. To do so, one should ideally produce a refactoring of their reference implementations to drop heap-memory dependence and then apply streamlining techniques as described in Section 3. We leave this work as a future direction.

In addition, recent works have proposed newer paradigms that improve upon MPCitH, called respectively Threshold-Computation-in-the-Head [FR23c, FR23b] and VOLE-in-the-Head [BBD+23, BBdSG+23]. Exploring avenues to streamline these emerging frameworks is a promising direction for future investigation.

Table 12: Comparison between the stack consumption and performance (in million of cycles) of PERK and MiRitH reference implementations optimized for Arm Cortex M4 on TARGET PLATFORM 1. For MiRitH, the numbers are extrapolated from [KKPY24, Tables 2 and 3].

	Algorithm	Stack Consumption			CPU Cycles		
		Keygen	Sign	Verify	Keygen	Sign	Verify
Section 4	`PERK-I-fast3`	7.70 KB	24.0 KB	20.7 KB	0.64 M	244 M	81.0 M
	`PERK-I-fast5`	9.03 KB	25.1 KB	21.8 KB	0.82 M	240 M	78.1 M
[ABB+24]	`mirith_hypercube_Ia_fast`	10.0 KB	75.1 KB	20.4 KB	1.00 M	59.0 M	53.6 M
	`mirith_hypercube_Ib_fast`	18.7 KB	94.7 KB	30.5 KB	1.88 M	83.8 M	78.1 M

Appendix

Streamlined PERK diagrams

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 $(\mathbb{F}_q^t \setminus \boldsymbol{0})^{\tau}$

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 $\boldsymbol{s}_0^{(e)} = \sum_{j \in [1,t]} \kappa_j^{(e)} \cdot \boldsymbol{x}_j$

$\diamond$ 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 2: Streamlined PERK - Sign algorithm

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 $(\mathbb{F}_q^t \setminus \boldsymbol{0})^\tau$

4. Recompute $(\alpha^{(e)})_{e \in [1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1,N])^\tau$

Step 2: Verification

5. $\bar{h}_1.\mathsf{state} = \mathsf{H}.\mathsf{init}(\mathsf{salt})$

6. $\bar{h}_1.\mathsf{state} = \bar{h}_2.\mathsf{state} = \mathsf{H}.\mathsf{update}(m, \mathsf{pk})$

7. Compute $\bar{h}_2.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_2.\mathsf{state}, h_1)$

8. For each iteration $e \in [1,\tau]$,

   ◇ $(\theta_i^{(e)})_{i \in [1,N] \setminus \alpha^{(e)}} \leftarrow \mathsf{PartialTreePRG}(z_2)$

   ◇ 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 $\bar{h}_1.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_{1,i}^{(e)})$

      - 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 $\bar{h}_1.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_{1,1}^{(e)})$

      - If $i = \alpha^{(e)}$, compute $\bar{h}_1.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_{1,\alpha}^{(e)})$

   ◇ Compute $\boldsymbol{s}_0^{(e)} = \sum_{j \in [1,t]} \kappa_j^{(e)} \cdot \boldsymbol{x}_j$ and $\boldsymbol{s}_\alpha^{(e)} = \boldsymbol{z}_1^{(e)}$

   ◇ 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 $\bar{h}_2.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_2.\mathsf{state}, \boldsymbol{s}_i^{(e)})$

      - If $i = \alpha^{(e)}$, compute $\bar{h}_2.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_2.\mathsf{state}, \boldsymbol{s}_\alpha^{(e)})$

   ◇ Compute $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \boldsymbol{H}\boldsymbol{s}_N^{(e)} - \sum_{j \in [1,t]} \kappa_j^{(e)} \cdot \boldsymbol{y}_j)$

   ◇ Compute $\bar{h}_1.\mathsf{state} = \mathsf{H}.\mathsf{update}(\bar{h}_1.\mathsf{state}, \mathsf{cmt}_1^{(e)})$,

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 3: Streamlined PERK - Verify algorithm

Tables

Table 13: Time/memory tradeoff in the sign algorithm by storing some of the $\boldsymbol{s}_i$. The experiment were run on the TARGET PLATFORM 1. Performance related data is given in millions (M) of CPU cycles. The reported values are obtained by averaging the results from 10 runs.

Algorithm	τ'	Stack Consumption	CPU Cycles
PERK-I-fast3	0	24.0 KB	244 M
	16	105 KB	229 M
	30	176 KB	214 M
PERK-I-short3	0	27.8 KB	1336 M
	5	230 KB	1285 M
	15	634 KB	1128 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 Rijneveld, Peter Schwabe, and Bas Westerbaaan. SPHINCS+. NIST's Post-Quantum Cryptography Standardization Project (Selected Algorithms), `https://sphincs.org/`, 2022.

[ABB+23a]   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.

[ABB+23b]   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. Specifications (Version 1.1), `https://pqc-perk.org/assets/downloads/PERK_2023_10_16.pdf`, 2023.

[ABB+23c]   Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vinçotte. RYDE. NIST's Post-Quantum Cryptography Standardization of Additional Digital Signature Schemes Project (Round 1), `https://pqc-ryde.org/`, 2023.

[ABB+23d]   Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, and Jean-Pierre Tillich. MIRA. NIST's Post-Quantum Cryptography Standardization of Additional Digital Signature Schemes Project (Round 1), `https://pqc-mira.org/`, 2023.

[ABB+24]   Gora Adj, Stefano Barbero, Emanuele Bellini, Andre Esser, Luis Rivera-Zamarripa, Carlo Sanna, Javier Verbel, and Floyd Zweydinger. Mirith: Efficient post-quantum signatures from minrank in the head. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):304–328, Mar. 2024.

[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.

[AF22]   Thomas Attema and Serge Fehr. Parallel repetition of $(k_1, \ldots, k_\mu)$-special-sound multi-round interactive proofs. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 415–443. Springer, Heidelberg, August 2022.

[AGH+23]   Carlos Aguilar Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The return of the SDitH. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 564–596. Springer, Heidelberg, April 2023.

[ARZV+23]   Gora Adj, Luis Rivera-Zamarripa, Javier Verbel, Emanuele Bellini, Stefano Barbero, Andre Esser, Carlo Sanna, and Floyd Zweydinger. MiRitH. NIST's Post-Quantum Cryptography Standardization of Additional Digital Signature Schemes Project (Round 1), https://pqc-mirith.org/, 2023.

[BBD+23]   Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 581–615. Springer, Heidelberg, August 2023.

[BBdSG+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. Faest. NIST's Post-Quantum Cryptography Standardization of Additional Digital Signature Schemes Project (Round 1), https://faest.info/, 2023.

[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.

[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.

[BKPV24]   Luk Bettale, Delaram Kahrobaei, Ludovic Perret, and Javier Verbel. Biscuit. Biscuit: Shorter MPC-based Signature from PoSSo (Version February 11, 2024), https://drive.google.com/file/d/1IYJVh0UJ6IERl331mlmq0repxPnt7WlS/view, 2024.

[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.

[EKR+18]   David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[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.

[FR23a]   Thibauld Feneuil and Matthieu Rivain. MQOM: MQ on my Mind Algorithm Specifications and Supporting Documentation (Version 1.0). https://www.mqom.org/docs/mqom-v1.0.pdf, 2023.

[FR23b]   Thibauld Feneuil and Matthieu Rivain. Threshold Computation in the Head: Improved Framework for Post-Quantum Signatures and Zero-Knowledge Arguments. *Cryptology ePrint Archive, Report 2023/1573*, 2023.

[FR23c]   Thibauld Feneuil and Matthieu Rivain. Threshold Linear Secret Sharing to the Rescue of MPC-in-the-Head. In *International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt)*, 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.

[FSL24]   Décio Luiz Gazzoni Filho, Tomás S. R. Silva, and Julio López. Efficient isochronous fixed-weight sampling with applications to ntru. Cryptology ePrint Archive, Paper 2024/548, 2024. https://eprint.iacr.org/2024/548.

[GMR85]   Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMW86]   Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract). In *27th FOCS*, pages 174–187. IEEE Computer Society Press, October 1986.

[IKOS07]   Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[KCC+23]   Seongkwang Kim, Jihoon Cho, Mingyu Cho, Jincheol Ha, Jihoon Kwon, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Sangyub Lee, Dukjae Moon, Mincheol Son, and Hyojin Yoon. Aimer. NIST's Post-Quantum Cryptography Standardization of Additional Digital Signature Schemes Project (Round 1), https://aimer-signature.org/, 2023.

[KKPY24]   Matthias J. Kannwischer, Markus Krausz, Richard Petri, and Shang-Yi Yang. pqm4: Benchmarking nist additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Paper 2024/112, 2024. https://eprint.iacr.org/2024/112.

[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.

[Lin20]   Yehuda Lindell. Secure multiparty computation (MPC). Cryptology ePrint
             Archive, Report 2020/300, 2020. https://eprint.iacr.org/2020/300.

[MFG+23]   Carlos Aguilar Melchor, Thibauld Feneuil, Nicolas Gama, Shay Gueron,
             James Howe, David Joseph, Antoine Joux, Edoardo Persichetti, Tovohery H.
             Randrianarisoa, Matthieu Rivain, and Dongze Yue. SDitH. NIST's Post-
             Quantum Cryptography Standardization of Additional Digital Signature
             Schemes Project (Round 1), https://sdith.org/, 2023.

[NIS23]   NIST.   Post-quantum   cryptography: Digital   signature
             schemes.   https://csrc.nist.gov/Projects/pqc-dig-sig/
             round-1-additional-signatures, 2023.

[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 Cryptogra-
             phy 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].

[ZCD+20]   Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder,
             Claudio   Orlandi,   Sebastian   Ramacher,   Christian   Rechberger,
             Daniel Slamanig, JonathanKatz, Xiao Wang, Vladmir Kolesnikov,
             and Daniel Kales.   Picnic. technical report.   NIST's Post-
             Quantum   Cryptography   Standardization   Project   (Round   3),
             https://csrc.nist.gov/Projects/post-quantum-cryptography/
             post-quantum-cryptography-standardization/round-3-submissions,
             2020.