Efficient isochronous fixed-weight sampling with applications to NTRU

Décio Luiz Gazzoni Filho[1,2], Tomás S. R. Silva[3] and Julio López[1]

[1] Universidade Estadual de Campinas (UNICAMP), Instituto de Computação, Campinas, Brazil
[2] State University of Londrina, Department of Electrical Engineering, Londrina, Brazil
[3] Universidade Estadual de Campinas (UNICAMP), Instituto de Matemática, Estatística e Computação Científica, Campinas, Brazil

Abstract. We present a solution to the open problem of designing an efficient, unbiased and timing attack-resistant shuffling algorithm for NTRU fixed-weight sampling. Although it can be implemented without timing leakages of secret data in any architecture, we illustrate with ARMv7-M and ARMv8-A implementations; for the latter, we take advantage of architectural features such as NEON and conditional instructions, which are representative of features available on architectures targeting similar systems, such as Intel. Our proposed algorithm improves asymptotically upon the current approach, which is based on constant-time sorting networks ($O(n)$ versus $O(n \log^2 n)$), and an implementation of the new algorithm is also faster in practice, by a factor of up to 6.91 (591%) on ARMv8-A cores and 12.58 (1158%) on the Cortex-M4; it also requires fewer uniform random bits. This translates into performance improvements for NTRU encapsulation, compared to state-of-the-art implementations, of up to 50% on ARMv8-A cores and 71% on the Cortex-M4, and small improvements to key generation (up to 2.7% on ARMv8-A cores and 6.1% on the Cortex-M4), with negligible impact on code size and a slight improvement in RAM usage for the Cortex-M4.

Keywords: Post-quantum cryptography · NTRU · Sampling · ARM

1 Introduction

In the late 1990s, the rise of quantum algorithms for database search and factorization [Gro96, Sho97] posed a threat to public-key cryptosystems based on integer factorization and/or discrete logarithms. Even though quantum computers capable of efficiently performing such computations do not exist yet, growing concern within the community led to seeking alternative cryptographic primitives capable of resisting attacks from quantum algorithms. Thus, *Post-Quantum Cryptography* (PQC) arises as an attempt to counter these attacks by developing new public-key cryptographic algorithms built on problems known to be resistant to quantum attacks, such as lattice-based problems.

One of the oldest lattice-based cryptosystems is NTRU, first presented in the rump session of CRYPTO '96 [HPS]. It remains relevant, as shown by advancing to the third round of the NIST PQC standardization process [CDH$^+$20, Nat17], and its standardization in other forums [IEE09, ASC10]. A performance bottleneck of NTRU is fixed-weight sampling of polynomials, i.e. those with a prescribed number of randomly permuted -1, 0 and 1 coefficients, employed in key generation and encapsulation. Unless carefully optimized, this contributes a large runtime cost, particularly to encapsulation.

E-mail: decio.gazzoni@ic.unicamp.br (Décio Luiz Gazzoni Filho), tomas@ime.unicamp.br (Tomás S. R. Silva), jlopez@ic.unicamp.br (Julio López)

Shuffling algorithms appear perfectly suited to solve the problem of fixed-weight sampling; however, there is no known efficient algorithm that is resistant to timing attacks for this problem. Instead, constant-time sorting is used to generate random permutations, as mandated by the NTRU submission to the NIST PQC contest [CDH+20]. We propose a new, timing attack-resistant shuffling algorithm to replace the sorting-based approach, with improved asymptotic running time and large performance improvements in actual implementations, especially for embedded architectures.

Prior to our work, the main proposal to avoid the cost of fixed-weight sampling for NTRU is NTRU-HRSS [HRSS17]. Their technique was later merged into the NTRU proposal for NIST's PQC standardization process [CDH+20]. Due to larger key and ciphertext sizes, it was adopted for only one out of the four suggested parameter sets.

There exist many shuffling algorithms, such as Fisher-Yates [FY38, Dur64, Knu97], Rao-Sandelius [Rao61, San62] and MergeShuffle [BBHL18]. Algorithms in the coin tossing model, aimed at minimizing the consumption of random bits, are reviewed in [BBHT17]. However, none of these are designed to resist side-channel attacks. Indeed, [Dan19] remarks that Fisher-Yates is the most straightforward implementation of fixed-weight sampling, but cautions that "implementing Fisher-Yates in such a way that there is no side channel is difficult." They opt for a constant-time sorting network, as proposed by Bernstein for use with the McEliece [BCS13] and NTRUPrime [BCLvV18] cryptosystems.

A constant-time version of Fisher-Yates has been proposed by Sendrier [Sen21] with running time $O(n^2)$ for the BIKE cryptosystem; as BIKE samples $O(\sqrt{n})$ out of n elements, its performance behaves acceptably as $O((\sqrt{n})^2) = O(n)$. In NTRU, a full shuffle is required, and Sendrier's algorithm cannot compete with $O(n \log^2 n)$ for the fastest practical sorting networks.

Our contributions. In §3, we solve the open problem of designing an unbiased shuffling algorithm resistant to timing attacks for the NTRU fixed-weight sampling problem. It is a drop-in replacement for NTRU's current sampling-by-sorting approach, improving the running time from $O(n \log^2 n)$ for the best practical sorting networks to $O(n)$, without impacting security; we discuss implementation aspects in §4. We show in §5 that an implementation of our proposed approach is considerably faster for the fixed-weight sampling step: up to $6.91\times$ (591%) on ARMv8-A cores and $12.58\times$ (1158%) on the Cortex-M4. This translates into considerable improvements for the KEM encapsulation operation (up to 50% on ARMv8-A cores and 71% on the Cortex-M4) and smaller improvements for key generation (up to 2.7% and 6.1% on the same respective platforms), with little effect on code size, and small gains in memory usage, for embedded architectures. We illustrate how to implement its main operations efficiently in the ARMv8-A and ARMv7-M architectures, as well as generic operations suitable for any architecture, and discuss possible implementations for Intel architectures. Our implementation is available under an open source license in `https://github.com/...` [1]

2 Preliminaries

2.1 NTRU random sampling

NTRU is a post-quantum public-key cryptosystem whose security relies on the difficulty of finding short vectors in high-dimensional lattices [Ajt96, MG02]. It is based on a polynomial ring over a finite field, and some of its parameters are random ternary polynomials, i.e., with coefficients in $\{-1, 0, +1\}$[2]. A subset of these are additionally restricted to being *fixed-weight*, i.e. having a prescribed number of -1, 0 and $+1$ coefficients. The NTRU

[1]Source code will be made available following the publication of the paper.

[2]Given that these are ternary polynomials, the coefficient 2 may be used interchangeably with -1.

specification [CDH+20] defines $\mathcal{T}(d)$ as the set of ternary "polynomials that have exactly $d/2$ coefficients equal to $+1$ and $d/2$ coefficients equal to -1".

The straightforward approach to sample from $\mathcal{T}(d)$ is to fix a representative of $\mathcal{T}(d)$ (e.g., -1 for the first $d/2$ coefficients, $+1$ for the next $d/2$ coefficients, and 0 for the remaining ones), and randomly permute its coefficients using a shuffling algorithm. However, known shuffling algorithms are not side-channel attack resistant [Dan19]. The usual solution to this problem, mandated by the NTRU specification [CDH+20], is based on sorting. Briefly, an array of key-value pairs are created, using uniformly random samples as keys, and values are coefficients from the chosen fixed representative of $\mathcal{T}(d)$. Sorting the random keys induces a random permutation of the coefficients. This approach is illustrated in Algorithm 1. Note that sorting must be performed in constant-time to avoid timing leaks; while most classical sorting algorithms are variable-time, sorting networks [Knu98, Section 5.3.4] are constant-time and have proven to be efficient enough in practice [BCS13, BCLvV18].

Algorithm 1 SampleFixedType: fixed-weight sampling by sorting [CDH+20, §1.10.5]

Input: $(b_0, b_1, \ldots, b_{l-1})$ (random bit string of length $l = 30(n-1)$)
Output: v (a polynomial in $\mathcal{T}(q/16-1)$)
Notes: We denote by $\textsc{Int}(x_0, \ldots, x_{k-1})$ the unsigned integer with x_j $(0 \le j \le k-1)$ at the j-th bit of its binary representation.

```
1: $\mathbf{a} \leftarrow [0, 0, \ldots, 0]$                    ▷ Array of $n-1$ zeros
2: $v \leftarrow 0$                                      ▷ The zero polynomial
3: $i \leftarrow 0$
4: for $i = 0$ to $q/16-2$ do
5:     $\mathbf{a}[i] \leftarrow 1 + 4 \cdot \textsc{Int}(b_{30i}, \ldots, b_{30i+29})$
6: for $i = q/16-1$ to $q/8-3$ do
7:     $\mathbf{a}[i] \leftarrow 2 + 4 \cdot \textsc{Int}(b_{30i}, \ldots, b_{30i+29})$
8: for $i = q/8-2$ to $n-2$ do
9:     $\mathbf{a}[i] \leftarrow 0 + 4 \cdot \textsc{Int}(b_{30i}, \ldots, b_{30i+29})$
10: Sort $\mathbf{a}$ in constant time
11: for $i = 0$ to $n-2$ do
12:     $v \leftarrow v + (\mathbf{a}[i] \bmod 4)x^i$
13: return $v$
```

2.2 Shuffling algorithms

Fisher-Yates. The Fisher-Yates shuffle algorithm, also known as Knuth's shuffle [FY38, Dur64, Knu97], is a classical technique for randomly and unbiasedly permuting elements in a collection. It is displayed in Algorithm 2.

Algorithm 2 Fisher-Yates($\mathbf{a}, n$)

Input: An array $\mathbf{a}$ of n elements
Output: A random permutation of $\mathbf{a}$

```
1: for $i = n-1$ downto 0 do
2:     $j \leftarrow$ random integer such that $0 \le j \le i$
3:     Exchange $\mathbf{a}[j]$ and $\mathbf{a}[i]$
4: return $\mathbf{a}$
```

Fisher-Yates has favourable performance characteristics: $O(n)$ running time with small constants. However, the pioneering work of [Ber04], applied to S-boxes in the AES cipher, revealed that array accesses indexed by secret data are susceptible to side-channel attacks, due to timing variabilities induced by the presence or absence of data in CPU caches. A similar attack can be mounted on Algorithm 2 by recovering the indices j in the accesses to $\mathbf{a}[j]$ in line 3, allowing the permutation to be reconstructed by an attacker. We recall that a constant-time version was proposed in [Sen21]; however, its running time for a full shuffle (as required in NTRU) is degraded to $O(n^2)$.

Rao-Sandelius. A relevant shuffling algorithm is Rao-Sandelius, independently proposed in the 1960s by [Rao61] and [San62]. It relies on a divide-and-conquer strategy.

Algorithm 3 RS($\mathbf{a}, n$): Rao-Sandelius shuffle

```
Input: An array $\mathbf{a}$ of $n$ elements
Output: A random permutation of $\mathbf{a}$
 1: if $n \leq 1$ then
 2:     return $\mathbf{a}$
 3: if $n = 2$ then
 4:     if rand-bit $= 1$ then
 5:         return $[\mathbf{a}[1], \mathbf{a}[0]]$
 6:     else
 7:         return $[\mathbf{a}[0], \mathbf{a}[1]]$
 8: Let $\mathbf{A}_0$ and $\mathbf{A}_1$ be two empty arrays
 9: for $i = 0$ to $n$ do
10:     Add $\mathbf{a}[i]$ into $\mathbf{A}_{\texttt{rand-bit}}$
11: return RS$(\mathbf{A}_0, |\mathbf{A}_0|)$ || RS$(\mathbf{A}_1, |\mathbf{A}_1|)$
```

The case $n = 2$ can be made constant-time using standard techniques. Line 10 directs each element $\mathbf{a}[i]$ to a different array depending on a random bit; by evicting both arrays from the cache for later probing, an attacker can find which array was written to. This can be countered by writing to both arrays regardless of the random bit drawn, but only incrementing the correct pointer. However, the random choice of array for assignment may lead to uneven growth of the arrays. We are unaware of any concrete analyses in the literature, but conjecture that this leaks enough data to mount a cache timing attack.

MergeShuffle. Finally, MergeShuffle, introduced in [BBHL18], "is an (easy to implement) extremely efficient algorithm to generate random permutations (or to randomly permute an existing array)". As with the Rao-Sandelius algorithm, MergeShuffle uses a divide-and-conquer strategy and is amenable to a parallel implementation.

Let k be a cut-off threshold to switch to Fisher-Yates. MergeShuffle splits an input array $(a_0, a_1, \ldots, a_{n-1})$ into 2^k blocks to be shuffled using Fisher-Yates (Algorithm 2), and then merges the resulting permutations as presented in Algorithm 4. The merging procedure is similar in spirit to that of e.g. mergesort, but it is performed in-place and uses a random bit to choose whether to swap elements from the two input arrays.

The use of Fisher-Yates as a subroutine of MergeShuffle renders it equally susceptible to cache timing attacks. It is also unclear whether the merging step can be vectorized, to attain competitive performance, and implemented in constant-time.

3 Fixed-weight sampling by constant-time shuffling

As just discussed, while shuffling is the natural solution to the fixed-weight sampling problem in NTRU, we are unaware of any shuffling algorithm resistant to side-channel attacks. In this section, we propose an efficient, unbiased and timing attack-resistant

Algorithm 4 MergeShuffle($\mathbf{a}, k$)

```
Input: An array $\mathbf{a}$ of $n$ elements
Output: A random permutation of $\mathbf{a}$
1: Divide $\mathbf{a}$ into $2^k$ blocks of roughly the same size
2: Shuffle each block independently using Fisher-Yates
3: $p \leftarrow k$
4: repeat
5:     Merge adjacent blocks of size $2^p$ into new blocks of size $2^{p+1}$    ▷ See text
6:     $p \leftarrow p + 1$
7: until $\mathbf{a}$ consists of a single block
8: return $\mathbf{a}$
```

shuffling algorithm suitable for NTRU fixed-weight sampling. Throughout this section, n is defined as in the NTRU specification and assumes values of either 509, 677 or 821.

We first describe a subroutine (Algorithm 5) to generate an array of random integers **si** such that $\mathbf{si}[i] \sim \mathcal{U}(0, n-1-i)$ for $0 \leq i < n-1$. It is a slightly modified version of [Lem19, Algorithm 5]. While other approaches exist to achieve the same result, some of which are discussed in the same paper, this method achieves the best performance among all methods we experimented with, while restricting costly (and, in all CPUs we're familiar with, variable-time) divisions by non-power-of-two integers to a pre-computation step.

Algorithm 5 RejSamplingMod(n, **rnd**): Unbiased uniform random number generation

```
Input: $n$
Input: rnd (array of random $L$-bit integers; refer to discussion in §4 for its length)
Output: si (output array of $n-1$ integers)
1: for $i = 0$ to $n-2$ do                                ▷ Precomputation
2:     $\mathbf{t}[i] = 2^L \bmod (n-1-i)$
3: $j \leftarrow 0$
4: for $i = 0$ to $n-2$ do
5:     repeat
6:         $m \leftarrow \mathbf{rnd}[j] \times (n-1-i)$
7:         $j \leftarrow j + 1$
8:         $l \leftarrow m \bmod 2^L$                      ▷ Implement with a bitmask
9:     until $l \geq \mathbf{t}[i]$
10:    $\mathbf{si}[i] \leftarrow \lfloor m/2^L \rfloor$          ▷ Implement using right-shift by $L$ bits
11: return si
```

Lemma 1 (Correctness and unbiasedness of Algorithm 5 [Lem19])**.** *Let $L \in \mathbb{N}^* = \mathbb{N} \setminus \{0\}$. Then, $\forall s \in [0, 2^L)$ and $\forall y \in [0, s)$, with s, y integers, there are $\lfloor 2^L/s \rfloor$ values of $x \in [0, 2^L)$ such that $\lfloor (x \cdot s)/2^L \rfloor = y$ and $(x \cdot s) \mod 2^L \geq 2^L \mod s$.*

Proof. Let $L \in \mathbb{N}^*$. Take $s \in [0, 2^L)$ and $y \in [0, s)$. Given $x \in [0, 2^L)$, the product $x \cdot s$ maps integers in $[0, 2^L)$ to integers in $[0, s \cdot 2^L)$. Take $z \in [0, s \cdot 2^L)$ an integer, the result of the integer division of z by 2^L is 0 if $z \in [0, 2^L)$, 1 if $z \in [2^L, 2^{L+1})$, and so on. Every interval $[i \cdot 2^L, (i+1) \cdot 2^L)$ contains $\lfloor ((i+1) \cdot 2^L - i \cdot 2^L)/s \rfloor = \lfloor 2^L/s \rfloor$ multiples of s; as such, the interval $[i \cdot 2^L + (2^L \mod s), (i+1) \cdot 2^L)$ contains $\lfloor ((i+1) \cdot 2^L - i \cdot 2^L - (2^L \mod s))/s \rfloor = \lfloor 2^L/s \rfloor$ multiples of s, since $(2^L - (2^L \mod s))$ is multiple of s. □

Lemma 1 implies that rejecting values in the interval $[i \cdot 2^L, (i+1) \cdot 2^L + 2^L \mod s)$ that are multiple of s produces exactly $\lfloor 2^L/s \rfloor$ multiples of s. We discuss issues of timing

Algorithm 6 SHUFFLE($n, c_0, c_1, \mathbf{rnd}$): Fixed-weight sampling by shuffling

```
Input: $n$
Input: $c_0, c_1$ (prescribed number of coefficients equal to 0, resp. 1)
Input: $\mathbf{rnd}$ (array of random $L$-bit integers; refer to discussion in §4 for its length)
Output: $\mathbf{v}$ (output array of $n-1$ integers)
 1: $\mathbf{si} \leftarrow$ RejSamplingMod($n, \mathbf{rnd}$)
 2: for $i = 0$ to $n-2$ do
 3:     if $\mathbf{si}[i] < c_0$ then        ▷ See text for discussion of constant-time implementation
 4:         $\mathbf{v}[i] \leftarrow 0$
 5:         $c_0 \leftarrow c_0 - 1$
 6:     else if $\mathbf{si}[i] < c_0 + c_1$ then
 7:         $\mathbf{v}[i] \leftarrow 1$
 8:         $c_1 \leftarrow c_1 - 1$
 9:     else
10:         $\mathbf{v}[i] \leftarrow -1$
11: return $\mathbf{v}$
```

attack resistance, as well as the choice of the performance-critical parameter L in §4. Algorithm 6 is our proposed shuffling approach for the fixed-weight sampling problem.

Evidently, the main loop of Algorithm 6, as presented, does not execute in constant time due to the use of branches. However, architecture-agnostic standard techniques, as well as architecture-specific conditional instructions, can be used to obtain a branchless, constant-time implementation; see §4. Moreover, all accesses to the arrays **si** and **v** are performed sequentially. We exploit the fact that $O(1)$ distinct values need to be shuffled (indeed, only 3: $-1, 0, 1$), a situation not considered in the usual shuffling algorithms. Intuitively, one could draw an analogy between Fisher-Yates shuffling and selection sort, and by replacing the latter with counting sort, arrive at our proposed algorithm.

Lemma 2 (Correctness and unbiasedness of Algorithm 6)**.** *Let n be an integer and* **rnd** *an array of random L-bit integers as in Algorithm 6. Let c_i for $i \in \{-1, 0, 1\}$ be the number of coefficients equal to i in the output polynomial, so that $c_{-1} + c_0 + c_1 = n - 1$, and write $\Sigma = \{-1\}^{c_{-1}} \times \{0\}^{c_0} \times \{1\}^{c_1}$. Then, Algorithm 6 produces an array $v = \sigma(\Sigma)$, where $\sigma \in \mathrm{Perm}(\Sigma; c_{-1}, c_0, c_1)$ is an uniformly taken permutation of Σ with c_{-1}, c_0, c_1 repetitions.*

Proof. It is immediate to verify that the output of Algorithm 6 is $\mathbf{v} = (\sigma(\Sigma_1), \ldots, \sigma(\Sigma_{n-1}))$ for some $\sigma \in \mathrm{Perm}(\Sigma; c_{-1}, c_0, c_1)$. To show unbiasedness, we proceed by directly computing $P(v = (\sigma(\Sigma_1), \ldots, \sigma(\Sigma_{n-1})))$ for an arbitrary σ. Recalling that $\mathbf{si}[j] \sim \mathcal{U}(0, n-1-j)$ for $0 \le j < n-1$, we have that

$$
\begin{aligned}
P(v = (\sigma(\Sigma_1), \ldots, \sigma(\Sigma_{n-1}))) &= \prod_{k=0}^{n-1} P(v_k = \sigma(\Sigma_K)) \\
&= \left(\prod_{l=0}^{c_{-1}-1} \frac{c_{-1} - l}{n-1-l}\right) \cdot \left(\prod_{m=0}^{c_0-1} \frac{c_0 - m}{n-1-c_{-1}-m}\right) \cdot \left(\prod_{p=0}^{c_1-1} \frac{c_1 - p}{n-1-c_{-1}-c_0-p}\right) \\
&= \frac{c_{-1}!}{\left(\prod_{l=0}^{c_{-1}-1} n-1-l\right)} \cdot \frac{c_0!}{\left(\prod_{m=0}^{c_0-1} n-1-c_{-1}-m\right)} \cdot \frac{c_1!}{\left(\prod_{p=0}^{c_1-1} n-1-c_{-1}-c_0-p\right)} \\
&= \frac{c_{-1}! c_0! c_1!}{(n-1)!} = \frac{1}{|\mathrm{Perm}(\Sigma; c_{-1}, c_0, c_1)|},
\end{aligned}
$$

where the fourth equality follows from the fact that $n = c_{-1} + c_0 + c_1$. Hence, Algorithm 6 covers all possible permutations $\sigma \in \mathrm{Perm}(\Sigma; c_{-1}, c_0, c_1)$ with uniform probability. □

Lemma 3. *Algorithm 6 executes in time $O(n)$ on average, assuming $2^L > 2n$.*

Proof. The loop of line 2 clearly executes in time $O(n)$. Thus, the remaining work consists in analyzing Algorithm 5. The outer loop (line 4) consists of $n-1$ iterations. Noticing that $\mathbf{t}[i] < n, \forall i$, the condition in line 9 will be satisfied with probability $1 - \frac{n}{2^L} > 50\%$. Thus, the expected number of iterations in the inner loop (line 5) is less than 2, so Algorithm 5 also executes in time $O(n)$ on average. □

We remark that the $O(n)$ running time of Algorithm 6 improves upon the $O(n \log^2 n)$ running time of sorting networks typically used for constant-time sorting implementations [BCS13, BCLvV18], such as Batcher's odd-even merge sort [Knu98].

The algorithm necessarily consumes at least $n - 1$ random L-bit integers and may, in principle, consume an infinite number of them due to rejections; however, in §4, we show that, for $L = 16$, generating just 4% to 5.5% extra random integers is sufficient in practice.

4 Implementation aspects

Architectural guarantees regarding constant-time execution. Both ARMv8-A and Intel architectures have recently introduced hardware flags that, when set, guarantee constant-time execution of a subset of CPU instructions, which should generally be sufficient to implement most cryptographic algorithms: FEAT_DIT for ARMv8-A [ARM23, Sections A2.6.1, B1.3.6, C5.2.4] and DOIT for Intel [Int23a, Int23b]. We verified that all instructions handling secret data in our ARMv8-A implementations are included in the affected subset.

These new features do not imply that CPUs launched prior to the introduction of these flags execute these instructions in variable time. Indeed, ARM is unaware of older CPUs with variable timing for instructions now covered by FEAT_DIT [ARM]; and Intel advises developers to assume older microarchitectures behave as if DOIT is enabled [Int23a].

This issue has garnered attention at the beginning of 2024, as Apple ARMv8-A cores (which are designed by Apple and not ARM) are subject to a microarchitectural attack called GoFetch [CWS+24]; setting the FEAT_DIT bit on the M3 disables the data memory-dependent prefetchers targeted by the attack, rendering it ineffective.

Resistance against timing attacks of Algorithm 5. There are some possible sources of timing leaks in Algorithm 5, which we enumerate and analyze.

The integer multiplication in line 6 must execute in constant-time, which is the norm in modern CPUs[3], although there are rare exceptions such as the ARM Cortex-M3 for $32 \times 32 = 64$-bit multiplications; however, $32 \times 32 = 32$-bit multiplication suffices for the purpose of this algorithm, and there is evidence that it executes in constant time in the Cortex-M3 [dG15, Por18].

Array accesses in lines 6 and 9 use sequential indices; thus, secret data is not leaked.

The loop in lines 5 to 9 performs rejection sampling based on public data, precomputed in line 2: the remainder of 2^L divided by integers in the sequence $n-1, n-2, \ldots, 1$, where L and n are public parameters. Nevertheless, given the attack of Guo et al. [GHJ+22] targeting rejection sampling in fixed-weight sampling algorithms for BIKE and HQC code-based cryptosystems, it is worth analyzing whether a similar attack could apply here. We note that their attack relies on two key assumptions:

[3]Note that multiplication instructions are covered by ARMv8-A's FEAT_DIT and Intel's DOIT flags.

1. A high rejection rate, leading to multiple calls to the `seedexpander` routine (equivalently in our case, the `randombytes` routine) which creates a timing distinguisher. As discussed later, the rejection rate for our chosen parameter $L = 16$ is sufficiently small that e.g. a full run of Algorithm 5 in the case $n = 509$ has $> 40\%$ probability of no rejections at all. Due to this low rejection rate, we sample enough uniform random integers from the outset so that the probability is overwhelmingly low ($< 2^{-74}$) that extra samples are required, avoiding potential extra calls to `randombytes` while introducing a negligible overhead.

2. Derivation of the random seed for fixed-weight sampling from secret data – namely, the output of decryption from the reencryption step of decapsulation, as required by the Fujisaki-Okamoto transform for IND-CCA security of the KEM. The attack starts by trial encrypting many candidate messages until finding an m that requires multiple calls to `seedexpander`, which gives rise to a timing distinguisher (made considerably harder, but not impossible, in our case by the first point above). Carefully constructed perturbations of the resulting ciphertext c are fed to the decapsulation procedure, while using the timing distinguisher to determine whether the decryption step of reencryption outputs the same m or a different message, allowing the attacker to learn information about the secret key. Repeated application of this procedure extracts the vast majority of key material, and the remaining bits are easily found. However, we note that NTRU does not require reencryption due to the rigidity of the NTRU DPKE [CDH+20, Figures 9 and 10]; indeed, the fixed-weight sampling algorithm is not executed at all during either decapsulation or decryption.

Thus, we conclude that Algorithm 5 does not render NTRU vulnerable to the attack of Guo et al [GHJ+22].

Choosing the parameter L. The choice of L in Algorithm 5 is a tradeoff between the cost of random number generation and the frequency of rejections; the latter lead to branch mispredictions and costly pipeline flushes in modern, highly-pipelined superscalar CPUs such as some of the ARMv8-A cores considered in this work. If samples are rarely rejected, a SIMD implementation of the algorithm becomes feasible; one can keep track of which lanes were rejected and resample them later (usually with scalar code). To minimize rejection, one must choose L such that $2^L \gg n - i$, but this translates into added cost for random number generation, and thus L should not be unreasonably larger than $n - i$.

We propose $L = 16$ as a natural choice, supported by all scalar and SIMD instruction sets we're aware of. The next smaller size, 8 bits, is insufficient for half or more of the values to be sampled in the standard NTRU parameter sets, and for most of the intervals where it is sufficient, it would lead to a high rejection rate, running counter to the SIMD philosophy. By exactly matching an available lane size, no bit shifts/masks/permutations are required to load random integers into SIMD registers, further improving performance. It is also the natural choice for storing the 11- or 12-bit NTRU polynomial coefficients; indeed, it is the representation used by the reference code and the state-of-the-art implementations we chose for performance comparisons, requiring no size conversions.

Finally and most importantly, rejections are relatively rare: a block of 16 samples is fully accepted (zero rejections) with probability at least 94.2%, 91.6% and 90.1% for $n = 509$, 677 and 821, respectively. These are minimum figures, and as $n - i$ decreases, the acceptance probability increases even further. Furthermore, the probability of accepting all $n - 1$ samples (i.e., no rejections at all during a complete execution of the algorithm) is 40.2%, 18.9% and 8.6% for $n = 509$, 677 and 821 respectively. These figures are obtained by modeling the number of required samples as a sum of geometric random variables and are displayed in a Jupyter notebook accompanying the source code of our implementation.

Due to the low rejection probability, it is sufficient to generate just a few extra random integers over the lower bound of n. For each n, we computed the cumulative distribution function $P(x \leq k)$ and sought the minimum k such that $1 - P(x \leq k) < 2^{-74}$, enough to sample $2^{10} > n$ integers for each of 2^{64} key exchanges. For $L = 16$, and rounding up to the next multiple of 8 (the number of 16-bit lanes in a NEON register), we find that 536, 704 and 856 random 16-bit integers are sufficient (i.e., an overhead of 5.5%, 4.1% and 4.4%) for $n = 509$, 677 and 821, respectively; these are the suggested lengths for the array **rnd** in Algorithms 5 and 6. This calculation is included in the aforementioned Jupyter notebook, and can be used to obtain suggested array lengths for other choices of L if desired.

One might argue that $L = 16$ is a "wasteful" choice, as it requires 123%, 109% and 103% more bits than the (unattainable) lower bound of $\log_2(n!)$ bits for $n = 509$, 677 and 821, respectively. Still, we note this is slightly more than half as many random bits as the approach dictated by the NTRU specification [CDH+20], which calls for $30 \times n$ bits.

Taking $L > 16$ appears counterproductive, e.g. due to reduced computational throughput from using larger SIMD lanes. On the other hand, in scenarios where pseudo-random number generation is expensive, SIMD is not available and pipeline flushes have less performance impact (i.e. deeply embedded cores such as the Cortex-M4), choosing $L < 16$ (say, 12 or 10) may result in better overall performance. One might even conceive of an adaptive choice, decreasing L along with $n - i$, although this results in more complex code.

SIMD implementation of Algorithm 5. To minimize the execution time of Algorithm 6, we seek to implement Algorithm 5 using SIMD instructions. At first glance, it is unsuitable for SIMD, as some lanes may be rejected while others are accepted during sampling. However, it is possible to sample a whole SIMD register and take note of which lanes, if any, were rejected, to be fixed up later using scalar code (recall that an adequate choice of L ensures that rejections only happen with low probability, so the effect of this fixup procedure on the running time is small.) We also break the dependency on the index j of the random array by using disjoint ranges of the array for SIMD sampling (indices 0 to $n - 2$) and the fixup procedure ($n - 1$ onwards). This idea is captured in Algorithm 7.

In addition to previously discussed issues of timing attack resistance of Algorithm 5, we note that any non-sequential accesses to the array **rnd** arise from switching between the ranges of indices $0 \leq i + k < n - 1$ and $j \geq n - 1$, that is, they are due to rejections and thus do not leak secret data; accesses within each range are sequential.

Line 10 should use SIMD comparison instructions (e.g. NEON's `CMHI` or AVX2's `VPCMPGT`). These create a mask with all bits set or clear in the corresponding lane, while Algorithm 7 as written calls for setting and clearing individual bits, a choice made purely for ease of exposition. Actual implementations are advised to tweak the representation to employ groups of bits instead, so as to achieve an efficient implementation of the inner loop of line 6. For instance, `VPMOVMSKB` is a natural choice in AVX2, resulting in 2-bit mask groups for 16-bit lanes. In NEON, we extract 8-bit masks with `UZP1`, and reduce them to 4-bit masks using `SHRN` by 4. NEON's 128-bit registers suggest a choice of $W = 8$ if $L = 16$. However, we achieved better performance by taking $W = 16$, implemented as an unrolled 2-iteration loop processing 8-element vectors. We attribute this to the fact that converting a mask with `UZP1` and `SHRN` costs the same for 8 or 16 values.

Constant-time implementation of Algorithm 6. We now discuss how to implement Algorithm 6 in constant-time. First, we rewrite it using the C language's ternary operator, as shown in Algorithm 8, and then discuss strategies to implement this operator in constant time, firstly as an architecture-agnostic solution, and then consider conditional instructions present in the ARMv8-A, ARMv7-M and Intel architectures. Note that this version replaces -1 coefficients by 2; this is not an issue as the sampled polynomial has coefficients in $\mathbb{Z}/3\mathbb{Z}$, and indeed, the reference NTRU code employs the same representation.

Expressions of the form $(x < y)\ ?-1 : 0$, in lines 4 and 5 of Algorithm 8, can be made constant-time by noticing that, in two's complement integer arithmetic (used in nearly all

Algorithm 7 SIMD-RejSamplingMod(n, **rnd**): SIMD version of Algorithm 5.

```
Input: $n$
Input: rnd (array of random $L$-bit integers; refer to previous discussion about its length)
Output: si (output array of $(W+1)\lfloor(n-1)/W\rfloor$ integer elements, of which only the first
    $n-1$ entries are valid.)
 1: for $i = 0$ to $n-2$ do                                      ▷ Precomputation
 2:     $\mathbf{t}[i] = 2^L \bmod (n-1-i)$
 3: $j \leftarrow n-1$
 4: for $i = 0, W, 2W, \ldots, W\lfloor(n-1)/W\rfloor$ do
 5:     $mask \leftarrow 0$
 6:     for $k = 0$ to $W-1$ do                  ▷ Loop body implemented using SIMD code
 7:         $\mathbf{m}[k] \leftarrow \mathbf{rnd}[i+k] \times (n-1-(i+k))$
 8:         $\mathbf{l}[k] \leftarrow \mathbf{m}[k] \bmod 2^L$
 9:         $\mathbf{si}[i+k] \leftarrow \lfloor m/2^L \rfloor$
10:         if $\mathbf{l}[k] < \mathbf{t}[i+k]$ then
11:             $mask_k \leftarrow 1$                      ▷ $mask_k$ denotes the $k$-th bit of $mask$
12:         else
13:             $mask_k \leftarrow 0$
14:     while $mask \neq 0$ do                   ▷ Loop body implemented using scalar code
15:         k = CountTrailingZeros($mask$)
16:         repeat
17:             $m' \leftarrow \mathbf{rnd}[j] \times (n-1-(i+k))$
18:             $j \leftarrow j+1$
19:             $l' \leftarrow m \bmod 2^L$
20:         until $l' \geq \mathbf{t}[i+k]$
21:         $\mathbf{si}[i+k] \leftarrow \lfloor m'/2^L \rfloor$
22:         $mask_k \leftarrow 0$
23: return si
```

modern architectures), -1 and 0 have all bits set and cleared, respectively. The sign (most significant) bit of $x - y$ is 1 if $x < y$ and 0 otherwise; an arithmetic right shift by $w - 1$ bits, where w is the word size, replicates the sign bit across the entire word. Concretely, the following C code implements line 4 for 16-bit signed integer variables:

```
t0 = (si[i] - c0) >> 15;
```

While already efficient, better performance is achievable. To that end, we analyze the critical path of the main loop of Algorithm 8, shown in Figure 1. We disregard memory loads and stores, which can be removed from the critical path by proper scheduling. For any mobile-, desktop- or server-class modern CPU, one can assume at least a 2-way superscalar pipeline and single-cycle latency for all used operations, in which case the critical path from lines 4 and 5 of one iteration to the next (the bold arrows in the figure) takes 3 cycles.

In ARMv8-A, arithmetic operations can be encoded so that one of the input operands is shifted; thus, a single instruction can compute both $t_0 = t_0 \gg (w-1)$ and $c_0 = c_0 + t_0$. Unfortunately, ARMv8-A CPUs considered in this work, such as the Apple M1 [Joh22] and Cortex-A72 [ARM15], execute these instructions with a 2-cycle latency, offering no gain in performance (but a slight reduction in code size).

By employing ARMv8-A conditional instructions such as `CINC` and `CSET`, it is possible to reduce the critical path to 2 cycles. However, Algorithm 8 calls for decrementing c_0 and c_{01}, and there is no `CDEC` instruction in ARMv8-A; we modify the algorithm to use negative values for c_0, c_{01} and $\mathbf{si}[i]$, so that we can increment c_0 and c_{01} using `CINC` instead. Thus, we arrive at the following code for the algorithm's main loop:

Algorithm 8 CT-Shuffle($n, c_0, c_1, \mathbf{rnd}$): Fixed-weight sampling by shuffling, implemented in constant-time

```
Input: $n$
Input: $c_0, c_1$ (prescribed number of coefficients equal to 0, resp. 1)
Input: rnd (array of random $L$-bit integers; refer to previous discussion about its length)
Output: v (output array of $n - 1$ integers)
Notes: We employ the C language ternary operator ? to denote constant-time selection
    between two values based on a condition. See text for implementation possibilities.
1: si ← SIMD-RejSamplingMod($n$, rnd)
2: $c_{01} \leftarrow c_0 + c_1$                ▷ Note this invariant is maintained in the loop body
3: for $i = 0$ to $n - 2$ do
4:     $t_0 \leftarrow (\mathbf{si}[i] < c_0)\ ? -1 : 0$
5:     $t_1 \leftarrow (\mathbf{si}[i] < c_{01})\ ? -1 : 0$
6:     $c_0 \leftarrow c_0 + t_0$
7:     $c_{01} \leftarrow c_{01} + t_1$
8:     $\mathbf{v}[i] \leftarrow 2 + t_0 + t_1$
9: return v
```

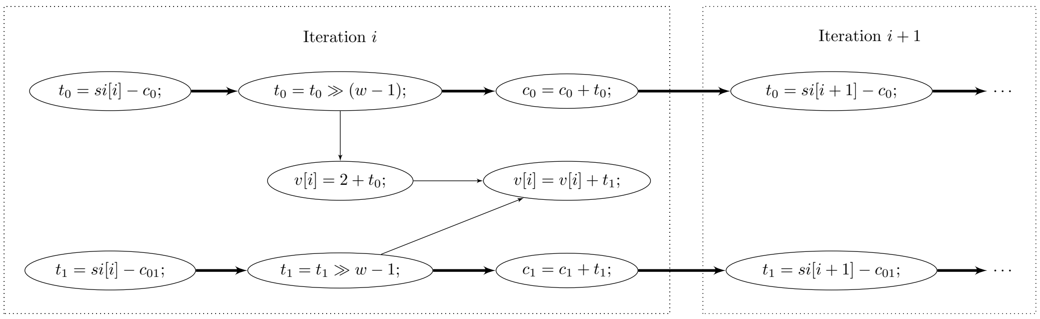

Figure 1: Critical path of the main loop of Algorithm 8.

```
cmp       c0, si[i]
cinc      c0,    c0, lt
cset    v[i],    ge
cmp      c01, si[i]
cinc     c01,   c01, lt
cinc    v[i],  v[i], ge
```

There are two critical paths: one from `cmp c0, r` to `cinc c0, c0, lt` to the next iteration's `cmp c0, r`; and the second for the same instructions involving `c01`. In all the considered ARM CPUs, all instructions in the code fragment above have single-cycle latency, and thus the loop has the potential to execute in 2 cycles/iteration.

Unfortunately, we run into throughput issues: in the Apple M1, reverse engineering efforts [Joh22] indicate that, although it is capable of executing 6 scalar instructions/cycle, only 3 execution units can execute flag-setting and conditional instructions, i.e. all instructions in the above code fragment. While theoretically sufficient to run the code at maximum throughput, we have observed instruction scheduling issues while attempting to software-pipeline Algorithms 7 and 8, preventing execution at maximum throughput. The following instruction sequence requires more μops, but performs better in the M1:

```
subs     tmp,   c0, si[i]
```

```
cinc      c0,   c0,    lt
add     v[i],  two,   tmp, asr #31
subs     tmp,  c01, si[i]
cinc     c01,  c01,    lt
add     v[i], v[i],   tmp, asr #31
```

We use 32-bit registers (w0, w1, etc.) and initialize two with the constant 2. It is also advantageous for the Cortex-A72, since the add instruction with shifted argument executes in the M pipeline, whereas all other instructions execute in the I0/I1 pipelines. While other bottlenecks come into play in the Cortex-A72, notably its 3-wide instruction decoder, this alternative instruction sequence performs better than the original.

Intel has conditional instructions for conditional moves (CMOVcc) and sets (SETcc), where cc are condition codes, but no conditional increments or decrements. For positive values of c_0 and c_{01}, as in the original version of Algorithm 8, an alternative is to decrement c0 and c01 and use CMOV to select between original and decremented values; decrements can execute in parallel with comparisons, thus the critical path is not lengthened.

Unfortunately, Intel instructions do not offer the three-operand form of ARMv8-A and other RISC architectures, so an extra MOV is required to create a copy prior to decrementing in order to avoid overwriting the original values; this doesn't necessarily increase the critical path, due to MOV elimination [Fog22], but it does increase front-end pressure. Implementers are advised to keep in mind the achievable performance given the critical path, to benchmark and analyze compiler-generated code if employing a high-level language, and to consider inline assembly (or a full assembly language implementation) to emit instructions that are well-matched to the decoder restrictions.

For the ARMv7-M architecture, a straightforward implementation of Algorithm 8, implementing lines 4 and 5 using the arithmetic right shift trick, works really well; this is aided by the ability to shift one of the input operands to data processing (logical and arithmetic) instructions, resulting in very compact and efficient code.

We have experimented with ARMv7-M's IT instruction, attempting to improve performance compared to the straightforward implementation, but we were unsuccessful. However, we did find an especially compact instruction sequence, devoid of conditional execution instructions, to implement the main loop of Algorithm 8:

```
cmp     si[i],  c0
sbc        c0,  #0
sbc      v[i], one
cmp     si[i], c01
sbc       c01,  #0
sbc      v[i], #-1
```

We set one to the constant 1. As the straightforward implementation is already efficient, this alternative saves one clock cycle per loop iteration, i.e. < 1000 cycles for the full algorithm. As fixed-weight sampling is performed only once during key generation and encapsulation, the speedup is just $< 0.02\%$ for the former and $\approx 0.15\%$ for the latter.

Software pipelining of Algorithms 7 and 8. Modern superscalar CPUs use distinct execution units for scalar and SIMD instructions. Most of the execution time of Algorithm 7 is spent in SIMD code, while Algorithm 8 is strictly scalar. This is amenable to software pipelining [Lam88]. In the best-case scenario, one can achieve execution time close to the maximum, rather than the sum, of the execution times of Algorithms 7 and 8.

Concretely, we inline Algorithm 7 into Algorithm 8, strip-mine the main loop of the latter, and then fuse the outer loops of both algorithms, processing W entries at a time.

With this approach, we were able to achieve, in the Apple M1, execution times only $\approx 12\%$ slower than the lower bound (2 cycles/iteration) for the main loop of Algorithm 8 alone. This includes all overhead such as function calls and returns, prologue and epilogue, initialization, and of course, the execution of Algorithm 7 itself, as seen in Table 4. The narrow (3-wide) decoder of the Cortex-A72 precludes achieving a similar result as the M1, but by interleaving instructions of both algorithms to improve scheduling, we achieved results not far from the limit dictated by the decoder bandwidth bottleneck.

Known Answer Tests. We note that the Known Answer Tests (KATs) in NTRU's specification [CDH+20] are tightly coupled to the fixed-weight sampling by sorting approach mandated there. Therefore, an implementation employing Algorithm 6 will fail these KATs for key generation and encapsulation. However, our sampled polynomials meet the fixed-weight requirement imposed by NTRU and are in principle indistinguishable from those generated by the existing approach. Thus, keys generated using our algorithm are valid, and the result of an encapsulation employing our algorithm will produce a correct decapsulation even by an unmodified implementation of the current NTRU proposal.

Given the simplicity and improved performance and code size characteristics of Algorithm 6, we suggest that future standardization attempts of NTRU specify our approach instead of sampling by sorting, and generate KATs accordingly. Implementers attempting to replicate our results, whether on ARMv8-A or other architectures, can use unofficial KATs generated by us, included in our source code package.

5 Experimental results

We now present experimental results for implementations of our proposed approach for various 64-bit ARMv8-A cores, as well as the 32-bit ARMv7-M Cortex-M4 core.

5.1 Methodology

We implemented reference versions of Algorithms 5 and 6, and optimized versions for ARMv7-M and ARMv8-A by replacing Algorithm 6 with Algorithm 8; for ARMv8-A specifically, we replaced Algorithm 5 by a NEON version of Algorithm 7. We integrated the reference and optimized implementations with existing state-of-the-art implementations of NTRU: `pqm4` [KPR+][4] for ARMv7-M and [GFBL24, NG21, CCHY23] for ARMv8-A. KATs were generated using the reference implementation and compared against the optimized implementations; we added tests to ensure interoperability between a conventional implementation (using sampling by sorting) and our proposed approach.

Testbeds and measurement methods. Our testbeds for performance measurement, with their corresponding CPU cores, are:

- Apple M1 P-core at 3200 MHz in an Apple MacBook Air laptop running macOS;
- Apple M3 P-core at 4064 MHz in an Apple MacBook Pro laptop running macOS;
- Cortex-A72 at 1500 MHz in a Raspberry Pi 4 single-board computer running Linux;
- Cortex-A57 at 1430 MHz in an Nvidia Jetson Nano single-board computer running Linux;
- Cortex-A53 at 1400 MHz in a Raspberry Pi 3 single-board computer running Linux;

[4]Although NTRU was removed from the most recent version of `pqm4`, after Kyber was selected in the NIST post-quantum standardization process, we used the most recent version prior to NTRU's removal.

- Cortex-M4 at 24 MHz in an STM32F4DISCOVERY development board.

Save for the ARMv7-M Cortex-M4 core, the remaining testbeds are ARMv8-A, running in 64-bit mode. Of these, the Apple M1, M3 and Cortex-A57 cores feature ARMv8-A Cryptographic Extensions, but the Cortex-A72 and the Cortex-A53 do not.

Our ARMv8-A performance measurements use the cycle counting routines originally introduced in [NG21]. Each routine is executed for 1,024 times and the average cycle count is reported. ARMv7-M measurements employ the `pqm4` [KPR+] benchmarking harness, which counts cycles using the Cortex-M4 SysTick timer. The number of iterations is set to 10, and the mean of results are reported; although this is a small number, the Cortex-M4 core is much simpler and more deterministic than the large out-of-order ARMv8-A cores, thus exhibiting little run-to-run variability.

While, to a first approximation, cycle counts are not influenced by CPU clock speed, there may be second-order effects such as the decoupling of CPU and bus/RAM/cache clocks. Thus, we take precautions to maximize the likelihood that benchmarks are performed at the nominal clock speeds quoted above: we use the `performance` scaling governor for Linux systems; in Apple systems, as far as we aware, there is no control over clock speeds, and there is no TurboBoost-like feature. In both cases, we try to avoid thermal throttling by inserting delays between benchmark runs to allow systems to cool down. The Cortex-M4 core does not automatically boost/throttle clock speeds; `pqm4` configures it to 24 MHz at startup, ensuring all benchmarks run at that fixed clock speed.

ARMv8-A binaries were compiled with Apple clang 15 (Apple M1 and M3), clang 17 (Cortex-A72 and Cortex-A53), and clang 10 (Cortex-A57), with `-O3` and core-specific `-mcpu` optimization flags. ARMv7-M binaries were compiled with gcc 12.2.1, passing the `-o speed` flag to the `pqm4` benchmark script. We enable the FEAT_DIT bit on ARMv8-A cores where it is available (in the case of our testbeds, only the Apple M1 and M3).

ARMv8-A implementation. Our implementation is based on the source code provided by [GFBL24], which contains their AMX implementation and the NEON implementations of [CCHY23, NG21]. As [CCHY23] is the state-of-the-art in NEON implementations, but targets only the HPS2048677 and HRSS701 parameter sets, [NG21] is included to display HPS2048509 and HPS4096821 results. Importantly, [GFBL24] backports optimized auxiliary routines of [CCHY23] to [NG21] (in particular a NEON implementation of constant-time sorting) and provides an optimized implementation of NIST's `randombytes()` AES-CTR-DRBG pseudo-random number generator (PRNG), using ARMv8-A Cryptographic Extensions. These routines are critical to the performance of fixed-weight sampling.

For CPUs without ARMv8-A Cryptographic Extensions, the ChaCha20 PRNG of [CCHY23] is used. As KATs are incompatible across different PRNGs, we supply two sets of KATs for validation, using ChaCha20 and AES-CTR-DRBG generators. We ensure that latter matches those provided in the NTRU specification, which uses the same PRNG.

ARMv7-M implementation. `pqm4` [KPR+] is the gold standard for Cortex-M4 implementations of PQC schemes. While its NTRU implementation has highly optimized polynomial multiplication and inversion routines, the constant-time sorting routine in use is the `portable3` variant of djbsort [Ber19], using a non-architecture-specific implementation of the core minimum/maximum operation of the sorting network. Compiler output inspection reveals that the compiler the minimum/maximum idiom was not recognized, thus generating suboptimal code without using e.g. conditional instructions. We performed some optimization work on this routine, so as to avoid casting our proposed approach in an excessively favorable light. We switched to the more efficient `portable4` variant of djbsort, wrote inline assembly versions of the core minimum/maximum operation using conditional operations and a reduced number of memory accesses, and replaced all `long long` (64-bit) variables by 32-bit `long` variables to avoid unnecessary use of multi-precision arithmetic, given that ARMv7-M is a 32-bit architecture. This range reduction does not present an issue in NTRU due to the small lengths (hundreds of elements) of the arrays to be sorted. While it is certainly possible to further optimize this routine, further experiments by us resulted in code size increases, which are undesirable in deeply embedded environments.

Table 1 compares the performance, code size and stack memory usage of encapsulation in the existing NTRU scheme (using sampling by sorting), for the original `pqm4` implementation and our optimized version in our STM32F4DISCOVERY testbed; we denote these as "[KPR+] original" and "[KPR+] optimized", respectively, in Table 1. It is seen that our optimizations result in large speedups (44–48%) with negligible effect on code size and none at all on stack usage. While we omit corresponding figures for key generation, our optimizations also outperformed stock `pqm4`, although by smaller amounts (5.4–6.0%); code size and stack usage differences are similar. Results for decapsulation and for the HRSS701 parameter set are not shown, as they do not call the constant-time sorting routine.

Table 1: Comparison of the original pqm4 [KPR+] NTRU implementation and our optimized version for encapsulation. Code size and stack usage are in bytes. For differences, positive values denote an increase in the optimized version relative to the original one.

Parameter set	Work	Cycle count	Code size	Stack usage
HPS2048509	[KPR+] original	557 131	192 560	14 084
	[KPR+] optimized	386 278	192 632	14 084
	Speedup	**44%**		
	Difference		**+0.04%**	**0%**
HPS2048677	[KPR+] original	801 357	282 292	19 996
	[KPR+] optimized	546 570	282 364	19 996
	Speedup	**47%**		
	Difference		**+0.03%**	**0%**
HPS4096821	[KPR+] original	997 084	370 808	23 436
	[KPR+] optimized	672 825	370 884	23 436
	Speedup	**48%**		
	Difference		**+0.02%**	**0%**

5.2 Performance figures and analysis

We present performance figures for NTRU KEM key generation and encapsulation in Tables 2 (for Apple SoCs) and 3 (for ARM Cortex cores); decapsulation does not employ fixed-weight sampling, thus its performance is unaffected by our proposed approach. We present NEON results from the implementations of [NG21] for the HPS2048509 and HPS4096821 parameter sets, and [CCHY23] for the HPS2048677 and HRSS701 parameter sets. AMX results are from the implementation of [GFBL24]. We emphasize that all ARMv8-A implementations use the NEON optimized constant-time sorting routine of [CCHY23]. For the Cortex-M4 core, we use the implementation of [KPR+], incorporating our optimizations for constant-time sorting. We present performance results as cycle counts, calculating speedups as $c_{\text{sorting}}/c_{\text{shuffling}} - 1$.

Results for the shuffling approach consist in replacing the `sample_fixed_type` routine by our proposed algorithms, and adjusting the amount of uniform random bits to match the requirements of the shuffling algorithms, as discussed in Section 4.

We also present performance figures for fixed-weight sampling, by measuring calls to the `sample_fixed_type` routine, whose results are presented in Table 4. Finally, we present code size (Flash) and stack (RAM) usage figures for the Cortex-M4 in Table 5.

Table 2: Cycle counts (in kilocycles) for NTRU KEM key generation (**KG**) and encapsulation (**Enc.**) in the Apple M1 and M3 SoCs.

Param. Set	**Sampling**	**Apple M1**				**Apple M3**			
		NEON		**AMX**		**NEON**		**AMX**	
		KG	**Enc.**	**KG**	**Enc.**	**KG**	**Enc.**	**KG**	**Enc.**
509	Sorting	218	16.1	170	12.5	214	15.5	164	11.7
	Shuffling	214	12.5	167	8.90	211	12.0	160	8.22
	Speedup	**1.7%**	**29%**	**2.1%**	**40%**	**1.7%**	**29%**	**2.2%**	**43%**
677	Sorting	307	20.6	283	17.1	296	19.4	266	16.0
	Shuffling	309	14.9	278	11.9	296	14.1	260	10.9
	Speedup	**-0.7%**	**39%**	**1.9%**	**44%**	**-0.2%**	**37%**	**2.1%**	**46%**
821	Sorting	498	28.0	384	19.4	491	27.2	371	18.1
	Shuffling	491	21.8	378	13.1	485	21.2	365	12.1
	Speedup	**1.3%**	**28%**	**1.6%**	**48%**	**1.2%**	**28%**	**1.8%**	**50%**
701	N/A	323	14.6	287	11.5	309	13.9	270	10.5
	Slowdown vs. 677 sorting	**5.4%**	**-29%**	**1.4%**	**-33%**	**4.4%**	**-28%**	**1.4%**	**-35%**
	Slowdown vs. 677 shuffling	**4.7%**	**-1.7%**	**3.3%**	**-3.9%**	**4.3%**	**-1.6%**	**3.5%**	**-4.2%**

Key generation and encapsulation. Our proposed approach achieved performance improvements across the board, for both key generation and encapsulation, save for a few outliers in the former. For Cortex-M4, these improvements come at a negligible cost to code size (Flash), and even a slight improvement in stack (RAM) usage, as seen in Table 5.

With regards to key generation, we see improvements of up to 2.7% for ARMv8-A cores and 6.1% for the Cortex-M4. We recall that NTRU key generation is computationally expensive; disregarding simpler operations, it requires a modulo-q inversion (usually realized by a modulo-2 inversion followed by 8 multiplications), a modulo-3 inversion, 5 extra multiplications, 2 different types of sampling (including `sample_fixed_type`) and pseudo-random number generation. Therefore, it is not surprising that optimizing a single sampling routine results in limited performance improvements.

Results are more significant for encapsulation, which are arguably of more interest than key generation, seeing as, for most cryptographic applications, the former will be run far more often than the latter. We see improvements of up to 44% and 50% for NEON and AMX implementations in ARMv8-A, and 71% for the Cortex-M4. Improvements correlate well with polynomial multiplication performance, which is fastest for NEON in the HPS2048677 parameter set (based on the faster TMVP approach of [CCHY23]) and in AMX implementations; this is expected due to Amdahl's law.

Fixed-weight sampling. Table 4 shows that our shuffling approach significantly improves performance of fixed-weight sampling compared to the sampling by sorting approach of previous works. We see very significant speedups for all platforms: up to 6.91 (591%) in ARMv8-A cores and 12.58 (1158%) in the Cortex-M4. Measurements do not include the cost of pseudo-random number generation (i.e. the `randombytes` routine), which is highly platform-dependent; recall that our approach requires slightly more than half as many pseudo-random bytes as sampling by sorting.

Comparison with NTRU-HRSS. It is instructive to compare NTRU-HPS2048677 to NTRU-HRSS701, as both are designed to the same NIST security level. Fortunately,

Table 3: Cycle counts (in kilocycles) for NTRU KEM key generation (**KG**) and encapsulation (**Enc.**) in ARM Cortex cores.

Param. Set	**Sampling**	**Cortex-A72**		**Cortex-A57**		**Cortex-A53**		**Cortex-M4**	
		KG	**Enc.**	**KG**	**Enc.**	**KG**	**Enc.**	**KG**	**Enc.**
509	Sorting	884	71.2	884	61.5	1243	111	2674	386
	Shuffling	860	54.1	892	51.8	1213	86.7	2520	234
	Speedup	**2.7%**	**32%**	**-0.8%**	**19%**	**2.4%**	**28%**	**6.1%**	**65%**
677	Sorting	1140	77.4	1146	65.1	1636	121	4317	547
	Shuffling	1123	55.3	1123	45.3	1600	84.9	4094	327
	Speedup	**1.5%**	**40%**	**2.1%**	**44%**	**2.3%**	**43%**	**5.4%**	**67%**
821	Sorting	2156	135	2131	121	2993	197	5705	673
	Shuffling	2111	109	2102	95.2	2951	156	5419	395
	Speedup	**2.1%**	**24%**	**1.4%**	**27%**	**1.4%**	**26%**	**5.3%**	**71%**
701	N/A	1200	58.0	1200	53.6	1579	77.7	4188	368
	Slowdown vs. 677 sorting	**5.2%**	**-25%**	**4.7%**	**-18%**	**-3.5%**	**-36%**	**-3.0%**	**-33%**
	Slowdown vs. 677 shuffling	**6.8%**	**4.8%**	**6.9%**	**18%**	**-1.3%**	**-8.5%**	**2.3%**	**13%**

Table 4: Cycle counts (in kilocycles) for fixed-weight sampling, excluding the cost of uniform random number generation.

Param. Set	**Sampling**	**Apple M1**	**Apple M3**	**Cortex-A72**	**Cortex-A57**	**Cortex-A53**	**Cortex-M4**
509	Sorting	4.49	4.36	13.0	14.0	23.2	152
	Shuffling	1.11	1.06	2.28	2.30	4.97	13.3
	Speedup	**4.04×**	**4.12×**	**5.71×**	**6.11×**	**4.67×**	**11.45×**
677	Sorting	6.43	6.25	19.5	21.2	35.1	221
	Shuffling	1.49	1.41	3.11	3.07	6.79	18.4
	Speedup	**4.32×**	**4.42×**	**6.25×**	**6.91×**	**5.18×**	**12.05×**
821	Sorting	7.65	7.48	22.6	24.9	40.9	280
	Shuffling	1.79	1.71	3.75	3.77	8.12	22.3
	Speedup	**4.26×**	**4.37×**	**6.03×**	**6.61×**	**5.03×**	**12.58×**

the state-of-the-art NEON implementation of [CCHY23] implements both parameter sets, allowing for a fair comparison. Tables 2 and 3 include rows marked "Slowdown vs. 677 sorting" and "Slowdown vs. 677 shuffling", computed as $c_{701}/c_{677} - 1$; thus, positive values indicate that HRSS701 is slower than HPS2048677, and the contrary for negative values.

Even with the sampling by sorting approach, HPS2048677 is usually faster than HRSS701 for key generation, with the exception of the Cortex-A53 and Cortex-M4 cores; with the shuffling approach, HPS2048677 key generation always outperforms HRSS701. As for encapsulation, HPS2048677 using sampling by sorting was significantly slower in all cases, by up to 33% in Apple SoCs and the Cortex-M4, and 36% in ARMv8-A Cortex cores. The shuffling approach closes this gap, with HPS2048677 slower by at most 4.2% in Apple SoCs, and 8.5% in the Cortex-A53; for other ARMv8-A cores, HPS2048677 is actually faster, by up to 18%, and in the Cortex-M4, it is also faster by 13%.

Table 5: Code size (Flash) and stack (RAM) usage, in bytes, for ARMv7-M binaries. Statically allocated data (`.data` and `.bss` sections) were reported as zero in all cases. "**Diff.**" refers to the percentual difference between implementations; positive values denote an increase in our version relative to [KPR+].

Param. set	**Work**	**Code size**	**Stack usage** Key gen.	Encaps.
HPS 2048509	[KPR+]	192 632	21 376	14 084
	Ours	193 092	20 544	13 244
	Diff.	**+0.2%**	**−3.9%**	**−6.0%**
HPS 2048677	[KPR+]	282 364	28 480	19 996
	Ours	283 164	27 352	18 868
	Diff.	**+0.3%**	**−4.0%**	**−5.6%**
HPS 4096821	[KPR+]	370 884	35 216	23 436
	Ours	371 964	33 856	22 076
	Diff.	**+0.3%**	**−3.9%**	**−5.8%**
HRSS701	[KPR+]	265 264	27 528	18 332

6 Conclusion

In this work, we showed that timing attack-resistant fixed-weight sampling can be performed without using constant-time sorting. We have proposed a new algorithm (Algorithm 8) which achieves a running time of $O(n)$, an improvement over $O(n \log^2(n))$ for previous, sorting network-based approaches. This results in performance improvements in actual implementations across a range of different platforms, from deeply embedded to high-performance laptop CPUs. Additionally, the amount of random data needed for sampling is reduced by almost half, which is advantageous for architectures without instructions to accelerate cryptographically secure PRNGs. Moreover, our proposed method may be simpler to implement in an optimized fashion than constant-time sorting networks.

This solves a long-standing open problem: to date, the best alternative was the NTRU-HRSS variant, which also seeks to eliminate the cost of constant-time sorting required for sampling fixed-weight polynomials. As discussed in §5, a modified NTRU-HPS2048677, using our proposed approach, closes the performance gap to NTRU-HRSS701 (recalling that both are designed to the same NIST security level). We also note that key and ciphertext sizes for NTRU-HPS2048677 are smaller: 930 (resp. 1138) bytes for the public key and ciphertext, and 1234 (resp. 1450) bytes for the private key, for NTRU-HPS2048677 (resp. NTRU-HRSS701). Finally, the need to support both NTRU-HPS and NTRU-HRSS to achieve different security levels results in increased implementation complexity, e.g. due to the HRSS-specific version of `Lift` [CDH+20, §1.9.3] and the additional `Ternary_Plus` sampling routine [CDH+20, §1.10.4]. In light of these arguments, we call into question the need for a separate NTRU-HRSS parameter set.

Future work Although NTRU is no longer being considered by NIST, we recall that it has been standardized in other forums [IEE09, ASC10]. Since our proposed Algorithm 8 improves upon the existing fixed-weight sampling by sorting approach mandated by the NTRU specification submitted to NIST [CDH+20], we suggest amending NTRU specifications to use Algorithm 8, and incorporating it into any future standardization efforts (for instance, we note that FrodoKEM [BCD+16] is also no longer under consideration by NIST, but is being considered for standardization by ISO [Int23c]). We also suggest developing implementations for other widely-used architectures, in particular, Intel (using

AVX2 and AVX-512 SIMD extensions) and the recently released ARMv8.1-M Helium SIMD instruction set for deeply embedded systems [Dir19].

Algorithm 8, as stated, is not amenable to vectorization, due to a loop-carried dependency between iterations of its main loop. Using a similar idea as the initial step of MergeShuffle (Algorithm 4), vectorization becomes possible; we developed a prototype implementation that confirms its potential for large speedups, especially on wide CPUs such as the M1 and M3. However, without applying the remaining steps of MergeShuffle, the resulting permutation is biased, which may create an avenue of attack. An alternative we envisioned involves sampling from the hypergeometric distribution; however, this is an uncommon distribution in cryptography, and we were unable to find any efficient, constant-time algorithms. We invite future work into either modifying MergeShuffle to be constant-time, or to propose efficient, constant-time hypergeometric sampling algorithms.

References

[Ajt96] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237838.

[ARM] ARM Limited. How is instruction timing affected by the FEAT_DIT architectural feature? URL: https://developer.arm.com/documentation/ddi0487/ja/.

[ARM15] ARM Limited. Cortex®-A72 software optimization guide, 2015. URL: https://developer.arm.com/documentation/uan0016/a/.

[ARM23] ARM Limited. Arm® architecture reference manual for a-profile architecture, 2023. URL: https://developer.arm.com/documentation/ka005181/1-0/.

[ASC10] ASC X9.98. Lattice-based polynomial public key establishment algorithm for the financial services industry. *ASC/X9 - ANSI X9.98*, 2010.

[BBHL18] Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie O. Lumbroso. MergeShuffle: a very fast, parallel random permutation algorithm. In Luca Ferrari and Malvina Vamvakari, editors, *Proceedings of the 11th International Conference on Random and Exhaustive Generation of Combinatorial Structures, GASCom 2018, Athens, Greece, June 18-20, 2018*, volume 2113 of *CEUR Workshop Proceedings*, pages 43–52, Aachen, Germany, 2018. CEUR-WS.org. URL: http://ceur-ws.org/Vol-2113/paper3.pdf.

[BBHT17] Axel Bacher, Olivier Bodini, Hsien-Kuei Hwang, and Tsung-Hsi Tsai. Generating random permutations by coin tossing: Classical algorithms, new analysis, and modern implementation. *ACM Trans. Algorithms*, 13(2), feb 2017. doi:10.1145/3009909.

[BCD$^+$16] Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2016. URL: https://doi.org/10.1145/2976749.2978425.

[BCLvV18] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: Reducing attack surface at low cost. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, pages 235–260, Cham, 2018. Springer International Publishing.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 250–272, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/papers.html#cachetiming, 2004.

[Ber19] Daniel J. Bernstein. djbsort. https://sorting.cr.yp.to, 2019.

[CCHY23] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic views of vectorized polynomial multipliers – NTRU. Cryptology ePrint Archive, Report 2023/1637, 2023. https://ia.cr/2023/1637. To appear at INDOCRYPT 2023.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M.Schanck, Tsunekazu Saito, Peter Schwabe, William Whyte, Keita Xagawa, Takashi Yamakawa, and Zhenfei Zhang. NTRU algorithm specifications and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. https://ntru.org/resources.shtml.

[CWS+24] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *USENIX Security*, 2024. URL: https://gofetch.fail/files/gofetch.pdf.

[Dan19] Oussama Danba. Optimizing NTRU using AVX2. Master's thesis, Radboud University, 2019.

[dG15] Wouter de Groot. A performance study of X25519 on Cortex-M3 and M4. Master's thesis, Eindhoven University of Technology, 2015.

[Dir19] Rhonda Dirvin. Next-generation Armv8.1-M architecture: Delivering enhanced machine learning and signal processing for the smallest embedded devices. https://www.arm.com/company/news/2019/02/next-generation-armv8-1-m-architecture, 2019.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, jul 1964. doi:10.1145/364520.364540.

[Fog22] Agner Fog. The microarchitecture of Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf, 2022.

[FY38] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver & Boyd, Oxford, England, 3rd edition, 1938.

[GFBL24] Décio Luiz Gazzoni Filho, Guilherme Brandão, and Julio López. Fast polynomial multiplication using matrix multiplication accelerators with applications to NTRU on Apple M1/M3 SoCs. Cryptology ePrint Archive, Paper 2024/002, 2024. URL: https://eprint.iacr.org/2024/002.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 223–263, June 2022. URL: http://dx.doi.org/10.46586/tches.v2022.i3.223-263, doi:10.46586/tches.v2022.i3.223-263.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery. URL: https://doi.org/10.1145/237814.237866.

[HPS] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: a new high speed public key cryptosystem. CRYPTO '96 rump session. https://web.securityinnovation.com/hubfs/files/ntru-orig.pdf.

[HRSS17] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 232–252, Cham, 2017. Springer-Verlag Berlin Heidelberg.

[IEE09] IEEE standard specification for public key cryptographic techniques based on hard problems over lattices. *IEEE Std 1363.1-2008*, pages 1–81, 2009. doi:10.1109/IEEESTD.2009.4800404.

[Int23a] Intel Corporation. Data operand independent timing instruction set architecture (ISA) guidance, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html.

[Int23b] Intel Corporation. Data operand independent timing instructions, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[Int23c] International Organization for Standardization. FrodoKEM: Learning with errors key encapsulation preliminary draft standard, 2023. URL: https://frodokem.org/files/FrodoKEM-ISO-20230314.pdf.

[Joh22] Dougall Johnson. Apple M1 microarchitecture research. https://dougallj.github.io/applecpu/firestorm.html, 2022.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1998.

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

[Lam88] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Not.*, 23(7):318–328, jun 1988. doi:10.1145/960116.54022.

[Lem19] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1), jan 2019. doi:10.1145/3230636.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, 2002.

[Nat17] National Institute of Standards and Technology. Post-Quantum Cryptography, 2017. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[NG21] Duc Tri Nguyen and Kris Gaj. Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 234–254, Cham, 2021. Springer International Publishing.

[Por18] Thomas Pornin. Constant-time multiplication. https://www.bearssl.org/ctmul.html, 2018.

[Rao61] C. Radhakrishna Rao. Generation of random permutations of given number of elements using random sampling numbers. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 23(3):305–307, 1961. URL: http://www.jstor.org/stable/25049166.

[San62] Martin Sandelius. A simple randomization procedure. *Journal of the Royal Statistical Society. Series B (Methodological)*, 24(2):472–481, 1962. URL: http://www.jstor.org/stable/2984238.

[Sen21] Nicolas Sendrier. Secure sampling of constant-weight words – application to BIKE. Cryptology ePrint Archive, Report 2021/1631, 2021. https://ia.cr/2021/1631.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. doi:10.1137/S0097539795293172.