Optimizing the Decoding Process of a Post-Quantum Cryptographic Algorithm

Antonio Guimarães[1], Diego F. Aranha[1], Edson Borin[1]

[1]Institute of Computing – University of Campinas (UNICAMP)
Av. Albert Einstein, 1251 - 13.083-852 - Campinas - SP - Brazil

antonio.junior@students.ic.unicamp.br, {dfaranha, edson}@ic.unicamp.br

Abstract. *QcBits is a state-of-the-art constant-time implementation of a code-based encryption scheme for post-quantum public key cryptography. This paper presents an optimized version of its decoding process, which is used for message decryption. Our implementation leverages SSE and AVX instructions extensions and performs 3.6 to 4.8 times faster than the original version, while preserving the 80-bit security level and constant time execution. We also provide experimental data that indicates a further 1.4-factor speedup supposing the existence of instructions for vectorial conditional moves and 256-bit register shifts. Finally, we implemented countermeasures for side-channel security and showed that they do not affect the overall performance.*

1. Introduction

Recent developments on quantum computers and cryptanalysis create the need for new efficient and secure algorithms for public key cryptography to replace current standards. Conventional algorithms, mostly represented by elliptic curves [Koblitz 1987] and RSA [Rivest et al. 1978], depend on the hardness of integer factorization and computing discrete logarithms, which can be efficiently solvable in a quantum computer by Shor's algorithm [Shor 1999].

Public-key algorithms that do not rely on these problems are known since the late 70s and are today called post-quantum cryptographic techniques. Among them, code-based encryption is becoming one of the most promising. First proposed by McEliece using Goppa codes [McEliece 1978], it was never considered a reasonable alternative until the quantum computing rise, since it was clearly outperformed by other algorithms. Besides the difference in execution time, the very large public keys were also one of the main issues preventing it from becoming competitive.

Aiming to reduce the key size and improve performance while maintaining the security level, many different choices of codes and decoding algorithms were proposed in the past few years. Misoczki et al. [Misoczki et al. 2013] proposed the McEliece cryptosystem instantiation with quasi-cyclic structure with moderate density parity check codes (QC-MDPC). Although recent implementations show that the decoding process is slower than the originally proposed Goppa Codes, it allows a key size reduction from 20,480 bits to 4,801 bits, while keeping the 80-bit classical security level.

Side-channel attacks are also an important factor to consider, and secure implementations should not leak information correlated with critical data (keys and plaintext). More recently, Chou presented QcBits [Chou 2016], a constant-time implementation of a

QC-MDPC code-based encryption scheme. Using polynomial representation and a technique called bitslicing, the implementation is fully protected against timing attacks and two times faster than the previous speed record. This last accomplishment is specially important since the decoding process is probabilistic and executing it in constant-time may be very inefficient and lead to high decoding error rates.

The QcBits implementation was provided in two versions: `ref`, which uses only C code, and `clmul`, which performs arithmetic using the 128-bit carry-less multiplication instruction. Although the original implementations are fully constant-time, they are vulnerable against other side-channel attacks based on power consumption [Rossi et al. 2017].

In this paper, we present the following contributions:

- We optimized the decoding process for both the QcBits versions, achieving a speedup of 3.6 times over the `clmul` version and 4.8 times over `ref`.
- We estimated that gains could be as high as 5.06 times on `clmul` version if new instructions for conditional vectorial moves and 256-bit register shifts were added on the architecture.
- We mitigate all currently known power vulnerabilities found in the original implementation with an almost negligible ($< 1\%$) impact to the overall performance.

Our performance improvement comes from vectorization using AVX instructions, loop unrolling on hot spots and pre-calculation of vector rotations. All the performance measurements were executed on Haswell and Skylake architectures. We focus on the 80-bit security level for comparisons against related work, but higher security levels can be achieved with minimal changes to the implementation and all optimization techniques presented in the paper are still applicable. To the best of our knowledge, our paper is the first to present a fully vectorized software optimization of QcBits.

This work is organized as follows: Section 2 described code-based cryptography and the QcBits algorithm; Section 3 discusses and presents the results of our optimization on the decoding process of QcBits; Section 4 explains the countermeasure applied to mitigate a power channel vulnerability found in the original implementation; Section 5 presents some related work; and Section 6 discusses the final conclusions and future work.

2. Code-based Cryptography and the QcBits algorithm

The McElice cryptosystem [McEliece 1978] was the first code-based encryption scheme ever proposed and still remains as the most relevant one. Its security is based on the hardness of decoding linear codes, which is an NP-complete problem. The original scheme used Goppa Codes, which enabled great performance due to very efficient decoding algorithms, but keys took 460Kb at the 80-bit security [Bernstein et al. 2008], making the system not competitive in terms of viability among the alternatives.

Equation 1 shows the encryption in the McEliece Cryptosystem: m is a message of length k; z is an error vector with Hamming Weight t; and G' is a $k \times n$ matrix defined on Equation 2, where S is a scrambling matrix, G is the generator matrix for the chosen code (e.g. Goppa Code) and P is a permutation matrix. All these matrices are randomly generated and the last 3 of them compose the cryptosystem private key, while their product G' is the public key. The decryption is shown on Equation 3, where $Decode$ is the decoding algorithm for the chosen code.

Using Goppa codes at the 80-bit security, the parameters k, n, and t are chosen respectively as 1632, 1269 and 34 bits, resulting in the 460Kb public key size. Many techniques were proposed in order to reduce the key size of Goppa codes. Misoczki and Barreto [Misoczki and Barreto 2009] proposed a dyadic structure, but although they successfully presented a viable small-key alternative with just 20Kb, it resulted in structural vulnerabilities [Faugere et al. 2010].

$$c' = mG' + z \tag{1}$$

$$G' = SGP \tag{2}$$

$$m = Decode(cP^{-1})S^{-1} \tag{3}$$

As an alternative to Goppa codes, QC-MDPC codes [Misoczki et al. 2013] were first introduced in 2013, allowing the use of very compact keys. Table 1 shows a key length comparison between QC-MDPC codes and some of the previous alternatives.

Table 1. Key length in bits for different codes (from [Misoczki et al. 2013])

Security Level	QC-MDPC	QD-Goppa	Goppa
80	4,801	20,480	460,647
128	9,857	32,768	1,537,536
256	32,771	65,536	7,667,855

This great reduction in key size was achieved thanks to the quasi-cyclic (QC) structure. As discussed before, the public and private keys in the cryptosystem are all matrices composed of circulant blocks and this quasi-cyclic structure allows them to be represented by the first row only, as illustrated below:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & \quad & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Besides exploiting the quasi-cyclic structure, another advantage of QC-MDPC codes is eliminating the need for scrambling and permutation matrices in the McEliece cryptosystem. The generator matrix G is the public-key and a parity check matrix H is the private key used in the decoding algorithm. Considering this, the decryption process boils down to the plain decoding, which is shown on Algorithm 1. For these codes, the generator matrix G is the row reduced echelon form of the parity check matrix H.

In Algorithm 1, H is the parity check matrix, c is the ciphertext and TH is an experimentally determined variable threshold, depending on the approach. The first line is the syndrome calculation, defined as the multiplication between the parity check matrix and the ciphertext. If it results in a zero vector, then there is no error in the message and therefore the decoding successfully finishes. Otherwise, the bit-flipping algorithm, represented by the `for` loop on Algorithm 1, must be executed. This algorithm works by calculating the Hamming Weight, which is the number of ones, of the `and` between each parity check matrix column and the calculated syndrome. If this Hamming Weight is greater than the threshold, the function $FlipBit$ flips the i-th bit (corresponding to the column) of the ciphertext c. Once the ciphertext is changed, the process restarts.

Algorithm 1: QC-MDPC Bit-flipping decoding.

```
Input  : $H$, $c$ and $TH$
Output: c
$s \leftarrow H \times c$
while $s \neq 0$ do
    foreach column $h_i$ in $H$ do
        $hd \leftarrow HammingWeight(h_i \wedge s)$
        if $hd > TH$ then
            $FlipBit(c, i)$
    end
    $s \leftarrow H \times c$
end
```

2.1. QcBits

QcBits [Chou 2016] is a state-of-the-art implementation of an encryption scheme based on QC-MDPC codes. It established new speed records and is the first fully constant-time implementation for this type of code. The speed improvement was achieved by representing the cryptosystem matrices, represented by its first row, as binary polynomials over $(x^r - 1)$, where r is a code size parameter. This was possible thanks to the quasi-cyclic structure. This way, all the syndrome calculations were done as polynomial multiplications instead of the less efficient general matrix multiplications. The polynomial view also helps with key generation, where the generator matrix was calculated using a polynomial inversion of the parity check matrix.

QcBits was presented in two versions: the C-only `ref` version, and the `clmul` version using the `PCLMULQDQ` instruction [Gueron and Kounavis 2010] to accelerate polynomial arithmetic. On both version, the bit-flipping in Algorithm 1 was implemented using constant-time vector rotations and bitslicing. Since it was the main target of our optimization, it will be further explained in the following subsection.

Aside from raw performance, constant-time execution was also an important implementation feature. It enabled the code to be resistant against timing side channel attacks, which was a problem for the previous implementation [Strenzke et al. 2008]. The decoding algorithm was the most challenging part of the implementation to protect. As shown in Algorithm 1, the original form of the algorithm is inherently variable time because the decoding only stops when all errors are corrected. To work around this problem, QcBits determined a maximum number of iterations for the decoding (6 at the 80-bit security level), and failure otherwise. There's no proof or strict estimate indicating that 6 iterations are enough for a practical secure use of the implementation, but empirical tests showed an acceptably low failure rate.

2.1.1. Bit-flipping algorithm

Algorithm 2 shows the implementation of decoding in QcBits. TH is the iteration threshold, s is the syndrome, c is the ciphertext and H' the sparse representation of the parity check matrix, which is an array of non-zero indices. The $BitSliceAdder$ function consists in adding each bit individually by positioning and storing each bit of the result in an array position (Algorithm 3), similarly to a half adder circuit. The $BitSliceSubtractor$ is implemented in the same way, but with a full adder or subtractor instead.

Algorithm 2: QcBits Bit-flipping implementation logic

```
  Input  : $H'$, $c$, $s$ and $TH$
  Output: $c$
1 $N \leftarrow 1 + \lceil log_2(|H'|) \rceil$
2 $sum[N] \leftarrow 0's$
3 foreach index $i$ in $H'$ do
4     $w \leftarrow s \lll i$
5     $sum \leftarrow BitSliceAdder(sum, w)$
6 end
7 $sum \leftarrow BitSliceSubtractor(sum, TH)$
8 $c \leftarrow \neg sum[N-1] \oplus c$
```

Algorithm 3: BitSlice Adder Implementation Logic

```
Input  : $N$, $sum$ and $w$
Output: $sum$
for $i = 0$ to $N$ do
    $c_{out} \leftarrow sum[i] \wedge w$
    $sum[i] \leftarrow sum[i] \oplus w$
    $w \leftarrow c_{out}$
end
```

Line 1 in Algorithm 2 calculates the number of bits necessary to represent the number of elements belonging to H', which is the maximum result that can be stored on the sum array by the $BitSliceAdder$. Line 2 initializes sum with zeros. The loop on line 3 iterates over the private key indices: for each index, the syndrome is rotated left on the index value (line 4) and the result is added to the sum array using the $BitSliceAdder$ function. This process is equivalent to calculating the Hamming Weight of the bitwise AND between each matrix column and the syndrome. However, for 80-bit security, instead of iterating over the 4801 rows of the parity check matrix, this method just needs to iterate over the 90 indices of the sparse matrix representation. At the end of the loop, the threshold is subtracted from the sum of each bit. If the most significant result bit is one on line 8, it indicates that the threshold is greater than the sum and the corresponding bit must not be flipped. Otherwise, the bit is flipped.

3. QcBits Optimization

We began our optimization by extending the vectorization to the whole code using SSE4 instructions for 128-bit registers, available since Intel Nehalem, and using AVX2 instructions for 256-bit registers, available since Haswell. Our initial expectation was obtaining a 2-factor speedup in the first case and 4 in the latter since these values correspond to the number of SIMD lanes found on these standards. Most of the code was composed of bitwise operations, such as XOR and AND of the bit slice adder, and were easily vectorizable, resulting in an immediate gain of 2.6 times when using the AVX2 instruction set. However, the absence of some instructions on the SIMD instruction sets prevented those expectations from materializing.

The main obstacle for vectorization was the implementation of 128-bit and 256-bit register shifts. These operations are necessary to perform the vector rotations shown on line 4 of Algorithm 2. For the 80-bit security level, the rotation target has 4801 bits and it is implemented in two steps using C language: first, the words that compose the vector are permuted following the rotation logic; next, the rotation is done inside each word, shifting its bits and inserting next word bits in the shifted area. For registers with size lesser or equal to 64 bits there's a single instruction to shift all the register bits, which facilitates the implementation. For larger registers we had to perform a custom multi-instruction logic, making the implementation slower and more complex.

Listing 1 shows our implementation of a shift right with carry on AVX2 registers, used in the vector rotation shown on line 4 of Algorithm 2. The code is composed by 10 intrinsics for vector instructions. It works by permuting 64-bit sets to reduce the shift amount to less than 64, then the Carry In is inserted using the `BLENDV` instruction and the shift is finished using instructions that shifts inside the 64-bit lanes. Some of the used instructions are very expensive, like the `PERMUTEVAR` instruction on line 12 and 19, which has 3-cycle latency in Skylake, according to Agner Fog's instruction tables [Fog 2011].

```
1   unit bitShiftRight256xmmCarry (unit data, int count, unit * carryOut, unit carryIn){
2     unit innerCarry, out, countVet;
3     unit idx = _mm256_set_epi32(0x7, 0x6, 0x5, 0x4, 0x3, 0x2, 0x1, 0x0);
4     const unit zeroMask = _mm256_set_epi64x(-1, -1, -1, 0);
5     unit zeroMask2 = _mm256_set_epi8(0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80,
6                                      0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82,
7                                      0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84,
8                                      0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86);
9
10    countVet = _mm256_set1_epi8((count >> 5) & 0xE);
11    idx = _mm256_add_epi8(idx, countVet);
12    data = _mm256_permutevar8x32_epi32(data, idx); // rotate
13    *carryOut = data;
14    zeroMask2 = _mm256_sub_epi8(zeroMask2, countVet);
15    data = _mm256_blendv_epi8 (carryIn, data, zeroMask2);
16    // shift less than 64
17    count = (count & 0x3F);
18    innerCarry = _mm256_blendv_epi8(carryIn, data, zeroMask);
19    innerCarry = _mm256_permute4x64_epi64(innerCarry, 0x39); // >> 64
20    innerCarry = _mm256_slli_epi64 (innerCarry, 64 - count);
21    out = _mm256_srli_epi64 (data, count);
22    out = _mm256_or_si256 (out, innerCarry);
23    return out;
24  }
```

Listing 1: 256-bit register shift implementation

For the `clmul` version vectorized with AVX2 instruction, the syndrome calculation was also a problem. Executed at the beginning of the decoding process, it was originally implemented using the carry-less multiplication instruction which is only available for 128-bit size registers. Therefore, this code snippet, which takes approximately 20% of the code execution time, is stuck at the 128-bit implementation.

3.1. Basic Vectorization Results

We compiled the implementations using the three industry-standard compilers: GCC 6.1.3, CLANG 3.9.1 and ICC 17.0.2. For all the compilers, the compilation optimization flags used were `-O3` and `-march=native`. The flag `-funroll-all-loops` was also used when compiling with GCC. Equivalent flags for aggressive loop unrolling on the other compilers were tested, but they didn't result in any performance improvement and therefore were removed. The implementations were executed in two machines: the first one, named Haswell, uses an Intel i7-4770 processor and the second, named Skylake, uses an Intel i7-6700k processor. Both machines run a Fedora 25 operating system and, aiming at experiment reproducibility and cycle accuracy, had the Intel Turbo Boost and Hyper-Threading mechanisms disabled. The performance results of this first vectorization are shown in the graph of Figure 1.

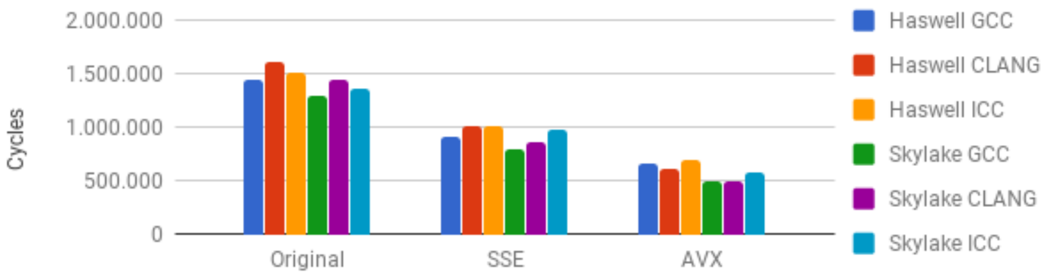

Figure 1. Initial vectorization results

As can be noted from the graph, the execution time, considering the compilation with GCC, reduced from 1,292,380 Skylake cycles and 1,441,220 Haswell cycles to respectively 803,970 and 912,498 cycles when using the SSE instruction set, which represents a speedup of 1.6 times; and to 501,473 and 669,596 cycles when using the AVX2 instruction set, which in turn represents a speedup of 2.6 times. The graph also shows the performance improvement between the two processors generations, especially for the vectorial versions: The Skylake processor is 10% faster than the Haswell processor on the original 64-bit version and on the SSE version, while for AVX2 version Skylake is 21% faster than Haswell. These conclusions are based on the average results obtained with the three compilers.

3.2. Vector Rotation Table

Although there is a likely more efficient implementation for Listing 1, it will probably be always inefficient without special hardware support. Instead of trying to optimize further our implementation, we focused on reducing the number of its executions. The word permutation of the vector rotation, which is shown on line 4 of Algorithm 2 and is composed of conditional copies and register shifts, represented almost 40% of the code execution time and 90 of them were calculated in the decoding implementation, one for each parity check matrix index. However, the permutation is done based on the first bits of each index and, using 256-bit registers and considering the 80-bit for the security level, there are only 32 possible permutations of words following the rotation logic.

Considering that, we construct a table of all possible word rotations in the beginning of the decoding process and just query that table instead of calculating the permutations every time. The graph in Figure 2 shows the correlation between the number of word rotations that were calculated and the number of possible rotations for each word size. As can be seen, the pre-calculated table of rotations would not be worth for the original 64-bit, but it is faster for all our optimized versions.

This approach, however, has some obstacles to be used in a constant time implementation. The table access pattern cannot depend on the private key because it would be leak cache-timing information that could be exploited on a side-channel attack [Strenzke et al. 2008]. Thus, every time the implementation needs to access the table

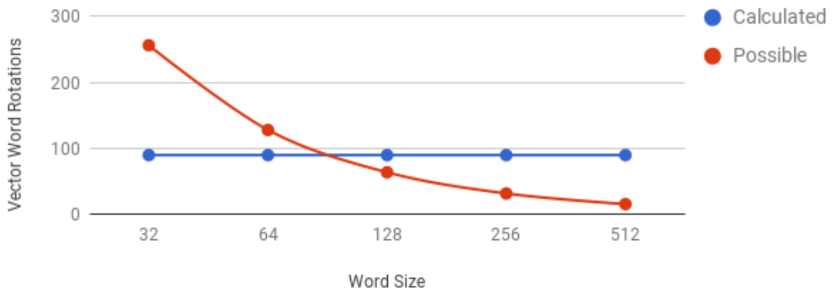

Figure 2. Number of word rotations computed and possible for each implementation

it must iterate over all the table elements conditionally copying each one of them. These extra memory accesses add a great performance penalty and the table of rotation alone became slower than the calculations even on 256-bit registers version.

Despite that, we were able to improve the rotation table by doing a trade-off between the calculation and the table access. Basically, we construct a table with just a small subset of the possible rotations. Then, when a rotation is needed, the implementation iterates over the table, picks the nearest rotation and calculates the pending rotation amount starting from the pre-calculated value. Since the rotation calculation is done based on each bit of the rotation amount, its performance is proportional to the logarithm of the maximum rotation amount. This way, we achieved a 1.19-factor speedup on the AVX version, when comparing to the basic vectorization time, using tables with 3 stored rotations. The number of Skylake cycles, when compiling with GCC, was further reduced from 501,473 cycles to 420,397 cycles and the overall speedup increase from 2.6 times to 3.1 times. The use of the rotation table also drastically reduced the number of iterations necessary to calculate the rotations. This reduction allowed a manual loop unrolling which leads to a 1.17-factor speedup over the best time, bringing the number of Skylake cycles when compiling with GCC down to 358,499 cycles.

All the presented optimization techniques were also applied to QcBits `ref` version, which uses only C code. Table 2 shows the results for all versions. The speedups relatively to the Original Version execution are shown in Figure 3.

Table 2. Final optimization results (in cycles)

Machine	Version	Compiler	Original	SSE	AVX2
Skylake	CLMUL	GCC	1,292,380	574,136	358,499
		CLANG	1,443,992	646,430	377,218
		ICC	1,368,697	878,976	449,620
	REF	GCC	2,097,282	844,992	492,390
		CLANG	2,236,178	944,803	470,578
		ICC	2,221,606	1,360,744	608,560
Haswell	CLMUL	GCC	1,441,220	788,436	529,956
		CLANG	1,610,954	829,896	528,188
		ICC	1,506,562	918,084	555,844
	REF	GCC	2,216,498	1,132,052	679,122
		CLANG	2,391,762	1,205,842	651,032
		ICC	2,337,726	1,306,476	716,208

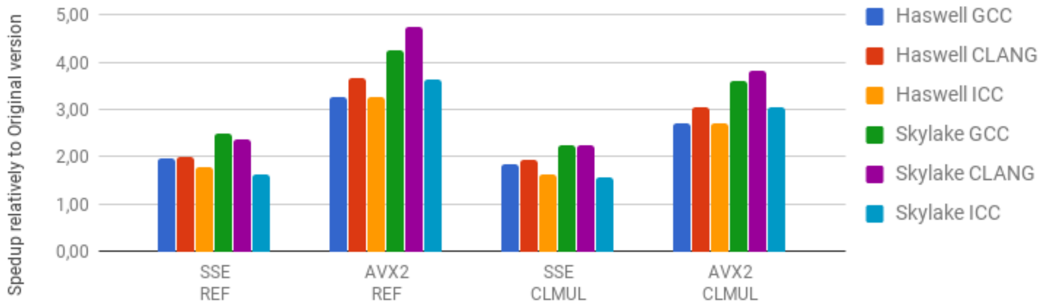

Figure 3. Final speedups achieved with the optimization (relatively to the corresponding Original version execution)

Analyzing the `clmul` version results in the graph, we can note that the results using SSE instructions overcome our initial expectation with a speedup of 2.25 times on Skylake with GCC. This was possible thanks to the rotation table use and the loop unrolling, previously explained. For the version vectorized with AVX2 instructions, however, the speedup is still below 4 times, being 3.6 times with GCC and 3.8 times with CLANG, both on Skylake. It happens mostly because of the absence of a 256-bit `clmul` instruction, which creates the need of the use of 128-bit register instructions in the syndrome calculation. This hurts the performance not only because of the use of small registers but also due to the transition between the instruction set extensions, which is known to be expensive [Lomont 2011].

The `ref` version uses the same constant-time rotation process showed in Algorithm 2 to calculate the matrix multiplications. The only modification in the algorithm is that the $BitSliceAdder$ is replaced by a simple `XOR`. Once this method doesn't rely on `clmul` instruction, the `ref` version could be better optimized, achieving a speedup of 4.75 times with a time of 470,578 cycles on Skylake with CLANG.

3.3. Estimations for possible improvements

As explained in the beginning of this section, the two main hindrances for the vectorization were the absence of vector instructions for register shifts and conditional moves. This last procedure is currently done by the `BLENDV` instruction, which is much more powerful and, hence, expensive than we need for this purpose. Although Fog [Fog 2011] reports a throughput of 1 cycle for this instruction, it is difficult to implement the instruction usage in a sequential way to achieve this time.

In order to estimate the possible gains if these two instructions exist, we experimented with the `clmul` version to suppose their existence. The experiment was done by replacing the `BLENDV` instruction with two addition instructions and the vector shift algorithm by a simple 64-bit lanes shift. This version, of course, doesn't result in the correct output, but it serves as a fair estimation. Testing on Skylake and compiling with GCC, we execute this version in 255.274 cycles, which represents a 1.4 times speedup compared to our best correct version and a total speedup of 5.06 times over the `clmul` version.

4. Power side-channel vulnerability

A simplified snippet of the original implementation code used for the word rotation is shown on Listing 2. As previously discussed, the rotation amount depends directly on the secret key bits and, therefore, must be executed mitigating all side channel leakages. On line 1, a mask is constructed using the variable `sk_bit`, which represents a secret key bit: If the bit is one, then the mask will be all ones, otherwise, the mask will be all zeros. Following this, on line 4, the vector is copied shifted or not depending on this mask.

```
1 mask = 0 - sk_bit;
2 us = 1 << i; // shift amount
3 for (j = 0; j < LEN; j++)
4     w[ j ] = (x[ j + us ] &  mask) ^ (x[ j ] & ~mask);
```

Listing 2: Vulnerable implementation of conditional copy for vector rotation

The problem lies on the fact that the power consumption of setting all bits in a register is perceptibly higher than keeping the register with all its bits zero. An attacker can exploit that fact and discover the secret key through a power measurement of the algorithm execution [Nascimento et al. 2016]. We are able to mitigate this vulnerability by using the instruction `BLENDV`, as shown in Listing 3. This vulnerability used to occur not only in the word rotation, but in all conditional copies implemented in the original version. We fix all of them in the same way and verified that this modification had very little impact on the overall performance ($< 1\%$). The performance results presented in Section 3 already include this modification.

```
mask = _mm_set1_epi8(sk_bit << 7);
us = 1 << i; // shift amount
for (j = 0; j < LEN; j++)
    w[ j ] = _mm_blendv_epi8(x[ j ], x[ j + us ], mask);
```

Listing 3: Secure implementation of conditional copy for vector rotation

5. Related Work

Most of the work related to the QcBits implementation is research on side-channel attacks. Rossi et al. [Rossi et al. 2017] presented a side-channel power-based attack against the syndrome calculation of QcBits. The attack exploited a power-leakage at the store of the rotated code-word (line 4 of Algorithm 2). They also provided a simple countermeasure in order to prevent the attack. We did not apply this countermeasure in this paper since the use of registers greater than 128 bits makes the attack complexity much higher than the target security level. Guo et al. [Guo et al. 2016] presented an attack exploiting a relation between the parity check matrix bits and the decoding failure rate of the algorithm. The attack was named Reaction Attack and is capable of recovering the private key of QcBits in minutes. No effective countermeasure was proposed for this attack yet. Considering implementation work, Hu and Cheung [Hu and Cheung 2017] presented a hardware implementation of QC-MDPC codes partially based on QcBits implementation. Using a Xilinx Virtex-6 FPGA, they achieved a 53% area-time product gain comparing to the previous designs for QC-MDPC codes.

To the best of our knowledge, our paper is the first to present a fully vectorized software optimization of QcBits. The use of vector instructions for cryptographic algorithm optimization, however, is quite common. Chang et al. [Chang et al. 2015] presented an optimized implementation of the RSA algorithm which achieved speedups of 4.3 to 5.9 times using the AVX-512 instruction set, and Hamburg [Hamburg 2012] presented an implementation of Elliptic-Curve Cryptography using SSE and AVX instructions. Considering the code-based cryptography field, specifically, there are also some optimization work using vector instructions. Maurich et al. [Maurich et al. 2015] presented an implementation of QC-MDPC codes using the SSE instruction set which was considered the speed record for these codes before the QcBits publication.

6. Conclusion

In this paper, we presented an optimized implementation of decoding process in QcBits. We vectorized the entire algorithm, inserted a table of pre-computed vector rotations and unrolled the rotation calculation loop for the versions `ref` and `clmul`. In the `ref` version, using the SSE and AVX2 instruction sets, we achieved a maximum speedup of 2.48 and 4.75 times, respectively, while in the `clmul` version we achieved a speedup of 2.23 and 3.6 times when using SSE and AVX2 instructions and compiling with GCC. We also implemented countermeasures for some known side channel vulnerabilities without any significant performance penalty.

Our results clearly demonstrate the algorithm's aptitude for vectorization. The `ref` version, which does not rely on the `clmul` instruction, presented higher gains than the register size increment, showing the great impact of the rotation pre-computation technique. The same occurred with `clmul` version vectorized with SSE instructions. The use of the table could also be much more efficiently implemented if the hardware provided constant-time memory accesses. Besides that, we also demonstrated that some hardware improvements, such as shifting and conditional move instructions for 128-bit and 256-bit registers, can be very useful for the algorithm performance, as shown by our 1.4-factor speedup estimation considering these instructions. A 256-bit version of the `clmul` instruction would also provide significant performance gains.

Considering the current post-quantum cryptography scenario, the code-based cryptography field is just beginning its rise and, considering the latest performance improvements, it is shaping up as one of the most promising candidates for that end. As future work, we intend to implement an AVX-512 version of the decoding process and to optimize the key-pair generation and the encryption process of QcBits. Also, some countermeasure must be developed to mitigate the Reaction Attack.

Acknowledgements

The authors would like to thank Intel Labs and the São Paulo Research Foundation (FAPESP) for supporting this research under grant 14/50704-7.

References

Bernstein, D. J., Lange, T., and Peters, C. (2008). Attacking and defending the mceliece cryptosystem. Cryptology ePrint Archive, Report 2008/318. `http://eprint.iacr.org/2008/318`.

Chang, C., Yao, S., and Yu, D. (2015). Vectorized big integer operations for cryptosystems on the intel mic architecture. In *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, pages 194–203. IEEE.

Chou, T. (2016). Qcbits: constant-time small-key code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 280–300. Springer.

Faugere, J.-C., Otmani, A., Perret, L., and Tillich, J.-P. (2010). Algebraic cryptanalysis of mceliece variants with compact keys. In *Eurocrypt*, pages 279–298. Springer.

Fog, A. (2011). Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*.

Gueron, S. and Kounavis, M. E. (2010). Intel® carry-less multiplication instruction and its usage for computing the gcm mode. *White Paper*.

Guo, Q., Johansson, T., and Stankovski, P. (2016). A key recovery attack on mdpc with cca security using decoding errors. In *Advances in Cryptology–ASIACRYPT 2016*, pages 789–815. Springer.

Hamburg, M. (2012). Fast and compact elliptic-curve cryptography. *IACR Cryptology ePrint Archive*.

Hu, J. and Cheung, R. C. (2017). Area-time efficient computation of niederreiter encryption on qc-mdpc codes for embedded hardware. *IEEE Transactions on Computers*.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*.

Maurich, I. V., Oder, T., and Güneysu, T. (2015). Implementing qc-mdpc mceliece encryption. *ACM Transactions on Embedded Computing Systems (TECS)*.

McEliece, R. J. (1978). A public-key cryptosystem based on algebraic. *Coding Thv*.

Misoczki, R. and Barreto, P. S. (2009). Compact mceliece keys from goppa codes. In *Selected Areas in Cryptography*, pages 376–392. Springer.

Misoczki, R., Tillich, J.-P., Sendrier, N., and Barreto, P. S. (2013). Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 2069–2073. IEEE.

Nascimento, E., Chmielewski, L., Oswald, D., and Schwabe, P. (2016). Attacking embedded ecc implementations through cmov side channels. *IACR Cryptology ePrint Archive*.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*.

Rossi, M., Hamburg, M., Hutter, M., and Marson, M. E. (2017). A side-channel assisted cryptanalytic attack against qcbits. Cryptology ePrint Archive, Report 2017/596. `http://eprint.iacr.org/2017/596`.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*.

Strenzke, F., Tews, E., Molter, H. G., Overbeck, R., and Shoufan, A. (2008). Side channels in the mceliece pkc. In *International Workshop on Post-Quantum Cryptography*, pages 216–229. Springer.