Quasi-Optimal Permutation Ranking and Applications to PERK

Slim Bettaieb[1], Alessandro Budroni[1],
Marco Palumbi[1], and Décio Luiz Gazzoni Filho[2,3]

[1] Technology Innovation Institute, Abu Dhabi, UAE,
{slim.bettaieb,alessandro.budroni,marco.palumbi}@tii.ae
[2] Instituto de Computação, Universidade Estadual de Campinas (UNICAMP), Campinas, Brazil,
[3] Dept. of Electrical Engineering, State University of Londrina, Londrina, Brazil,
decio.gazzoni@ic.unicamp.br, dgazzoni@uel.br

Abstract. A ranking function for permutations maps every permutation of length n to a unique integer between 0 and $n!-1$. For permutations of size that are of interest in cryptographic applications, evaluating such a function requires multiple-precision arithmetic. This work introduces a quasi-optimal ranking technique that allows us to rank a permutation efficiently without needing a multiple-precision arithmetic library. We present experiments that show the computational advantage of our method compared to the standard lexicographic optimal permutation ranking. As an application of our result, we show how this technique improves the signature sizes and the efficiency of PERK digital signature scheme.

Keywords: Efficient Compression · PERK · Permutation ranking · Post-quantum Cryptography.

1 Introduction

Permutations have been employed in Cryptography in various scenarios, including the construction of block ciphers and hash functions. Some of the fundamental hard problems in Post-quantum Cryptography, e.g., Permutation Code Equivalence [Leo82] and Permuted Kernel Problem [Sha90], guarantee a hardness based on the application of a secret permutation on certain geometrical or algebraic structures. Motivated by the ongoing standardization initiatives led by the authorities of different countries, such as NIST [NIS17,NIS23] and CACR [fCR20], several new cryptosystems featuring public-key encryption, key-encapsulation mechanisms and digital signatures have been proposed. Some of these introduced new technical and engineering challenges related to permutations such as efficient permutation sampling and composition. This work addresses the challenge of efficiently compressing a permutation.

The PERK signature scheme [ABB+23a] is a new digital signature whose security relies on a variant of the Permuted Kernel Problem, and it is currently

one of the candidates for round 1 of the NIST post-quantum competition for additional signatures [NIS23]. Essentially, the signature scheme functions as a zero-knowledge proof of knowledge, leveraging the Multi-Party Computation in-the-Head (MPCitH) paradigm [IKOS07]. Subsequently, the Fiat-Shamir transform [FS87], is employed to transition the interactive proof into a signature within the random oracle model. PERK provides three sets of parameters, to meet the different security levels I, III, and V defined by NIST [NIS23]. For each security level, there are two trade-offs: the first involves parameters labeled *short* aiming to reduce signature's size, and the second involves parameters labeled *fast* focusing on minimizing computational costs to achieve faster signature computation. The authors announced an updated version of the scheme [ABB+23b] a few months after the first submission. Here, a novel compression technique was introduced specifically for the *short* parameter sets based on a ranking/unranking algorithm for permutations. While this strategy achieves optimal permutation compression sizes, leading to an approximate 5% reduction in the overall size of *short* PERK signatures, it comes with the trade-off of increased compression time and a requirement for a multiple-precision arithmetic library. Notice that the percentage of PERK signatures occupied by compressed permutations ranges from 15% to 25%. Hence, any substantial improvement in the compression technique for permutations would translate into an improvement in the whole signature sizes.

Contributions. In this work, we introduce a new permutation compression technique that offers simultaneously three significant advantages with respect to previously known compression techniques:

- it gives quasi-optimal compression sizes, that is, a few bits only larger than the best compression possible,
- both compression and decompression routines are highly efficient to compute and easily parallelizable,
- it does not require heap-memory allocations or a multiple-precision arithmetic library, making it portable to resource-constrained devices.

We implement our novel compression method both in pure C and with AVX2 optimization.[4] We show that, by applying it to the PERK signature scheme, we obtain the following improvements:

- we reduce the signature sizes for the *fast* parameter sets by around 5%, at the price of a negligible increase in execution time,
- while maintaining equivalent signature sizes for the *short* parameter sets, we drop the dependency from heap-memory allocations and the `GMP` library for multiple-precision arithmetic. Moreover, we obtain a speed-up of about 2%.

The above considerations significantly increase PERK's code portability. Because of the simplicity of our implementation, we expect our contribution to be even more impactful on resource-constrained devices.

[4] The implementation will be made available together with the final version of the paper under an open-source licence.

Paper Organization. We give in Section 2 the necessary background to understand our work. We introduce our quasi-optimal ranking and unranking methods in Section 3 and present the impact on PERK together with the results of our experiments in Section 4. Finally we give our conclusions in Section 5.

2 Preliminaries

Let $\mathbb{N}$ and $\mathbb{Z}$ denote the sets natural and integer numbers respectively. Let $[n]$ represent the set of integers $\{0, 1, \ldots, n-1\} \subset \mathbb{Z}$, and $\mathcal{S}_n$ denote the group of permutations of $[n]$ with the operation of permutation composition. We write permutations $\pi \in \mathcal{S}_n$ using the so-called *one-line* notation,

$$\pi = (\pi(0), \pi(1), \ldots, \pi(n-1)),$$

that is, as the resulting list of the elements of the ordered set $[n]$ after the permutation is applied.

We define the function that gives the minimum number of bits to represent an integer as

$$\texttt{bitlen} : \mathbb{N} \to \mathbb{N} \qquad x \mapsto \lceil \log_2(x) \rceil .$$

2.1 Ranking/unranking of permutations

In the context of permutations, we call *ranking* a bijective function from $\mathcal{S}_n$ to $\{0, 1, \ldots, n! - 1\}$. The reverse function is called *unranking*. Ranking algorithms typically rank permutations in lexicographic ordering [Leh60,Bon08], but others using a different order also exist [MR01].

Ranking. The ranking of $\pi \in \mathcal{S}_n$ in lexicographic order is computed as follows. Consider the list of integers $(d_0, d_1, \ldots, d_{n-1})$, where

$$d_{n-i-1} = \sum_{j=i+1}^{n-1} \mathbb{1}_{\pi(j)<\pi(i)}, \quad \text{for } i = 0, \ldots, n-2, \qquad d_0 = 0, \tag{1}$$

where $\mathbb{1}_{\pi(j)<\pi(i)}$ is the characteristic function that returns 1 if $\pi(j) < \pi(i)$, 0 otherwise. The list of d_i is known as the *factorial representation* of the permutation. Notice that $d_i \in \{0, \ldots, i\}$, for $i = 0, \ldots, n-1$. The ranking of π is defined as

$$r : \mathcal{S}_n \to \{0, \ldots, n! - 1\}, \quad r(\pi) \mapsto \sum_{i=1}^{n} d_i \cdot i!. \tag{2}$$

An equivalent but recursive expression for $r(\pi)$ is

$$r(\pi) = d_0 + 1 \cdot (d_1 + \cdots + (n-2) \cdot (d_{n-2} + (n-1) \cdot d_{n-1}) \cdots) \tag{3}$$

The sequence $(d_0, d_1, \ldots, d_{n-1})$ is different for every $\pi \in \mathcal{S}_n$, and it is the factorial base representation of $r(\pi)$. It follows that $r(\cdot)$ is a bijection and so the ranking of each permutation is unique.

We report the complete ranking procedure in the variant using Equation 2 in Algorithm 1. Note that Equation 1 has an asymptotic cost of $O(n^2)$. However, Bonet provided an equivalent algorithm with an asymptotic cost of $O(n \log n)$ [Bon08]. For permutations of length $n > 20$, computing $r(\pi)$ requires, in practice, a multiple-precision arithmetic library. Equation 3 comes with the practical advantage, w.r.t. Equation 2, of allowing computation of $r(\pi)$ by performing multiplications by factors $< n$. Certain libraries (e.g., `GMP` [Pro23]) come with efficient and dedicated bignum multiplications by 32-bit integer types (e.g., `uint32_t`), and so this formula is preferred in this case. On the other hand, Equation 2, paired with a look-up table for the precomputed factorials, might be a better option when such an optimized function is not available.

Algorithm 1: Ranking of a permutation

Input: permutation $\pi \in \mathcal{S}_n$
Output: Rank $0 \leq r(\pi) < n!$

$R \leftarrow 0$;
for $i \leftarrow 0, \ldots, n-1$ **do**
  $c \leftarrow 0$; `// Equation 1`
  for $j \leftarrow i+1, \ldots, n-1$ **do**
    if $\pi(i) < \pi(j)$ **then**
      $c \leftarrow c + 1$;
  $R \leftarrow R + c \cdot i!$; `// Equation 2`
return R

Unranking. The unranking procedure is the inverse of ranking, i.e., from an integer $R \leq n! - 1$, one obtains the unique permutation $\pi \in \mathcal{S}_n$ such that $r(\pi) = R$. We report this procedure as Algorithm 2. Similarly to Algorithm 1, there exists a variant that makes use of a recursive formula to obtain the indexes d_i. Furthermore, its asymptotic complexity is $O(n^2)$. However, also in this case, Bonet proposed an equivalent algorithm that runs in time $O(n \log(n))$ [Bon08].

2.2 Permutation compression in PERK

According to the specifications of PERK (see Table 2 [ABB+23b]), the signature includes a set of τ compressed permutations of length n aimed at reducing the overall size of the signature. There are two compression approaches optimized for *fast* and *short* parameters, respectively. In the following sections, we detail these techniques, starting with the compression tailored for *fast* parameters and followed by the one designed for *short* parameters. Table 1 provides a breakdown of the compressed permutation sizes within the signature for all PERK parameter sets, and the relative percentage over the whole signature size.

Algorithm 2: Unranking of a permutation

Input: rank $0 \leq R < n!$
Output: permutation $\pi \in \mathcal{S}_n$

1 `used` $\leftarrow [false : \textbf{for } i \leftarrow 0, \ldots, n-1]$;
2 **for** $i \leftarrow 0, \ldots, n-1$ **do**
3 $\quad d_{n-i-1} = \left\lfloor \frac{R \mod (i+1)!}{i!} \right\rfloor$;
4 $\quad c \leftarrow 0$;
5 $\quad$ **for** $j \leftarrow 0, \ldots, n-1$ **do**
6 $\quad\quad$ **if** *not* `used`$[j]$ **then**
7 $\quad\quad\quad c \leftarrow c + 1$;
8 $\quad\quad$ **if** $c == d_{n-i-1} + 1$ **then**
9 $\quad\quad\quad \pi(i) \leftarrow j$;
10 $\quad\quad\quad$ `used`$[j] = true$;
11 $\quad\quad\quad$ ***break;***
12 **return** π

Pack-in-pairs permutation compression. Let $\pi \in \mathcal{S}_n$ be a permutation. Instead of representing π, with the one-line notation, as a list of elements of $[n]$, one packs its elements in pairs and represents it as a sequence of $\lceil n/2 \rceil$ elements in $[n^2]$. This is possible when

$$\left\lceil \log_2(n^2) \right\rceil < 2 \left\lceil \log_2(n) \right\rceil ,$$

which is true for $n \leq 181$, covering all PERK's parameters. More specifically, let us denote by L the list of $\in n \cdot \tau$ coefficients of all permutations to compress. The packing procedure works as follows. Let $A = \lfloor 2^b \rfloor$, where $b = 6.5$ for Level I, $b = 7$ for Level III, and $b = 7.5$ for Level V. For any two consecutive coefficients $L(i)$ and $L(i+1)$ of L, the compact representation cp is set as $cp = A \cdot L(i) + L(i+1)$. Subsequently, the resulting compact representations cp, comprising $2 \cdot b$ bits each, are concatenated and stored as a byte string. The aggregate size of the compressed permutations within the signature, employing these techniques, is $(2 \cdot b) \cdot \frac{n \cdot \tau}{2}$ bits. The unpacking procedure, is done by computing $L(i) = \lfloor \frac{cp}{A} \rfloor$ and $L(i+1) = cp \bmod A$.

Optimal permutation compression via ranking For each permutation, one computes its ranking $r(\pi)$ in lexicographic order. Specifically, the algorithm by Bonet [Bon08] is implemented using the `gmp` library for multiple-precision computations. In the ranking procedure, the recursive formula from Equation 3 is used to exploit the fast multiplication times a `uint32_t` available in `gmp`. On the other hand, when unranking, the standard unranking subroutine to obtain the factorial base representation (line 3 of Algorithm 2) is used in pair with a look-up table for storing the pre-computed factorials $0!, 1!, \ldots, (n-1)!$. This kind of permutation compression is *optimal*, i.e., each permutation is represented

Parameter Set	n	τ	size of permutations	%
PERK-I-fast3	79	30	1926 B	23.0
PERK-I-fast5	83	28	1889 B	24.0
PERK-I-short3	79	20	980 B	15.7
PERK-I-short5	83	18	936 B	16.2
PERK-III-fast3	112	46	4508 B	24.0
PERK-III-fast5	116	43	4365 B	24.3
PERK-III-short3	112	31	2356 B	16.5
PERK-III-short5	116	28	2240 B	17.0
PERK-V-fast3	146	61	8350 B	25.0
PERK-V-fast5	150	57	8016 B	25.3
PERK-V-short3	146	41	4346 B	17.3
PERK-V-short5	150	37	4070 B	17.7

Table 1: The fourth column presents the sizes of permutations in bytes as components of the overall signature for all parameter set. The fifth column gives the percentage over the whole signature size covered by compressed permutations.

uniquely by the minimum number of bits possible, equal to $\texttt{bitlen}(n! - 1)$, the resulting size of compressed permutations in the signature is $\tau \cdot \texttt{bitlen}(n! - 1)$ bits.

3 Quasi-optimal Permutation Ranking

A major drawback of compressing permutations through ranking is that it requires performing arithmetic operations between integers of up to $\texttt{bitlen}(n! - 1)$ bits. As already mentioned, the sizes of permutations in PERK make the use of a library for multiple-precision integer operations necessary. Consequently, the compression algorithm is relatively slow, and the portability of the implementation is reduced, especially when targeting resource-constrained devices.

This section, presents a *quasi-optimal* ranking approach for permutations that extends the optimal ranking method outlined in Section 2.1. Specifically, we map permutations uniquely to a set of integers with a maximum bit-size slightly exceeding $\texttt{bitlen}(n! - 1)$. This slight increase in size facilitates the ranking evaluation in one fundamental aspect: all computations are performed using only 32-bit words, eliminating the need for a multiple-precision arithmetic library.

3.1 Quasi-optimal ranking routine

Let $\pi \in \mathcal{S}_n$ be a permutation and let $(d_0, d_1, \ldots, d_{n-1})$ be its factorial base representation (see Equation 1). Let N be the target word-size for our computations

(e.g., $N = 32$ bits). We split $d_0, d_1, \ldots, d_{n-1}$ into subsequences and see each one of them as a factorial base representation relative to a different interval such that, for each subsequence, the corresponding integer is not larger than 2^N.

Let us assume that $n < 2^N$ and let $j_1 < j_2 < ... < j_\ell$ be the largest integers possible such that

$$j_0 = 0, \quad \frac{j_k!}{j_{k-1}!} < 2^N, \quad j_\ell = n, \quad \text{for every } k = 1, \ldots, \ell. \tag{4}$$

Define the following integers

$$s_k = \sum_{i=j_{k-1}}^{j_k-1} d_i \cdot \frac{i!}{j_{k-1}!}, \quad k = 1, \ldots, \ell. \tag{5}$$

Equivalently, analogously to Equation 3, the following recursive formula holds:

$$s_k = d_{j_{k-1}} + (j_{k-1}+1) \cdot (d_{j_{k-1}+1} + \cdots + (j_k - 2) \cdot (d_{j_k-2} + (j_k - 1) \cdot d_{j_k-1}) \cdots).$$

Notice that $s_k \leq j_k!/j_{k-1}! - 1$, an so $s_k < 2^N$ for every $k = 1, \ldots, \ell$. Let

$$M = \sum_{k=1}^{\ell} \texttt{bitlen}\left(\frac{j_k!}{j_{k-1}!} - 1\right),$$

and define the following function

$$s : \mathcal{S}_n \to \{0,1\}^M, \quad s(\pi) \mapsto (s_1 \| s_2 \| \cdots \| s_\ell).$$

Our compression method represents the permutation π as $s(\pi)$, and is displayed as Algorithm 3. In general, the bit size of the compression M is (slightly) larger than $\log_2(n!)$, the size obtained with the compression via ranking. However, since every s_k is bounded by 2^N, each one of them can be computed using only N-bit size registers. Choosing $N = 16$, 32, or 64 allows compressing permutations of a certain length without requiring a multiple-precision arithmetic library. In practice, to produce a code portable to several different architectures; in this work, we consider $N = 32$.

3.2 Quasi-optimal unranking routine

To invert the quasi-optimal ranking procedure detailed in Section 3.1, one first must obtain the factorial representation $d_1, \ldots, d_{n-1}$ of the permutation π, as follows.

$$d_{j_{k-1}+j_k-i-1} = \left\lfloor \frac{s_k \mod (i+1)!/(j_{k-1})!}{i!/(j_{k-1})!} \right\rfloor, \quad \text{for } i = j_{k-1}, \ldots, j_k - 1, \tag{6}$$

for $k = 1 \ldots, \ell$. Then, one obtains the one-line permutation representation from the factorial representation using, for example, the sub-routine line 4-11 of Algorithm 2, or Bonet unranking algorithm [Bon08, Figure 4]. We display this idea

Algorithm 3: Quasi-optimal ranking of a permutation

Input: permutation $\pi \in \mathcal{S}_n$; j_k as in Equation 4
Output: Rank $0 \leq r(\pi) < 2^M$

```
for i ← 0, ..., n − 1 do
    for j ← i + 1, ..., n − 1 do
        if π(i) < π(j) then
            d[i] ← d[i] + 1;
for k ← 0, ..., ℓ − 1 do
    s[k] ← 0;                                   // Equation 5
    for i ← j_{k+1} − 1, ..., j_k do
        s[k] ← s[k] · i + d[i];
R ← s[0] ‖ s[1] ‖ ... ‖ s[ℓ − 1];               // pack compactly
return R
```

as Algorithm 4.

An additional advantage of our proposed method, when compared to optimal ranking, lies in its facilitation of vectorization (SIMD instructions) in both ranking and unranking processes. This is made possible by utilizing 32-bit registers only for all computations.

3.3 Further improvements

One can obtain some further improvements in size thanks to the following idea. Let us consider the factorial

$$n! = 1 \cdot 2 \cdots (n-1) \cdot n.$$

Then we have that

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n-1) + \log_2(n).$$

Following the approach outlined in Section 3.1, one could pack such number as follows

$$\underbrace{1 \cdot 2 \cdots (j_1 - 1)}_{s_1} \cdot \underbrace{j_1 \cdots (j_2 - 1)}_{s_2} \cdots \underbrace{j_{\ell-1} \cdots n}_{s_\ell},$$

and so one has that

$$\texttt{bitlen}(n!) \leq \texttt{bitlen}(s_1) + \texttt{bitlen}(s_2) + \cdots + \texttt{bitlen}(s_\ell). \tag{7}$$

However, such an attempt to maximize $s_1, s_2, \ldots, s_\ell$ (subject to $\texttt{bitlen}(s_k) < N$) in this order is unlikely to result in a minimal packing. Instead, we propose using the well-known A* search algorithm [RN09][Sec. 3]. Given the small size of n in the cases of our interest, and using the bit-size of the remaining factors

Algorithm 4: Quasi-optimal unranking of a permutation

Input: rank $R = s[0] \,\|\, s[1] \,\|\, \ldots \,\|\, s[\ell-1]$ $(0 \le R < 2^M)$
Output: permutation $\pi \in \mathcal{S}_n$

for $k \leftarrow 0, \ldots, \ell-1$ **do**
  for $i \leftarrow j_k + 1, \ldots, j_{k+1}$ **do**
    $d[n-1-i] \leftarrow s[k] \bmod i$; `// Equation 6`
    $s[k] \leftarrow \lfloor s[k]/i \rfloor$;
for $i \leftarrow 0, \ldots, n-1$ **do**
  $c \leftarrow 0$;
  for $j \leftarrow 0, \ldots, n-1$ **do**
    if *not* `used`$[j]$ **then**
      $c \leftarrow c + 1$;
    if $c == d[n-i-1]+1$ **then**
      $\pi(i) \leftarrow j$;
      `used`$[j] = true$;
      break;
return π

to be packed as the heuristic in the A* search algorithm (which is easily seen to be admissible), a packing of the factors that minimizes the overall size can be quickly found. Nevertheless, we choose to introduce additional constraints that lead to a faster implementation when using vectorization (and specifically the AVX2 instruction set):

- the number of words is chosen as either 16, 24, or 32 depending on n, which is a multiple of 8 (the 256-bit vector length of AVX2 divided by $N = 32$);
- we bound the minimum and maximum number of indexes per word and introduce heuristic penalties to seek a balanced distribution between words, especially within the natural 8-lane boundaries of AVX2.

A further generalization of this concept would be to dispense with the requirement of consecutive indexes within each word. This would immensely increase the search space, making it likely that packings that are either optimal or within very few bits of it are found; however, non-consecutive indexes would also complicate the implementation, and thus, we chose not to pursue this idea.

Example 1. Let us compute the compression size of a permutation π of length $n = 79$, as in PERK-I-fast3 and PERK-I-short3. We choose the word length of $N = 32$ bits. In Table 2, we report the values for the indexes j_k and the size in bits for each s_k, for $k = 1, \ldots, 78$. The total size in bits of our quasi-optimal ranking is $M = \texttt{bitlen}(s(\pi)) = 394$. Note that the size in bits of the optimal ranking is $\texttt{bitlen}(r(\pi)) = 389$. Therefore, our compression is only 5 bits larger than the compression via optimal ranking, a small price to pay ($\approx 1.3\%$) for the benefit of being able to perform all computations more efficiently using 32-bit registers only.

k	**factorial base subsequence**	j_k	`bitlen`(s_k)
1	$d_1,\ d_2,\ d_3,\ d_4,\ d_5$	6	10
2	$d_6,\ d_7,\ d_8,\ d_9,\ d_{10}$	11	16
3	$d_{11},\ d_{12},\ d_{13},\ d_{14}$	15	15
4	$d_{15},\ d_{16},\ d_{17},\ d_{18},\ d_{19}$	20	21
5	$d_{20},\ d_{21},\ d_{22},\ d_{23}$	24	18
6	$d_{24},\ d_{25},\ d_{26},\ d_{27},\ d_{28}$	29	24
7	$d_{29},\ d_{30},\ d_{31},\ d_{32},\ d_{33}$	34	25
8	$d_{34},\ d_{35},\ d_{36},\ d_{37},\ d_{38}$	39	27
9	$d_{39},\ d_{40},\ d_{41},\ d_{42},\ d_{43}$	44	27
10	$d_{44},\ d_{45},\ d_{46},\ d_{47},\ d_{48}$	49	28
11	$d_{49},\ d_{50},\ d_{51},\ d_{52},\ d_{53}$	54	29
12	$d_{54},\ d_{55},\ d_{56},\ d_{57},\ d_{58}$	59	30
13	$d_{59},\ d_{60},\ d_{61},\ d_{62},\ d_{63}$	64	30
14	$d_{64},\ d_{65},\ d_{66},\ d_{67},\ d_{68}$	69	31
15	$d_{69},\ d_{70},\ d_{71},\ d_{72},\ d_{73}$	74	31
16	$d_{74},\ d_{75},\ d_{76},\ d_{77},\ d_{78}$	79	32
	Total		394

Table 2: Quasi-optimal permutation compression parameters and sizes for $n = 79$ and $N = 32$. Note that $d_0 = 0$ always, hence there is no need to encode it.

3.4 Comparison

We report in Table 3 the comparison of our sub-optimal permutation against the other two methods used in PERK, for the relevant values of n. One can see that our method gives compression sizes very close to *optimal*, while beating significantly *pack-in-pairs*.

4 Experiments and Applications to PERK

In this section, we give the details regarding the C implementation of our algorithm, the results of our benchmark tests, and the impact of our work on PERK signature both computationally and on the signature size. To start, we describe our testing environment.

4.1 Testing environment

We preformed all experiments presented in this section on a machine with 96GB of memory and an Intel® Core™ i7-13700K CPU @ 3.40GHz. As a compiler, we used `clang` (version 17.0.2), and the version of the `GMP` multiple-precision library installed on the machine is 6.3.0. Especially when testing the integration of our

n	**pack-in-pairs**	**optimal ranking**	**sub-optimal ranking**
79	514	389	394
83	540	414	421
112	784	606	615
116	812	633	643
146	1095	845	860
150	1125	873	889

Table 3: Amount in *bits* required to compress a permutation. The second and third columns are the values resulting in PERK's *fast* and *short* parameter sets respectively. The last column represent the values of our novel compression method.

algorithms on the full PERK scheme, we expect the impact to be relatively small and hard to detect if the testing environment is not adequately set. Hence, we took the following countermeasure to reduce possible noise in benchmarks, not only due to hardware and OS factors but also because of issues such as code and data layout [MDHS09], as this noise is of similar magnitude to the improvements we are trying to measure on full scheme benchmarks.

On the hardware and OS side, we disabled HyperThreading and TurboBoost CPU features, installed Ubuntu's low-latency Linux kernel, set the scaling governor to `performance` mode, isolated a CPU and pinned our benchmarks to run on it while masking interrupts to avoid running on that CPU, used Linux's `NOHZ` feature to reduce tick interrupts and disabled address space layout randomization. Before running the benchmarks, we turned off WiFi and Bluetooth and removed the Ethernet cable. The machine is placed in a temperature-controlled room, and its cooling system is adequate to ensure clock speed is not throttled.

On the software side, we employed a feature of the `lld` link, which randomizes the order of functions in the binary, one of the factors explicitly pointed out by [MDHS09]. We also renamed the binary before each run with a differently-sized name (varying this over a range of 64 consecutive sizes), which has the effect of realigning the stack memory for the process, another effect discussed by [MDHS09]. We set code alignment to 64 bytes to match the cache line size of the CPU. We ran each test with 8 different randomized linking orders and 64 consecutive alignments; for the full-scheme benchmarks, each routine was run 12 times, discarding the first 2 results, which serve as a warm-up for the CPU's branch predictor and its caches.

We also perform a statistical hypothesis test (Student's independent two sample t-test) to determine whether speedups/slowdowns between the baseline and our proposed implementation are statistically significant at the $p = 0.05$ level.

4.2 Implementation of our compression/decompression algorithms

We implemented the compression and decompression algorithms presented in Section 3 in pure C without any external library dependency. In addition, we have developed an AVX2-optimized implementation, the details of which are provided below in this section. The code will be made available together with the final version of this paper under an open-source license.

We report in Table 4 the results of our experiment that compares the quasi-optimal ranking implementation against the compression from PERK version 1.1. To do so, we imported the compression and decompression algorithms from the official repository of PERK [ABB+23b]. The code has been compiled with compilation flags `-O3 -funroll-loops -march=native -mavx2` to make the comparison fair against our AVX2 implementation, and to highlight that this achieves vectorizations undetected by the compiler. Looking at Table 4, the first consideration is that our approach is always considerably faster than the optimal ranking method. Then, one can notice that, due to its simplicity, *pack-in-pairs* is still the fastest compression method in general. The pack-in-pairs implementation displays non-linear scaling for compression and decompression for the cases $n = 112, 116$, and as such, our AVX2 compression implementation is actually faster in these cases. The reason for such scaling is unclear, but may be related to code alignment issues or compiler heuristics not being satisfied for these particular values. For the other cases, our AVX2 implementation gives timings quite close to *pack-in-pairs* while providing much shorter compression sizes. Finally, our AVX2 code gives a speed-up against our pure C implementation of 5.6-6.9× in compression and 9.6-13.3× in decompression.

Details of our AVX2 optimization At a high level, our AVX2 implementation is a straightforward translation of the C code, exploiting the considerable parallelization opportunities presented by the algorithm itself within a single compression or decompression. We wrote *compact* and manually *unrolled* versions of the code, but as the latter performed consistently better, we choose to present performance results only for it.

We employ vectors with 8 elements of 32 bits each for ranking and unranking. Referring to the example of Table 2, we operate first on $k = 1, \ldots, 8$, processing each "column" in sequence, and then $k = 9, \ldots, 16$. As can be seen, our A* search strategy ensures that vectors are fully utilized due to the choice of the number of words as 16, 24, or 32 depending on n. On the other hand, since, in general, $8 \nmid n$, it is inevitable that there will be some gaps in the distribution of indices, as seen in the cases $k = 3$ and 5 in Table 2. In these cases, we still process the full vector (therefore using invalid data for the gaps) but use the AVX2 `blend` instruction to choose whether to include the result (in the example, for the cases $k = 1, 2, 4, 6, 7, 8$) or not (for $k = 3$ and 5).

We note that the loads in the ranking algorithm map well to the `gather` instructions of AVX2, whereas the stores in the unranking algorithm are a clear use case for `scatter` instructions, which, unfortunately, are unavailable on AVX2,

only AVX-512. Thus, we expect that an AVX-512 implementation of decompression could perform even better.

For computing the factorial representation, we employ vectors with 32 elements of 8 bits each, which is sufficient as $n \leq 150$. For the first iteration of the algorithm, the number of vectors required varies from $\lceil 79/32 \rceil = 3$ to $\lceil 150/32 \rceil = 5$ across PERK parameter sets. This number decreases as the algorithm restricts itself to progressively shorter ranges of the full array of indexes. We ensure that computations are performed only on the minimum number of vectors required at each iteration. If the length of the current range is k, then we use $\lfloor k/32 \rfloor$ full vectors and a partially masked vector, using only $k \bmod 32$ out of the 32 available lanes. As with ranking and unranking, we still process the entire vector and conditionally select only the lanes performing useful work, this time using masks and the bitwise AND operator.

We expect that a batched implementation, compressing or decompressing multiple permutations at once, opens up possibilities for further instruction-level parallelism and better utilization of 32-byte vectors for computing the factorial representation. This should lead to more considerable speedups in exchange for more complex code. We leave such an investigation to future work.

n	**Pack-in-pairs**	**Optimal**	**Quasi-optimal**	**Quasi-optimal AVX2**
		Compression		
79	309	11307	2606	411
83	330	11995	2792	419
112	887	17775	4271	623
116	920	18582	4508	649
146	652	25772	6172	1086
150	694	26552	6408	1130
		Decompression		
79	280	30586	7172	743
83	292	33048	7838	773
112	824	54572	13219	1104
116	873	58013	14101	1140
146	601	89232	21226	1670
150	606	93817	22902	1718

Table 4: Comparison in CPU cycles of the compression and decompression routines from PERK version 1.1 and the one introduced in Section 3. The results of each parameter set were obtained by computing the mean from $64 \cdot 64 \cdot 10,000 = 40,960,000$ random instances.

4.3 On the impact of our work on PERK signature scheme

To assess the impact of the compression technique presented in Section 3 to PERK [ABB+23b], we have integrated it into PERK's official implementation and conducted experiments. In order to not introduce any vulnerability to PERK, we included in our decompression algorithm (Algorithm 4) a check that each $s[k] \leq \frac{j_k!}{j_{k-1}!}$, for $k = 1 \cdots, \ell$, to ensure that the compression is bijective and one cannot easily generate another valid signature. If the bound does not hold, the signature gets rejected. Note that constant-time code is not required in our case, as we compress and decompress public data.

We start by reporting in Table 5 a comparison of the signature sizes of PERK when using the compression explained in Section 3 compared to PERK version 1.1. One can see that our quasi-optimal ranking increases the signature sizes of PERK by a negligible fraction (never more than 0.25%) for the *short* parameter sets. On the other hand, it always reduces the sizes of PERK's *fast* parameter sets by more than 5%.

Parameter Set	**PERK v. 1.1 Signature Size**	**PERK + Section 3 Signature Size**	**gain/loss** %
PERK-I-fast3	8345 B	**7897 B**	−5.37
PERK-I-fast5	8026 B	**7611 B**	−5.17
PERK-I-short3	**6251 B**	6256 B	+0.08
PERK-I-short5	**5780 B**	5792 B	+0.21
PERK-III-fast3	18820 B	**17849 B**	−5.16
PERK-III-fast5	17968 B	**17060 B**	−5.05
PERK-III-short3	**14280 B**	14308 B	+0.20
PERK-III-short5	**13164 B**	13175 B	+0.08
PERK-V-fast3	33339 B	**31547 B**	−5.37
PERK-V-fast5	31664 B	**29983 B**	−5.30
PERK-V-short3	**25141 B**	25203 B	+0.25
PERK-V-short5	**23040 B**	23082 B	+0.18

Table 5: Signature size gain and loss using quasi-optimal ranking for compressing permutations compared to PERK version 1.1 [ABB+23b]. For each row, we write in bold the compression that gives the shortest signatures. The last column reports the gain/loss in percentage of our method against PERK version 1.1.

In Tables 6 and 7, we present the CPU-cycle performance on the aforementioned benchmark platform when utilizing various compression methods for each parameter set of PERK across both reference and optimized (AVX2) implementations. For the reference version, there are generally small slowdowns for the

fast parameters and small speedups for the *short* parameters, on the order of $< 1\%$ for signing and $< 3\%$ for verification. The optimized version compares better: there are a few slowdowns, most of which are negligible ($\leq 0.35\%$, with the exception of verification for PERK-V-fast5 at 0.88%). On the other hand, there are speedups for all *short* parameters, and even for some *fast* parameters, of up to 2.9%.

Summarizing the results of Tables 5 to 7, PERK would get the following impact from our compression:

- significantly smaller signature sizes for *fast* parameter sets, for a negligible computational cost increase,
- equivalent signature sizes for the *short* parameter sets, with either equivalent or slightly faster signature and verification algorithms,
- more straightforward and more portable code, free of any dependency from a multiple-precision arithmetic library. In addition, our compression drops the need for any heap-memory allocation required by `GMP`, which is a critical issue for resource-constrained devices; indeed, the `pqm4` project [KKPY24], mirroring best practices in the embedded industry, excludes any implementations that perform dynamic memory allocations.

One additional benefit is that PERK could use the same compression algorithm (and code) for both *short* and *fast* parameter sets.

5 Conclusions

We introduced a quasi-optimal permutation ranking that, unlike its optimal counterpart, allows it to be computed without using a multiple-precision arithmetic library. This allowed us to define a new permutation compression technique. Our experiments suggest that our technique achieves the best trade-off of efficiency and compression size for the permutation sizes considered in this work. We applied our result to the digital signature PERK, obtaining a considerable improvement in the signature size for the *fast* versions of the scheme and an overall more straightforward and more portable code. Moreover, we expect our code to yield significant improvements in efficiency for implementations of PERK on resource-constrained devices since the `GMP` library cannot be ported there.

Additionally, we believe that our result might be useful also outside the realm of Cryptography, in applications such as heuristic search, combinatorial optimization and data structure indexing.

Parameter set	PERK v. 1.1		PERK + Sec. 3		Speedup	
	Sign	Verify	Sign	Verify	Sign	Verify
I-fast3	20.7	10.2	20.8	10.5	**−0.65%**	**−2.46%**
I-fast5	20.4	9.90	20.5	10.2	**−0.76%**	**−2.81%**
I-short3	111	55.6	110	54.6	**0.53%**	**1.91%**
I-short5	106	52.2	105	51.2	**0.85%**	**1.98%**
III-fast3	49.9	25.2	50.1	25.9	**−0.50%**	**−2.45%**
III-fast5	48.5	24.2	48.8	24.8	**−0.45%**	**−2.48%**
III-short3	268	136	268	135	**0.09%**	**0.68%**
III-short5	252	127	252	125	**0.22%**	**1.51%**
V-fast3	104	54.9	104	55.9	**−0.16%**	**−1.79%**
V-fast5	99.9	52.4	100	53.7	**−0.43%**	**−2.35%**
V-short3	556	297	557	296	**−0.03%**	**0.63%**
V-short5	519	274	517	271	**0.31%**	**1.03%**

Table 6: Performance in millions of CPU-cycles of the quasi-optimal ranking for compressing permutations compared to the reference implementation of PERK version 1.1 [ABB+23b]. The results of each parameter set were obtained by computing the mean from $8 \cdot 64 \cdot 10 = 5120$ random instances. Speedups in bold indicate statistically significant results at the $p = 0.05$ significance level.

References

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 version 1.0. 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 version 1.1. https://pqc-perk.org/resources.html, 2023.

Bon08. Blai Bonet. Efficient algorithms to rank and unrank permutations in lexicographic order. *Workshop on Search in Artificial Intelligence and Robotics - Technical Report*, 01 2008.

fCR20. Chinese Association for Cryptographic Research. National cryptography algorithm design competition. https://www.cacrnet.org.cn/site/content/854.html, 2020.

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.

Parameter set	PERK v. 1.1		PERK + Section 3		Speedup	
	Sign	Verify	Sign	Verify	Sign	Verify
I-fast3	7.53	5.07	7.53	5.08	**−0.02%**	**−0.22%**
I-fast5	7.21	4.91	7.21	4.92	**−0.07%**	**−0.35%**
I-short3	39.4	26.9	39.2	26.3	**0.46%**	**2.13%**
I-short5	36.0	24.6	35.8	24.0	**0.56%**	**2.27%**
III-fast3	15.9	12.2	15.8	12.2	**0.57%**	**−0.10%**
III-fast5	15.4	11.7	15.3	11.7	**0.39%**	**−0.06%**
III-short3	83.2	65.1	82.6	64.2	**0.75%**	**1.52%**
III-short5	77.4	59.5	76.8	57.8	**0.77%**	**2.90%**
V-fast3	36.4	27.5	36.4	27.8	**−0.14%**	**−0.88%**
V-fast5	34.9	26.5	34.5	26.4	**0.99%**	**0.40%**
V-short3	193	146	192	142	**0.57%**	**2.73%**
V-short5	179	133	178	130	**0.59%**	**2.62%**

Table 7: Performance in millions of CPU-cycles of the AVX2 quasi-optimal ranking for compressing permutations compared to the optimized implementation of PERK version 1.1 [ABB+23b]. The results of each parameter set were obtained by computing the mean from $8 \cdot 64 \cdot 10 = 5120$ random instances. Speedups in bold indicate statistically significant results at the $p = 0.05$ significance level.

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.

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.

Leh60. Derrick H Lehmer. Teaching combinatorial tricks to a computer. *Combinatorial Analysis*, pages 179–193, 1960.

Leo82. Jeffrey Leon. Computing automorphism groups of error-correcting codes. *IEEE Transactions on Information Theory*, 28(3):496–511, 1982.

MDHS09. Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

MR01. Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

NIS17. NIST. Post-quantum cryptography standardization. https://csrc.nist.gov/projects/post-quantum-cryptography, 2017.

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

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

RN09. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

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