Generating a random string with a fixed weight

Nir Drucker and Shay Gueron

University of Haifa, Israel,
and
Amazon, Seattle, USA

Abstract. Generating, uniformly at random, a binary or a ternary string with a fixed length L and a prescribed weight W, is a step in several quantum safe cryptosystems (e. g., BIKE, NTRUEncrypt, NTRU LPrime, Lizard, McEliece).
This *fixed weight vector selection* generation is often implemented via a shuffling method or a rejection method, but not always in "constant time" side channel protected flow. A recently suggested constant time algorithm for this problem, uses Network Sorting and turns out to be quite efficient. This paper proposes a new method for this computation, with a side channel protected implementation. We compare it to the other methods for different combinations of L and W values. Our method turns out to be the fastest approach for the cases where L is (relatively) short and $0.1 < W/L \leq 0.5$. For example, this range falls within the parameters of NTRU LPrime, where our method achieves a $3\times$ speedup in the string generation. This leads to an overall $1.14\times$ speedup for the NTRU LPrime key generation.

Keywords: Software optimization · Combinatorics · Post Quantum Cryptography · Coding

1 Introduction

This paper deals with efficient methods for what we call the Fixed Weight Vector Selection (FWVS) problem, defined as follows.

Definition 1. *Let L, W, q, m be integers such that $0 < W \leq L$ and let $q \geq 2$ be a prime. Denote the set of all vectors with L symbols from $\mathbb{F}_{q^m}$ having weight W by S_L^W (the weight of a vector is defined to be the number of its nonzero symbols).*

Problem 1 (Fixed Weight Vector Selection (FWVS)). Select, uniformly at random, a vector from S_L^W (here, uniformly at random means that every vector from S_L^W has equal probability to be selected in the process).

To illustrate, we give a simple example.

Example 1. $L = 5, W = 4, q = 2, m = 1$. Here ($q = 2$), the elements of the vectors are bits. With these parameters, $S_L^W = \{11110, 11101, 11011, 10111, 01111\}$. The FWVS problem is to select X from S_L^W such that $Pr(X = s) = \frac{1}{5}$ for every $s \in S_L^W$.

Problem 1 is one of the steps in several cryptosystems that have been proposed to the first round [2] of NIST Post-Quantum Cryptography (PQC) project. Five such schemes (BIKE, HQC, McEliece, NTRUEncrypt, and NTRU LPrime) were recently selected for the second round of the project [3]. A selection of the relevant parameters that shows the different range of interest for L and W, is shown in Table 1. Note that these algorithms use symbols from $\mathbb{F}_2$ (binary strings) or $\mathbb{F}_3$ (ternary strings). These correspond to $m = 1$ and $q = 2$ or $q = 3$.

Table 1. Some of the proposals submitted to the first round of NIST PQC project [2] for which the FWVS Problem is a step in their key generation/encapsulation/decapsulation. The schemes BIKE, HQC, McEliece, NTRUEncrypt, and NTRU LPrime were recently selected for the second round of the project [3]. The cryptosystems are sorted in alphabetic order. Note the different ranges of L and W values.

Cryptosystem	NIST Security Category	L	W
BIG QUAKE	1	3,600	78
BIG QUAKE	5	9,000	135
BIKE1 (keygen)	1	10,163	71
BIKE1 (encaps)	1	20,326	134
BIKE1 (keygen)	5	32,749	137
BIKE1 (encaps)	5	65,498	261
HQC	1	22,229	67
HQC	5	59,011	133
Lizard	1	536	140
Lizard	5	1024	200
McElice6960119	5	6960	119
McElice8192128	5	8192	128
NTRUEncrypt	1	443	143
NTRUEncrypt	5	743	247
NTRU LPrime	5	761	250

We point out that a construction for the case $q = 3$ can be implemented by means of post processing the results of a construction for the case $q = 2$ (see Section 5). Therefore, we hereafter deal only with $q = 2$. We seek fast implementations of the FWVS problem with the following constraint: the execution should be in "constant time"[1]. By constant time we mean that the algorithm: a) does not involve access to elements of the vectors in a way that depends on the values of the symbols; b) does not involve branches that depend on the values of the symbols.
Proper software implementations of such algorithms are considered to be se-

[1] "Constant time" is the standard term for algorithms/implementations that are secure against (some) side channel attacks. Other terms that are used are "side channel protected" and "Isochronous."

cure against the known micro-architectural side channel analyses (side channel attacks hereafter)[2]. To illustrate the term "constant time" we give two examples:

Example 2. Assume an implementation that sets one bit in an array of bits by accessing only the relevant location in the array. A spy process that monitors the CPU caches can infer the memory address that was accessed, and therefore can learn the bit location.

Example 3. Assume an implementation that sets a bit only if it equals zero by performing the "read" operation but performing a "write" operation only when needed. A spy process that can measure the implementation's execution time can distinguish between the two cases (read or read+write) and learn if the relevant bit was set before the operation. A constant time implementation should always write a value.

Remark 1. The *overall execution time* does not have to be fixed (constant) in order to satisfy these requirements, but since "constant time" is the common term used for describing such software implementations, we adopt it.

Remark 2. It is straightforward to write side channel protected software for an inherently constant time algorithm. For other algorithms, the software implementations need to use special techniques in order to become side channels safe.

Our contributions. we proposed a new method (RepeatedAND) to address the FWVS problem and compare it to the alternative methods for different L and W combinations that are relevant to the Post-Quantum (PQ) proposals mentioned above. Our method(s) speedup the vectors' generation significantly for some size combinations. They lead to a noticeable speedup in the overall performance of e. g., NTRU LPrime [6].

The paper is organized as follows. Section 2 starts with preliminaries and notation. Section 3 describes the "RepeatedAND" method. Conversions (from binary to ternary) are described in Section 5. Our results are reported in Section 6 and a conclusion is brought in Section 7.

2 Preliminaries, notation, and conventions

We use $\mathbb{F}_{q^m}$ to denote a finite field with q^m symbols $\{0, 1, \ldots, q^m\}$, where in this paper we set $m = 1$ and $q \in \{2, 3\}$. Polynomials $a(x) = a_{L-1}x^{L-1} + \ldots a_1 x + a_0 \in \mathbb{F}_2[x]$ are represented, interchangeably, as strings of L bits. If A is such a string, then $A[i] = a(i)$, $0 \leq i < L$. Polynomials $b(x) = b_{L-1}x^{L-1} + \ldots b_1 x + b_0 \in \mathbb{F}_3[x]$ are represented as strings of 2-bit symbols ($2 \cdot L$ bits). If B is such a string, then $B[2i + 1 : 2i] = b(i)$, $0 \leq i < L$.

[2] Differential power attacks and fault injection attacks are outside the scope of this paper.

Example 4. The polynomial $x^9+x^6+x^2+x+1 \in \mathbb{F}_2[x]$ corresponds to the bits string 1001000111 of length $L = 10$. The polynomial $2x^9+x^6+x^2+x+2 \in \mathbb{F}_3[x]$ corresponds to the bits string 10000001000000010110 of length $L = 2 \cdot 10$.

Let A be an array of bits. The weight of A is denoted by $wt(A)$ and is the number of set bits in A. We denote by 1^ℓ (resp. 0^ℓ) the ℓ-bit string of all ones (resp. zeros). Bit-wise "and", "or" and "not" are denoted by $\wedge$ and $\vee$ and $\neg$, respectively. In the algorithms below, unless otherwise stated, uninitialized variables equal 0.

A Pseudorandom Function (PRF) for generating uniform random samples. We assume access to a PRF that consists of two functions. It is initialized by calling $st = \mathsf{InitPRF}(seed)$, where *seed* is some input seed (256-bit in our case) and the the output is a PRF state st. Subsequently, the function $r = \mathsf{GetRand}(st, \ell)$ is called in order to obtain a pseudorandom integer r in the range $0 \leq r < \ell$.

Remark 3. In general generating a long sequence of true random bits is slow (e. g., using `RDRAND` and `RDSEED` on x86 platforms), therefore we use some PRF to efficiently collect pseudorandom numbers.

2.1 The shuffling method

The classical approach to the FWVS problem is the Fisher-Yates Shuffle algorithm [11] (a.k.a known as Knuth shuffle). One of its variants [10] is described in Algorithm 1. It performs an in-place shuffle on a pre-initialized array. Steps 6-8 depend on the values in the array A. Thus, their (software) implementation is not inherently "constant time" unless the implementation takes the performance penalty for sweeping over the whole array in every iteration.

Algorithm 1 "inside-out", Fisher-Yates Shuffle algorithm [10, 11]

Input: *seed*, L, W
Output: A (an L bits string with weight W)
1: **procedure** GenString(*seed*, L, W)
2: $A = 0^{L-W}1^W$
3: st = InitPRF(seed)
4: **for** i = 0 to L **do**
5: $j = \mathsf{GetRand}(st, L)$
6: $t = A[j]$
7: $A[j] = A[i]$
8: $A[i] = t$
9: **return** A

2.2 Rejection method

Algorithm 2 is commonly known as the Rejection method (e. g., used in NTRUEncrypt [12, 13] and BIKE [4, 8]). Here, Step 6 is a branch that depends on the

value of A. Thus, its software implementation is not inherently "constant time" unless the implementation takes the performance penalty for sweeping over the whole array in every iteration.

Algorithm 2 Rejection method

Input: *seed*, L, W
Output: A (an L bits string with weight W)
procedure GenString(*seed*, L, W)
  $A = 0^L$
  st = InitPRF(seed)
  while $wt(A) \neq W$ **do**
    j = GetRand(st, L)
    if A[j]=0 **then**
      $A[j] = 1$
  return A

2.3 Sorting method

Algorithm 3 shows another approach to Problem 1 that we call here Sorting method. This method has been recently proposed for NTRU LPrime [6]. It is based on constant time sorting algorithms (details and review are provided in [5]), and enjoys an efficient software implementation (using AVX2) demonstrated in [5]. Note that the code in NTRU LPrime generates strings with $q = 3$, but as explained above, we describe Algorithm 3 for the case $q = 2$. This algorithm is inherently constant time.

Algorithm 3 Sorting method (based on [5])

Input: *seed*, L, W
Output: A (an L bits string with weight W)
Comment: B is a $32L$ bits string. Sort operates on a 32-bit integers array.
procedure GenString(*seed*, L, W)
  st = InitPRF(seed)
  $B[32L-1:0]$ = GetRand($st, 2^{32L}-1$)
  for i in 0 to $W-1$ **do**
    $B[32i] = 1$
  for i in W to $L-1$ **do**
    $B[32i] = 0$
  Sort(B, L)
  for i in 0 to $L-1$ **do**
    $A[i] = B[32i]$
  return A

3 RepeatedAND method

We propose an alternative method to tackle Problem 1, as illustrated in Algorithm 4. The main idea is that if X and Y are two (pseudo)random independent strings of length L then: a) the expected weight of X and of Y is $\mathbb{E}\left[wt(X)\right] = \mathbb{E}\left[wt(Y)\right] = \frac{L}{2}$; b) $\mathbb{E}\left[wt(X \wedge Y)\right] = \frac{L}{4}$.

Steps 10-14 in Algorithm 4 generate a string $\bar{A}$ of length L with $wt(\bar{A}) \leq W$. The algorithm takes a sequence A_i, $i = 1, 2, , \ldots, J$ of (pseudo)random independent strings of length L, where $J \approx \left\lfloor log_2(L/W) \right\rfloor$, and sets

$$\bar{A} = \bigwedge_{j=0}^{J} A_j \tag{1}$$

The value of J is not pre-determined, but its expected value is $\mathbb{E}\left[J\right] = \left\lfloor log_2(L/W) \right\rfloor$. To illustrate, we give an example.

Example 5. Let $L = 1,024$, $W = 128$. Then

$$\begin{aligned}
&\mathbb{E}\left[wt(A_0)\right] = 512 \\
&\mathbb{E}\left[wt(A_0 \wedge A_1)\right] = 256 \\
&\mathbb{E}\left[wt(A_0 \wedge A_1 \wedge A_2)\right] = 128
\end{aligned}$$

The expected number of iterations required for generating $\bar{A}$ as in (1) is $J = 3$. In practice, J is typically 3 or 4.

Steps 6-17 use $\bar{A}$ in order to generate a new vector A with $wt(A) = W$ (exactly). Starting from $A = \bar{A}$ (with $wt(\bar{A}) \leq W$), we set $w = W - wt(\bar{A}) \geq 0$. Next, another string $\bar{A}'$ of length L with $wt(\bar{A}') \leq w$ is generated (Steps 10-14). The value $A = A \vee \bar{A}'$ satisfies $wt(\bar{A}) \leq wt(A) \leq W$. Steps 6-17 are repeated while $w > 0$. The expected number of rounds is at most $\lceil log_2 W \rceil$.

Remark 4. Algorithm 4 requires independent (pseudo)random values A_j. Sampling A_j from $\mathsf{GetRand}(\cdot)$ is (performance wise) costly, so reusing sampled strings may seem like a tempting shortcut. However, this approach violates the independence property. For example, circular rotation of $\bar{A}$ on Step 12, generates correlation between the bits of A. In practice, some bit manipulation techniques could possibly lead to sufficiently uncorrelated vectors, and perhaps make the selection acceptable. We do not adopt this approach here.

Lemma 1. *Algorithm 4 generates every string in S_L^W with equal probability.*

Proof. Algorithm 4 outputs only strings from S_L^W. For every round of Algorithm 4, we have, Step 15,

$$\begin{aligned}
wt(A) + wt(\bar{A}) &< wt(A) + w \\
&= wt(A) + W - wt(A) = W
\end{aligned}$$

Algorithm 4 RepeatedAND method

Input: $seed$, L, W
Output: A (an L bits string with weight W)

1: **procedure** GenString($seed$, L, W)
2: $st = \mathsf{InitPRF}(seed)$
3: $ctr = 0$
4: $w = W$
5: $A[L-1:0] = 0^L$
6: **do**
7: $j = 0$
8: $A_j = \mathsf{GetRand}(st, L)$
9: $\bar{A} = A_j \wedge \neg A$ ▷ Optimization
10: **do**
11: $j = j + 1$
12: $A_j = \mathsf{GetRand}(st, L)$
13: $\bar{A} = \bar{A} \wedge A_j$
14: **while** $(wt(\bar{A}) > w)$
15: $A = A \vee \bar{A}$
16: $w = W - wt(A)$
17: **while** $(w \neq 0)$
18: **return** A

Therefore, in Step 18, we have $wt(A) \leq W$. Since $Pr(wt(\bar{A}) = 0) < 1$, we can conclude that $\mathbb{E}\left[wt(A)\right] \leq \mathbb{E}\left[wt(A) + wt(\bar{A})\right]$. This implies that the algorithm stops after some number of rounds, and when it does, we have $wt(A) = W$. Clearly, Algorithm 4 outputs every string in S_L^W with equal probability because $Pr(A[i] = 1) = \frac{W}{L}$ for every bit $0 \leq i < L$, independently of the other bits. □

4 Different representations of strings

In general, algorithms use the sparse polynomials (in $\mathbb{F}_2[x]$ or $\mathbb{F}_3[x]$) in different ways that may use different representations of the data. As a result, Algorithm 4 that generates binary strings, may be less efficient than its alternatives, due to the performance cost of moving across the different representations used in a specific scheme (especially in constant time implementations). We give two examples. The additional implementation of BIKE [4] uses three representations for $a(x) \in \mathbb{F}_2[x]$: a) an L-bit string A; b) an $8L$-bit string (B) where each byte holds one bit $(B[8i] = A[i],\ 0 \leq i < L)$; c) a list of indexes $\{i : A[i] = 1, 0 \leq i < L\}$. The implementations of NTRU LPrime [6] represent a polynomial $b(x) \in \mathbb{F}_3[x]$ in three ways: a) a 32-bit integers array C, where $C[32i + 1 : 32i] = b_i + 1 \pmod 3$, $0 \leq i < L$ (used for sorting); b) a $2L$-bit array D, where $D[2i + 1 : 2i] = b_i + 1 \pmod 3$, $0 \leq i < L$; c) An $8L$-bit array E, where $E[8i + 7 : 8i] = D[2i + 1 : 2i] - 1$ (the values are `00000000`, `00000001`, and `11111111`).

Indeed, converting a binary string into a list of indexes in constant time is expensive and therefore the Rejection method is first used for generating the list

of indexes, which is later converted into a binary string. By contrast, moving across the representation in NTRU LPrime is sufficiently fast, thus using the RepeatedAND method is preferred (see Section 6).

5 Handling the case $q = 3$

An algorithm for converting a binary string into a ternary string is described in Algorithm 5. The algorithm gets a binary string S with a fixed weight W and an auxiliary (pseudo)random L-bit string B with $wt(B \wedge S) < W$ as its inputs ($B = B' \wedge S$, where B' is a (pseudo)random L bit vector). It outputs a $2L$ bits ternary string D where the "zero" symbol is `01` and the other two symbols are `10` and `00` (as in NTRU LPrime [6]). Step 1 of Algorithm 5 maps $0 \rightarrow$ `01` and $1 \rightarrow$ `10`. Step 2 uses B to decide which `10` symbol will be converted into a `00` symbol. To this end, it squares B (adds a 0 bit next to each bit) and multiplies the result by x. This ensures that only bits in even positions can equal 1. Finally, we achieve the results by XORing the two strings. Note that for cases like NTRUEncrypt that use a ternary vector with a fixed number of 1 symbols (d) and -1 symbols (e), we require that $wt(B \wedge S) = d$. To generate B, we use a modified version of Algorithm 4, where we replace Step 9 with $\bar{A} = A_j \wedge (\neg A) \wedge S$ and set $W = d$.

A fast implementation on modern $x86$ platforms can use the carry less multiplication (`PCLMUL`) instruction for squaring a polynomial in $\mathbb{F}_2[x]$, bit-wise XOR for addition and shift left for multiplication by x.

Algorithm 5 Converting a binary string into a ternary string

Input: S (an L bits string with weight W), B (an L bits string with $wt(B \wedge S)$)
Output: $C \in \mathbb{F}_3[x]$ (a $2L$ bits string with weight W)
Comment: The operations are in $\mathbb{F}_2[x]$
1: **procedure** ConvBinary2Ternary
2: $tmp = S^2 + \sum_{i=0, i-even}^{l} x^i$
3: $D = (x \cdot B^2) + tmp$
4: **return** D

6 Results

To compare the RepeatedAND method to the studied alternatives, we prepared optimized software implementations (specifically, in `C` using AVX/AVX2 intrinsics). For the Rejection method, we used our Additional implementation of BIKE [4] (written in `C` and x86 assembly). For the Sorting method, we used the implementation given in [5] (written in `C` and uses AVX2 intrinsics). We did not implement the Shuffling method because arguably, its constant time implementation would have roughly the same performance as the Rejection methods.

For all these methods we used the same PRF. Specifically, an efficient implementation (using AES-NI) of AES256 streaming mode, with a 256-bit seed (key) over a 32-bit counter. For computing the weight of an array in constant time we divided the array to 64-bit sub-arrays, and used the `POPCNT` instruction. This instruction receives a 64-bit value and returns its hamming weight (in 3 cycles). The algorithms were compiled with gcc (version 5.4.0) in 64-bit mode, using the "O3" Optimization level, and run on a Linux (Ubuntu 16.04.3 LTS) OS. We carried out the experiments on an Intel® desktop of the 7^{th} Intel® Core^{TM} Generation (Micro-architecture Codename "Kaby Lake"[KBL]) 3.60 GHz Core^{TM} i7 − 7700. This platform had 16 GB RAM, 32K L1d and L1i cache, 256K L2 cache, and 8,192K L3 cache. The Intel® Turbo Boost, Intel® Hyper-Threading Technology, and the Enhanced Intel Speedstep® Technology were disabled.

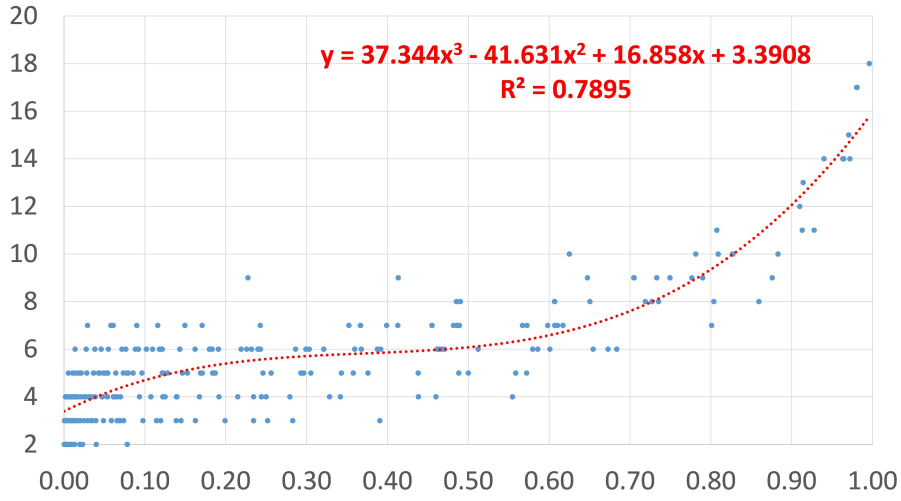

Fig. 1. The average number of rounds (vertical axis) in the RepeatedAND method (lower is better). The points in the graph represent (L, W) pairs and displayed according to the ratio $\frac{W}{L}$.

We performed several experiments. In the first experiment we tested 18 arbitrary values[3] of L, and 32 arbitrary values [4] of W. Altogether, there were 290 legitimate cases (i.e., $W < L$). For every legitimate pair, we ran the RepeatedAND method 30,000 times and recorded the average number of rounds. Figure 1 shows the results. We encountered a small number of rounds $(2 - 4)$

[3] $L = 128, 251, 437, 512, 761, 1,024, 1,493, 2,048, 4,096, 5,312, 8,192, 6,451, 10,163, 16,384, 24,567, 32,749, 32,768, 65,536$.

[4] $W = 10, 30, 50, 71, 110, 250, 286, 350, 512, 897, 1,200, 1,900, 2,500, 3,012, 3,981, 4,196, 4,691, 5,890, 7,891, 9,801, 12,010, 14,909, 15,901, 19,876, 23,090, 27,090, 32,123, 40,954, 51,209, 52,908, 59,908, 65,536$.

only when $\frac{W}{L} < 0.5$ with more occurrences when $\frac{W}{L} < 0.2$. For $\frac{W}{L} > 0.6$, we encountered a relatively large number of rounds (10+). Note that the FWVS problem is symmetric, i. e., solving it for some L and W is the same as solving it for the same L but with $\bar{W} = L - W$ and then negate the results. Thus, it is possible to ignore the cases where $\frac{W}{L} > 0.5$.

The second experiment is the same as above where we disabled the optimization of Algorithm 4, Step 9. Figure 2 shows the difference between the average number of rounds for the same (L, W) pairs in both experiments. Algorithm 4 (with the optimization) is significantly more efficient when $\frac{W}{L} \geq 0.2$ (the difference is usually more than 5 rounds).

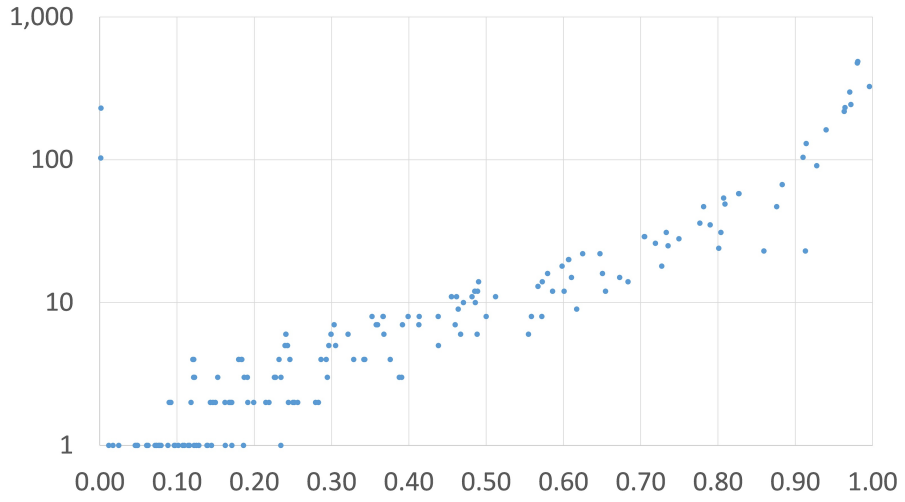

Fig. 2. Number of additional rounds per pair (L, W) (See pairs values in the text) aggregated according to the ratio $\frac{W}{L}$, when the optimization of Algorithm 4, Step 9 is disabled.

Table 2 compares the constant time implementations of the RepeatedAND , to the Rejection, and Sorting methods over the parameters of BIKE1, Lizard, NTRUEncrypt, and NTRU LPrime. Note that the final measurements for NTRUEncrypt should take into account the generation time of vector B in Alg. 5.

For easy reproducibility of the results, we used the benchmarking system SUPERCOP [7] in order to measure the impact of Algorithm 4 on the overall performance of NTRU LPrime (ntrulpr4591761)[5]. We modified the function `small_seeded_weightw` in the NTRU LPrime implementation so that it calls Algorithms 4 and 5. Subsequently, we used the function `small_decode` (of the

[5] Note that SUPERCOP uses checksums that were generated by running NTRU LPrime with its original Sorting method. Thus, to use SUPERCOP for measuring the performance with our (different) method, we removed the checksums

Table 2. Method comparison over some of the parameters of BIKE1, Lizard, NTRUEncrypt (NTRUEn), and NTRU LPrime (NTRULPr). The reported cycles include the (pseudo)random data generation. Fastest result for each parameter choice is marked with bold. The RepeatedAND and the Sorting implementations generate ternary strings while the Rejection implementation generate binary strings

Alg.	Cat.	L (bits)	W (bits)	$\frac{W}{L}$	RepeatedAND method (cycles)	Rejection method (cycles)	Sorting method (cycles)
BIKE1	1	10,163	71	0.007	57,086	**17,460**	160,313
BIKE1	1	20,326	134	0.006	137,130	**45,301**	390,546
BIKE1	5	32,749	137	0.004	189,791	**43,699**	643,000
BIKE1	5	65,498	261	0.004	426,333	**141,499**	1,502,388
Lizard	1	536	140	0.261	**2,377**	41,510	7,589
Lizard	5	1,024	200	0.195	**5,072**	81,251	13,811
NTRUEn	1	443	143	0.322	**2,006**	33,871	6,710
NTRUEn	5	743	247	0.332	**3,414**	82,928	10,722
NTRULPr	5	761	250	0.328	**3,086**	81,674	9,000

original code) to convert the resulting string into the required format. Figure 3 compares the 32 measurements taken by SUPERCOP for the key generation/encapsulation/decapsulation steps of the original implementation and of our optimization. The original implementations run them in constant time, unlike our optimized implementation, where Algorithm 4 has a variable number of rounds (but no secret information is leaked). Our optimization outperforms the baseline in $27/32 = 84\%$ (point A - key generation), $25/32 = 78\%$ (point B - encapsulation), and $21/32 = 65\%$ (point C - decapsulation), achieving a speedup of around $1.07\times$, $1.12\times$, $1.21\times$, respectively.

7 Conclusion

We presented a new algorithm for speeding up the FWVS problem. The implementation of Algorithm 4 achieves speedup of $3\times$ over the equivalent latest implementation in NTRU LPrime one of the schemes that were selected for the NIST PQC project round 2 [3].

In addition, our experiments show that the RepeatedAND method can speed-up other cryptosystems (e. g., Lizard) when L is small. If the value of L is high (as in BIKE1) it is preferable to use the Rejection method (or the Shuffling method). In addition, our experiments show that the RepeatedAND method is usually faster than the Sorting method (on the measured sizes). We point out that in cases where constant-time implementation is not required, the Shuffling or the Rejection methods are probably the fastest.

The performance of the RepeatedAND method depends on the performance of the underlying PRF. In our experiments, we used AES256 as an efficient choice. However, if a 128 bits key is sufficient (e. g., for NIST Category 1 algo-

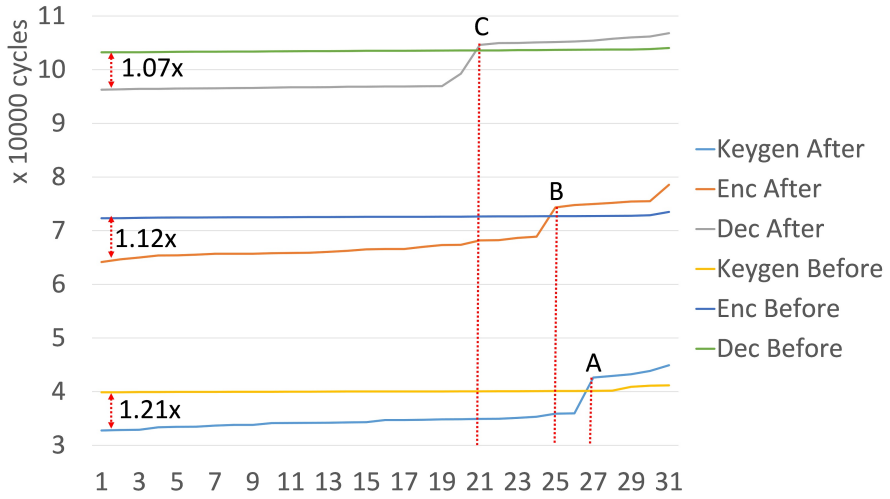

Fig. 3. Cycles count comparison for 32 NTRU LPrime (ntrulpr4591761) keygen/encapsulate/decapsulate measurements taken by SUPERCOP. Lower number of cycles is better. Here, "Before" refers to the original implementation and "After" refers to the optimization through Algorithm 4.

rithms), using AES128 will speed-up the algorithm. In addition, our experiments target x86 platforms with AES-NI enabled. On these platforms, using AES in counter mode leads to a very fast PRF. Implementations that target other platforms (without AES-NI) may choose to use a different PRF (e. g., based on hash functions).

Better news are coming soon: vector AES and POPCNT new instructions Intel has recently announced [1] that its future architecture, microarchitecture codename "Ice Lake", will add vectorized capabilities to the existing AES-NI instructions. These instructions are intended to push the performance of AES software further down, to a new (theoretically achievable) throughput of 0.16 C/B [9]. In addition, CPUs with AVX512 capabilities arrive with a new vectorized `POPCNT` instruction that performs eight 64-bit `POPCNT` in one instruction. Judicious use of these new instructions will significantly accelerate the RepeatedAND algorithm.

Acknowledgments

We thank an anonymous reviewer for the comment that led to Algorithm 6.
This research was supported by: The Israel Science Foundation (grant No. 1018/16); The BIU Center for Research in Applied Cryptography and Cyber Security, in conjunction with the Israel National Cyber Bureau in the Prime Minister's Of-

fice; the Center for Cyber Law & Policy at the University of Haifa in conjunction with the Israel National Cyber Directorate in the Prime Ministers Office.

References

1. −: Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf (October 2017)
2. NIST Post Quantum Cryptography - Round 1 Submissions. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions (2018)
3. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. Tech. rep. (2019). https://doi.org/10.6028/NIST.IR.8240, https://doi.org/10.6028/NIST.IR.8240
4. Aragon, N., Barreto, P.S.L.M., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Guneysu, T., Melchor, C.A., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J.P., Zmor, G.: BIKE: Bit Flipping Key Encapsulation (2017), https://bikesuite.org/files/BIKE.pdf
5. Bernstein, D.J.: djbsort. https://sorting.cr.yp.to/index.html (2018)
6. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU Prime: Reducing Attack Surface at Low Cost. In: Adams, C., Camenisch, J. (eds.) Selected Areas in Cryptography – SAC 2017. pp. 235–260. Springer International Publishing, Cham (2018)
7. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/ (December 2018)
8. Drucker, N., Gueron, S.: A toolbox for software optimization of QC-MDPC code-based cryptosystems. Cryptology ePrint Archive, Report 2017/1251 (2017), https://eprint.iacr.org/2017/1251
9. Drucker, N., Gueron, S., Krasnov, V.: Making AES great again: the forthcoming vectorized AES instruction. Cryptology ePrint Archive, Report 2018/392 (2018), https://eprint.iacr.org/2018/392
10. Durstenfeld, R.: Algorithm 235: random permutation. Communications of the ACM **7**(7), 420 (1964)
11. Fisher, R.A., Yates, F., et al.: Statistical tables for biological, agricultural and medical research. Statistical tables for biological, agricultural and medical research. (Ed. 3.) (1949)
12. Hoffstein, J., Howgrave-Graham, N., Pipher, J., Whyte, W.: Practical Lattice-Based Cryptography: NTRUEncrypt and NTRUSign, pp. 349–390. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-02295-1_11, https://doi.org/10.1007/978-3-642-02295-1_11
13. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Buhler, J.P. (ed.) Algorithmic Number Theory. pp. 267–288. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)

A A variant of the RepeatedAND method

Step 13 of Algorithm 4 ($\bar{A} = \bar{A} \wedge A_j$) can be replaced with $\bar{A} = \bar{A} \wedge \neg A_j$ without affecting the correctness or the performance characteristics of the algorithm. This is because

$$wt(\bar{A} \wedge A_j) \approx wt(\bar{A} \wedge \neg A_j) \approx \frac{wt(\bar{A})}{2} \tag{2}$$

and

$$\mathbb{E}\left[wt(\bar{A} \wedge A_j)\right] = \mathbb{E}\left[wt(\bar{A} \wedge \neg A_j\right] = \mathbb{E}\left[wt(\bar{A})/2\right] \tag{3}$$

Algorithm 6 is a variant of Algorithm 4 that leverages this fact. It replaces Step 13 in Alg. 4 with Steps 13-19. The following example illustrates this optimization.

Algorithm 6 A variant of the RepeatedAND method

```
    Input: seed, L, W
    Output: A (an L bits string with weight W)
 1: procedure GenString(seed, L, W)
 2:     st = InitPRF(seed)
 3:     ctr = 0
 4:     w = W
 5:     A[L − 1 : 0] = 0^L
 6:     do
 7:         j = 0
 8:         A_j = GetRand(st, L)
 9:         Ā = A_j ∧ ¬A                                  ▷ Optimization
10:         do
11:             j = j + 1
12:             A_j = GetRand(st, L)
13:             if wt(Ā ∧ A_j) ≤ w and wt(Ā ∧ ¬A_j) ≤ w then
14:                 Ā = max(Ā ∧ A_j, Ā ∧ ¬A_j)
15:             else
16:                 if wt(Ā ∧ A_j) ≤ w then
17:                     Ā = Ā ∧ A_j
18:                 else
19:                     Ā = Ā ∧ ¬A_j
20:         while (wt(Ā) > w)
21:         A = A ∨ Ā
22:         w = W − wt(A)
23:     while (w ≠ 0)
24:     return A
```

Example 6. Let $L = 2,048$, $W = 500$, and assume that $wt(A_0) = 1,024$ and $wt(A_0) \wedge wt(A_1) = 524$. Then $wt(A_0) \wedge \neg wt(A_1) = 1,024 - 524 = 500 = W$. Therefore, Algorithm 6 ends after one round, while Algorithm 4 will ends after at least two rounds.

Algorithm 6 is only one example of a greedy algorithm that uses (2) to optimize Algorithm 4. Other optimizations may apply for specific choices of L and W. For example, when $L = 2,048$ and $W = 200$ a sequence $\bar{A}_{i\geq 0} = \{\ldots, 800, 400, 200\}$ will probably lead to a smaller number of rounds compared to the expected sequence $\bar{A}_{i\geq 0} = \{\ldots, 512, 256, 128\}$.

B Bounding the probability that the RepeatedAND algorithm does not stop

In theory, Algorithm 4 can enter an infinite loop if $wt(A)$ does not change for an infinite number of times at a) Step 13 ;b) Step 15.

We first explain the claim in Lemma 1 that Algorithm 4 stops almost surely (i. e., the probability that it does not stops is negligible). We start by calculating the probability that the loop in Steps 10-14 ends after $L-W$ iterations. Suppose that the vector A with weight $x = wt(A)$ is converted to the vector A' with $y = wt(A')$ at the end of a single iteration. We consider the Markov chain that corresponds to transition from x to y, where we label the $L+1$ weights (states) by $0, \ldots, L$. The transition matrix $P_{x,y}$ is:

$$P_{x,y} = \begin{cases} \frac{1}{2^x} \cdot \binom{x}{y} & W < x \leq L,\ 0 \leq y \leq x, \\ 1 & 0 \leq x = y \leq W \\ 0 & \text{otherwise} \end{cases}$$

In particular, $P_{x,x} = \frac{1}{2^x} < \frac{1}{2^W}$ for $x > W$ and therefore

$$P\left(x > y\right) = 1 - \frac{1}{2^x} > 1 - \frac{1}{2^W}$$

If $x > y$ in at least $L - W$ iterations we get $y < W$ (because every iteration reduces the weight by at least 1). Since the loop iterations are independent, after $L - W$ iterations we get (if $2^W \geq 5$)

$$\begin{aligned} P\left(wt(A) \leq W\right) &> \left(1 - \frac{1}{2^W}\right)^{L-W} \\ &= \left(\left(1 - \frac{1}{2^W}\right)^{2^W}\right)^{\frac{L-W}{2^W}} \\ &> \left(\frac{1}{e} - 0.05\right)^{\frac{L-W}{2^W}} > \left(\frac{1}{2}\right)^{\frac{L-W}{2^{(W-1)}}} \end{aligned}$$

For example, in NTRU LPrime, $L = 761$, $W = 250$, and $\frac{L-W}{2^{(W-1)}} = \frac{511}{2^{249}} \approx \frac{1}{2^{239}}$. Consequently, the probability that the loop ends after at most $L - W = 511$ iterations is almost 1.

Case (b) is the case where $y = 0$ for an infinite number of rounds of the external loop (Steps 6-17), i. e., the Markov chain hits the absorbing state with weight 0 an infinite number of times. The probability to hit this absorbing state is $0 < h_{n,0} < \frac{1}{2^W}$ (for every $x < W$, $P_{x,0} = \frac{1}{2^x} < \frac{1}{2^W}$) and by the same reasoning as above, the probability to avoid it in $L - W$ rounds/attempts is almost 1.