Speeding up R-LWE post-quantum key exchange

Shay Gueron[1,2] and Fabian Schlieker[3]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Corporation, Israel Deveopment Center, Haifa, Israel
[3] Horst Görtz Institute for IT-Security, Ruhr University Bochum, Germany

Abstract. Post-quantum cryptography has attracted increased attention in the last couple of years, due to the threat of quantum computers breaking current cryptosystems. In particular, the key size and performance of post-quantum algorithms became a significant target for optimization. In this spirit, Alkim et al. have recently proposed a significant optimization for a key exchange scheme that is based on the R-LWE problem. In this paper, we build on the implementation of Alkim et al., and focus on improving the algorithm for generating a uniformly random polynomial. We optimize three independent directions: efficient pseudo-random bytes generation, decreasing the rejection rate during sampling, and vectorizing the sampling step. When measured on the latest Intel processor Architecture Codename Skylake, our new optimizations improve over Alkim et al. by up to 1.59x on the server side, and by up to 1.54x on the client side.

Keywords: Post-quantum key exchange, Ring-LWE, software optimization, AVX2, AVX512, AES-NI

1 Introduction

Cryptographic algorithms that are based on number theoretical problems like factorization and discrete logarithm can be broken if and when quantum computers are available. Sufficiently large quantum computers do not exist today, but can be expected to be built in the foreseeable future (e. g., in 10-15 years [1]). Fortunately, lattice theory offers mathematical problems that seem to not be vulnerable to such attacks. Therefore, lattice-based cryptosystems emerge as a viable secure post-quantum alternative.

Lattice-based algorithms have already been proposed for the important cryptographic primitives such as digital signatures, encryption, and key exchange. Specifically, Ding et al. introduced a lattice-based key exchange scheme, which was improved by Peikert [4,13]. A concrete instantiation of this algorithm has been recently proposed by Bos et al. [3]. The work of [3] is quite substantial, and includes a software implementation that can be directly integrated into the OpenSSL library. The implementation is optimized at the algorithmic level (e. g., uses NTT for polynomial multiplication), but since it relies on generic arithmetic

libraries and sampling from the Gaussian distribution, its performance can be improved.

Alkim, Ducas, Pöppelmann and Schwabe [2] (ADPS hereafter) addressed the performance issue of the implementation in [3] by using a more optimal parameter choice, and by optimizing the key exchange scheme with hand crafted low level assembly code. In addition, they showed that for key exchange (in contrast to digital signature schemes), it suffices to sample secret random values from a centered binomial distribution rather than from discrete Gaussian distributions. This reduces the associated computational costs significantly. As a result, their implementation is an order of magnitude faster than [3]. The source code was published online. In this paper, we build on the ADPS implementation and improve its performance further.

Our contribution. The work of ADPS focused on optimizing the polynomial multiplication arithmetic part of the algorithm. As a result, its relative weight in the overall computation time was reduced. With that, the pseudorandom polynomial generation (called "`parse`" in the paper), which was previously a small building block in the protocol, becomes ~45% of the computation time (for both the server and the client sides). We therefore focus our efforts on the `parse` function, and present optimizations at three independent levels:

- Reduce the rejection rate of pseudorandom candidates during the sampling step from 25% to 6%.
- Parallelize the rejection sampling step using AVX2 (and furthermore AVX512) instructions.
- Replace the SHAKE-128 extendable-output function (XOF) [14], for generating pseudorandom bytes, by a faster, parallel implementation of SHA-256. Alternatively, replace the hash based generation with one based on AES256.

We remark that the source code for our optimizations is made available online at `https://github.com/fschlieker/newhope`.

Organization of this paper. The paper is organized as follows. In Section 2 we give some background on how ADPS works. In particular, we explain the `parse` function in detail. Section 4 details our proposed optimizations, and the resulting performance is presented in Section 5. We conclude and compare to the performance of the standardized ECDH key exchange in Section 6.

2 Preliminaries

We follow the notation of ADPS, so computations are carried out in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n+1)$, the ring of integer polynomials modulo the polynomial (X^n+1) and with coefficients reduced modulo q. The implementation of ADPS is instantiated with $n = 1024$ and $q = 12289$ (a 14-bit prime). We denote polynomials in this ring by boldface characters.

We briefly outline the protocol[4] (consider the ADPS paper for details [2]).

Server: The server side creates a random seed (e. g., from `/dev/urandom`). A hash function, seeded with this seed, defines a stream of pseudorandom bytes. Uniformly distributed coefficients for a public polynomial **a** are then sampled from this stream, using a function called `parse`. Subsequently, a secret polynomial **s** and an error polynomial **e** (with small coefficients) are sampled from a centered binomial distribution. The server computes $\mathbf{b} = \mathbf{as} + \mathbf{e}$ and sends to the client **b** and the seed.

Client: The client re-generates the same **a** (from the seed) calling the `parse` function. Polynomials **s'**, **e'** and **e"** are sampled from the binomial distribution. Then, it computes $\mathbf{u} = \mathbf{as'} + \mathbf{e'}$ and $\mathbf{r} = \mathbf{HelpRec(v)}$, and sends these values to the server. Additionally, the client calculates $\mathbf{v} = \mathbf{bs'} + \mathbf{e"} = \mathbf{ass'} + \mathbf{es'} + \mathbf{e"}$.

Server: The server computes $\mathbf{v'} = \mathbf{us} = \mathbf{ass'} + \mathbf{e's}$. Now **v** and **v'** on both sides are "close" though not identical, due to the different error polynomials. The small errors can be corrected by a reconciliation mechanism (for which **r** is needed). Finally, server and client can compute the shared key as the SHA-3 hash over the reconciled data that is identical on both sides.

3 Considerations in generating the public polynomial

The proposal in [3] uses a fixed polynomial **a** as a system parameter. In contrast, [2] recommends to generate a fresh **a** for every key exchange, giving two reasons:

*Fend off possible concerns about a backdoored choice of **a**.* The polynomial **a** could be carefully chosen in a way that all the intermediate calculations during the protocol run would have values that are smaller than q. In such case, no reduction takes place, and the secret polynomial **s** can be recovered easily using calculations in $\mathbb{Z}$. This subtle backdoor could potentially allow key escrow to e. g., a standardization body that specified a weak **a**.

Avoid relying only on a single instance of a lattice problem. A fixed **a** gives a powerful attacker the possibility to focus on finding a short basis for that particular lattice (using a lot of computation power). All traffic exchanged under a key that is generated from **a** could then be possibly decrypted. Generating a fresh **a** for every key exchange mitigates this "all-for-the-price-of-one" attack.

The straightforward approach is to let one party generate **a** and send it to the other during the protocol run. This consumes a lot of network bandwidth because a polynomial is stored in 2 KB of data. A better way is to let *both* parties generate the polynomial independently from pseudorandom bytes that are produced under a shared random seed. With this method, as proposed by Galbraith [5], only the 256-bit seed needs to be transmitted.

[4] A comprehensible overview can also be found in a blog post by A. Langley; `https://www.imperialviolet.org/2015/12/24/rlwe.html`

When a fresh **a** per session is required, fast pseudorandom generation is obviously needed in order to assure that the generation does not become a performance bottleneck.

Pseudorandom generation methods. The authors of [2] argue that a security reduction is (only) possible under the Random Oracle Model (ROM), and therefore instantiate their scheme with SHAKE-128 XOF ([14]) that provides 128 bits of post-quantum security. When this XOF is seeded with a 256-bit random seed, it deterministically defines a stream of pseudorandom bytes. The assumption is that the probability to find a "malicious backdoored" polynomial **a** by sampling from this stream, is negligible. In other words, it is infeasible to try many different seeds until a malicious **a** is found.

We propose two alternatives to using of SHAKE-128, both of them offer 128 bits of post-quantum security, and achieve better performance.

1. Parallelized SHA-256. Concatenate the seed with a running counter value, to produce as many hash digests as needed for collecting a sufficient amount of pseudorandom bytes. This procedure can be parallelized, because the digests are computed from independent blocks.
2. Using a block cipher (AES256). First, hash the seed (using SHA-3) and generate a 256-bit value to be used as a key for AES256. Then, produce as many blocks as are needed, by encrypting and incrementing a counter value.

Remark 1. By first hashing the seed before using it directly as a key for a block cipher (AES256), we make sure that crafting a malicious **a** would not be possible by only finding certain AES keys that result in a desired ciphertext (a task, which is, by itself, computationally infeasible). In this scenario, one also needs to find a SHA-3 input that produces the desired key. Therefore, the construction can also be seen as a ROM instantiation.

Remark 2. We show below that the block cipher alternative performs better than hashing. However, if one wishes to operate directly under the same ROM assumption as in [2], it is possible to choose SHA-256 and still enjoy performance improvements.

4 Our optimizations

This section describes our optimizations and their software implementation.

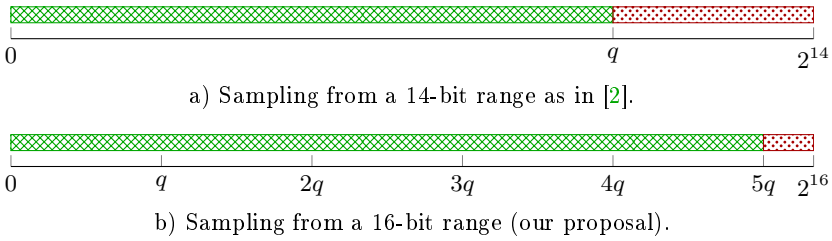
4.1 Decreasing the rejection rate

The function `parse`, that generates **a**, receives a seed and generates (using SHAKE-128 XOF) pseudorandom bytes as "candidates". These pseudorandom bytes are post-processed to sample the $n = 1024$ coefficients for **a**. Every pair of bytes of the pseudorandom stream is viewed as a 16-bit candidate. In ADPS,

the two most significant bits are zeroed to create a 14-bit value. If this value is smaller than q, it is accepted as a coefficient, and otherwise it is discarded. On average, this process accepts only $\frac{q}{2^{14}} = \frac{12289}{16384} \approx \frac{3}{4} = 75\%$ of the candidates (see Figure 1 a)). To accumulate $n = 1024$ uniformly distributed (over $\mathbb{Z}_q$) coefficients from two-byte words, `parse` needs to check $1024 \cdot 2 \cdot \frac{4}{3} \approx 2730$ bytes on average.

Ignoring two bits of every sample with a rejection rate of 25% is not optimal and we propose to use the full 16-bit sample. While we still need to reject some values from the pseudorandom stream (those that are $\geq 5q$ since $\left\lfloor \frac{2^{16}}{q} \right\rfloor = 5$), the overall acceptance probability is now $\frac{5 \cdot q}{2^{16}} = \frac{61445}{65536} \approx 94\%$. However, we need to subtract q up to four times from the accepted candidates to retrieve values in $\mathbb{Z}_q$ (which then remain uniformly random). See Figure 1 b) for an illustration and Listing 1 for the corresponding pieces of source code.

Note that this small change benefits twice: with less values rejected, we need to generate fewer pseudorandom bytes to begin with. Consequently, less values need to be conditionally checked. On average, the proposed routine needs only $1024 \cdot 2 \cdot \frac{65536}{61445} \approx 2184$ pseudorandom bytes in order to populate the coefficients of $\mathbf{a}$.



a) Sampling from a 14-bit range as in [2].

b) Sampling from a 16-bit range (our proposal).

Fig. 1. Sampling uniformly random values in $\mathbb{Z}_q$ from different input ranges. Candidates that are accepted are indicated by the crossed area and candidates in the dotted area are rejected. The acceptance rate is significantly higher when sampling from the 16-bit range. In this case, q might have to be subtracted up to four times from an accepted candidate in order to obtain a coefficient in $\mathbb{Z}_q$, but it remains uniformly random.

Remark 3. Note that since the seed and the generated polynomial $\mathbf{a}$ are meant to be public, the implementation does not need to execute the generation in constant time.

4.2 Vectorized rejection sampling

The process of filtering pseudorandom 16-bit candidates can be accelerated by using SIMD instructions. Specifically, it is possible to handle 16 candidates with

```
a) Rejection-sampling from 14 bits:

candidate = (buf[pos] | ((uint16_t) buf[pos+1] << 8)) & 0x3fff; // take only lower 14 bits
if(candidate < PARAM_Q) // accept as coefficient if < q
  a->v[ctr++] = candidate;

b) Rejection-sampling from 16 bits (our proposal):

candidate = (buf[pos] | ((uint16_t) buf[pos+1] << 8)); // take full 16 bits
r = candidate / PARAM_Q;
if (r < 5) // accept as coefficient if < 5q, since floor(2^16/q) = 5
  a->v[ctr++] = candidate - r * PARAM_Q; // subtract q up to 4 times to end up in Zq
```

Listing 1. Code snippets for a) Rejection sampling from two a bytes input, when the two most significant bits are discarded [2]; b) From the full 16-bit range (our proposal). `buf` contains the pseudorandom bytes and `pos` the position in that buffer. `a->v` points to the coefficients of `a` and `ctr` is incremented until we have 1024 accepted coefficients.

AVX2 instructions (using 256-bit registers) and 32 candidates with AVX512 (using 512-bit registers) [11,12].

Our AVX2 implementation uses a mixture of vector comparisons and permutations in order to compress and align the accepted candidates ($< q$). An illustrative excerpt of the code is given in Appendix A.

Processors with AVX512 support are not available yet, but we verified correctness of our AVX512-vectorized sampling using the Intel Software Development Emulator (SDE) tool.[5] We expect additional performance improvements due to: a) mask operands and VPCOMPRESSD (see Appendix A); b) faster parallelized SHA-256 (see Sections 3 and 4.3) to be visible when processors that support this architecture become available in the near future.

4.3 Fast generation of pseudorandom bytes.

After acquiring a 256-bit random seed (from `/dev/urandom`), the implementation of [2] uses SHAKE-128 XOF to generate the pseudorandom bytes stream. We investigate two alternatives for such generation.

Using SHA-256 with modern SIMD architectures. The AVX2 (AVX512) instructions can be used for computing 8 (16) hashes in parallel [8]. To this end, we built a highly optimized implementation that produces bytes at the rate of 2.75 cycles per byte (C/B) with AVX2 (and much faster on the coming AVX512 architectures).

Using AES (with AES-NI). We used the pipelined AES implementation of [7,6], which performs at 0.92 C/B on our test platform ("Skylake"). We run it in counter mode (CTR), so incrementing counter values are used as plaintexts and encrypted under a fixed key. This has the advantage that the key schedule only needs to be computed once and ciphertext generation can be efficiently pipelined.

[5] Intel Software Development Emulator (SDE) https://software.intel.com/en-us/articles/intel-software-development-emulator

5 Results

This section presents the results of our different optimizations. The performance numbers were obtained by using the test bench included in the implementation of [2]. The measurements were obtained on a platform with the latest Intel® Core™ Generation processor (Architecture Codename Skylake), with the Intel® Turbo Boost Technology, Intel® Hyper-Threading Technology, and Enhanced Intel Speedstep® Technology disabled. The code was compiled with `gcc` version 5.2.0 and full optimizations enabled ("`-O3`"). For consistent comparison, we compiled and measured the baseline implementation [2] on the same system.

Remark 4. During our work, we discovered a bug in this test bench, that leads to somewhat overoptimistic results, presumably due to caching of fixed values across multiple tests. We reported the bug to the authors, together with the appropriate fix, and assume that it will be corrected in the final version of [2]. The results we report here were measured with the *already fixed* version, and therefore deviate from the numbers reported in [2] in January 2016.

The results are presented in Table 1, showing the contribution of the different optimizations. We indicate the distinct optimization methods by abbreviations: reduction of the rejection rate (`I`), vectorization of rejection sampling (`II`), pseudorandom bytes generation using SHA-256 (`III`) and AES256 (`IV`). Note that the last two optimizations (`III` and `IV`) are mutually exclusive. The other optimizations (`I`, `II`) are independent, and are therefore combinable. The difference between the cycles count of the server and the client can be explained as follows. The server needs to obtain a seed from a (typically slow) randomness source, but on the other hand, the client needs to compute one more NTT and polynomial addition during the computations of its part of the exchange.

Table 1. The performance of the different optimizations, compared to ADPS [2] as the baseline. The numbers represent the cycles counts, measured using the test bench (lower is better) and the speedup factor compared to the baseline that is set to 1 (i. e., higher is better).

	Method	`parse` cycles	Server cycles	Server speedup	Client cycles	Client speedup
Baseline	ADPS [2]	59,627	127,712		129,349	
This work	`I`	47,044	113,361	1.13x	115,909	1.12x
	`I, II`	38,466	100,343	1.27x	104,120	1.24x
	`I, II, III`	32,080	94,183	1.36x	97,688	1.32x
	`I, II, IV`	17,053	80,087	1.59x	84,119	1.54x

Figure 2 illustrates our results with all our optimizations enabled, and compares them relatively to the baseline [2] that is set to 1. With the reduced

rejection rate method, vectorized rejection sampling and SHA-256 for pseudorandom generation, the speedup factor is 1.36x for the server and 1.32x for the client. The best speedup is achieved when AES-NI are used for generating pseudorandomness. This increases performance by a factor of 1.59x and 1.54x, on the server and the client side, respectively.

Fig. 2. The highest relative speedup factors on the server and the client sides, achieved by the proposed optimizations (the baseline implementation [2] is set to 1).

6 Conclusion

This paper demonstrated several optimizations that can be used for speeding up R-LWE-based key exchange. Our results show that the server and the client sides can profit from a speedup factor of up to 1.59x and 1.54x, respectively.

For comparison, we also measured the performance of the best available implementation of the standardized ECDH over P-256 key exchange ([9] and its improved version [10]), and found that the key exchange computations consume roughly 223,000 cycles on both sides. With all our optimizations, the R-LWE key exchange takes 80,087 cycles (server) and 84,119 cycles (client) (see Table 1). We point out that the amount of transferred data during the key exchange with R-LWE (4096 bytes) is higher than with ECDH (64 bytes). Note that the parameters in [2] were chosen quite conservatively. An appropriate level of security would be probably achieved with e. g., $n = 512$, which would halve the amount

of transferred data, and speed up computations. The authors justify the choice of $n = 1024$ by being able to thwart possible future advances in cryptanalysis. In any case we can see that, even with the discussed choice of parameters, post-quantum key exchange is already practical on current platforms.

Acknowledgments

This research was supported by the PQCRYPTO project, which was partially funded by the European Commission Horizon 2020 research Programme, Grant #645622.

References

1. IBM's stunning breakthrough: Quantum computing finally 'within reach'. http://www.foxnews.com/tech/2012/02/28/ibm-quantum-computing-as-little-as-10-years-off.html (Feb 2012)
2. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - a new hope. IACR Cryptology ePrint Archive 2015, 1092 (2015), http://eprint.iacr.org/2015/1092
3. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In: 2015 IEEE Symposium on Security and Privacy. pp. 553–570. IEEE Computer Society (May 2015)
4. Ding, J., Xie, X., Lin, X.: A simple provably secure key exchange scheme based on the learning with errors problem. IACR Cryptology ePrint Archive 2012, 688 (2012), http://eprint.iacr.org/2012/688
5. Galbraith, S.D.: Space-efficient variants of cryptosystems based on learning with errors (2013), https://www.math.auckland.ac.nz/~sgal018/compact-LWE.pdf
6. Gueron, S.: Intel ® Advanced Encryption Standard (AES) New Instructions Set. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf (Sept 2012)
7. Gueron, S.: Intel's new AES instructions for enhanced performance and security. In: Dunkelman, O. (ed.) Fast Software Encryption – FSE 2009. Lecture Notes in Computer Science, vol. 5665, pp. 51–66. Springer (Feb 2009)
8. Gueron, S., Krasnov, V.: Simultaneous hashing of multiple messages. J. Information Security 3(4), 319–325 (2012)
9. Gueron, S., Krasnov, V.: Fast prime field elliptic-curve cryptography with 256-bit primes. J. Cryptographic Engineering 5(2), 141–151 (2015)
10. Gueron, S., Krasnov, V.: Improved P256 ECC performance by means of a dedicated function for modular inversion modulo the P256 group order. https://mta.openssl.org/pipermail/openssl-dev/2015-April/001197.html (Apr 2015)
11. Intel Corporation: Intel ® 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf (Sept 2015)
12. Intel Corporation: Intel ® Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf (Aug 2015)

13. Peikert, C.: Lattice cryptography for the internet. In: Mosca, M. (ed.) Post-Quantum Cryptography – PQCrypto 2014. Lecture Notes in Computer Science, vol. 8772, pp. 197–219. Springer (Oct 2014)
14. National Institute of Standards, Technology: FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (2015)

A Vectorized rejection sampling - code snippets

The relevant part of our AVX2 optimizations in the source code is shown in Listing 2. Listing 3 shows the relevant part of our AVX512 optimizations. Note that the AVX512 sampling gets much easier thanks to the new masks feature that gives more targeted data-control in almost all instructions. In particular, the VPCOMPRESSD instruction allows us to write back only specific values instead of a whole vector.

In both these approaches, we incorporate our proposal to reduce the rejection rate as explained in Section 4.1. Since we are working on vectors of integers, we do not have a division function in the AVX integer instructions (like in Listing 1) and implement this by repeatedly comparing and subtracting.

```
1  const __m256i zero = _mm256_setzero_si256();
2  const __m256i modulus8 = _mm256_set1_epi32(PARAM_Q);
3  const __m256i modulus16 = _mm256_set1_epi16(PARAM_Q);
4
5  uint32_t good = 0;
6  uint32_t offset = 0;
7  while(ctr < PARAM_N-16)
8  {
9      __m256i tmp0, tmp1, tmp2;
10
11     tmp0 = _mm256_loadu_si256((__m256i *)&buf[pos]);
12
13     // normalize the values in range
14     tmp1 = _mm256_min_epu16(tmp0,modulus16);
15     tmp1 = _mm256_cmpeq_epi16(tmp1,modulus16);
16     tmp2 = _mm256_and_si256(tmp1, modulus16);
17     tmp0 = _mm256_sub_epi16(tmp0, tmp2);
18     tmp1 = _mm256_min_epu16(tmp0,modulus16);
19     tmp1 = _mm256_cmpeq_epi16(tmp1,modulus16);
20     tmp2 = _mm256_and_si256(tmp1, modulus16);
21     tmp0 = _mm256_sub_epi16(tmp0, tmp2);
22     tmp1 = _mm256_min_epu16(tmp0,modulus16);
23     tmp1 = _mm256_cmpeq_epi16(tmp1,modulus16);
24     tmp2 = _mm256_and_si256(tmp1, modulus16);
25     tmp0 = _mm256_sub_epi16(tmp0, tmp2);
26     tmp1 = _mm256_min_epu16(tmp0,modulus16);
27     tmp1 = _mm256_cmpeq_epi16(tmp1,modulus16);
28     tmp2 = _mm256_and_si256(tmp1, modulus16);
29     tmp0 = _mm256_sub_epi16(tmp0, tmp2);
30
31     tmp1 = _mm256_unpacklo_epi16(tmp0, zero); // transition to epi32
32     tmp2 = _mm256_cmpgt_epi32(modulus8, tmp1); // compare to modulus
33     good = _mm256_movemask_ps((__m256)tmp2);
34     tmp2 = _mm256_permutevar8x32_epi32(tmp1, perm_lut[good]);
35     // ctr includes offset, possible bad values are overwritten
36     _mm256_storeu_si256((__m256i *)&a->v[ctr], tmp2);
37
38     offset = __builtin_popcount(good); // we get this many good (< modulus) values
39     ctr += offset;
40
41     // the very same thing as above, only with unpackhi
42     tmp1 = _mm256_unpackhi_epi16(tmp0, zero); // transition to epi32
43     tmp2 = _mm256_cmpgt_epi32(modulus8, tmp1); // compare to modulus
44     good = _mm256_movemask_ps((__m256)tmp2);
45     tmp2 = _mm256_permutevar8x32_epi32(tmp1, perm_lut[good]);
46     // ctr includes offset, possible bad values are overwritten
47     _mm256_storeu_si256((__m256i *)&a->v[ctr], tmp2);
48
49     offset = __builtin_popcount(good); // we get this many good (< modulus) values
50     ctr += offset;
51
52     pos += 32; // proceed in the pseudorandom buffer
53
54     [...]
```

Listing 2. Vectorized rejection-sampling using AVX2 instructions. First, the candidate values are repeatedly compared to q and q is subtracted up to four times (ll. 14-29). This is the vectorized reduced rejection rate. It is folinesowed by the rejection step, in which the vector is permuted such that the values to be rejected are aggregated in one side of the vector (ll. 31-34, 42-45). A precomputed 8 KB lookup table is needed, in order to hold the 256 possible masks for this permutation. The pointer to the memory destination is increased such that the rejected values are overwritten (ll. 36-39,47-50).

```
1  const __m512i zero = _mm512_setzero_si512();
2  const __m512i modulus16 = _mm512_set1_epi32(PARAM_Q);
3  const __m512i modulus32 = _mm512_set1_epi16(PARAM_Q);
4
5  uint32_t offset = 0;
6  __mmask16 good = 0;
7
8  while(ctr < PARAM_N-32)
9  {
10     __m512i tmp0, tmp1;
11     __mmask32 mask;
12
13     tmp0 = _mm512_loadu_si512((__m512i *)&buf[pos]);
14
15     // normalize the values in range
16     mask = _mm512_cmple_epu16_mask(modulus32, tmp0);
17     tmp0 = _mm512_mask_sub_epi16(tmp0, mask, tmp0, modulus32);
18     mask = _mm512_cmple_epu16_mask(modulus32, tmp0);
19     tmp0 = _mm512_mask_sub_epi16(tmp0, mask, tmp0, modulus32);
20     mask = _mm512_cmple_epu16_mask(modulus32, tmp0);
21     tmp0 = _mm512_mask_sub_epi16(tmp0, mask, tmp0, modulus32);
22     mask = _mm512_cmple_epu16_mask(modulus32, tmp0);
23     tmp0 = _mm512_mask_sub_epi16(tmp0, mask, tmp0, modulus32);
24
25     tmp1 = _mm512_unpacklo_epi16(tmp0, zero);
26     good = _mm512_cmplt_epi32_mask(tmp1, modulus16);
27     _mm512_mask_compressstoreu_epi32((__m512i *)&a->v[ctr], good, tmp1);
28     offset = __builtin_popcount(good); // we get this many good (< modulus) values
29     ctr += offset;
30
31     tmp1 = _mm512_unpackhi_epi16(tmp0, zero);
32     good = _mm512_cmplt_epi32_mask(tmp1, modulus16);
33     _mm512_mask_compressstoreu_epi32((__m512i *)&a->v[ctr], good, tmp1);
34     offset = __builtin_popcount(good); // we get this many good (< modulus) values
35     ctr += offset;
36
37     pos += 64; // proceed in the pseudorandom buffer
38
39     [...]
```

Listing 3. Vectorized rejection-sampling using AVX512 instructions. The preparation step is much shorter, due to mask operands providing more control over the data in vector registers (ll. 16-23). With the VPCOMPRESSD, we can selectively write only specific values to memory and save the expensive permutation from our AVX2 approach (ll. 25-35).