SoK: Methods for Sampling Random Permutations in Post-Quantum Cryptography

Alessandro Budroni, Isaac A. Canales-Martínez, Lucas Pandolfo Perin

Cryptography Research Center, Technology Innovation Institute, Abu Dhabi, UAE
{alessandro.budroni,isaac.canales,lucas.perin}@tii.ae.

Abstract

In post-quantum cryptography, permutations are frequently employed to construct cryptographic primitives. Careful design and implementation of sampling random unbiased permutations is essential for efficiency and protection against side-channel attacks. Nevertheless, there is a lack of systematic research on this topic. Our work seeks to fill this gap by studying the most prominent permutation sampling algorithms and assessing their advantages and limitations. We combine theoretical and experimental comparisons and provide a C library with the implementations of the algorithms discussed. Furthermore, we introduce a new sampling algorithm tailored for cryptographic applications.

Keywords: Fisher-Yates, Permutation Sampling, Post-quantum Cryptography, Secure Implementation, SoK, Sorting

1 Introduction

Permutations are mathematical objects that refer to the arrangement of the elements of a set in a specific order. They serve as building blocks for various cryptographic algorithms. On a high level, permutations generate seemingly random outcomes that prevent adversaries from deducing patterns that may lead to mounting successful attacks.

Permutations find applications in some constructions within symmetric-key cryptography. We mention a couple of examples. A Substitution-Permutation Network (SPN) is a construction used to design block ciphers. In short, an SPN is comprised of several rounds, where each round consists of a (non-linear) substitution operation followed by a permutation; we refer to the existing literature (e.g., [1]) for further details on SPNs. The Advanced Encryption Standard [2, 3] is a cipher following the SPN approach. The sponge construction [4] is a design that has found applications in hash functions, message authentication codes, stream ciphers, authenticated encryption and pseudo-random generators, e.g., [5–8]. The sponge construction is a simple iterated construction for building a function with variable-length input and arbitrary-length output; it employs a fixed-length permutation operating on a fixed number of bits.

In post-quantum cryptography, permutations have been used to construct several algorithms. For example, the key generation of Classic McEliece [9], a key encapsulation mechanism and a candidate to the NIST[1] standardization process [10], and the digital signatures LESS [11] and EHT [12], submitted to the NIST standardization for additional signature schemes [13], include a subroutine that samples a random permutation. On the other hand, the signatures PKP-DSS [14] and PERK [15], submitted to the standardization processes run by CACR[2] [16] and NIST [13], respectively, are based on a hard problem known as *Permuted Kernel Problem* introduced by Shamir in 1989 [17]. Other proposed protocols employing permutations are [18–22].

Permutations are chosen and fixed in some cryptographic algorithms as part of the specification. That is the case for most, if not all, algorithms within symmetric-key cryptography. In other cases, like the post-quantum cryptography examples above, sampling permutations is a step within the algorithm. In this manuscript, we are interested in the latter case.

Sampling random permutations in a cryptographic context is a non-trivial operation not to be underestimated. Flaws or weaknesses in random permutation sampling algorithms can lead to catastrophic consequences, leaving cryptographic systems vulnerable to attacks. Failing to produce unbiased random permutations may leak information and expose sensitive data to adversaries. To the best of our knowledge, no systematic effort has been made to comprehensively analyze and compare the various methodologies employed in the existing literature. On the contrary, individual research has been conducted to tackle specific use cases. The objective of this manuscript is to fill this gap and provide a single source of knowledge on the topic of sampling random permutations in cryptography.

Contributions

In this work, we present a systematic study of the most widely used permutation sampling algorithms in cryptography. This study adheres to the guidelines for systematic mapping studies proposed by Petersen et al. [23]. For each of the selected algorithms, we analyse its benefits and limitations. In addition, we introduce a new algorithm specifically designed to sample permutations in constant time. Together with this manuscript, we introduce a publicly available library in C containing the implementations of all the algorithms selected. We use this library to perform a comparison both on a theoretical and experimental level.

[1]National Institute of Standards and Technology
[2]Chinese Association for Cryptographic Research

Organization

In Section 2 we give the preliminaries to address the topic. The methodology employed in our study is presented in Section 3. Section 4 and Section 5 report the known algorithms for sampling random permutations in cryptography, plus a new one of our design. In Section 6 we show the result of our theoretical and experimental comparisons. Finally, Section 7 contains our conclusions together with some potential future development on the topic. In addition, we report in Appendix A the details and experimental comparisons of the algorithms to perform permutation composition and inversion.

2 Background and Related Work

We denote the set of natural, integer and real numbers as $\mathbb{N}, \mathbb{Z}$ and $\mathbb{R}$, respectively. We use calligraphic capital letters (e.g., $\mathcal{A}$) to denote sets. The notation $U(\mathcal{A})$ denotes the uniform distribution over a finite set $\mathcal{A}$. The notation $r \leftarrow U(\mathcal{A})$ indicates that the element r has been sampled uniformly at random from the set $\mathcal{A}$. The notation $|\cdot|$ is used to indicate either the cardinality of a set (e.g., $|\mathcal{A}|$) or the absolute value of a real number (e.g., $|a|$, for $a \in \mathbb{R}$).

We use the notation $\ell = [x_0, x_1, \ldots, x_{n-1}]$ to denote a list of elements that are given in that specific order.

Given $x_1, x_2 \in \mathbb{N}$, we denote as $x_1 \parallel x_2$ the integer resulting by the concatenation of the bit representation of x_1 and x_2, with the most significant bit to the left.

2.1 Permutations

Let $\mathcal{C} = \{x_0, ..., x_{n-1}\}$ be a finite set with n elements. The set of all bijections $\mathcal{C} \to \mathcal{C}$ is denoted by $\mathcal{S}_{\mathcal{C}}$ and is called the *symmetric group* of $\mathcal{C}$. A *permutation* of $\mathcal{C}$ is an element of $\mathcal{S}_{\mathcal{C}}$.

One way to represent a permutation $\pi \in \mathcal{S}_{\mathcal{C}}$ is to write

$$\pi = \begin{bmatrix} x_0 & x_1 & \ldots & x_{n-1} \\ \pi(x_0) & \pi(x_1) & \ldots & \pi(x_{n-1}) \end{bmatrix},$$

meaning that π acts on the elements of $\mathcal{C}$ by sending x_i to $\pi(x_i) = x_j \in \mathcal{C}$, for some $0 \leq j < n$.

Let $\mathcal{C} = \{0, 1, \ldots, n-1\}$, for a positive integer n. Then, the symmetric group of $\mathcal{C}$ is denoted by $\mathcal{S}_n$ and is referred to as the *symmetric group of order* n. In this case, it is enough to represent a permutation $\pi \in \mathcal{S}_n$ as a list of elements

$$\pi = [\pi(0), \pi(1), ..., \pi(n-1)].$$

We will write the i-th element of π as $\pi(i)$.

$\mathcal{S}_n$ together with composition of functions form a group. The identity element is the identity permutation $id = [0, 1, \ldots, n-1] \in \mathcal{S}_n$ and $|\mathcal{S}_n| = n!$. Also, we write π^{-1} to denote the inverse of a permutation $\pi \in \mathcal{S}_n$. In what follows, unless otherwise specified, we will consider permutations in $\mathcal{S}_n$.

2.2 Permutation Sampling Algorithms

We define a permutation sampling algorithm as a randomized algorithm $A_n^{\$}$ that, for a positive integer n, gives as output an element of $\mathcal{S}_n$. In cryptography, we are interested in permutation sampling algorithms with the following properties:

- **Unbiased.** Let $\mathcal{D}_{A_n} \subseteq \mathcal{S}_n$ be the set of all permutations output by $A_n^{\$}$. We say that $A_n^{\$}$ is unbiased if

$$\left| Pr(\pi \leftarrow A_n^{\$}) - \frac{1}{|\mathcal{D}_{A_n}|} \right| \leq \mathsf{negl}(n), \qquad \text{for every } \pi \in \mathcal{D}_{A_n},$$

  and the elements of $\mathcal{D}_{A_n}$ are uniformly distributed over $\mathcal{S}_n$. Typically, in cryptography one requires that $1/|\mathcal{D}_{A_n}| \leq \mathsf{negl}(n)$.

- **Efficient.** We say that $A_n^{\$}$ is efficient if its time and space complexity are polynomial in n.

- **Constant-time implementation friendly.** In cryptography, algorithms that handle secret data are required to be resistant to side-channel analysis such as time and cache attacks. We say an algorithm runs in constant-time if its execution time is constant and does not depend on the input data. To be precise, in a cryptographic context, algorithms are considered to be constant-time when their execution time does not depend on secret data. In this way, a time analysis of the algorithm does not leak any information related to sensitive values that must be kept secret. A common good practice to achieve constant-time implementations is to avoid branching and memory access based on secret data. There exist applications in cryptography where a permutation algorithm is employed but does not handle sensitive data. In this scenario, our main concern is that the algorithm implementation is as efficient as possible. For this reason, it is crucial also to consider non-constant-time implementations. A related aspect is whether different algorithms are *compatible* with each other, that is, given the same seed to the PRNG, whether they return the same permutation or not.

In this manuscript, we are interested in permutation sampling algorithms in a cryptographic context. Thus, we focus on unbiased sampling algorithms, both constant and non-constant-time.

Overview of the algorithms. A trivial way of sampling a permutation is to repeatedly sample elements from $\{0, \ldots, n-1\}$ and save them in the order they are sampled until one obtains a permutation. Notice that as more elements are obtained, the probability of sampling one which has not been already sampled decreases. This in turn increases rapidly the number of trials needed to get a new number at each step. Hence, this method is highly inefficient.

In cryptography, two main approaches have been used for constructing efficient random permutation sampling algorithms. The first one consists of using a sorting algorithm as a subroutine and will be explained in Section 4. The second approach is based on sampling from a set without replacement and sees the Fisher-Yates algorithm as its main instantiation [24]. We will detail this second family of algorithms in Section 5.

Another approach studied usually outside the scope of cryptographic applications is the one involving *ranking* and *unranking* algorithms. A ranking algorithm maps each permutation of length n to a unique integer between 0 and $n! - 1$. To sample a permutation, one can sample an integer in such a range and then retrieve the corresponding permutation via the corresponding unranking algorithm. Lehmer formalized this idea in 1960 [25], and several improvements have been proposed subsequently (e.g. [26, 27]). However, this approach does not seem suitable for random sampling in cryptography because it requires to perform multiplications and divisions with multiple-precision arithmetic that are both computationally expensive and challenging to implement in constant-time.

2.3 On Sampling Vectors with Prefixed Hamming Weight

We say that a vector of length n has Hamming weight equal to k if exactly k of its entries are non-zero and $n - k$ are zero. Algorithms for sampling random permutations are closely connected to algorithms for sampling vectors with prefixed Hamming weight. Typically, one tries to sample k different indices in the range $[0, 1, \ldots, n-1]$, and these give the positions of the non-zero entries of the vector. If the indices are obtained without any bias on the order in which they are sampled, then setting $k = n$ gives all n indices in a random order, i.e., a permutation. In other words, one can see certain algorithms for sampling prefixed Hamming weight vectors as the truncated version of algorithms to sample permutations.

On the other hand, a binary vector with Hamming weight equal to k and length n is nothing else than a permutation of the vector

$$(\underbrace{1, \ldots, 1}_{k}, \underbrace{0, \ldots, 0}_{n-k}).$$

2.4 Random Integers within a Range

Many algorithms reported in this manuscript require sampling integers uniformly at random within a range. When the range is of the kind $\{0, \ldots, 2^k - 1\}$, for some integer $k \geq 0$, then the operation is relatively easy to implement. It is enough to sample $h \geq k$ bits at random, where h is a multiple of 8, and perform masking/shifting operations to obtain a random integer in the desired range.

However, when the range is of the kind $\{0, \ldots, i-1\}$, where i is not a power of 2, shifts and masking operations are not enough to obtain a uniform random integer within the range. Let k be an integer such that $2^k > i$. A common implementation mistake is to sample $r \leftarrow U(\{0, \ldots, 2^k - 1\})$, and then calculate

$$(r \bmod i) \in \{0, \ldots, i-1\}.$$

Indeed, the obtained number will be biased because there are some numbers that are more likely to appear after modulo reduction.

To obtain an unbiased distribution, we consider the strategy of rejection sampling. First, one samples $r \leftarrow U(\{0, 1, \ldots, 2^k - 1)$. If $r < i$, then one accepts the sample. On the contrary, if $r \geq i$, the number gets rejected and a new one will be sampled. This simple method produces unbiased sampling, but has the drawback that, when i is only slightly larger than 2^{k-1}, the rejection probability is close to 50%.

To reduce the rejection probability, we use a scaling approach that allows to arbitrarily reduce the rejection probability at the cost of increasing the necessary number of random bits to be sampled. This method seems to be folklore in the literature [28], however, in certain scenarios, side-channel attacks can be used to leak sensitive information due to rejection of bad samples [29]. A strategy to reduce leakage is referred to as *bounded rejection sampling* in [30]. The idea is to repeat the rejection sampling a fixed number of times N before returning the result. The rejection probability is upper bounded to 50%, so at least one valid result can be successfully obtained with probability $1 - (1/2)^N$.

Algorithm 1 Sampling integers uniformly at random within a range in constant-time.

```
Input: Integers $i, k \geq 0$ such that $2^k > i$
Output: a random integer $0 \leq r < i$
1: Set $r = 0$
2: Set $k'$ as the smallest integer s.t. $2^{k'} > i$
3: for 1 to $N$ do
4:     $x \leftarrow U(\{0, \ldots, 2^k - 1\})$
5:     $x = x \wedge (k' - 1)$
6:     $r = (i > x) \ ? \ x \ : \ r$
7: end for
8: Return $r$
```

We employ Algorithm 1 to execute sampling of integers within ranges smaller than powers of two in constant-time. The masking in line 5 of the algorithm guarantees a rejection probability of no more than 50%. We also remark that the return value in line 6 should be assigned using a constant-time conditional assignment. We compare the performance of the constant-time sampling method (CT) and the one proposed in [28] (Lemire) in Section 6.2. We remark that Lemire sampling could also be considered constant-time, depending on the flexibility of the constant-time definition and what is considered a secret value during the sampling method. However, we consider the conservative approach where Algorithm 1 always runs in constant time.

2.5 Related Work

A comprehensive theoretical study on permutation sampling algorithms outside the realm of cryptography, hence not taking the side-channel resistance perspective into account, can be found in [31]. A qualitative study on Fisher-Yates and some of its variants can be found in [32]. More recently, the preprint of a concurrent work proposing a verification method for permutation sampling implementations, with a focus on hiding countermeasure against side-channel attacks, has been made public [33].

Some algorithms presented here employ a sorting procedure as a subroutine. A recent comparison of some of the fastest, not necessarily constant-time, algorithms can be found in [34]. However, such comparison does not include *djbsort* [35], the constant-time sorting algorithm implementation that we employ in this work.

Given the connections to the topic of sampling fixed-weight vectors highlighted in Section 2.3, this study took inspiration from works on that field, in particular the work of Nicolas Sendrier [36]. A subsequent detailed study on the secure implementations of the most used fixed-weight vector sampling algorithms can be found in [30].

3 Methodology

3.1 Research Questions

The objective of this systematic study is twofold. First, we want to identify the permutation sampling algorithms currently used in cryptography. Second, we want to understand how their performance (in execution time) compare to each other. To accomplish this, we based our study on the following research questions (RQ):

- RQ1. What are the techniques used for sampling random permutations?
- RQ2. How do properties relevant for cryptography affect the techniques above?
- RQ3. How do these techniques compare against each other?

As shown later (in Section 5), there are algorithms that use sampling of integers. We clarify that this study is focused on permutation sampling. Even though various methods for integer sampling were found, we did not analyse them. When we report the results of our experiments (see Section 6), we used two integer sampling methods: one for constant-time and one for efficient sampling.

3.2 Search of Research Material

The selection of the source of information used to conduct this study started by querying the most used databases of research material such as *ACM Library*, *IEEEXplore*, *SpringerLink*, *Google Scholar* and *ResearchGate*. Additionally, we found important material on pre-print databases such as *arXive* and *ePrint*. The queries can be grouped by topic into four categories:

1. *Permutation sampling algorithms in general*, that is, without restricting the scope to a cryptographic context. This allowed us to enlarge our knowledge on the set of existing algorithms and trace back the line of research that brought to light the algorithms used nowadays.
2. *Permutation sampling in existing cryptographic algorithms and protocols*. Since we could not find any detailed survey on permutation sampling algorithms in cryptography, we looked for research works in the field involving permutations. We discovered that independent research was conducted repeatedly to assess the best approach for the use case among the known ones. When the details of the algorithm were not specified in the manuscript or report, an associated implementation was usually publicly available, allowing us to reconstruct the algorithm.

3. *Fixed-weight vectors sampling in cryptography.* Given the similarities among the algorithms used for sampling permutations and fixed-weight vectors, as highlighted in Section 2.2, we studied several research works concerning these related algorithms in a cryptographic context. Indeed, it was relatively easy to translate some of these algorithms for fixed-weight vectors to algorithms for permutations. However, in some cases, we could not perform such adaptation.
4. *Sampling integers within a range.* This subroutine is used in some of the algorithms presented in this manuscript (see Section 5). Therefore, we investigated the techniques used in the literature and adapted them to our use case. Noteworthy, many of the sources on this topic were related to fixed-weight vectors sampling algorithms as well.

The queries to the aforementioned research databases have been made using combinations of keywords in English. Moreover, we visited the official web pages of cryptographic algorithms which employ permutations or require sampling random permutations, as well as web pages of researchers who are known to work on these and related topics. Finally, we discovered material through backward snowball sampling.

3.3 Selection of Material

We decided whether to include or exclude articles based on titles, abstracts, and full-text reading relevant sections. Particularly, we focused on those sections related to permutation sampling, fixed-weight vector sampling or integer sampling and the desired properties for their use in cryptography. The review was performed independently by all authors. When an article was considered relevant, the other authors were informed about it and a collective decision on whether to include or exclude the material was made.

The following is the criteria used to include material:

- It presents an algorithm or a description of the sampling method.
- When it is a random permutation sampling method, the material shows that it is unbiased. If such discussion is not there, we included the study if it was not trivial to argue that it is not the case.
- When the material claims the proposed method is constant-time, it shows why that is the case.
- The material provides an implementation of the sampling method.

It is not necessary for the included material to comply with all the points above. The following is the criteria used to exclude material:

- Material is not presented in English.
- Material is not available in full-text.

3.4 Data Extraction, Analysis and Classification

To extract data from the selected material, we used a form as shown in Table 1. Data extraction was performed independently by all authors as well. Each author performed the extraction from the material found individually. Later, all authors collectively checked and corrected when necessary.

Data item	Value	RQ
Title	Name of the document	
Author(s)	Name of authors	
Context	General or cryptographic	RQ1
Type of sampling	Permutation, fixed-weight vector, integer	RQ1
If applicable, cryptographic algorithm or protocol which uses this sampling method	Name of cryptographic algorithm or protocol	RQ1 & RQ2
Is it constant-time?	Yes/No	RQ2
Is there an available implementation?	Yes/No	RQ3a
Are there benchmark results?	Yes/No	RQ3b

Table 1: Data extraction form.

During analysis and classification, we first grouped the material by the context in which the sampling algorithms are presented. That is, whether it is in a general or in a cryptographic context. Then, for the latter case, we classified the material according to the general technique used for sampling random permutations (see Sections 4 and 5) and whether a constant-time variant is available. Finally, we also classified them according to whether an implementation is publicly available.

3.5 Validity Evaluation

According to the classification by Petersen et al. [23], we identified threats to descriptive validity and theoretical validity.

As stated by Petersen et al., descriptive validity is the extent to which observations are described accurately and objectively. Similarly to the study conducted by those authors, we used a form which objectified the data extraction process and could be updated iteratively. Thus, descriptive validity was controlled this way.

Regarding theoretical validity, the identified threats are the following: (i) search strategy, since we restricted the search by the databases and keywords used; (ii) selection process, where we limited the selected studies by availability, language and level of analysis and description done by the authors when claiming that the sampling strategy is unbiased or constant-time; and (iii) bias in the selection and data extraction, analysis and classification processes.

We may have omitted relevant keywords while searching for material. Threat (i) was mitigated via the snowballing process, which allowed us to discover material initially not found when querying databases or visiting the relevant web pages. No material was excluded due to unavailability or not being written in English. Thus, threat (ii) mainly depends on the analysis provided within the considered studies. We could have omitted relevant material if such analysis was not present. In cryptography, however, it is common to avoid using techniques, primitives, etc., until they are considered to be well analysed. Since the sampling methods we ended up examining are already in use within cryptographic algorithms or protocols, they have already been analysed. Hence, we consider this threat mitigated. Threat (iii) was controlled by carrying out the selection and data processing independently and then collectively verifying and, when necessary, correcting.

4 Sorting-based permutation sampling (RQ1)

Consider sampling a list $[r_0, \ldots, r_{n-1}]$ of length n of random positive integers, such that $r_i < M$, for an integer $M \geq n$. Assume for now that they are all different from each other. From this list, we can extrapolate a permutation by looking at the position of the elements in the list in increasing order. For example, if r_i and r_j are respectively the smallest and second smallest integers of the list, then the relative permutation would have 0 in the $i-$th position, 1 in the $j-$th position, and so on.

Based on this idea, one constructs Algorithm 2 for sampling random permutations. This method was employed in [9], [19] and [15].

Algorithm 2 Sorting-based random permutation sampling.

```
Input: Positive integers $n, M$ such that $n \leq M$, a sorting algorithm $\mathsf{sort}(\cdot)$
Output: A random permutation $\pi \in \mathcal{S}_n$
 1: Initialize $\ell = [0, \ldots, 0]$
 2: repeat
 3:     for $i = 0$ to $n - 1$ do
 4:         $r \leftarrow U(\{0, 1, \ldots, M - 1\})$
 5:         $\ell(i) = (r \parallel i)$
 6:     end for
 7:     $\ell' = \mathsf{sort}(\ell)$
 8:     Extract the lists $\tau$, $\pi$ from $\ell'$ such that
                $\ell'(i) = (\tau(i) \parallel \pi(i)), \quad 0 \leq i \leq n-1$
 9: until There are no repetitions in $\tau$
10: Return $\pi$
```

The most expensive operation in the main loop of Algorithm 2 is sorting the constructed list ℓ (Line 7). Hence, the specific choice of the sorting algorithm sort highly affects the complexity of Algorithm 2. There exist fast sorting algorithms having an average running time of $O(n \log n)$, (for example, see [37–39]).

Line 9 consists of checking that the list of random integers sampled does not contain any repetition. This is done to avoid any kind of bias produced by sort. Indeed, in case two or more elements are the same, it is left to the design of sort to choose which one to put first. Given that the list s is already sorted, this check has only a linear cost in n. If one were to do the check before sorting, the cost would be quadratic.

To estimate the number of iterations of the main loop of Algorithm 2, one must compute the probability P_r of rejecting a sampled permutation, that is, the probability of sampling at least twice the same element in the inner loop (Line 3). This can be computed using the *birthday-paradox* formula

$$P_r = 1 - \frac{M!}{M^n(M-n)!}.$$

The magnitude of M with respect to n determines whether the number of repetitions of the main loop in Algorithm 2 is negligible or not. It was shown in [40] that, for certain parameters M and n, one could afford skipping the collision check (Line 9) as the output distribution would still be close to uniform. However, in this work, we only consider instantiations of Algorithm 2 with rejection enabled.

Properties for use in cryptography (RQ2). In a cryptographic context, it is important that a permutation sampling algorithm exhibits no sensitive bias. The distribution of permutations generated by Algorithm 2 underwent scrutiny in [40], revealing that any bias introduced through this method is negligible for cryptographic applications.

Additionally, secure implementations necessitate both the random sampler (Line 4) and the sorting algorithm (Line 7) to be resistant against side-channel attacks. Achieving side-channel resistance for the random sampler involves a straightforward approach: set $M = 2^k$ and utilize a secure implementation of a PRNG or XOF to acquire k pseudo-random bits at each iteration. On the other hand, for the sorting algorithm, one can consider the *djbsort* software library [35]. Initially introduced in [41] for fixed-weight vectors sampling, this library incorporates an efficient constant-time sorting algorithm.

5 Fisher-Yates style permutation sampling (RQ1)

Introduced in an early version by Ronald Fisher and Frank Yates [24], but firstly described in its modern version by Richard Durstenfeld [42], the so-called Fisher-Yates algorithm has been a standard choice for sampling permutations. Before giving the details and its variants, we start by giving a generalization of it.

Generalization of Fisher-Yates. On a high level, one can sample a random permutation in $\mathcal{S}_n$ by sampling uniformly at random without replacement from the set $\{0, 1, \ldots, n-1\}$, and storing the elements in a list keeping the sampling order. This idea is summarized in Algorithm 3.

At each step of the iteration, the index is sampled uniformly at random from all possible indices. This ensures that one can sample any permutation with the same probability. In practice, sampling without replacement from a set is a non-trivial computational task. As we will see in the following subsections, Fisher-Yates and its variants can be seen as different instantiations of Algorithm 3, each one with different routines for sampling without replacement.

More generally, one can obtain a new permutation sampling algorithm by proposing an alternative method for sampling integers without replacement from a set, and this is what we do in Section 5.3.

Algorithm 3 Sampling without replacement basic idea.

```
Input: Integer $n \geq 1$
Output: a permutation $\pi \in \mathcal{S}_n$
1: Set $\mathcal{A} = \{0, 1, \ldots, n-1\}$
2: Initialize $\pi = [\cdot]$
3: for $i = 0$ to $n-1$ do
4:     $\pi(i) \leftarrow U(\mathcal{A})$
5:     $\mathcal{A} = \mathcal{A} \setminus \{\pi(i)\}$
6: end for
7: Return $\pi$
```

5.1 Classic Fisher-Yates

We report in Algorithm 4 the Fisher-Yates algorithm in its classical instantiation. It represents an instantiation of Algorithm 3 where the operation swap is used to perform sampling without replacement from a set. On a high level, the swap moves the sampled element to the left of the list and leaves the ones still to be sampled to the right. An alternative mathematical interpretation of it is that it exploits the fact that any permutation can be decomposed into a sequence of swaps.

Algorithm 4 Classic Fisher-Yates.

```
Input: A positive integer $n$
Output: A random permutation $\pi \in \mathcal{S}_n$
1: Initialize $\pi = [0, 1, \ldots, n-1]$
2: for $i = n-1$ down to $0$ do
3:     $j \leftarrow U(\{0, \ldots, i\})$
4:     swap$(\pi(i), \pi(j))$
5: end for
6: Return $\pi$
```

In [43], a variant of Algorithm 4 is presented where the vector initialization (Line 1) and the swap (Line 4) are replaced by two instructions $\pi(i) = \pi(j)$ and $\pi(j) = i$. This simplification is the idea at the basis of Sendrier's variant, which will be described in Section 5.2. Another alternative and equivalent version exists with the for loop going in increasing order and the index sampling being done in the set $\{i, \ldots, n-1\}$.

Algorithm 4 is fast and easy to implement. The time complexity is $O(n)$, however, it comes with a hidden cost dictated by swap (Line 4). The unpredictable random memory access necessary to perform the swap slows down the performance when n is large.

Properties for use in cryptography (RQ2). As mentioned at the beginning of the section, sampling indices without replacements yields a permutation sampled with no bias. From a cryptographic standpoint, the vulnerability of Algorithm 4 to side-channel attacks arises from secret-dependent memory access, leaking timing information related to the permutation, particularly when the permutation is a part of sensitive data. Nevertheless, Classic Fisher-Yates is used in the key-generation routines of the digital signatures LESS [11] and EHT [12]. The variants detailed in Section 5.2 and Section 5.3 address this issue but come at the expense of an increased time complexity.

Another drawback of Fisher-Yates, especially when compared against Algorithm 2, is that sampling from an interval that is not necessarily a power of two (Line 3) requires additional care in order to exclude the possibility of any unwanted bias. In this work, we explore techniques previously documented in Section 2.4 to navigate this challenge.

5.2 Sendrier's Fisher-Yates variant

In order to address the vulnerabilities that a naïve implementation of Algorithm 4 comes with, Nicolas Sendrier proposed two constant-time variants of Fisher-Yates that avoid data-dependant memory access [36]. The work of Sendrier was done for sampling low-weight vectors. In this case, only one of the two algorithms presented by Sendrier [36, Algorithm 3] can be adapted for sampling permutations. The other variant [36, Algorithm 4] allows a slightly more efficient implementation when sampling low-weight vectors, but it samples the indices in a pre-fixed order, and, for this reason, it cannot be used for sampling random permutations.

We report in Algorithm 5 the adaptation of [36, Algorithm 3] to random permutation sampling. One can see that the memory accesses are independent from the sensitive data (the permutation values), and Line 5 allows non-branching constant-time implementations. The asymptotic time complexity increased to $O(n^2)$. More precisely, Line 5 is repeated $\frac{n(n-1)}{2}$ times.

Algorithm 5 Sendrier's Fisher-Yates.

```
Input: A positive integer $n$
Output: A random permutation $\pi \in \mathcal{S}_n$
1: Initialize $\pi = [0, \ldots, 0]$
2: for $i = n - 1$ down to $0$ do
3:     $\pi(i) \leftarrow U(\{i, \ldots, n-1\})$
4:     for $i = i + 1$ to $n - 1$ do
5:         $\pi(j) = (\pi(j) = \pi(i))\ ?\ i\ :\ \pi(j)$
6:     end for
7: end for
8: Return $\pi$
```

A nice property of Algorithm 5 is that the shuffle produces the same permutation as Algorithm 4 when the input seed to the PRNG is the same. This property is useful in cryptography when one is required to sample the same permutation both in constant and non-constant time (for example, during signing and verification in a digital signature protocol).

Properties for use in cryptography (RQ2). Being a variant of Fisher-Yater, this technique also samples a permutation with no bias. Sendrier's variant specifically addresses the issue that made classic Fisher-Yates vulnerable.

5.3 Natural Fisher-Yates: a New Variant

Let us represent the sampling set $\mathcal{A}$ in Algorithm 3, at each iteration, as a sorted list according to the *natural* order of integers. In this case, instead of keeping track of the sampling set (i.e., the full set minus the already sampled elements), one can keep track of the positions of the *missing* numbers (the ones sampled at the previous iterations) in the list. Based on this idea, we construct Algorithm 6.

Algorithm 6 Natural Fisher Yates.

```
Input: Integer $n \geq 1$
Output: A permutation $\pi \in \mathcal{S}_n$
 1: Initialize $\ell, \pi = [0, \dots, 0]$
 2: for $i = 0$ to $n - 1$ do
 3:     $\ell(i) \leftarrow U(\{0, 1, \dots, n - 1 - i\})$
 4:     $T = i$
 5:     for $j = 0$ to $i - 1$ do
 6:         $M = (\ell(j) \geq \ell(i))\ ?\ 1\ :\ 0$
 7:         $T = T - M$
 8:         $\ell(j) = \ell(j) - M$
 9:     end for
10:     $\pi(i) = \ell(i) + T$
11: end for
12: Return $\pi$
```

Proposition 1. *Algorithm 6 returns a uniformly random permutation of $\mathcal{S}_n$.*

Proof. We want to prove that, at each iteration i, $\pi(i)$ is a positive integer sampled uniformly at random from the set

$$\{0, 1, \dots, n - 1\} \setminus \{\pi(0), \pi(1), \dots, \pi(i - 1)\}.$$

Let $a_0 = [0, 1, \dots, n - 1]$ be the list representing the sampling set at the first iteration. Let $\ell(0) \coloneqq R_0 \leftarrow U(\{0, \dots, n-1\})$ be the sampled index (Line 3). Trivially, $\pi(0) \coloneqq \ell(0) = a_0(R_0) = R_0$ is uniform in $\{0, 1, \dots, n - 1\}$.

At the second iteration, we define the sampling list as the set $\{0, 1, \dots, n - 1\} \setminus \{\pi(0)\}$ sorted according to the natural order, i.e.,

$$a_1 = [0, \dots, \ell(0) - 1, \ell(0) + 1, \dots, n - 1].$$

Then, one samples $\ell(1) \coloneqq R_1 \leftarrow U(\{0, \dots, n-2\})$. If $\ell(1) \geq \ell(0)$, then we are sampling a number to the right of the "skipped" position at $\ell(0)$ in a_1, and so we must take $\pi(1) \coloneqq a_1(R_1) = R_1 + 1$. Otherwise, if $\ell(1) < \ell(0)$, we have that $\pi(1) \coloneqq a_1(R_1) = R_1$. Therefore, $\pi(1)$ can take any value in $\{0, 1, \dots, n-1\} \setminus \{R_0\}$ with the same probability. However, in the second case next iteration, one must take into consideration that in the sampling list

$$a_2 = [0, \ldots, \ell(1) - 1, \ell(1) + 1, \ldots, \ell(0) - 1, \ell(0) + 1, \ldots, n - 1],$$

the index corresponding to the skipped position $\ell(0)$ gets moved by one position to the left, i.e., $\ell(0) - 1$, therefore we must update $\ell(0) \coloneqq \ell(0) - 1$ (Line 8).

In general, at the i-th iteration, we sample $\ell(i) \coloneqq R_i \leftarrow U(\{0, \ldots, i\})$ and we get that

$$\pi(i) \coloneqq \ell(i) + \sum_{0 \leq j < i} \mathbf{1}_{\ell(i) > \ell(j)},$$

where the $\ell(j)$ correspond to the current skip positions. Hence, with the same probability, $\pi(i)$ can take any value from $\{0, 1, \ldots, n - 1\} \setminus \{\pi(0), \pi(1), \ldots, \pi(i - 1)\}$.

Note that, at end of the for loop, we have that $\ell(i) = 0$, for $i = 0, \ldots, n-1$, because every time we sample a $R_j < \ell(i)$, we decrease $\ell(i)$ by 1. Given that there are exactly R_i elements left to be sampled to the left of $a_i(R_i)$ at iteration i, we will decrease $\ell(i)$ by 1 exactly R_i times. □

Properties for use in cryptography (RQ2). The time complexity of Algorithm 6 is $O(n^2)$, same as Sendrier's Fisher-Yates variant. Also, it allows a constant-time implementation without secret-dependant memory accesses or branching.

5.4 On Parallelizing Fisher-Yates

Several efforts have been made outside the scope of cryptography to parallelize Fisher-Yates, and several works on this topic can be found easily using the search engines mentioned in Section 3.2. We decided to exclude this line of research from our analysis for the following reasons. Most of the works that we have been able to list were not intended to be used in cryptography, and so lacked, in some cases, an analysis of the randomness of the outcome, and in all cases, a constant-time implementation. We emphasize once more here that making an algorithm constant-time usually has non-trivial implications for its efficiency. Therefore, to extrapolate any conclusions relevant for cryptography from these works, one would need to write a dedicate constant-time implementation version of these, a task that is outside the scope of this systematization of knowledge. Finally, the few works we could find on this line of research that were actually intended for cryptographic purposes had a focus on hardware implementations, which, again, goes beyond the focus of this work.

6 Comparisons and Experimental Results

6.1 Theoretical Comparison (RQ3)

Table 2 summarizes the main differences between the algorithms presented in Section 4 and Section 5. At an asymptotic complexity level, the fastest algorithm for permutation sampling allowing constant-time implementations is Algorithm 2, with the condition that the number of repetitions in its main loop is negligible. Classic Fisher-Yates (Algorithm 4) seems to be the best choice when a secure implementation is not required. However, the hidden cost coming with the operation `swap` makes it hard to state, on

	Constant-Time	Time Complexity	Space Complexity
Algorithm 2	Yes	$O(n \log(n))$	$O(n)$
Algorithm 4	No	$O(n)$	$O(n)$
Algorithm 5	Yes	$O(n^2)$	$O(n)$
Algorithm 6	Yes	$O(n^2)$	$O(n)$

Table 2: Theoretical Comparison of the Algorithms

a theoretical level, whether this is faster or slower than Algorithm 2. Typically, one expects `swap` to be very fast when the array is small, and become slower with longer arrays. Such kind of conclusion is left for experimental comparison.

Sendrier's and Natural Fisher-Yates variants (Algorithm 5 and Algorithm 6, respectively) share the same asymptotic time complexity and are both constant-time. For most applications, they do not represent the best choice of permutation sampling algorithm in terms of efficiency. However, for very large permutations, if a constant-time sorting algorithm that sorts big data-types is not available, the random buffer might not be enough to build a low-rejection sorting-based sampling algorithm (i.e., there are too many repetitions in the main loop in Algorithm 2). In this case, Algorithm 5 and Algorithm 6 might become a valid choice. Among the two, Sendrier's variant has the advantage that it is compatible with Classic Fisher-Yates. Therefore, the latter can be used in tandem with the former when the same permutation needs to be sampled twice, once securely and once non-securely. On the other hand, Natural Fisher-Yates zeroes the random buffer by design of the algorithm, an operation required in cryptography to not leave sensitive data in memory.

This theoretical analysis does not take into consideration the cost of sampling random data in the format required by the algorithm. This operation is intuitively more costly in Fisher-Yates algorithms than in Algorithm 2. The magnitude of this cost will be evaluated in the experimental comparison in the next section.

6.2 Experimental Comparison (RQ3)

To validate the conclusions resulted from the theoretical comparisons, we implemented all algorithms reported in this study in pure `C` and executed benchmark tests for different permutation sizes[3]. We instantiated Algorithm 2 using the *djbsort* constant-time software library [35] (version 20190516). In addition, with the objective of obtaining a faster non-constant-time alternative to *djbsort* and based on the analysis in [34], we chose to employ *vqsort* [44, 45]. We used the *xkcp* SHAKE implementation [46] (commit 7fa59c0ec4) to instantiate a PRNG used for sampling integers.

Optimizations. We run the experiments with and without optimization. In the former case, we enabled AVX2 compilation flags. In addition, we also use AVX2 optimized routines for the internal SHAKE functions, for *djbsort-32*, for *vqsort-32/64*, for Natural Fisher-Yates and Sendrier Fisher-Yates. We remark that Classic Fisher-Yates and *djbsort-64* are not optimized with AVX2 code and any optimization to these operations come directly from the compiler.

[3]Our code is available at https://github.com/AlessandroBudroni/Sampling_Random_Permutations.

Constant-time evaluation. We evaluate the constant-time execution of all the permutation sampling algorithms implemented using the software tool *dudect* [47] for $n = 1024$ and timeout of 20 minutes. Additionally, for the constant-time evaluation only, we disable any aborts from the sampling algorithm due to failure. As expected, Classic Fisher-Yates quickly fails the constant-time test with dudect. During the 20 minutes execution time for each of the constant-time algorithms (Sendrier/Natural Fisher Yates, Algorithm 2 with *djbsort*), dudect does not find evidence of non constant-time execution. Surprisingly, also the non-optimized implementations of Algorithm 2 using *vqsort-32* and *vqsort-64* seem to run in constant-time (probably due to the emulation of the vectorized instructions). Nonetheless, the AVX2 optimized versions of the Algorithm 2 with *vqsort* did not run in constant-time.

Benchmark evaluation. The benchmarks were executed on an Intel Core i9-13900K (3 Ghz) CPU with Hyper-Threading and Turbo Boost features disabled. Each plot in this section corresponds to the total time (in clock cycles) given in log scale of computing the respective operation 10^5 times. We run the experiments for permutations of size 2^n with $5 \leq n \leq 12$.

Given that the algorithms based on Fisher-Yates require random integer sampling, we start by discussing the performance implications of this operation. In Figure 1, we benchmark the constant-time (CT) and Lemire integer samplings with an additional buffered version of each technique, without AVX2 (either optimized code or compilation flags). The buffered version samples all random values once, instead of on-demand. As depicted in the figure, the performance of the constant-time sampling is over one order of magnitude slower due to oversampling and no aborts. In addition, we see a considerable performance speedup when using the buffered version of the sampling algorithms. For this reason, we benchmark all permutation sampling algorithms using the buffered CT sampling, with the exception of Classic Fisher-Yates since it is already non-constant-time.

At first glance, we remark that the running time of the Classic Fisher-Yates, Natural Fisher-Yates and Sendrier Fisher-Yates algorithms are dominated by the underlying integer sampling algorithm, that is highlighted in Figure 2 and Figure 3 using dotted lines. In the case of Classic Fisher-Yates the integer sampling seems to dominate for all values of n. For Natural Fisher-Yates and Sendrier Fisher-Yates, the algorithm running time diverges for higher values of n.

Considering only the constant-time Fisher-Yates based sampling algorithms, we observe that Natural Fisher-Yates has better performance than Sendrier Fisher-Yates for larger sizes of permutation, when optimization is not available. Otherwise, both achieve the same performance.

Clearly the fastest alternative to permutation sampling in our experiments is the sort sampling. We show how the constant-time *djbsort* sampling performs when compared with the recently claimed fastest sorting algorithm, *vqsort*. For the unoptimized experiments, *djbsort* unequivocally beats *vqsort* at sampling permutations smaller than 2^{12}, see Figure 2. This test uses *vqsort* "EMU128" compilation option to disable AVX2, which might make it slower as it does not support non-vectorized sorting. For a fairer comparison, we observe that *djbsort-32-avx2* still outperforms *vqsort-32-avx2*

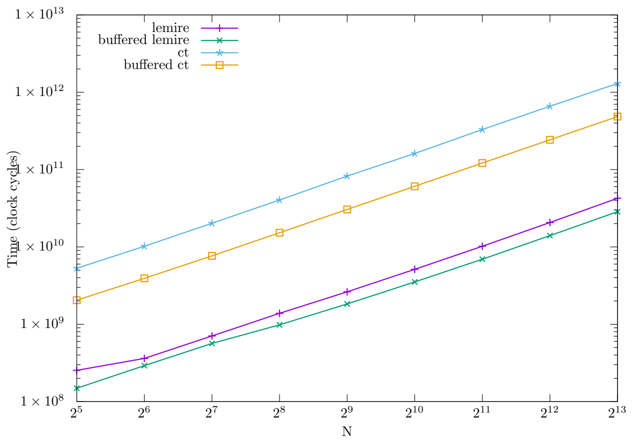

Fig. 1: Uniform integer sampling from a range: comparisons of the considered methods for uniform integer sampling from a range. No AVX2 optimization was used in this case.

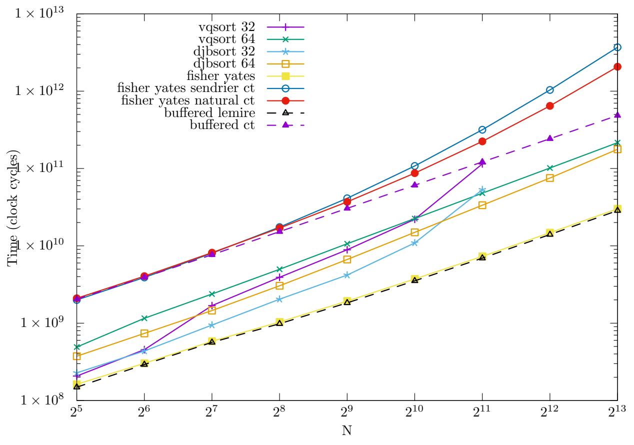

Fig. 2: Permutation sampling: performance comparison of the non-optimized (non-AVX2) implementations of the presented algorithms.

in Figure 3 for $n \geq 2^8$, while the same does not hold for *djbsort-64-avx2* compared to *vqsort-64-avx2*. This could be due to the lack of available AVX2 code optimization for *djbsort* with *uint64_t*, recalling that the AVX2 optimization for this scheme is left for the compiler.

A final observation of the sort sampling is the behavior when $n > 2^{10}$. Because the concatenation of the random value and the permutation index in Line 5 of Algorithm 2 is limited to the size of the sorted register, the probability Algorithm 2 exits the repeat loop is decreased as the size of the permutation increases. This probability degrades quickly after $n = 2^{10}$ and is visible in both figures with sampling running times. Indeed,

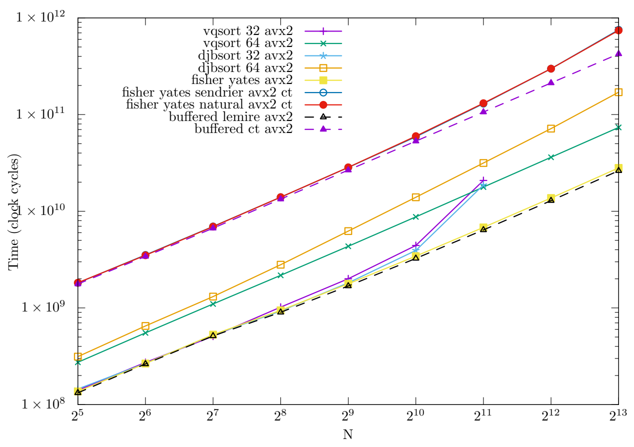

Fig. 3: Optimized permutation sampling: performance comparison of the AVX2 optimized implementations of the presented algorithms.

sort sampling is faster with 64 bit integers for $n = 2^{11}$ without optimization and should be also true for AVX2 optimization if *djbsort-64* was instructed with optimized code. Nonetheless, sort-sampling is unfeasible with 32 bit integers when $n = 2^{12}$, as the algorithm will repeat the loop iteration with overwhelming probability.

7 Conclusions and Future Directions

In this manuscript, we systematically studied algorithms for random permutation sampling in cryptography. Our experimental results suggest that the *djbsort*-based instantiations of Algorithm 2 dominate over the Fisher-Yates constant-time variants. Moreover, the subroutine of sampling integers from a range present in Fisher-Yates-like algorithms requires a non-trivial dedicate implementation effort. Sorting-based permutation samplers do not have this problem since the size of their random buffer is a power of 2.

Given that we failed in finding a faster and not necessarily constant-time alternative to *djbsort-32*, this task remains open. Succeeding in doing so would help speeding-up certain computations in protocols where constant-time is not required, (e.g., verification in [15]), and we leave this for future investigation.

Contrary to our expectations, Figure 2 and Figure 3 show that for Fisher-Yates-like algorithms, the integer sampling subroutine within a certain range dominates over the shuffle. We leave as future work improving such subroutine. Our Fisher-Yates variant (Algorithm 6) shows a slight advantage over Sendrier's (Algorithm 5). However, both of them are significantly less efficient compared to the instances of Algorithm 2. It would be interesting to find another faster variant to use in combination with non-secure Classic Fisher-Yates.

In addition, given the current state-of-the-art implementation, we believe that AVX2 versions of *djbsort-16* and *djbsort-64* would give better performance results for very small and very large permutations, respectively, and for permutation compositions and inversions (see Appendix A).

One final direction for further investigation is to study shuffling algorithms for sampling permutations that were proposed outside the realm of cryptography (e.g. [48, 49]), analyze their bias, and produce constant-time implementations to evaluate their effective efficiency for cryptographic applications.

References

[1] Katz, J., Lindell, Y.: Introduction to Modern Cryptography, 2nd edn. Cryptography and Network Security. CRC Press, (2014)

[2] Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., Dray, J.: Advanced Encryption Standard (AES). Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD (2001)

[3] Daemen, J., Rijmen, V.: The Design of Rijndael. Information Security and Cryptography. Springer, (2002)

[4] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic sponge functions (2011). https://keccak.team/papers.html Accessed 2023-08-18

[5] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak reference (2011). https://keccak.team/papers.html Accessed 2023-08-18

[6] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge-based pseudo-random number generators. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 33–47. Springer, (2010). https://doi.org/10.1007/978-3-642-15031-9_3

[7] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the sponge: Single-pass authenticated encryption and other applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, (2012). https://doi.org/10.1007/978-3-642-28496-0_19

[8] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Permutation-based encryption, authentication and authenticated encryption (2012). https://keccak.team/papers.html Accessed 2023-08-18

[9] Albrecht, M.R., Bernstein, D.J., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., Maurich, I., Misoczki, R., Niederhagen, R., Paterson, K.G., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Tjhai, C.J., Tomlinson, M., Wang, W.: ClassicMcEliece. Tech. rep., National Institute of Standards and Technology (2020). https://classic.mceliece.org/

[10] NIST: Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography (2017)

[11] Baldi, M., Beckwith, A.B.L., Biasse, J.-F., Esser, A., Gaj, K., Mohajerani, K., Pelosi, G., Persichetti, E., Saarinen, M.-J.O., Santini, P., Wallace, R.: LESS (version 1.1). Tech. rep., National Institute of Standards and Technology (2023). https://www.less-project.com/

[12] Semaev, I., Feussner, M.: EHTv3 and EHTv4. Tech. rep., National Institute of Standards and Technology (2023). https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures

[13] Standards, N.I., Technology: Post-Quantum Cryptography: Digital Signature Schemes. https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures (2023)

[14] Beullens, W., Faugère, J.-C., Koussa, E., Macario-Rat, G., Patarin, J., Perret, L.: PKP-Based Signature Scheme. Cryptology ePrint Archive, Paper 2018/714. https://eprint.iacr.org/2018/714 (2018). https://eprint.iacr.org/2018/714

[15] Aaraj, N., Bettaieb, S., Bidoux, L., Budroni, A., Dyseryn, V., Esser, A., Gaborit, P., Kulkarni, M., Mateu, V., Palumbi, M., Perin, L., Tillich, J.-P.: PERK. Tech. rep., National Institute of Standards and Technology (2023). https://pqc-perk.org/

[16] Cryptographic Research, C.A.: National Cryptography Algorithm Design Competition. https://www.cacrnet.org.cn/site/content/854.html (2020)

[17] Shamir, A.: An efficient identification scheme based on permuted kernels (extended abstract) (rump session). In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435, pp. 606–609. Springer, (1990). https://doi.org/10.1007/0-387-34805-0_54

[18] Bidoux, L., Gaborit, P.: Compact post-quantum signatures from proofs of knowledge leveraging structure for the pkp, sd and rsd problems. In: El Hajji, S., Mesnager, S., Souidi, E.M. (eds.) Codes, Cryptology and Information Security, pp. 10–42. Springer, Cham (2023)

[19] Beullens, W.: Sigma protocols for MQ, PKP and SIS, and Fishy signature schemes. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 183–211. Springer, (2020). https://doi.org/10.1007/978-3-030-45727-3_7

[20] Feneuil, T.: Building MPCitH-based Signatures from MQ, MinRank, Rank SD and PKP. Cryptology ePrint Archive, Paper 2022/1512. https://eprint.iacr.org/2022/1512 (2022). https://eprint.iacr.org/2022/1512

[21] Feneuil, T., Joux, A., Rivain, M.: Shared permutation for syndrome decoding: new zero-knowledge protocol and code-based signature. Designs, Codes and Cryptography **91**, 1–46 (2022) https://doi.org/10.1007/s10623-022-01116-1

[22] Bidoux, L., Gaborit, P., Kulkarni, M., Mateu, V.: Code-based signatures from new proofs of knowledge for the syndrome decoding problem. Designs, Codes and Cryptography **91**, 1–48 (2022) https://doi.org/10.1007/s10623-022-01114-3

[23] Petersen, K., Vakkalanka, S., Kuzniarz, L.: Guidelines for conducting systematic mapping studies in software engineering: An update. Information and Software Technology **64**, 1–18 (2015)

[24] Fisher, R.A., Yates, F.: Statistical Tables for Biological, Agricultural and Medical Research, 6th ed. edn. Oliver & Boyd, London, (1938)

[25] Lehmer, D.H.: Lehmer, d. h. teaching combinatorial tricks to a computer. Proceedings of Symposia in Applied Mathematics **10**, 179–193 (1960) https://doi.org/10.1090/psapm/010

[26] Bonet, B.: Efficient algorithms to rank and unrank permutations in lexicographic order. Workshop on Search in Artificial Intelligence and Robotics - Technical Report (2008)

[27] Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79**(6), 281–284 (2001)

[28] Lemire, D.: Fast random integer generation in an interval. ACM Trans. Model. Comput. Simul. **29**(1) (2019) https://doi.org/10.1145/3230636

[29] Guo, Q., Hlauschek, C., Johansson, T., Lahr, N., Nilsson, A., Schröder, R.L.: Don't Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE. Cryptology ePrint Archive, Paper 2021/1485. https://eprint.iacr.org/2021/1485 (2021). https://eprint.iacr.org/2021/1485

[30] Krausz, M., Land, G., Richter-Brockmann, J., Güneysu, T.: A holistic approach towards side-channel secure fixed-weight polynomial sampling. In: Boldyreva, A., Kolesnikov, V. (eds.) Public-Key Cryptography - PKC 2023 - 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7-10, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13941, pp. 94–124. Springer, (2023). https://doi.org/10.1007/978-3-031-31371-4_4 . https://doi.org/10.1007/978-3-031-31371-4_4

[31] Arndt, J.C.: Generating Random Permutations. Australian National University. https://jjj.de/pub/arndt-rand-perm-thesis.pdf (2010). https://jjj.de/pub/arndt-rand-perm-thesis.pdf

[32] Nasim, Z., Bano, Z., Ahmad, M.: Analysis of efficient random permutations generation for security applications. In: 2015 International Conference on Advances in Computer Engineering and Applications, pp. 337–341 (2015). https://doi.org/10.1109/ICACEA.2015.7164726

[33] Park, J.-Y., Ju, J.-W., Lee, W., Kang, B.-G., Kachi, Y., Sakurai, K.: A Statistical Verification Method of Random Permutations for Hiding Countermeasure Against Side-Channel Attacks. Cryptology ePrint Archive, Paper 2023/1750. https://eprint.iacr.org/2023/1750 (2023). https://eprint.iacr.org/2023/1750

[34] Bergdoll, L.: 10 17x faster than what? A performance analysis of Intel's x86-simd-sort (AVX-512). https://github.com/Voultapher/sort-research-rs/blob/main/writeup/intel_avx512/text.md (2023)

[35] Bernstein, D.J.: djbsort software library. https://sorting.cr.yp.to/ (2019)

[36] Sendrier, N.: Secure Sampling of Constant-Weight Words – Application to BIKE. Cryptology ePrint Archive, Paper 2021/1631. https://eprint.iacr.org/2021/1631 (2021). https://eprint.iacr.org/2021/1631

[37] Ajtai, M., Komlós, J., Szemerédi, E.: An o(n log n) sorting network. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing. STOC '83, pp. 1–9. Association for Computing Machinery, New York, NY, USA (1983). https://doi.org/10.1145/800061.808726 . https://doi.org/10.1145/800061.808726

[38] Sedgewick, R.: Implementing quicksort programs. Commun. ACM **21**(10), 847–857 (1978) https://doi.org/10.1145/359619.359631

[39] Kim, P.-S., Kutzner, A.: Ratio based stable in-place merging. In: Agrawal, M., Du, D., Duan, Z., Li, A. (eds.) Theory and Applications of Models of Computation, pp. 246–257. Springer, Berlin, Heidelberg (2008)

[40] Bernstein, D.J.: Divergence bounds for random fixed-weight vectors obtained by sorting. https://ntruprime.cr.yp.to/divergence-20180430.pdf (2020)

[41] Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU prime: Reducing attack surface at low cost. In: Adams, C., Camenisch, J. (eds.) SAC 2017. LNCS, vol. 10719, pp. 235–260. Springer, (2017). https://doi.org/10.1007/978-3-319-72565-9_12

[42] Durstenfeld, R.: Algorithm 235: Random permutation. ACM **7**(7), 420 (1964) https://doi.org/10.1145/364520.364540

[43] Knuth, D.E.: The Stanford GraphBase: A Platform for Combinatorial Computing. Association for Computing Machinery, New York, NY, USA (1993)

[44] Blacher, M., Giesen, J., Sanders, P., Wassenberg, J.: Vectorized and performance-portable Quicksort (2022)

[45] Blacher, M., Giesen, J., Sanders, P., Wassenberg, J.: Vectorized and performance-portable Quicksort. https://github.com/google/highway/tree/master/hwy/contrib/sort (June 2023)

[46] Team, K.: eXtended Keccak Code Package. https://github.com/XKCP/XKCP

[47] Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? Cryptology ePrint Archive, Paper 2016/1123. https://eprint.iacr.org/2016/1123 (2016). https://eprint.iacr.org/2016/1123

[48] Bacher, A., Bodini, O., Hollender, A., Lumbroso, J.O.: Mergeshuffle: a very fast, parallel random permutation algorithm. In: International Conference on Random and Exhaustive Generation of Combinatorial Structures (2015). https://api.semanticscholar.org/CorpusID:13919449

[49] Lawnik, M., Banasik, A.: Generating and numbering permutations with the use of chaotic maps. In: 2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), vol. 1, pp. 387–390 (2021). https://doi.org/10.1109/IDAACS53288.2021.9661044

A On Composing and Inverting Permutations

Some cryptographic protocols require to make operations with permutations such as composition and inversion. As for sampling random permutations, we give here an overview of the different methods and we report an experimental comparison of these.

Standard Permutation Inversion and Composition. Let $\pi_1, \pi_2 \in \mathcal{S}_n$ be two permutations, and assume one wants to compute the composition $\tau := \pi_1 \circ \pi_2$. Then, it is enough to compute $\tau(i) = \pi_1(\pi_2(i))$, for $i = 0, \dots, n-1$. Similarly, given a permutation $\pi \in \mathcal{S}_n$, one can evaluate its inverse π^{-1} by setting $\pi^{-1}(\pi(i)) = i$, for $i = 0, \dots, n-1$.

A naïve implementation of both these operations share the same problem of data-dependent memory access with Algorithm 4, making them not secure for a cryptographic context. The constant-time versions of these algorithms are given in Algorithm 7 and Algorithm 8. Both of them have a time complexity of $O(n^2)$.

Algorithm 7 Standard composition in constant-time

```
Input: $n \in \mathbb{N}$, $\pi_1, \pi_2 \in \mathcal{S}_n$
Output: $\tau := \pi_1 \circ \pi_2 \in \mathcal{S}_n$
1: Initialize $\tau = [0, \dots, 0]$
2: for $i = 0$ to $n-1$ do
3:     for $j = 0$ to $n-1$ do
4:         $\tau(i) = (i == \pi_2(j))\ ?\ \pi_1(i) :\ \tau(i)$
5:     end for
6: end for
7: Return $\tau$
```

Algorithm 8 Standard inversion in constant-time.

```
Input: $n \in \mathbb{N}$, $\pi \in \mathcal{S}_n$
Output: $\tau := \pi^{-1} \in \mathcal{S}_n$
1: Initialize $\tau = [0, \dots, 0]$
2: for $i = 0$ to $n - 1$ do
3:     for $j = 0$ to $n - 1$ do
4:         $\tau(j) = (j == \pi(i))\ ?\ i :\ \tau(j)$
5:     end for
6: end for
7: Return $\tau$
```

Sorting-based Composition and Inversion. Sorting algorithms can be employed also to compose and invert permutations. We give in Algorithm 9 and Algorithm 10 the relative algorithms in pseudo-code. Unlike Algorithm 2, these do not have any rejection. Hence, the complexity depends only on that of the sorting algorithm sort, which can be $O(n \log n)$. In practice, using the *djbsort* software library [35], one gets faster running times compared to secure implementation of Algorithm 7 and Algorithm 10 for both composition and inversion.

Algorithm 9 Sorting-based composition.

```
Input: $n \in \mathbb{N}$, $\pi_1, \pi_2 \in \mathcal{S}_n$
Output: $\tau := \pi_1 \circ \pi_2 \in \mathcal{S}_n$
1: $\ell = [\cdot]$
2: Compute $\pi_2^{-1}$ the inverse of $\pi_2$
3: for $i = 0$ to $n - 1$ do
4:     $\ell(i) = (\pi_2^{-1}(i) \parallel \pi_1(i))$
5: end for
6: $\ell' = \mathsf{sort}(\ell)$
7: Extract $\tau$ from $\ell'$ such that
       $$\ell(i)' = (i \mid\mid \tau(i)), \quad 0 \leq i < n$$
8: Return $\tau$
```

Note that in Algorithm 9, the sorting algorithm is called twice. Indeed, one call to sort is contained in Line 2. If one were to skip that line and define ℓ such that $\ell(i) = (\pi_2(i) \parallel \pi_1(i))$, one would obtain $\tau = \pi_1 \circ \pi_2^{-1}$. This explains why one needs to first invert π_2. However, when computing the composition of several permutations

$$\pi = \pi_1 \circ \pi_2 \circ \ldots \circ \pi_m,$$

one can proceed as follows. First invert π_m, then compute $\pi_m^{-1} \circ \pi_{m-1}^{-1}$ without the need to invert π_{m-1} and with only one query to sort (as explained above). Hence, one continues in the same way for all the remaining permutations in decreasing order. We

Algorithm 10 Sorting-based inversion.

```
Input: $n \in \mathbb{N}$, $\pi \in \mathcal{S}_n$
Output: $\tau := \pi^{-1} \in \mathcal{S}_n$
1: $\ell = [\cdot]$
2: for $i = 0$ to $n-1$ do
3:     $\ell(i) = (\pi(i) \parallel i)$
4: end for
5: $\ell' = \mathsf{sort}(\ell)$
6: Extract $\tau$ from $\ell'$ such that
       $\ell(i)' = (i \mid\mid \tau(i)), \quad 0 \le i < n$
7: Return $\tau$
```

have then

$$\pi^{-1} = \pi_m^{-1} \circ \pi_{m-1}^{-1} \circ \ldots \circ \pi_1^{-1}.$$

Finally, it is enough to invert π^{-1} to obtain the desired permutation π. This approach allows to save (almost) half of the calls to sort, and was used, for example, in [19] and [15].

Experimental Comparison. Figure 4 and Figure 5 report the result of the experimental comparisons of the algorithms reported in this section. The experiments have been performed on the same machine specified in Section 6.2, and no AVX2 optimized code or compilation flag was used in this case.

The resulting plots confirm the estimated asymptotical complexity of the algorithms. The sorting-based approach (Algorithm 9 and Algorithm 10) is the best choice in most of the cases for constant-time implementations. The constant-time standard composition (Algorithm 7) presents an advantage over the sorting-based algorithms only for small permutations. As expected, the classic non-secure operations dominates in speed over all the secure methods.

Figure 5 shows a strange behavior regarding the standard inversion operation for $2^5 \le n \le 2^7$. We believe this is due to some optimization performed by the compiler.

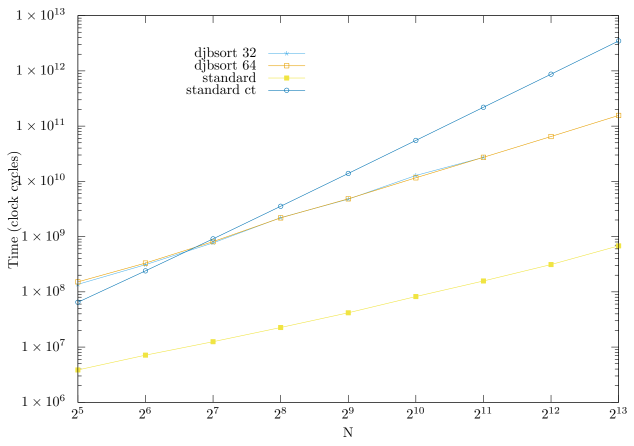

Fig. 4: Composition of permutations: The plot presents the number of clock cycles for preforming 100.000 permutation compositions for each value of n and for each algorithm. *djbsort-32* and *djbsort-64* correspond to the selected sorting algorithm chosen to instantiate Algorithm 9. *standard* stands for the naïve (not constant-time) version of Algorithm 7, which in turn is referred to as *standard ct*.

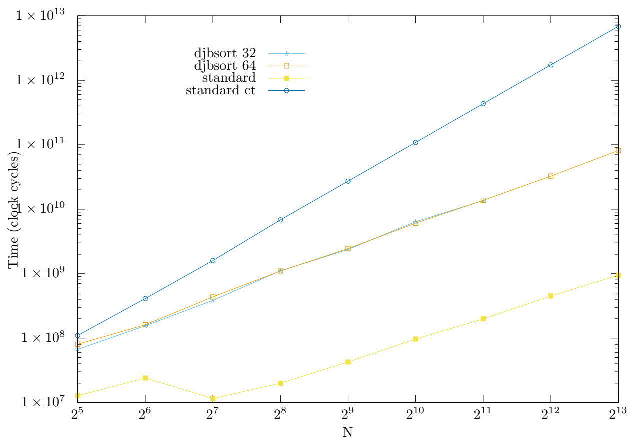

Fig. 5: Inversion of permutations: The plot presents the number of clock cycles for preforming 100.000 permutation inversions for each value of n and for each algorithm. *djbsort-32* and *djbsort-64* correspond to the selected sorting algorithm chosen to instantiate Algorithm 10. *standard* stands for the naïve (not constant-time) version of Algorithm 8, which in turn is referred to as *standard ct*.