Software Speed Records for Lattice-Based Signatures

Tim Güneysu[1], Tobias Oder[1], Thomas Pöppelmann[1], and Peter Schwabe[2] [⋆]

[1] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
[2] Digital Security Group, Radboud University Nijmegen, The Netherlands

Abstract. Novel public-key cryptosystems beyond RSA and ECC are urgently required to ensure long-term security in the era of quantum computing. The most critical issue on the construction of such cryptosystems is to achieve security *and* practicability at the same time. Recently, lattice-based constructions were proposed that combine both properties, such as the lattice-based digital signature scheme presented at CHES 2012. In this work, we present a first highly-optimized SIMD-based software implementation of that signature scheme targeting Intel's Sandy Bridge and Ivy Bridge microarchitectures. This software computes a signature in only 634988 cycles on average on an Intel Core i5-3210M (Ivy Bridge) processor. Signature verification takes only 45036 cycles. This performance is achieved with full protection against timing attacks.

Keywords: Post-quantum cryptography, lattice-based cryptography, cryptographic signatures, software implementation, AVX, SIMD

1 Introduction

Besides breakthroughs in classical cryptanalysis the potential advent of quantum computers is a serious threat to the established discrete-logarithm problem (DLP) and factoring-based public-key encryption and signature schemes, such as RSA, DSA and elliptic-curve cryptography. Especially when long-term security is required, all DLP or factoring-based schemes are somewhat risky to use. The natural consequence is the need for more diversification and investigation of potential alternative cryptographic systems that resist attacks by quantum computers. Unfortunately, it is challenging to design secure *post-quantum* signature schemes that are efficient in terms of speed and key sizes. Those which are known to be very efficient, such as the lattice-based NTRU-sign [15] have been shown to be easily broken [19]. Multivariate quadratic (MQ) signatures, e.g., Unbalanced Oil and Vinegar (UOV), are fast and compact, but their public keys are huge with around 80 kB and thus less suitable on embedded systems – even with optimizations the keys are still too large (around 8 Kb) [20].

[⋆] This work was supported by the National Institute of Standards and Technology under Grant 60NANB10D004. Permanent ID of this document: `ead67aa537a6de60813845a45505c313`. Date: March 28, 2013

The introduction of special ring-based (ideal) lattices and their theoretical analysis (see, e.g., [18]) provides a new class of signature and encryption schemes with a good balance between key size, signature size, and speed. The speed advantage of ideal lattices over standard lattice constructions usually stems from the applicability of the Number Theoretic Transform (NTT), which allows operations in quasi-linear runtime of $\mathcal{O}(n \log n)$ instead of quadratic complexity. In particular, two implementations of promising lattice-based constructions for encryption [12] and digital signatures [14] were recently presented and demonstrate that such constructions can be efficient in reconfigurable hardware. However, as the proof-of-concept implementation in [12] is based on the generic NTL library [22], it remains still somewhat unclear how these promising schemes perform on high-performance processors that include modern SIMD multimedia extensions such as SSE and AVX.

Contribution. The main contribution of this work is the first optimized software implementation of the lattice-based signature scheme proposed in [14]. It is an aggressively optimized variant of the scheme originally proposed by Lyubashevsky [17] without Gaussian sampling. We use security parameters $p = 8383489, n = 512, k = 2^{14}$ that are assumed to provide an equivalent of about 80 bits of security against attacks by quantum computers and 100 bits of security against classical computers. With these parameters, public keys need only 1536 bytes, private keys need 256 bytes and signatures need 1184 bytes. On one core of an Intel Core i5-3210M processor (Ivy Bridge microarchitecture) running at 2.5 GHz, our software can compute more than 3900 signatures per second or verify more than 55000 signatures per second. To maximize reusability of our results we put the software into the public domain[3]. We will additionally submit our software to the eBACS benchmarking project [4] for public benchmarking.

Outline. In Section 2 we first provide background information on the implemented signature scheme. Our implementation and optimization techniques are described in Section 3 and evaluated and compared to previous work in Section 4. We conclude with future work in Section 5.

2 Signature Scheme Background

In this section we briefly revisit the lattice-based signature scheme implemented in this work. For more detailed information as well as security proofs, please refer to [14, 17].

2.1 Notation

In this section we briefly recall the notation from [14]. We use a similar notation and denote by $\mathcal{R}^{p^n}$ the polynomial ring $\mathbb{Z}[x]_p\langle x^n + 1\rangle$ with integer coefficients in the range $[-\frac{p-1}{2}, \frac{p-1}{2}]$ where n is a power of two. The prime p must satisfy the

[3] The software is available at `http://cryptojedi.org/crypto/\#lattisigns`

congruence relation $p \equiv 1 \pmod{2n}$ to allow us to use the quasi-linear-runtime NTT-based multiplication. For any positive integer k, we denote by $\mathcal{R}_k^{p^n}$ the set of polynomials in $\mathcal{R}^{p^n}$ with coefficients in the range $[-k, k]$. The expression $a \overset{\$}{\leftarrow} D$ denotes the uniformly random sampling of a polynomial a from the set D.

2.2 Definition

According to the description in [14] we have chosen a to be a randomly generated global constant. For the key generation described in Algorithm 1 we therefore basically perform sampling of random values from the domains $\mathcal{R}_1^{p^n}$ followed by a polynomial multiplication with the global constant and an addition. The private key sk consists of the values s_1, s_2 while t is the public key pk. Algorithm 2 signs

Algorithm 1: Key generation algorithm GEN(p, n)

```
   Input: Parameters $p, n$
   Output: $(t)_{pk}$, $(s_1, s_2)_{sk}$
1  $s_1, s_2 \overset{\$}{\leftarrow} \mathcal{R}_1^{p^n}$
2  $t \leftarrow as_1 + s_2$
```

a message m specified by the user. In step 1 two polynomials y_1, y_2 are chosen uniformly at random with coefficients in the range $[-k, k]$. In step 2 a hash function is applied on the higher-order bits of $ay_1 + y_2$ which outputs a polynomial c by interpreting the first 160-bit of the hash output as a sparse polynomial. In step 3 and 4, y_1 and y_2 are used to mask the private key by computing z_1 and z_2. The algorithm only continues if z_1 and z_2 are in the range $[-(k-32), k-32]$ and restarts otherwise. The polynomial z_2 is then compressed into $z_2^{'}$ in step 7 by Compress. This compression is part of the aggressive size reduction of the signature $\sigma = (z_1, z_2^{'}, c)$ since only some portions of z_2 are necessary to maintain the security of the scheme. For the implemented parameter set Compress has a chance of failure of less than two percent which results in the restart of the whole signing process.

The verification algorithm VER as described in Algorithm 3 first ensures that all coefficients of $z_1, z_2^{'}$ are in the range $[-(k-32), k-32]$ and rejects the input otherwise by returning $b = 0$ to indicate an invalid signature. In the next step, $az_1 + z_2^{'} - tc$ is computed, transformed into the higher-order bits and then hashed. If the polynomial c from the signature and the output of the hash match, the signature is valid and the algorithm outputs $b = 1$ to indicate its success.

In Algorithm 4 the transformation of a polynomial into a higher-order representation is described. This algorithm exploits the fact that every polynomial $Y \in \mathcal{R}^{p^n}$ can be written as

$$Y = Y^{(1)}(2(k-32)+1) + Y^{(0)}$$

Algorithm 2: Signing algorithm SIGN(s_1, s_2, m)

```
Input: $s_1, s_2 \in \mathcal{R}_1^{p^n}$, message $m \in \{0,1\}^*$
Output: $z_1, z_2^{'} \in \mathcal{R}_{k-32}^{p^n}$, $c \in \{0,1\}^{160}$
1 $y_1, y_2 \xleftarrow{\$} \mathcal{R}_k^{p^n}$
2 $c \leftarrow \mathrm{H}(\mathsf{Transform}(ay_1 + y_2),\, m)$
3 $z_1 \leftarrow s_1 c + y_1$
4 $z_2 \leftarrow s_2 c + y_2$
5 if $z_1$ or $z_2 \notin \mathcal{R}_{k-32}^{p^n}$ then
6     go to step 1
7 $z_2^{'} \leftarrow \mathsf{Compress}(ay_1 + y_2 - z_2, z_2, p, k-32)$
8 if $z_2^{'} = \perp$ then
9     go to step 1
```

Algorithm 3: Verification algorithm VER$(z_1, z_2^{'}, c, t, m)$

```
Input: $z_1, z_2^{'} \in \mathcal{R}_{k-32}^{p^n}$, $t \in \mathcal{R}^{p^n}$, $c \in \{0,1\}^{160}$, message $m \in \{0,1\}^*$
Output: $b$
1 if $z_1$ or $z_2^{'} \notin \mathcal{R}_{k-32}^{p^n}$ then
2     $b \leftarrow 0$
3 else
4     if $c = H(\mathit{Transform}(az_1 + z_2^{'} - tc),\, m)$ then
5         $b \leftarrow 1$
6     else
7         $b \leftarrow 0$
```

where $Y^{(0)} \in \mathcal{R}_{k-32}^{p^n}$ and thus every coefficient of $Y^{(0)}$ is in the range $[-(k-32), k-32]$. Due to this bijectional relationship, every polynomial Y can be also written as the tuple $(Y^{(1)}, Y^{(0)})$.

Algorithm 5 describes the compression algorithm Compress which takes a polynomial y, a polynomial z with small coefficients and the security parameter k as well as p as input. It is designed to return a polynomial $z^{'}$ that is compacted but still maintains the equality between the higher-order bits of $y+z$ and $y+z^{'}$ so that $(y+z)^{(1)} = (y+z^{'})^{(1)}$. In particular, the parameters of the scheme are chosen in a way that the if-condition specified in step 3 is true only for rare cases. This is important since only values assigned to $z^{'}[i]$ in step 6 to step 12 can be efficiently encoded.

The hash function H maps an arbitrary-length input $\{1,0\}^*$ to a 512-coefficient polynomial with 32 coefficients in $\{-1,1\}$ and all other coefficients zero. The whole process of generating this string and its transformation into a polynomial with the above described character is shown in Algorithm 6. In step 1 the message is concatenated with a binary representation of the polynomial x generated

Algorithm 4: HIGHER-ORDER TRANSFORMATION ALGORITHM TRANSFORM(y, k)

```
Input: $y \in \mathcal{R}^{p^n}$, $k$
Output: $y^{(1)}$
1 for $i$=0 to $n-1$ do
2     $y^{(0)}[i] \leftarrow y[i] \mod (2(k-32)+1)$
3     $y^{(1)}[i] \leftarrow \frac{y[i]-y^{(0)}[i]}{2(k-32)+1}$
4 return $y^{(1)}$
```

by the algorithm BinRep. It takes a polynomial $x \in \mathcal{R}^{p^n}$ as input and outputs a (somehow standardized) binary representation of this polynomial. The 160-bit hash value is processed by partitioning it into 32 blocks of 5 side-by-side bits (beginning with the lowest ones) that each correspond to a particular region in the polynomial c. These bits are $r_4r_3r_2r_1r_0$ where $(r_3r_2r_1r_0)_2$ represents the position in the region interpreted as a 4-bit unsigned integer and the bit r_4 determines if the value of the coefficient is -1 or 1.

2.3 Parameters and Security

Parameters that offer a reasonable security margin of approximately 100 bits of comparable classical symmetric security are $n = 512$, $p = 8383489$, and $k = 2^{14}$This parameter set is the primary target of this work. For some intuition on how these parameters were selected, how the security level has been computed, for a second parameter set and a security proof in the random-oracle model we refer again to [14].

In general, the security of the signature scheme is based on the Decisional Compact Knapsack (DCK$_{p,n}$) problem and the hardness of finding a preimage in the hash function. For solving the DCK problem one has to distinguish between uniform samples from $\mathcal{R}^{p^n} \times \mathcal{R}^{p^n}$ and samples from the distribution $(a, as_1 + s_2)$ with a being chosen uniformly at random from $\mathcal{R}^{p^n}$ and s_1, s_2 being chosen uniformly at random from $\mathcal{R}_1^{p^n}$. In comparison to the Ring-LWE problem [18], where s_1, s_2 are chosen from a Gaussian distribution of a certain range, this just leads to s_1, s_2 with coefficients being either ±1 or zero. Therefore, the DCK problem is an "aggressive" variant of the LWE problem but is not affected by the Arora-Ge algorithm as only one sample is given for the DCK problem and not the required polynomially-many [1]. Note also that extraction of the private key from the public key requires to solve the search variant of the DCK problem. In [14] the hardness of breaking the signature scheme for the implemented parameter set is computed based on the root Hermite factor of 1.0066 and stated to provide roughly 100 bits of security. Finding a preimage in the hash function has classical time complexity of 2^l but is lowered to $2^{l/2}$ by Grover's quantum algorithm [13]. As we use an output bit length of $l = 160$ from the hash function the implemented

Algorithm 5: Compression Algorithm Compress(y, z, p, k)

```
Input: $y \in \mathcal{R}_k^{p^n}$, $z \in \mathcal{R}_{k-32}^{p^n}$, $p$, $k$
Output: $z^{'} \in \mathcal{R}_k^{p^n}$
1  $uncompressed \leftarrow 0$
2  for $i$=0 to $n-1$ do
3      if $|y[i]| > \frac{p-1}{2} - k$ then
4          $z^{'}[i] \leftarrow z[i]$
5          $uncompressed \leftarrow uncompressed + 1$
6      else
7          write $\mathbf{y}[i] = \mathbf{y}[i]^{(1)}(2k+1) + \mathbf{y}[i]^{(0)}$ where $-k \leq \mathbf{y}[i]^{(0)} \leq k$ if
           $y[i]^0 + z[i] > k$ then
8              $z[i]^{'} \leftarrow k$
9          else if $y[i]^0 + z[i] < -k$ then
10             $z[i]^{'} \leftarrow -k$
11         else
12             $z[i]^{'} \leftarrow 0$
13 if $uncompressed \leq \frac{6kn}{p}$ then
14     return $z^{'}$
15 else
16     return $\perp$
```

scheme achieves a security level of roughly 80 bits of security against attacks by a quantum computer.

3 Software Optimization

In this section we show our approach to high-level optimization of algorithms and low-level optimization to make best use of the target micro-architecture.

3.1 High-Level Optimization

In the following we present high-level ideas to speed-up the polynomial multiplication, runtime behavior as well as randomness generation.

Polynomial multiplication. In order to achieve quasi-linear speed in $\mathcal{O}(n \log n)$ when performing the essential polynomial-multiplication operation we use the Fast Fourier Transform (FFT) or more specifically the Number Theoretic Transform (NTT) [21]. The advantages offered by the NTT have recently been shown by a hard- and software implementation of an ideal lattice-based public key cryptosystem [12]. The NTT is defined in a finite field or ring for a given primitive n-th root of unity ω. The generic forward $\mathrm{NTT}_\omega(a)$ of a sequence

Algorithm 6: HASH FUNCTION INVOCATION $\mathrm{H}(x, m)$

```
Input: Polynomial $x \in \mathcal{R}^{p^n}$, message $m \in \{0,1\}^*$, hash function
        $\tilde{H}(\{0,1\}^*) \to \{0,1\}^{160}$
Output: $c \in \mathcal{R}_1^{p^n}$ with at most 32 coefficients being -1 or 1
1 $r \leftarrow \tilde{H}(m||\mathsf{BinRep}(x))$
2 for $i$=0 to $n-1$ do
3     $c[i] = 0$
4 for $i$=0 to 31 do
5     $pos \leftarrow 8 \cdot r_{5i+3} + 4 \cdot r_{5i+2} + 2 \cdot r_{5i+1} + r_{5i}$
6     if $r_{5i+4} = 0$ then
7         $c[i \cdot 16 + pos] \leftarrow -1$
8     else
9         $c[i \cdot 16 + pos] \leftarrow 1$
```

$\{a_0, .., a_{n-1}\}$ to $\{A_0, \ldots, A_{n-1}\}$ with elements in $\mathbb{Z}_p$ and length n is defined as $A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \bmod p,\ i = 0, 1, \ldots, n-1$ with the inverse $\mathrm{NTT}_\omega^{-1}(A)$ just using ω^{-1} instead of ω.

For lattice-based cryptography it is also convenient that most schemes are defined in $\mathbb{Z}_p[\mathbf{x}]/\langle x^n + 1\rangle$ and require reduction modulo $x^n + 1$. As a consequence, let ω be a primitive n-th root of unity in $\mathbb{Z}_p$ and $\psi^2 = \omega$. Then when $a = (a_0, \ldots a_{n-1})$ and $b = (b_0, \ldots b_{n-1})$ are vectors of length n with elements in $\mathbb{Z}_p$ let $d = (d_0, \ldots d_{n-1})$ be the negative wrapped convolution of a and b (thus $d = a * b \mod x^n + 1$). Let $\bar{a}, \bar{b}$ and $\bar{d}$ be defined as $(a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1})$, $(b_0, \psi b_1, \ldots, \psi^{n-1} b_{n-1})$, and $(d_0, \psi d_1, \ldots, \psi^{n-1} d_{n-1})$. It then holds that $\bar{d} = NTT_w^{-1}(NTT_w(\bar{a}) \circ NTT_w(\bar{b}))$ [24], where $\circ$ means componentwise multiplication. This avoids the doubling of the input length of the NTT and also gives us a modular reduction by $x^n + 1$ for free. If parameters are chosen such that n is a power of two and that $p \equiv 1 \bmod 2n$, the NTT exists and the negative wrapped convolution can be implemented efficiently.

In order to achieve high NTT performance, we precompute all constants $\omega^i, \omega^{-i}, \psi^i$ as well as $n^{-1} \cdot \psi^i$ for $i \in 0 \ldots n-1$. The multiplication by n^{-1}, which is necessary in the NTT^{-1} step, is directly performed as we just multiply by $n^{-1} \cdot \psi^{-i}$.

Storing parameters in NTT representation. The polynomial a is used as input to the key-generation algorithm and can be chosen as a global constant. By setting $\tilde{a} = \mathrm{NTT}(a)$ and storing $\tilde{a}$ we just need to perform $\mathrm{NTT}^{-1}(\tilde{a} \circ \mathrm{NTT}(y_1))$, which consists of one forward transform, one point multiplication and one backward transform. This is implemented in the `poly_mul_a` function and is superior to the general-purpose NTT multiplication, which requires three transforms.

Random polynomials. During signature generation we need to generate two polynomials with random coefficients uniformly distributed in $[-k, k]$. To obtain these polynomials, we first generate $4 \cdot (n + 16) = 2112$ random bytes using the Salsa20 stream cipher [2] and a seed from the Linux kernel random-number generator /dev/urandom. We interprete these bytes as an array of $n+16$ unsigned 32-bit integers. To convert one such a 32-bit integer r to a polynomial coefficient c in $[-k, k]$ we first check whether $r \geq (2k+1) \cdot \lfloor 2^{32}/(2k+1) \rfloor$. If it is, we discard this integer and move to the next integer in the array. Otherwise we compute $c = (r \mod (2k+1)) - k$.

The probability that an integer is discarded is $(2^{32} \mod (2k+1))/2^{32}$. For our parameters we have $(2^{32} \mod (2k+1)) = 4$. The probability to discard a randomly chosen 32-bit integer is thus $4/2^{32} = 2^{-30}$. The 16 additional elements in our array (corresponding to one block of Salsa20) make it extremely unlikely that we do not sample enough random elements to set all coefficients of the polynomial. In this highly unlikely case we simply sample another 2112 bytes of randomness.

During key generation we use the same approach to generate polynomials with coefficients in $\{-1, 0, 1\}$. The difference is that we sample bytes instead of 32-bit integers. We again sample one additional block of Salsa20 output, now corresponding to 64 additional elements. A byte is discarded only if its value is 255, the chance to discard a random byte is thus 2^{-8}.

3.2 Low-Level Optimization

The performance of the signature scheme is largely determined by a small set of operations on polynomials with $n = 512$ coefficients over $\mathbb{Z}_p$ where p is a 23-bit prime. This section first describes how we represent polynomials and what implementation techniques we use to accelerate operations on these polynomials.

Representation of polynomials. We represent each 512-coefficient polynomial as an array of 512 double-precision floating-point values. Each such array is aligned on a 32-byte boundary, meaning that the address in memory is divisible by 32. This representation has the advantage that we can use the single-instruction multiple-data (SIMD) instructions of the AVX instruction-set extension in modern Intel and AMD CPUs. These instructions operate on vectors of 4 double-precision floats in 256-bit-wide, so called ymm vector registers. These registers and the corresponding AVX instructions can be found, for example, in the Intel Sandy Bridge, Intel Ivy Bridge, and AMD Bulldozer processors. The following performance analysis focuses on Ivy Bridge processors; Section 4 also reports benchmarks from a Sandy Bridge processor.

Both Sandy Bridge and Ivy Bridge processors can perform one AVX double-precision-vector multiplication and one addition every cycle. This corresponds to 4 multiplications (vmulpd instruction) and 4 additions (vaddpd instruction) of polynomial coefficients each cycle. However, arithmetic cost is not the main bottleneck in our software as loads and stores are often necessary because only 64 polynomial coefficients fit into the 16 available ymm registers. The performance of loads and stores is more complex to determine than arithmetic throughput. In principle, the processor can perform two loads and one store every two cycles. However, this maximal throughput can be reduced by bank conflicts. For details see [10, Section 8.13].

Modular reduction of coefficients. To perform a modular reduction of a coefficient x, we first compute $c = x \cdot \overline{p^{-1}}$, then round c, then multiply c by p and then subtract c from x. The first step uses a precomputed double-precision approximation $\overline{p^{-1}}$ of the inverse of p. When reducing all coefficients of a polynomial, the multiplications and the subtraction are performed on four coefficients in parallel with the `vmulpd` and `vsubpd` AVX instructions, respectively. The rounding is also done on four coefficients in parallel using the `vroundpd` instruction. Note that depending on the rounding mode we can obtain the reduced value of x in different intervals. If we perform a truncation we obtain x in $[0, p-1]$, if we round to the nearest integer we obtain x in $[-((p-1)/2), (p-1)/2]$. We only need rounding to the nearest integer (`vroundpd` with rounding-mode constant `0x08`). Both representations are required at different stages of the computation; `vroundpd` supports choosing the rounding mode.

Lazy reduction. The prime p has 23 bits. A double-precision floating-point value has a 53-bit mantissa and one sign bit. Even the product of two coefficients does not use the whole available precision, so we do not have to perform modular reduction after each addition, subtraction or even multiplication. We can thus make use of the technique known as *lazy reduction*, i.e., of performing reduction modulo p only when necessary.

Optimizing the NTT. The most speed-critical operation for signing is polynomial multiplication and we can thus use the NTT transformation as described above. We start from a standard fast iterative algorithm (see, e.g., [9]) for computing the FFT/NTT and adapt it to the target architecture. The transformation of a polynomial f with coefficients $f_0, \ldots, f_{511}$ to or from NTT representation consist of an initial permutation of the coefficients followed by $\log_2 n = 9$ levels of operations on coefficients. On level 0, pick up f_0 and f_1, multiply f_1 with a constant (a power of ω), add the result to f_0 to obtain the new value of f_0 and subtract the result from f_0 to obtain the new value of f_1. Then pick up f_2 and f_3 and perform the same operations to find the new values for f_2 and f_3 and so on. The following levels work in a similar way except that the distance of pairs of elements that are processed together is different: on level i process elements that are 2^i positions apart. For example, on level 2 pick up and transform f_0 and f_4, then f_1 and f_5 etc. On level 0 we can omit the multiplication by a constant, because the constant is 1.

The obvious bottleneck in this computation are additions (and subtractions): Each level performs 256 additions and 256 subtractions accounting for a total of $9 \cdot 512 = 4608$ additions requiring at least 1152 cycles. In fact the lower bound of cycles is much higher, because after each multiplication by a constant we need to reduce the coefficients modulo p. This takes one `vroundpd` instruction and one subtraction. The `vroundpd` instruction is processed in the same port as additions and subtractions, we thus get a lower bound of $(9 \cdot 512 + 8 \cdot 512)/4 = 2176$ cycles. To get close to this lower bound, we need to make sure that all the additions can be efficiently processed in AVX instructions by minimizing overhead from memory access, multiplications or vector-shuffle instructions.

Starting from level 2, the structure of the algorithm is very friendly for 4-way vector processing: For example, we can load (f_0, f_1, f_2, f_3) into one vector register, load (f_4, f_5, f_6, f_7) in another vector register, load the required constants (c_0, c_1, c_2, c_3) into a third vector register and then use one vector multiplication, one vector addition and one vector subtraction to obtain $(f_0+c_0f_4, f_1+c_1f_5, f_2+c_2f_6, f_3+c_3f_7)$ and $(f_0-c_0f_4, f_1-c_1f_5, f_2-c_2f_6, f_3-c_3f_7)$. However, on levels 0 and 1 the transformations are not that straightforwardly done in vector registers. On level 0 we do the following: Load f_0, f_1, f_2, f_3 into one register; perform vector multiplication of this register with $(1, -1, 1, -1)$ and store the result in another register; perform a `vhaddpd` instruction of these two registers which results exactly in $(f_0 + v_1, f_0 - f_1, f_2 + f_3, f_2 - f_3)$. On level 1 we do the following: Load f_0, f_1, f_2, f_3; multiply with a vector of constants, reduce the result modulo p; use the `vperm2f128` instruction with constant argument `0x01` to obtain $c_2f_2, c_3f_3, c_0f_0, c_1f_1$ in another register and perform vector register multiplication of this register by $(1, 1, -1, -1)$; add the result to (f_0, f_1, f_2, f_3) to obtain the desired $(f_0 + c_2f_2, f_1 + c_1f_1, f_0 - c_2f_2, f_1 - c_3f_3)$.

A remaining bottleneck is memory access. To minimize loads and stores, we merge levels 0,1,2, levels 3,4,5 and levels 6,7,8. The idea is that on one level two pairs of coefficients are interacting; through two levels it is 4-tuples of coefficients that interact and through 3 levels it is 8-tuples of coefficients that interact. On levels 0,1 and 2 we load these 8 coefficients; perform all transformations through the 3 levels and store them again, then proceed to the next 8 coefficients. On higher levels we load 32 coefficients, perform all transformations through 3 levels on them, store them and then proceed to the next 32 coefficients.

In total, one NTT transformation takes 4484 cycles on the Ivy Bridge processor. This includes about 500 cycles for the initial coefficient permutation. We are continuing to investigate the difference between the lower bound on cycles dictated by vector additions and the cycles actually taken by our software.

Addition and subtraction. Addition and subtraction of polynomials simply means loading coefficients, performing double-precision floating-point addition or subtraction, and storing the result coefficient. This is completely parallel, so we do this in 256 vector loads, 128 vector additions or subtractions, and 128 vector stores.

Higher-order transformation. The higher-order transformation described in Algorithm 4 is a nice example of the power of representing polynomial coefficients as double-precision floats: The only operation required is the multiplication by the precomputed value $\overline{(2(k-32)+1)^{-1}}$ (a double-precision approximation of $(2(k-32)+1)^{-1}$) and a subsequent rounding towards the nearest integer. As for the coefficient reduction we perform these computations using the `vmulpd` and `vroundpd` instructions.

4 Performance Analysis and Benchmarks

In this section we analyze the performance of our software and report benchmarks for key generation (`crypto_keypair`), as well as the signing (`crypto_sign`) and verification (`crypt_sign_open`) algorithm. Our software implements the eBATS API [4] for signature software, but we did *not* use SUPERCOP for benchmarking. The reason is that SUPERCOP reports the *median* of multiple runs to filter out benchmarks that are polluted by, for example, an interrupt that occurred during some of the computations. Considering the median of timings when signing would be overly optimistic and cut off legitimate benchmarks of signature generations that took very long because they required many attempts. Therefore, for signing we report the *average* of 100000 signature generations; for key-pair generation, verification and lower-level functions we report the median of 1000 benchmarks. However, we will submit our software to eBACS for public benchmarking and discuss the issue with the editors of eBACS. Note that our software for signing is obviously not running in constant time but the timing variation is independent of secret data; our software is fully protected against timing attacks.

We performed benchmarks on two different machines:

- a machine called `h9ivy` at the University of Illinois at Chicago with an Intel Core i5-3210M CPU (Ivy Bridge) at 2500 MHz and 4 GB of RAM; and
- a machine called `h6sandy` at the University of Illinois at Chicago with an Intel Core i3-2310M CPU (Sandy Bridge) at 2100 MHz and 4 GB of RAM.

All software was compiled with gcc-4.7.2 and compiler flags `-O3 -msse2avx -march=corei7-avx -fomit-frame-pointer`. During the benchmarks Turbo-Boost and hyperthreading were switched off. The performance results for the most important operations are given in Table 1. The message length was 59 bytes for the benchmarking of `crypto_sign` and `crypto_sign_open`.

Table 1. Cycle counts of our software; $n = 512$ and $p = 8383489$.

Operation	**Sandy Bridge cycles**	**Ivy Bridge cycles**
crypto_sign_keypair	33894	31140
crypto_sign	681500	634988
crypto_sign_open	47636	45036
ntt	4480	4484
poly_mul	16052	16096
poly_mul_a	11100	11044
poly_setrandom_maxk	12788	10824
poly_setrandom_max1	6072	5464

Polynomial-multiplication performance. The multiplication of two polynomials (`poly_mul`) takes 16096 cycles on the Ivy Bridge. Out of those, $3 \cdot 4484 = 13452$ cycles are for 3 NTT transformations (`ntt`).

Key-generation performance. Generating a key pair takes 31140 cycles on the Ivy Bridge. Out of those, $2 \cdot 5464 = 10928$ cycles are required to generate two random polynomials (`poly_setrandom_max1`); 11044 cycles are required for a multiplication by the constant system parameter a (`poly_mul_a`); the remaining 9168 cycles are required for one polynomial addition, compression of the two private-key polynomials and packing of the public-key polynomial into a byte array.

Signing performance. Signing takes 634988 cycles on average on the Ivy Bridge. Each signing *attempt* takes 85384 cycles. We need 7 attempts on average, so those attempts account for about $7 \cdot 85384 = 597688$ cycles; the remaining cycles are required for constant overhead for extracting the private key from the byte array, copying the message to the signed message etc. *Some* of the remaining cycles may also be due to some measurements being polluted as explained above.

Out of the 85384 cycles for each signing attempt, $2 \cdot 10824 = 21648$ cycles are required to generate two random polynomials (`poly_setrandom_maxk`); $2 \cdot 16096 = 32192$ cycles are required for two polynomial multiplications; 11084 cycles are required for a multiplication with the system parameter a; the remaining 20460 cycles are required for hashing, the higher order transformation, four polynomial additions, one polynomial subtraction and testing whether the polynomial can be compressed.

Verification performance. Verifying a signature takes 45036 cycles on the Ivy Bridge. Out of those, 16096 cycles are required for a polynomial multiplication; 11084 cycles are required for a multiplication with a; the remaining 17856 cycles are required for hashing, the high-order transformation, a polynomial addition and a polynomial subtraction, decompression of the signature, and unpacking of the public key from a byte array.

Comparison. As we provide the first software implementation of the signature scheme we cannot compare our result to other software implementations. In [14] only a hardware implementation is given which is naturally hard to compare to. For different types of FPGAs and parallelism, an implementation of sign/verify of 931/998 (Spartan-6 LX16) up to 12627/14580 (Virtex-6 LX130) messages/signatures per second is reported. However, the architecture is quite different; in particular it uses a configurable number of high-clock-frequency schoolbook multipliers instead of an NTT multiplier. The explanation for the low verification performance on the FPGA, compared with the software implementation, is that only one such multiplier is used in the verification engine.

Another target for comparison is a recently reported implementation of an ideal lattice-based encryption system in soft- and hardware [12]. In software, the necessary polynomial arithmetic relies on Shoup's NTL library [22]. Measurements confirmed that our basic arithmetic is faster than their prototype implementation (although their parameters are smaller) as we can rely on AVX, a hand-crafted NTT implementation and optimized modular reduction.

Various other implementations of post-quantum signature schemes have been described in the literature and many of them have been submitted to eBACS [4]. In Table 2 we compare our software in terms of security, speed, key sizes and signature size to the Rainbow, TTS, and C^* (pFLASH) software presented in [8], and the MQQ-Sig software presented in [11]. The cycle counts of these implementations are obtained from the eBACS website and have been measured on the same Intel Ivy Bridge machine that we used for benchmarking (`h9ivy`). We reference these implementations by their names in eBACS (in `typewriter` font) and their corresponding paper. For most of these multivariate schemes, the signing performance is much better, verification performance is somewhat better, but they suffer from excessive public-key sizes.

We furthermore compare to software described in the literature that has not been submitted to eBACS, specifically the implementation of the parallel-CFS code-based signature scheme presented in [16], the implementation of the treeless signature scheme TSS12 presented in [23], and the implementation of the hash-based signature scheme XMSS [6]. For those implementations we give the performance numbers from the respective paper and indicate the CPU used for benchmarking. Parallel-CFS not only has much larger keys, signing is also several orders of magnitude slower than with the lattice-based signature software presented in this paper. However, we expect that verification with parallel-CFS is very fast, but [16] does not give performance numbers for verification. The TSS software is using the scheme originally proposed in [17]. It makes an interesting target for comparison as it is similar to our scheme but relies on weaker assumptions. However, the software is much slower for both signing and verification. Hash-based signature schemes are also an interesting post-quantum signature alternative due to their well understood security properties and relatively small keys. However, the `XMSS` software presented in [6] is still an order of magnitude slower than our implementation and produces considerably larger signatures.

Finally we include two non-post-quantum signature schemes in the comparison in Table 2. First, the Ed25519 elliptic-curve signature scheme [3] and second, RSA-2048 signatures based on the OpenSSL implementation (`ronald2048`). Comparing to those schemes shows that our implementation and also most of the multivariate-signature software can even be faster or at least quite comparable to established schemes in terms of performance. However, the key and signature sizes of those two non-post-quantum signature are not beaten by any post-quantum proposal, yet.

Other lattice-based signature schemes that have a security reduction in the standard model are given in [7] and [5]. However, those papers do not give concrete parameters, security estimates or describe an implementation.

5 Future Work

As the initial implementation work has been carried out it is now necessary in future work to evaluate the security claims of the scheme by careful cryptanalysis and development of potential attacks. Especially, as the implemented scheme relaxes some assumptions that are required for connection to worst-case lattice problems more confidence is needed for real world usage. Other future work is

Table 2. Comparison of different post-quantum signature software; **pk** stands for public key; **sk** stands for private key. The sizes are given in bytes. All software was benchmarked on `h9ivy` if not indicated otherwise.

Software	**Security**	**Cycles**		**Sizes**	
This work	100 bits	**sign:**	634988	**pk:**	1536
		verify:	45036	**sk:**	256
				sig:	1184
`mqqsig160` [11]	80 bits	**sign:**	1996	**pk:**	206112
		verify:	33220	**sk:**	401
				sig:	20
`mqqsig192` [11]	96 bits	**sign:**	3596	**pk:**	333540
		verify:	63488	**sk:**	465
				sig:	24
`mqqsig224` [11]	112 bits	**sign:**	3836	**pk:**	529242
		verify:	65988	**sk:**	529
				sig:	28
`mqqsig256` [11]	128 bits	**sign:**	4560	**pk:**	789552
		verify:	87904	**sk:**	593
				sig:	32
`rainbow5640` [8]	80 bits	**sign:**	53872	**pk:**	44160
		verify:	34808	**sk:**	86240
				sig:	37
`rainbowbinary16242020` [8]	80 bits	**sign:**	29364	**pk:**	102912
		verify:	17900	**sk:**	94384
				sig:	40
`rainbowbinary256181212` [8]	80 bits	**sign:**	33396	**pk:**	30240
		verify:	27456	**sk:**	23408
				sig:	42
`pflash1` [8]	80 bits	**sign:**	1473364	**pk:**	72124
		verify:	286168	**sk:**	5550
				sig:	37
`tts6440` [8]	80 bits	**sign:**	33728	**pk:**	57600
		verify:	49248	**sk:**	16608
				sig:	43
Parallel-CFS [16]	80 bits	**sign:**	4200000000[a]	**pk:**	20968300
$(20, 8, 10, 3)$		**verify:**	-	**sk:**	4194300
				sig:	75
TSS12 [23]	80 bits	**sign:**	93633000[b]	**pk:**	13087
$(n = 512)$		**verify:**	13064000[b]	**sk:**	13240
				sig:	8294
XMSS [6]	82 bits	**sign:**	7261100[c]	**pk:**	912
$(H = 20, w = 4, \text{AES-128})$		**verify:**	556600[c]	**sk:**	19
				sig:	2451
`ed25519` [3]	128 bits	**sign:**	67564	**pk:**	32
		verify:	209328	**sk:**	64
				sig:	64
`ronald2048`	112 bits	**sign:**	5768360	**pk:**	256
(RSA-2048 based on		**verify:**	77032	**sk:**	2048
OpenSSL)				**sig:**	256

[a] Benchmarked on an Intel Xeon W3670 (3.20 GHz)
[b] Benchmarked on an AMD Opteron 8356 (2.3 GHz)
[c] Benchmarked on an Intel i5-M540 (2.53 GHz)

the investigation of efficiency on more constrained devices like ARM (which, in some versions, also feature a SIMD unit) or even low-cost 8-bit processors.

Acknowledgments

We would like to thank Michael Schneider, Vadim Lyubashevsky, and the anonymous reviewers for their helpful comments.

References

1. Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.
2. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs – The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008.
3. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.
4. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. `http://bench.cr.yp.to` (accessed 2013-01-25).
5. Xavier Boyen. Lattice mixing and vanishing trapdoors: A framework for fully secure short signatures and more. In Phong Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 499–517. Springer, 2010.
6. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.
7. David Cash, Dennis Hofheinz, Eike Kiltz, and Chris Peikert. Bonsai trees, or how to delegate a lattice basis. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 523–552. Springer, 2010.
8. Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2009.
9. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
10. Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2010. `http://www.agner.org/optimize/microarchitecture.pdf` (version 2012-02-29).
11. Danilo Gligoroski, Rune Steinsmo Ødegård, Rune Erlend Jensen, Ludovic Perret, Jean-Charles Faugère, Svein Johan Knapskog, and Smile Markovski. MQQ-SIG – an ultra-fast and provably CMA resistant digital signature scheme. In Liqun Chen,

Moti Yung, and Liehuang Zhu, editors, *Trusted Systems*, volume 7222 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2011.

12. Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2012.
13. Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996.
14. Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 530–547. Springer, 2012.
15. Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSIGN: Digital signatures using the NTRU lattice. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2003.
16. Grégory Landais and Nicolas Sendrier. Implementing CFS. In Steven Galbraith and Mridul Nandi, editors, *Progress in Cryptology – INDOCRYPT 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2012.
17. Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.
18. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.
19. Phong Q. Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 271–288. Springer, 2006.
20. Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. Small public keys and fast verification for multivariate quadratic public key systems. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 475–490. Springer, 2011.
21. John M. Pollard. The Fast Fourier Transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.
22. Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/` (accessed 2013-03-18).
23. Patrick Weiden, Andreas Hülsing, Daniel Cabarcas, and Johannes Buchmann. Instantiating treeless signature schemes. IACR Crptology ePrint archive report 2013/065, 2013. `http://eprint.iacr.org/2013/065`.
24. Franz Winkler. *Polynomial Algorithms in Computer Algebra (Texts and Monographs in Symbolic Computation)*. Springer, 1 edition, 1996.