New software speed records for cryptographic pairings

Michael Naehrig[1], Ruben Niederhagen[2,3], and Peter Schwabe[3] *

[1] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
mnaehrig@microsoft.com
[2] Department of Electrical Engineering
National Taiwan University, 1 Section 4 Roosevelt Road, Taipei 106-70, Taiwan
ruben@polycephaly.org
[3] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
peter@cryptojedi.org

Abstract. This paper presents new software speed records for the computation of cryptographic pairings. More specifically, we present details of an implementation which computes the optimal ate pairing on a 257-bit Barreto-Naehrig curve in only 4,470,408 cycles on one core of an Intel Core 2 Quad Q6600 processor.
This speed is achieved by combining 1.) state-of-the-art high-level optimization techniques, 2.) a new representation of elements in the underlying finite fields which makes use of the special modulus arising from the Barreto-Naehrig curve construction, and 3.) implementing arithmetic in this representation using the double-precision floating-point SIMD instructions of the AMD64 architecture.

Keywords: Pairings, Barreto-Naehrig curves, ate pairing, AMD64 architecture, modular arithmetic, SIMD floating-point instructions

1 Introduction

The use of pairings in cryptography has enabled practical realizations of numerous protocols. The implementation of such protocols demands the ability to efficiently compute the pairing while guaranteeing a required level of security. Most cryptographic pairings are derived from the Tate pairing on elliptic curves.

Since the introduction of Miller's algorithm [26, 27] for computing pairings on elliptic curves, a lot of research has been devoted to finding the most efficient Tate-pairing variants for different security levels by constructing suitable pairing-friendly elliptic curves [18] and by making specific choices for the groups and parameters involved in the computation [7, 8]. Several variants of the Tate pairing

* This work has been supported in part by the European Commission through the ICT Programme under Contract ICT–2007–216499 CACE, and through the ICT Programme under Contract ICT-2007-216646 ECRYPT II. Permanent ID of this document: `4d8d6cd8dc32f9524bb84bbe9c148076`. Date: May 28, 2010

have been proposed like the eta, ate and twisted ate pairings [6, 23], the R-ate [24] and optimal ate pairings [32] (see also [5]), often increasing the computational efficiency over that of their predecessors. Overall, these improvements have led to a remarkable increase in the efficiency of pairing-based protocols. Still, new protocols including the computation of a large number of pairings or pairing products [13, 21] demand even faster pairings.

Pairing-based protocols involve elliptic-curve point groups as well as subgroups of the multiplicative group of a finite field. To achieve most efficient implementations, it is desirable to choose parameters such that the discrete logarithm problems in all groups have roughly the same difficulty.

At the 128-bit security level, a nearly optimal choice for a pairing-friendly curve is a Barreto-Naehrig (BN) curve [9] over a prime field of size roughly 256 bits with embedding degree $k = 12$. According to [4], such a curve achieves 128 bits of security, while according to [15] this is 124 bits. This paper describes a constant-time implementation of an optimal ate pairing on a BN curve over a prime field $\mathbb{F}_p$ of size 257 bits. The prime p is given by the BN polynomial parametrization $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$, where $u = v^3$ and $v = 1966080$. The curve equation is $E : y^2 = x^3 + 17$.

We are the first to propose a software pairing implementation exploiting the polynomial parametrization of the prime p to speed up the underlying field arithmetic. For most pairing-friendly curves, the primes defining the base field are constructed using polynomial parametrizations. These parametrizations have been used to speed up the final exponentiation [31], but so far have not been successfully exploited for field arithmetic in software. Nevertheless, Fan, Vercauteren, and Verbauwhede [16] use the polynomial shape of the prime p to achieve computational speedups in hardware.

To maximize reusability of our results we put all software[4] described in this paper into the public domain.

Comparison to previous work. There exist several descriptions and benchmarks of software implementations of cryptographic pairings. Implementations targeting the 128-bit security level usually use 256-bit BN curves.

The software presented in [22] takes 10,000,000 cycles to compute the R-ate pairing over a 256-bit BN curve on one core of an Intel Core 2 processor; the same computation on one core of an AMD Opteron processor also takes 10,000,000 cycles. Unpublished benchmarks of a newer version of that software (included in the Miracl library [25]) are claimed to take 7,850,000 cycles on an Intel Core 2 Duo T5500 processor [30]. The software presented in [29] takes 29,650,000 cycles to compute the ate pairing over a 256-bit BN curve on one core of a Core 2 Duo processor. Software presented in [19] takes 23,319,673 cycles to compute the ate pairing over a 256-bit BN curve on one core of an Intel Core 2 Duo processor; another implementation described in the same paper takes 14,429,439 to compute the ate pairing on two cores of an Intel Core 2 Duo processor.

The software presented in this paper computes the optimal ate pairing in $4,470,408$ cycles on one core of an Intel Core 2 Quad Q6600 processor and is

[4] Available at `http://www.cryptojedi.org/crypto/#dclxvi`.

thus more than twice as fast as the fastest previously published result and more than 40 percent faster than other unpublished results we are aware of. We are not aware of any other software implementation of a cryptographic pairing at the 128-bit security level that achieves speeds of 7,850,000 cycles or faster on any AMD64 processor.

Organization of the paper. In Section 2 we give a short review of the optimal ate pairing for BN curves. Section 3 collects state-of-the-art high-level optimization techniques for the computation of cryptographic pairings on BN curves from the literature as we use them in our software. Section 4 describes our new approach to represent elements of the underlying finite field $\mathbb{F}_p$ and algorithms to perform arithmetic using this representation in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. In Section 5 we explain how we use the double-precision floating-point SIMD instructions of the AMD64 instruction set (SSE, SSE2, SSE3) to efficiently implement these algorithms. Section 6 gives benchmarking results of our software on different microarchitectures.

2 An optimal ate pairing over Barreto-Naehrig curves

For a Barreto-Naehrig (BN) curve, the most efficient pairings are the R-ate pairing [24] and optimal ate pairings [32]. In this section, we provide the basic background and notation, and describe the algorithm for an optimal ate pairing that is used in our implementation.

Let $E : y^2 = x^3 + b$ be a BN curve over the prime field $\mathbb{F}_p$. This means that there is a $u \in \mathbb{Z}$ such that both p and n, given by

$$\begin{aligned} p &= p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1, \\ n &= n(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1, \end{aligned}$$

are prime. The number of $\mathbb{F}_p$-rational points on E is $\#E(\mathbb{F}_p) = n$, and E has embedding degree $k = 12$ with respect to n. We denote by $\mathcal{O}$ the point at infinity, i.e. the neutral element of the group operation on E. For $m \in \mathbb{Z}$, we write $[m]$ for the multiplication-by-m map on E.

Let ϕ_p be the p-power Frobenius endomorphism on E and $E[n]$ the n-torsion subgroup of E. We define $G_1 = E[n] \cap \ker(\phi_p - [1]) = E(\mathbb{F}_p)$, $G_2 = E[n] \cap \ker(\phi_p - [p]) \subseteq E(\mathbb{F}_{p^{12}})[n]$, $G_3 = \mu_n$, where $\mu_n \subset \mathbb{F}_{p^{12}}^*$ is the group of n-th roots of unity.

An optimal ate pairing on E is given in [32] as

$$a_{\text{opt}} : G_2 \times G_1 \to G_3,\ (Q, P) \mapsto (f_{6u+2,Q}(P) \cdot g_{6u+2,Q}(P))^{(p^{12}-1)/n},$$

where $g_{6u+2,Q}(P) = l_{Q_3,-Q_2}(P) \cdot l_{-Q_2+Q_3,Q_1}(P) \cdot l_{Q_1-Q_2+Q_3,[6u+2]Q}(P)$ with $Q_1 = \phi_p(Q)$, $Q_2 = \phi_p^2(Q)$, and $Q_3 = \phi_p^3(Q)$. The value $l_{R,S}(P) \in \mathbb{F}_{p^{12}}$ is the function of the line through the points R and S on the curve, evaluated at P.

There is no need to compute Q_3. Instead, $g_{6u+2,Q}(P)$ can be replaced by

$$h_{6u+2,Q}(P) = l_{[6u+2]Q,Q_1}(P) \cdot l_{[6u+2]Q+Q_1,-Q_2}(P).$$

The reason for this is that for BN curves $Q_1 - Q_2 + Q_3 + [6u+2]Q = \mathcal{O}$, which can be easily derived from Lemma 2.17 in [28]. By writing down the divisors of the functions $g_{6u+2,Q}$ and $h_{6u+2,Q}$, it can be seen that they only differ by vertical line functions. When evaluated at P, such line functions produce values in proper subfields of $\mathbb{F}_{p^{12}}$ that are mapped to 1 by the final exponentiation.

Algorithm 1 Optimal ate pairing on BN curves for $u > 0$

```
Input: $P \in G_1$, $Q \in G_2$, $m_{\mathrm{opt}} = 6u + 2 = (1, m_{s-1}, \ldots, m_0)_2$.
Output: $a_{\mathrm{opt}}(Q, P)$.
 1: $R \leftarrow Q$, $f \leftarrow 1$
 2: for ($i \leftarrow s-1$; $i \geq 0$; $i--$) do
 3:     $f \leftarrow f^2 \cdot l_{R,R}(P)$, $R \leftarrow [2]R$
 4:     if ($m_i = 1$) then
 5:         $f \leftarrow f \cdot l_{R,Q}(P)$, $R \leftarrow R + Q$
 6:     end if
 7: end for
 8: $Q_1 = \phi_p(Q)$, $Q_2 = \phi_{p^2}(Q)$
 9: $f \leftarrow f \cdot l_{R,Q_1}(P)$, $R \leftarrow R + Q_1$
10: $f \leftarrow f \cdot l_{R,-Q_2}(P)$, $R \leftarrow R - Q_2$
11: $f \leftarrow f^{p^6-1}$
12: $f \leftarrow f^{p^2+1}$
13: $f \leftarrow f^{(p^4-p^2+1)/n}$
14: return $f$
```

Algorithm 1 shows how $a_{\mathrm{opt}}(Q, P)$ can be computed in the case $u > 0$. Lines 2 to 7 are called the Miller loop. It contains doubling steps in Line 3 and addition steps in Line 5. The value $h_{6u+2,Q}(P)$ is multiplied to the result of the Miller loop in Lines 8 to 10 by two addition steps. Lines 11 to 13 together carry out the final exponentiation to the power $(p^{12} - 1)/n$, where Lines 11 and 12 comprise its easy part. It can be done by applying the p^6-power Frobenius automorphism on $\mathbb{F}_{p^{12}}$, a single inversion and a multiplication in $\mathbb{F}_{p^{12}}$. Line 13 represents the hard part of the final exponentiation.

As usual [9, 23, 14], we use a sextic twist $E' : y^2 = x^3 + b/\xi$ defined over $\mathbb{F}_{p^2}$ to represent the points in G_2 by points on the twist using the twist isomorphism $\psi : E' \to E,\ (x', y') \mapsto (\omega^2 x', \omega^3 y')$. The element $\xi \in \mathbb{F}_{p^2}$ (neither a cube nor a square in $\mathbb{F}_{p^2}$) is chosen such that the twist has the right order, i.e. it holds $n \mid \#E'(\mathbb{F}_{p^2})$. The field $\mathbb{F}_{p^{12}}$ is generated over $\mathbb{F}_{p^2}$ by ω via the irreducible polynomial $X^6 - \xi$, i.e. $\omega^6 = \xi$.

The map ψ induces a group isomorphism between $G_2' = E'(\mathbb{F}_{p^2})[n]$ and G_2. So, all points $R \in G_2$ should be seen as being represented by a corresponding point $R' \in G_2'$, i.e. $R = \psi(R')$. All curve arithmetic is done on the twist and intermediate results can be kept in their representation on E'. This means that all curve arithmetic requires $\mathbb{F}_{p^2}$-arithmetic only. Arithmetic in $\mathbb{F}_{p^{12}}$ is also based on arithmetic in $\mathbb{F}_{p^2}$. Overall, there are no $\mathbb{F}_p$ computations other than those involved in $\mathbb{F}_{p^2}$ computations during the optimal-ate-pairing algorithm. Thus an improvement of $\mathbb{F}_{p^2}$-arithmetic–even without improving $\mathbb{F}_p$-arithmetic–leads to an improvement of all parts of the computation.

3 High-level techniques: field extensions, Miller loop and final exponentiation

In this section, we describe the high-level structure of our implementation. We use state-of-the-art optimization techniques from the literature for this implementation level. Our focus is on the construction of the higher degree field extensions such as $\mathbb{F}_{p^6}$ and $\mathbb{F}_{p^{12}}$, the Miller loop to compute $f_{6u+2,Q}(P)$, the function value $h_{6u+2,Q}(P)$ and the structure of the final exponentiation.

The construction of field extensions and the efficiency of the optimal ate pairing depend on the chosen curve parameters. Since our new base-field representation needs the parameter u to be a third power $u = v^3$, we are strongly restricted in the choice of curves. Field multiplications and squarings in $\mathbb{F}_{p^{12}}$ are very expensive, so another important condition on u is that $6u + 2$ should have a Hamming weight as low as possible, to save as many addition steps during the Miller loop as possible.

Our specific choice here is $u = v^3$ with $v = 1966080$. This value provided the lowest Hamming weight for $6u + 2$ among all possibilities for u a third power that gave primes p of 256 or 257 bits. All ideas in this paper also work for other choices of v, but possibly lead to less efficient implementations.

3.1 Field extensions

The above choice for u implies $p \equiv 1 \pmod 4$ which unfortunately means that the field extension $\mathbb{F}_{p^2}$ can not be constructed using $\sqrt{-1}$. Instead we use $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, where $i^2 = -7$. The value ξ to construct the twist and higher-degree extensions is $\xi = i + 6$.

On top of the quadratic extension we build the field $\mathbb{F}_{p^{12}}$ as a tower, first $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}(\tau)$ with $\tau^3 = \xi$ and then $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}(\omega)$ with $\omega^2 = \tau$. This is the same construction as in [14] and we follow [14] in implementing the extension field arithmetic.

3.2 Miller loop

The value $6u + 2$ determines the number of doubling and addition steps in the Miller loop of the optimal ate pairing. The number of doubling steps is 65. The Hamming weight of $6u + 2$ is 9, so there are 8 addition steps. Throughout the pairing computation we use the group G_2' to represent points in G_2. We use Jacobian coordinates for the curve arithmetic in G_2'. In particular, for the doubling and addition steps, we use the formulas given by Arène et al. in [3]. The points in G_1, at which line functions are evaluated, are kept in affine coordinates.

The multiplication of the intermediate variable f with the line function values in the Miller loop is done via a special multiplication function exploiting the fact that line function values are sparse elements of $\mathbb{F}_{p^{12}}$, where only half of the coefficients over $\mathbb{F}_{p^2}$ are different from zero.

After the Miller loop, the points Q_1 and Q_2 are computed by applying the p-power and the p^2-power Frobenius endomorphisms. We do two final addition steps with Q_1 and $-Q_2$, respectively, to multiply the result of the Miller loop by the function value $h_{6u+2,Q}(P)$.

3.3 Final exponentiation

The final exponentiation in our implementation is done as indicated in Lines 11 to 13 of Algorithm 1. It is divided into the easy part (Lines 11, 12) and the hard part (Line 13). The easy part has low computational costs compared to the hard part. Raising the element to the power $p^6 - 1$ is simply a conjugation in the extension $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$ and a single division in $\mathbb{F}_{p^{12}}$. The exponentiation by $p^2 + 1$ is done by applying the p^2-power Frobenius automorphism and one multiplication.

For the hard part, we use the method proposed by Scott et al. in [31]. The main advantage here is that the exponentiation is essentially split into three exponentiations by the sparse exponent u. For our choice the Hamming weight of u is 8. The final result is then obtained by applying the Frobenius automorphism and by using the polynomial representation in u of the fixed exponent $(p^4 - p^2 + 1)/n$.

Note that after the easy part of the final exponentiation, the resulting element in $\mathbb{F}_{p^{12}}$ lies in the cyclotomic subgroup of $\mathbb{F}^*_{p^{12}}$, i.e. the subgroup of order $\Phi_{12}(p) = p^4 - p^2 + 1$, where Φ_{12} is the 12-th cyclotomic polynomial. Granger and Scott [20] recently showed how to exploit this fact to obtain very efficient squaring formulas for such elements. We use these formulas during the hard part of the final exponentiation.

4 Mid-level techniques: arithmetic in $\mathbb{F}_{p^2}$ and $\mathbb{F}_p$

This section explains the new approach for representing integers modulo p where p is given by the BN polynomial $36u^4 + 36u^3 + 24u^2 + 6u + 1$. Inspired by Bernstein's implementation of Curve25519 [12], we suggest to split such an integer into 12 coefficients each of which will be stored in a double-precision floating-point variable in the software implementation. We now give the details of our approach.

4.1 Representing base field elements

Elements in the base field $\mathbb{F}_p$ are integers modulo the prime $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ for some $u \in \mathbb{Z}$. The technique in this section does not depend on the specific fixed value for u that we chose in our implementation. We only need to make the assumption that there exists an integer $v \in \mathbb{Z}$ with $u = v^3$. Furthermore, let $\delta = \sqrt[6]{6}$. Then we have $(\delta v x)^3 = \sqrt{6}ux^3$.

We represent integers by polynomials in the ring

$$R = \mathbb{Z}[x] \cap \overline{\mathbb{Z}}[\delta v x],$$

where $\overline{\mathbb{Z}}$ denotes the ring of algebraic integers in $\mathbb{C}$. Note that the ring homomorphism $R \mapsto \mathbb{Z},\ F \mapsto f = F(1)$ is surjective and thus we may represent an integer f by any polynomial F in the preimage of f under the above map. The product of two integers can be computed by multiplying the corresponding polynomials in R and evaluating the product at 1.

Since δ is an algebraic integer, we see that the polynomial

$$\begin{aligned} P &= 36u^4x^{12} + 36u^3x^9 + 24u^2x^6 + 6ux^3 + 1 \\ &= (\delta v x)^{12} + \delta^3(\delta v x)^9 + 4(\delta v x)^6 + \delta^3(\delta v x)^3 + 1 \end{aligned}$$

is an element of R representing the prime p.

Let $\alpha = \delta v x$. Any integer f can be represented by a polynomial $F \in R$ with $F(1) = f$ of the following form:

$$\begin{aligned} F &= f_0 + f_1\delta^5\alpha + f_2\delta^4\alpha^2 + f_3\delta^3\alpha^3 + f_4\delta^2\alpha^4 + f_5\delta\alpha^5 \\ &\quad + f_6\alpha^6 + f_7\delta^5\alpha^7 + f_8\delta^4\alpha^8 + f_9\delta^3\alpha^9 + f_{10}\delta^2\alpha^{10} + f_{11}\delta\alpha^{11} \\ &= f_0 + f_1 \cdot 6(vx) + f_2 \cdot 6(vx)^2 + f_3 \cdot 6(vx)^3 + f_4 \cdot 6(vx)^4 \\ &\quad + f_5 \cdot 6(vx)^5 + f_6 \cdot 6(vx)^6 + f_7 \cdot 36(vx)^7 + f_8 \cdot 36(vx)^8 \\ &\quad + f_9 \cdot 36(vx)^9 + f_{10} \cdot 36(vx)^{10} + f_{11} \cdot 36(vx)^{11}, \end{aligned}$$

where $f_i \in \mathbb{Z}$ for all i. The integer f corresponds to the vector of coefficients $(f_0, f_1, \dots, f_{11})$ of F.

4.2 Multiplication modulo p

Multiplication modulo p in the new representation is done in two stages, first a polynomial multiplication of the two polynomials representing the integers and second a reduction step.

Let $f, g \in \mathbb{Z}$ be two integers with corresponding polynomials $F, G \in R$ and coefficient vectors $(f_0, f_1, \dots, f_{11})$ and $(g_0, g_1, \dots, g_{11})$. The product $H = FG$ then has coefficient vector $(h_0, h_1, \dots, h_{22})$ and has the form $H = h_0 + h_1\delta^5\alpha + \dots + h_{21}\delta^2\alpha^{21} + h_{22}\delta\alpha^{22}$.

We next represent the result of the multiplication by a polynomial of degree 11 which has 12 coefficients. For the degree-reduction, we use the polynomial P representing the BN prime p. Reducing polynomials modulo P corresponds to reducing the corresponding integers modulo p. We have $P = \alpha^{12} + \delta^3\alpha^9 + 4\alpha^6 + \delta^3\alpha^3 + 1$, thus we can use the equation

$$\alpha^{12} = -\delta^3\alpha^9 - 4\alpha^6 - \delta^3\alpha^3 - 1$$

to reduce the degree of H. The degree reduction is given in Algorithm 2.

Algorithm 2 Degree reduction after polynomial multiplication

```
Input: Coefficient vector $(h_0, h_1, \dots, h_{22}) \in \mathbb{Z}^{23}$ of $H \in R$ with $H(1) = h$.
Output: Reduced coefficient vector $(h'_0, h'_1, \dots, h'_{11})$ of $H'$ with $H'(1) = h$.
 1: $h'_0 \leftarrow h_0 - h_{12} + 6h_{15} - 2h_{18} - 6h_{21}$
 2: $h'_1 \leftarrow h_1 - h_{13} + h_{16} - 2h_{19} - h_{22}$
 3: $h'_2 \leftarrow h_2 - h_{14} + h_{17} - 2h_{20}$
 4: $h'_3 \leftarrow h_3 - h_{12} + 5h_{15} - h_{18} - 8h_{21}$
 5: $h'_4 \leftarrow h_4 - 6h_{13} + 5h_{16} - 6h_{19} - 8h_{22}$
 6: $h'_5 \leftarrow h_5 - 6h_{14} + 5h_{17} - 6h_{20}$
 7: $h'_6 \leftarrow h_6 - 4h_{12} + 18h_{15} - 3h_{18} - 30h_{21}$
 8: $h'_7 \leftarrow h_7 - 4h_{13} + 3h_{16} - 3h_{19} - 5h_{22}$
 9: $h'_8 \leftarrow h_8 - 4h_{14} + 3h_{17} - 3h_{20}$
10: $h'_9 \leftarrow h_9 - h_{12} + 2h_{15} + h_{18} - 9h_{21}$
11: $h'_{10} \leftarrow h_{10} - 6h_{13} + 2h_{16} + 6h_{19} - 9h_{22}$
12: $h'_{11} \leftarrow h_{11} - 6h_{14} + 2h_{17} + 6h_{20}$
13: return $(h'_0, h'_1, \dots, h'_{11})$.
```

Polynomial multiplication and degree reduction make the coefficients grow in their absolute value. Whenever the coefficients get too large we need to do a coefficient reduction. For that we use Algorithm 3. We address the relevant bounds on the coefficients and how to guarantee them in Section 5.1. After the reduction, we have

$$h_0, h_6 \in [-|3v|, |3v|), \; h_1, h_3, h_4, h_7, h_9, h_{10} \in [-|v/2|, |v/2|),$$

the coefficients h_2, h_5, h_8, h_{11} may have an absolute value only slightly larger than $|v/2|$. The function rnd in Algorithm 3 denotes rounding to the nearest integer.

5 Low-level techniques: using SIMD floating-point arithmetic

Implementations of large-integer arithmetic on 64-bit processors usually decompose a large integer into limbs of 64 bits. Arithmetic is then performed using fast 64-bit integer-multiply and -add instructions [1, 25, 2]. For our implementation we do not make use of these instructions but instead use double-precision floating-point arithmetic. Many modern microprocessors including all microprocessors implementing the AMD64 architecture have very fast floating-point units. This is due to the fact that the performance of many applications such as image and video processing relies on fast floating-point arithmetic rather than integer processing and that many CPU benchmarks focus on the speed of floating-point operations.

It has been shown before that one can use these fast floating-point units for high-speed cryptography and for arithmetic on large integers, see for example Bernstein in [12] and [11]. In contrast to the implementation in [12] which uses

Algorithm 3 Coefficient reduction

```
Input: Coefficient vector $(h_0, h_1, \dots, h_{11}) \in \mathbb{Z}^{12}$ of $H \in R$ with $H(1) = h$.
Output: Reduced coefficient vector $(h'_0, h'_1, \dots, h'_{11})$ of $H'$ with $H'(1) = h$.
 1: $r \leftarrow \mathrm{rnd}(h_1/v), h_1 \leftarrow h_1 - rv, h_2 \leftarrow h_2 + r$
 2: $r \leftarrow \mathrm{rnd}(h_4/v), h_4 \leftarrow h_4 - rv, h_5 \leftarrow h_5 + r$
 3: $r \leftarrow \mathrm{rnd}(h_7/v), h_7 \leftarrow h_7 - rv, h_8 \leftarrow h_8 + r$
 4: $r \leftarrow \mathrm{rnd}(h_{10}/v), h_{10} \leftarrow h_{10} - rv, h_{11} \leftarrow h_{11} + r$
 5: $r \leftarrow \mathrm{rnd}(h_2/v), h_2 \leftarrow h_2 - rv, h_3 \leftarrow h_3 + r$
 6: $r \leftarrow \mathrm{rnd}(h_5/v), h_5 \leftarrow h_5 - rv, h_6 \leftarrow h_6 + r$
 7: $r \leftarrow \mathrm{rnd}(h_8/v), h_8 \leftarrow h_8 - rv, h_9 \leftarrow h_9 + r$
 8: $r \leftarrow \mathrm{rnd}(h_{11}/v), h_{11} \leftarrow h_{11} - rv$
 9: $h_9 \leftarrow h_9 - r$
10: $h_6 \leftarrow h_6 - 4r$
11: $h_3 \leftarrow h_3 - r$
12: $h_0 \leftarrow h_0 - r$
13: $r \leftarrow \mathrm{rnd}(h_0/(6v)), h_0 \leftarrow h_0 - r \cdot 6v, h_1 \leftarrow h_1 + r$
14: $r \leftarrow \mathrm{rnd}(h_3/v), h_3 \leftarrow h_3 - rv, h_4 \leftarrow h_4 + r$
15: $r \leftarrow \mathrm{rnd}(h_6/(6v)), h_6 \leftarrow h_6 - r \cdot 6v, h_7 \leftarrow h_7 + r$
16: $r \leftarrow \mathrm{rnd}(h_9/v), h_9 \leftarrow h_9 - rv, h_{10} \leftarrow h_{10} + r$
17: $r \leftarrow \mathrm{rnd}(h_1/v), h_1 \leftarrow h_1 - rv, h_2 \leftarrow h_2 + r$
18: $r \leftarrow \mathrm{rnd}(h_4/v), h_4 \leftarrow h_4 - rv, h_5 \leftarrow h_5 + r$
19: $r \leftarrow \mathrm{rnd}(h_7/v), h_7 \leftarrow h_7 - rv, h_8 \leftarrow h_8 + r$
20: $r \leftarrow \mathrm{rnd}(h_{10}/v), h_{10} \leftarrow h_{10} - rv, h_{11} \leftarrow h_{11} + r$
21: return $(h'_0, h'_1, \dots, h'_{11})$.
```

80-bit floating-point values (with a 64-bit mantissa), we decided to use 64-bit floating-point values (with a 53-bit mantissa). This allows us to use the single-instruction multiple-data (SIMD) instructions of the SSE2 and SSE3 instruction set operating on double-precision (64-bit) floating-point values.

These instructions perform two double-precision floating-point operations at once on two independent inputs layed out in 128-bit vector registers called XMM registers. The AMD64 architecture defines 16 architectural XMM registers. For example the instruction `addpd %xmm1, %xmm2` takes the low 64 bits from register `%xmm1` and the low 64 bits of register `%xmm2`, adds them as double-precision floating-point values and stores the result in the low 64 bits of register `%xmm2`; at the same time it takes the high 64 bits from register `%xmm1` and the high 64 bits of register `%xmm2`, adds them as double-precision floating-point values and stores the result in the high 64 bits of register `%xmm2`.

The most important SSE2 instructions for our implementation are the `addpd` and the `mulpd` instructions. The Intel Core 2 processors (both, 65 nm and 45 nm models) can issue up to one `mulpd` and one `addpd` instruction per cycle and thus execute 4 floating-point operations in one cycle. However, it can not execute 2 `mulpd` or 2 `addpd` instructions in the same cycle (for details see [17]). To arrange data in the XMM vector registers our implementation requires additional non-arithmetic instructions such as `shufpd`, `unpckhpd` and `unpcklpd`. In the implementation of the squaring in $\mathbb{F}_{p^2}$ we also need the `addsd` instruction which adds the low double values of two XMM registers and leaves the high double value of the destination register unchanged.

Note that all arithmetic instructions only have 2 operands, one of the inputs is overwritten by the output. This sometimes requires additional `mov` instructions to copy data to other registers or memory.

5.1 Avoiding overflows

Double-precision floating point registers hold real numbers of the form $2^e f$, with $f \in \{-2^{53}-1, \ldots, 2^{53}-1\}$ and $e \in \{-1022, \ldots, 1023\}$. The result of an operation on two such numbers is guaranteed to be exact if it is in $\{-2^{53}-1, \ldots, 2^{53}-1\}$, otherwise the result value may *overflow*. To make sure that such overflows do not occur, we cannot simply run the code on some inputs and check whether it produces the correct results; we have to make sure that an overflow cannot occur for any valid inputs.

We first implemented all algorithms in the C++ programming language (not using SIMD instructions) and replace the `double` data type by a self-written class `CheckDouble` to represent 64-bit floating-point values. This class performs all arithmetic operations on a member variable d of type `double`. Furthermore it stores the "worst-case" absolute value of the mantissa m in a member variable of type `uint64_t` which is updated with each operation. Before actually performing an operation it checks whether the worst-case result overflows; if it does, the program is aborted. Updating m is straight-forward: Multiplying (d_1, m_1) and (d_2, m_2) yields $(d_1 d_2, m_1 m_2)$, adding (d_1, m_1) and (d_2, m_2) yields $(d_1 + d_2, m_1 + m_2)$, and subtracting (d_1, m_1) from (d_2, m_2) yields $(d_2 - d_1, m_1 + m_2)$. The only divisions are by the constants v and $6v$, for those divisions it is safe to set the result to $(d/v, |m/v|)$ or $(d/6v, |m/6v|)$ respectively. The remainder of a (rounding) division by v is always between $-|v/2|$ and $|v/2|$, so we can just set the maximal mantissa to $|v/2|$ when computing the remainder of a division by v. Analgously, the maximal mantissa of the remainder of a division by $6v$ is $|3v|$.

For all constants involved in the pairing computation we can initialize the maximal mantissa with the actual value. For the inputs to the pairing we assume that they are worst-case output of the reduction described in Algorithm 3.

In order to obtain the targeted performance we replaced the `CheckDouble` class again by the `double` data type and re-implemented the speed-critical functions in the `qhasm` programming language [10] using SIMD instructions where possible. The operations on double-precision floating-point values in this `qhasm` implementation are the same as in the C version for which we automatically verified that no overflows can occur. Tools to verify this property on the assembly level would be very helpful but we do not know of such tools. The resulting software has passed a bilinearity and non-degeneracy test on 1,000,000 random inputs, each test involving 3 pairing computations.

5.2 Implementation of field-arithmetic operations

The 12 coefficients $f_0, \dots, f_{11}$ of a polynomial F representing an element $f \in \mathbb{F}_p$ (see Section 4) are stored consecutively in a `double` array of size 12. The 24 coefficients $g_0, \dots, g_{11}$ and $h_0, \dots, h_{11}$ representing an element $(gi + h) \in \mathbb{F}_{p^2}$ are stored interleaved in a `double` array of size 24 as $(h_0, g_0 | h_1, g_1 | \dots | h_{11}, g_{11})$. In the following descriptions, all SIMD instructions operate on every two adjacent double values of this representation. Observe that the implementations do not minimize the number of *instructions* but try to minimize the number of *cycles*.

$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication. Multiplication of $ai + b$ and $ci + d$, layed out in memory as $op_1 = (b_0, a_0 | \dots | b_{11}, a_{11})$ and $op_2 = (d_0, c_0 | \dots | d_{11}, c_{11})$, is done by duplicating $b_0, \dots, b_{11}$ to obtain

$$t_1 = (b_0, b_0 | b_1, b_1 | \dots | b_{11}, b_{11}).$$

We then perform a digit-sliced multiplication of t_1 and op_2 to obtain

$$t_2 = (bd_0, bc_0 | \dots | bd_{22}, bc_{22}).$$

In a second step we duplicate $a_0, \dots, a_{11}$, obtain

$$t_1 = (a_0, a_0 | a_1, a_1 | \dots | a_{11}, a_{11}),$$

multiply digit-sliced with op_2 and obtain

$$t_3 = (ad_0, ac_0 | \dots | ad_{22}, ac_{22}).$$

We then multiply the high double values of t_3 by $i^2 = -7$ and obtain

$$t_3 = (ad_0, -7ac_0 | \dots | ad_{22}, -7ac_{22}).$$

Swapping in t_3 yields

$$t_3 = (-7ac_0, ad_0 | \dots | -7ac_{22}, ad_{22}).$$

Finally we add t_3 to t_2 and apply polynomial reduction (Algorithm 2) and coefficient reduction (Algorithm 3) to obtain interleaved coefficients of $(ai+b)(ci+d) = ((ad + bc)i + (bd - 7ac))$. During this computation we keep values in XMM registers as much as possible.

The parallel digit-sliced multiplication uses the schoolbook algorithm resulting in 144 multiplications (`mulpd`), 121 additions (`addpd`), and 10 more multiplications by 6 (`mulpd`). We experimented with Karatsuba multiplication but did not gain any performance – we are planning to further examine possible speedups by using Karatsuba multiplication.

Computing the rounded quotient and the remainder in the coefficient reduction could be done using multiplication by $1/v$, using the `roundpd` instruction on the result, multiplying by v and subtracting the result from the original value to obtain the remainder. As the `roundpd` instruction is part of the SSE4.1 instruction set which is not available on 65 nm Core 2 processors and all AMD processors we decided to implement rounding as addition and subsequent subtraction of a constant as explained for example by Bernstein in [11].

$\mathbb{F}_{p^2}$ squaring. When squaring an element $ai + b \in \mathbb{F}_{p^2}$, layed out in memory as $op_1 = (b_0, a_0| \dots |b_{11}, a_{11})$, we swap the coefficients to obtain

$$t_1 = (a_0, b_0| \dots |a_{11}, b_{11}).$$

We then copy t_1 and apply the `addsd` instruction with op_1 to obtain

$$t_2 = (a_0 + b_0, b_0| \dots |a_{11} + b_{11}, b_{11}).$$

Then we multiply the low double values in t_1 by $i^2 = -7$ and obtain

$$t_1 = (-7a_0, b_0| \dots | -7a_{11}, b_{11}).$$

Applying the `addsd` instruction on t_1 and op_1 yields

$$t_1 = (b_0 - 7a_0, b_0| \dots |b_{11} - 7a_{11}, b_{11}).$$

Now we use digit-sliced multiplication on t_1 and t_2 to obtain

$$r = (((b - 7a)(a + b))_0, ab_0| \dots |((b - 7a)(a + b))_{22}, ab_{22}).$$

Copying r and duplicating the high double values yieds

$$d = (ab_0, ab_0| \dots |ab_{22}, ab_{22}).$$

Multiplying the low double values in d by $-i^2 - 1 = 6$ yields

$$d = (6ab_0, ab_0| \dots |6ab_{22}, ab_{22}), \text{and}$$

adding d to r and applying polynomial reduction and coefficient reduction yields the coefficients of the result $(2ab)i + (b^2 - 7a^2)$.

$\mathbb{F}_{p^2} \times \mathbb{F}_p$ multiplication. Evaluating the line functions requires multiplications of an element of $\mathbb{F}_{p^2}$ with an element of $\mathbb{F}_p$. This is implemented using the same parallel schoolbook multiplication which is used in the $\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication. This requires first duplicating the coefficients of the $\mathbb{F}_p$ element in memory.

$\mathbb{F}_{p^2}$ short coefficient reduction. Additions, subtractions and multiplications with small constants in $\mathbb{F}_{p^2}$ can all be implemented using 12 SIMD instructions. They produce results which may have coefficients that are too large to go as input into a multiplication but are still so small that they do not require the full-fledged coefficient reduction from Algorithm 3. If the output of an addition or subtraction is used as input to a multiplication we apply a short coefficient reduction which first carries from f_{11} to f_0, f_3, f_6 and f_9. Then it carries from all odd coefficients $f_1, f_3, \dots, f_9$ and then from all even coefficients $f_0, f_2, \dots, f_{10}$.

$\mathbb{F}_p$ inversion. The final exponentation involves one inversion in $\mathbb{F}_{p^{12}}$. This can be computed with only one inversion in $\mathbb{F}_p$ and several multiplications as described in, e.g., [22, Sec. 2]. We implement inversion in $\mathbb{F}_p$ as exponentiation with $p - 2$ using a simple square-and-multiply algorithm. There exist certainly faster methods to compute inverses in $\mathbb{F}_p$, but this way we can easily ensure constant-time behaviour of the inversion and the single inversion in $\mathbb{F}_{p^2}$ accounts for less than 3 percent of the total computation time.

6 Benchmarking results

This section gives benchmarking results of the pairing computation on different microarchitectures. All benchmarks were obtained by iteratively calling the function to benchmark and a function reading the CPU cycle counter 1000 times and then computing the median of the differences of every two consecutive cycle counts. The call to the function reading the cycle counter and the loop control incurs some overhead so Table 2 also gives the cycles obtained when no function is called between two consecutive cycle counts. The counts for the Miller loop in Table 2 include the two final addition steps in the optimal ate pairing before the final exponentiation.

Name	Affiliation	CPU	OS	Compiler
`latour`	Eindhoven University of Technology	Intel Core 2 Quad Q6600 2394 MHz	Linux 2.6.28	gcc 4.3.3
`behemoth`	National Taiwan University	Intel Core 2 Quad Q9550 2830 MHz	Linux 2.6.27	gcc 4.3.2
`dragon`	National Taiwan University	Intel Xeon E5504 2000 MHz	Linux 2.6.27	gcc 3.4.6
`mace`	University of Illinois at Chicago	AMD Athlon 64 X2 3800+ 2000 MHz	Linux 2.6.31	gcc 4.4.1
`chukonu`	University of Illinois at Chicago	AMD Phenom II X4 955 3210.298 MHz	Linux 2.6.31	gcc 4.4.1

Table 1. Machines used for benchmarking

	`latour`	`behemoth`	`dragon`	`mace`	`chukonu`
no function	72	34	24	11	75
$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication	693	672	676	1732	737
$\mathbb{F}_{p^2}$ squaring	558	510	504	1220	606
$\mathbb{F}_{p^2} \times \mathbb{F}_p$ multiplication	432	391	388	977	434
$\mathbb{F}_{p^2}$ short coefficient reduction	144	110	84	195	152
$\mathbb{F}_{p^2}$ inversion	127071	126718	134524	339835	127067
Miller loop	2,267,811	2,320,908	2,365,812	5,666,343	2,506,893
optimal ate pairing	4,470,408	4,480,716	4,736,408	10,961,234	4,989,872

Table 2. Cycle counts of relevant operations on different machines. Parameters: $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$, with $u = v^3$ and $v = 1966080$, BN curve: $y^2 = x^3 + 17$ over $\mathbb{F}_p$.

Acknowledgements: We thank Tanja Lange and the anonymous referees for their valuable comments that helped us improve the paper.

References

1. The GNU MP bignum library. `http://gmplib.org/` (accessed Mar 31, 2010).
2. MPFQ - a finite field library. `http://mpfq.gforge.inria.fr/` (accessed Mar 31, 2010).
3. Christophe Arène, Tanja Lange, Michael Naehrig, and Christophe Ritzenthaler. Faster pairing computation. Cryptology ePrint Archive, Report 2009/155, to appear in the Journal of Number Theory, 2010. `http://eprint.iacr.org/2009/155/`.
4. Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management - part 1: General (revised), 2007. Published as NIST Special Publication 800-57, `http://csrc.nist.gov/groups/ST/toolkit/documents/SP800-57Part1_3-8-07.pdf`.
5. Paulo S. L. M. Barreto. A survey on craptological pairing algorithms. *Journal of Craptology*, 7, 2010. `http://www.anagram.com/~jcrap/Volume_7/Pairings.pdf`.
6. Paulo S. L. M. Barreto, Steven D. Galbraith, Colm Ó' hÉigeartaigh, and Michael Scott. Efficient pairing computation on supersingular abelian varieties. *Designs, Codes and Cryptography*, 42(3):239–271, 2007.
7. Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. In *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2002.
8. Paulo S. L. M. Barreto, B. Lynn, and Michael Scott. Efficient implementation of pairing-based cryptosystems. *Journal of Cryptology*, 17(4):321–334, 2004.
9. Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2006.
10. Daniel J. Bernstein. qhasm: tools to help write high-speed software. `http://cr.yp.to/qhasm.html` (accessed Mar 31, 2010).
11. Daniel J. Bernstein. Floating-point arithmetic and message authentication, 2004. Document ID: dabadd3095644704c5cbe9690ea3738e, `http://cr.yp.to/papers.html#hash127`.
12. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. Document ID: 4230efdfa673480fc079449d90f322c0, `http://cr.yp.to/papers.html#curve25519`.
13. Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004.
14. Augusto J. Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over Barreto-Naehrig curves. In *Pairing-Based Cryptography – Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 197–207. Springer-Verlag, 2007.
15. Nigel Smart (editor). ECRYPT2 yearly report on algorithms and keysizes (2008-2009). Technical report, ECRYPT II – European Network of Excellence in

Cryptology, EU FP7, ICT-2007-216676, 2009. published as deliverable D.SPA.7 `http://www.ecrypt.eu.org/documents/D.SPA.7.pdf`.

16. Junfeng Fan, Frederik Vercauteren, and Ingrid Verbauwhede. Faster $\mathbb{F}_p$-arithmetic for cryptographic pairings on Barreto-Naehrig curves. In *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 240–253, 2009. `http://www.cosic.esat.kuleuven.be/publications/article-1256.pdf`.
17. Agner Fog. Software optimization ressources, 2010. `http://http://www.agner.org/optimize/` (accessed Mar 31, 2010).
18. David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010.
19. Philipp Grabher, Johann Großschädl, and Dan Page. On software parallel implementation of cryptographic pairings. In *Selected Areas in Cryptography – SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009.
20. Robert Granger and Michael Scott. Faster squaring in the cyclotomic subgroup of sixth degree extensions. Cryptology ePrint Archive, Report 2009/565, 2009. `http://eprint.iacr.org/2009/565/`, to appear in PKC 2010.
21. Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2008.
22. Darrel Hankerson, Alfred Menezes, and Michael Scott. Software implementation of pairings. In Marc Joye and Gregory Neven, editors, *Identity-Based Cryptography*. IOS Press, 2008. Draft available online: `http://www.math.uwaterloo.ca/~ajmeneze/publications/pairings_software.pdf`.
23. Florian Heß, Nigel P. Smart, and Frederik Vercauteren. The eta pairing revisited. *IEEE Transactions on Information Theory*, 52:4595–4602, 2006.
24. Eunjeong Lee, Hyang-Sook Lee, and Cheol-Min Park. Efficient and generalized pairing computation on abelian varieties. Cryptology ePrint Archive, Report 2008/040, 2008. `http://eprint.iacr.org/2008/040/`.
25. Shamus Software Ltd. Multiprecision integer and rational arithmetic C/C++ library. `http://www.shamus.ie/` (accessed Mar 31, 2010).
26. Victor S. Miller. Short programs for functions on curves. Unpublished manuscript, 1986. `http://crypto.stanford.edu/miller/miller.pdf`.
27. Victor S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17:235–261, 2004.
28. Michael Naehrig. *Constructive and Computational Aspects of Cryptographic Pairings*. PhD thesis, Technische Universiteit Eindhoven, 2009. `http://www.cryptojedi.org/users/michael/data/thesis/2009-05-13-diss.pdf`.
29. Michael Naehrig, Paulo S. L. M. Barreto, and Peter Schwabe. On compressible pairings and their computation. In *Progress in Cryptology - AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2008.
30. Michael Scott. Personal communication, March 2010.
31. Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Pairing-Based Cryptography – Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2009.
32. Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1).