High-Speed Parallel Software Implementation of the η_T Pairing

Diego F. Aranha[1,⋆], Julio López[1], and Darrel Hankerson[2]

[1] University of Campinas
{dfaranha,jlopez}@ic.unicamp.br
[2] Auburn University
hankedr@auburn.edu

Abstract. We describe a high-speed software implementation of the η_T pairing over binary supersingular curves at the 128-bit security level. This implementation explores two types of parallelism found in modern multi-core platforms: vector instructions and multiprocessing. We first introduce novel techniques for implementing arithmetic in binary fields with vector instructions. We then devise a new parallelization of Miller's Algorithm to compute pairings. This parallelization provides an algorithm for pairing computation without increasing storage costs significantly. The combination of these acceleration techniques produce serial timings at least 24% faster and parallel timings 66% faster than the best previous result in an Intel Core platform, establishing a new state-of-the-art implementation of this pairing instantiation in this platform.

Keywords: Efficient software implementation, vector instructions, multi-core architectures, bilinear pairings, parallelization.

1 Introduction

The computation of bilinear pairings is the most expensive operation in Pairing-based Cryptography, especially for high levels of security. For this reason, implementations must employ all the resources found in the target platform to obtain maximum efficiency. A resource being increasingly introduced in computing platforms is parallelism, in the form of vector instructions (data parallelism) and multiprocessing (task parallelism). This trend is observed even in the embedded space, with proposals of resource-constrained multi-core architectures and vector instruction sets for multimedia processing in portable devices.

This work describes a high-performance implementation of the η_T pairing [1] over binary supersingular curves at the 128-bit security level which employs these two forms of parallelism in a very efficient way. The target platform is the Intel Core architecture [2], the most popular 64-bit computing platform. Our main contributions are:

⋆ Supported by FAPESP under grant no. 2007/06950-0.

J. Pieprzyk (Ed.): CT-RSA 2010, LNCS 5985, pp. 89–105, 2010.

- *Novel techniques for implementing arithmetic in binary fields:* we explore powerful SIMD instructions to accelerate arithmetic in binary fields. We focus on the SSE family of vector instructions, but the same techniques can be employed with other SIMD instruction sets such as Altivec and the upcoming AMD SSE5.
- *Parallelization of Miller's Algorithm to compute pairings:* we develop a simple algorithm for parallel pairing computation which does not increase storage costs. Our parallelization is independent of the underlying pairing instantiation, allowing a parallel implementation to reach scalability in a variable number of processors unrelated to the pairing mathematical definition. This parallelization provides good scalability in fields of small characteristic.
- *Static load balancing technique:* we present a simple technique to balance the costs of parallel pairing computation between the available processing units. The technique is successfully applied for latency minimization, but its flexibility allows the implementation to determine controlled non-optimal partitions of the algorithm.
- *Experimental results:* speedups of parallel implementations over serial implementations are estimated and experimentally verified for platforms up to 8 processors. We also obtain an approximation of the performance up to 32 processing units and compare our serial and parallel execution times with the current state-of-the-art implementations with the same parameters.

The results of this work can improve serial and parallel implementations of pairings. The parallelization may be important to reduce the latency of pairing computation in two scenarios: (i) desktop-class processors running real-time applications with strict response time requirements; (ii) embedded multiprocessor architectures with weak processing units. The availability of parallel algorithms for application in these scenarios is suggested as an open problem by [3] and [4]. Our features of flexible load balancing and small storage overhead are critical for the second scenario, because they can support static scheduling schemes for compromises between pairing computation time and power consumption; and memory capacity is commonly restricted in embedded devices.

2 Finite Field Arithmetic

In this section we will represent the elements of $\mathbb{F}_{2^m}$ using a polynomial basis. Let $f(z)$ be an irreducible binary polynomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$. A field element $a(z) = \sum_{i=0}^{m-1} a_i z^i$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_1, a_0)$ of length m. In a software implementation, these bit coefficients are packed and stored in an array $(a[0], \ldots, a[n-1])$ of n W-bit words, where W is the word size of the processor. For simplicity, we assume that n is always even.

2.1 Vector Instruction Sets

Vector instructions, also called SIMD (Single Instruction, Multiple Data) because they operate in several data objects simultaneously, are widely supported

in recent families of processor architectures. The number, functionality and efficiency of these instructions have been improved with each new generation of processors, and natural applications include multimedia processing, scientific applications or any software with high arithmetic density. Some well-known SIMD instruction sets are the Intel MMX and SSE [5] families, the Altivec extensions introduced by Apple and IBM in the Power architecture specification and AMD 3DNow. Instruction sets supported by current technology are restricted to 128-bit registers and provide simple orthogonal operations across 8, 16, 32 or 64-bit data units stored inside these registers, but future extensions such as Intel AVX and AMD SSE5 will support 256-bits registers with the added inclusion of a heavily-anticipated carry-less multiplier [6].

The Intel Core microarchitecture is equipped with several vector instruction sets which operate in 16 architectural 128-bit registers. A small subset of these instructions can be used to implement binary field arithmetic, some found in the Streaming SIMD Extensions 2 (SSE2) and others in the Supplementary SSE3 instructions (SSSE3). The SSE2 instruction set is also supported by the recent VIA Nano processors, AMD processors since the K8 family and Intel processors since the Pentium 4.

A non-exhaustive list of SSE2 instructions relevant for our work is given below. Each instruction described will be referred in the algorithms by the short mnemonic which follows the instruction opcode:

- `MOVDQU/MOVDQA` (*load*/*store*): implements load/store between unaligned/aligned memory addresses and registers. In our implementation, all allocated memory is stored in 128-bit aligned base addresses so that the faster `MOVDQA` instruction can always be used.
- `PSLLQ/PSRLQ` ($\ll_{\dagger 8}, \gg_{\dagger 8}$): implements bitwise left/right shifts of a pair of 64-bit integers while shifting in zero bits. This instruction does not propagate bits from the lower 64-bit integer to the higher 64-bit integer, thus additional shifts and additions are required to implement bitwise shifts of 128-bit values.
- `PSLLDQ/PRLLDQ` ($\ll_8, \gg_8$): implements byte-wise left/right shifts of a 128-bit register. Since this instruction propagates bytes from the lower half to the higher half of a 128-bit register, this instruction is preferred over the previous one when the shift amount is a multiple of 8. Thus shifts by multiples of 8 bits should be used whenever possible. The latency of this instruction is 2 cycles in the first generation of Core 2 Conroe/Merom (65nm) processors and 1 cycle in the more recent Penryn/Wolfdale (45nm) microarchitecture.
- `PXOR/PAND/POR` ($\oplus, \wedge, \vee$): implements bitwise XOR/AND/OR of two 128-bit registers. These instructions have a high throughput, reaching 3 instructions per cycle when the operands are registers and there are no dependencies between consecutive operations.
- `PUNPCKLBW/PUNPCKHBW` (*interlo*/*interhi*): interleaves the lower/higher bytes in a register with the lower/higher bytes of another register.

We also find application for powerful but often-missed SSSE3 instructions:

- `PALIGNR` ($\lhd$): takes registers r_a and r_b, concatenate their values, and pull out a 128-bit section from an offset given by a constant immediate; in other

words, implements a right byte-wise shift with propagation of shifted out bytes from r_a to r_b. This instruction can be used to implement a left shift by s bytes with the immediate $(16 - s)$.
- `PSHUFB` (*lookup* or *shuffle* depending on functionality): takes registers of bytes $r_a = a_0, a_1, \ldots, a_{16}$ and $r_b = b_0, b_1, \ldots, b_{16}$ and replaces r_a with the permutation $a_{b_0}, a_{b_1}, \ldots, a_{b_{16}}$; except that it replaces a_i with zero if the most significant bit of b_i is set. A powerful use of this instruction is to perform 16 simultaneous lookups in a 16-byte lookup table. This can be easily done by storing the lookup table in r_a and the lookup indexes in r_b. Intel introduced a specific *Super Shuffle Engine* in the latest microarchitecture to reduce the latency of this instruction from 3 cycles to 1 cycle.

Alternate vector instruction sets present functional analogues of these instructions. In particular, the `PSHUFB` permutation instruction is implemented as `VPERM` in Altivec and as `PPERM` in SSE5, although the `PPERM` instruction is reportedly more powerful as it can also operate at bit level. SIMD instructions are critical for the performance of binary field arithmetic and can be easily accessed with compiler intrinsics. In the remainder of this section, the optimization techniques applied during the implementation of each field operation are detailed. We will describe algorithms in terms of *vector operations* using the mnemonics defined above so that algorithms can be easily transcribed to other target platforms. Specific instruction choices based on latency or functionality will be focused on the SSE family.

2.2 Squaring

Since the square of a finite field element $a(z) \in \mathbb{F}_{2^m}$ is given by $a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1} z^{2m-2} + \cdots + a_2 z^4 + a_1 z^2 + a_0$, the binary representation of $a(z)^2$ can be computed by inserting a zero bit between each pair of consecutive bits on the binary representation of $a(z)$. This operation can be accelerated by introducing a lookup table as discussed in [7]. This method can be improved further if the table lookups can be executed simultaneously. This way, for an implementation which processes 4 bits per iteration, squaring can be implemented mainly in terms of permutation instructions which convert groups of 4 bits (*nibbles*) to the corresponding expanded bytes. The proposed optimization is shown in Algorithm 1. The algorithm receives a field element a stored in a vector of n 64-bit words (or $\frac{n}{2}$ 128-bit values) and expands the input into a double-precision vector t which can be reduced modulo $f(z)$. At each iteration of this algorithm, a 128-bit value $a[2i]$ is loaded from memory and separated by a bit mask into two registers containing the low nibbles (a_L) and the high nibbles (a_H). Each group of nibbles is then expanded from 4 bits to 8 bits by a parallel table lookup. The proper order of bytes is restored by interleaving instructions which pick alternately the lower or higher bytes of a_L or a_H to form two consecutive 128-bit values $(t[2i], t[2i+1])$ produced as the result.

Algorithm 1. Proposed implementation of squaring in $\mathbb{F}_{2^m}$.

Input: $a(z) = a[0..n-1]$.
Output: $c(z) = c[0..n-1] = a(z)^2 \bmod f(z)$.

1: ▷ Store in *table* the squares $u(z)^2$ of all 4-bit polynomials $u(z)$.
2: $table \leftarrow$ `0x5554515045444140,0x1514111005040100`
3: $mask \leftarrow$ `0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F`
4: **for** $i \leftarrow 0$ **to** $\frac{n}{2} - 1$ **do**
5: $\quad a_0 \leftarrow load(a[2i])$
6: $\quad a_L \leftarrow a_0 \wedge mask, \quad a_L \leftarrow lookup(table, a_L)$
7: $\quad a_H \leftarrow a_0 \gg_{\dagger 8} 4, \quad a_H \leftarrow a_H \wedge mask, \quad a_H \leftarrow lookup(table, a_H)$
8: $\quad t[2i] \leftarrow store(interlo(a_L, a_H)), \quad t[2i+1] \leftarrow store(interhi(a_L, a_H))$
9: **end for**
10: **return** $c = t \bmod f(z)$

2.3 Square Root

Given an element $a(z) \in \mathbb{F}_{2^m}$, the field element $c(z)$ such that $c(z)^2 = a(z) \bmod f(z)$ can be computed by $c(z) = a_{even} + \sqrt{z} \cdot a_{odd} \bmod f(z)$, where a_{even} represents the concatenation of even coefficients of $a(z)$, a_{odd} represents the concatenation of odd coefficients of $a(z)$ and $\sqrt{z}$ is a constant depending on the irreducible polynomial $f(z)$ [8]. When $f(z)$ is a suitable trinomial $f(z) = z^m + z^t + 1$ with odd exponents m, t, $\sqrt{z}$ has the sparse form $\sqrt{z} = z^{\frac{m+1}{2}} + z^{\frac{t+1}{2}}$ and multiplication by this constant can be computed with shifts and additions only.

This algorithm can also be implemented with simultaneous table lookups. Algorithm 2 presents our implementation of this method with vector instructions. The algorithm processes 128 bits of a in each iteration and progressively separates the coefficients of $a[2i]$ in even or odd coefficients. First, a permutation mask is used to divide $a[2i]$ in bytes of odd index and bytes of even index. The bytes with even indexes are stored in the lower 64-bit part of a_0 and the bytes with odd indexes are stored in the higher 64-bit part of a_0. The high and low nibbles of a_0 are then divided into a_L and a_H and additional lookup tables are applied to further separate the bits of a_L and a_H into bits with odd and even indexes. At the end of the 128-bit section, a_0 stores the interleaving of odd and even coefficients of a packed into groups of 4 bits. The remaining instructions in the 128-bit sections separate the even and odd coefficients into u and v, which can be reordered and multiplied by $\sqrt{z}$ inside the 64-bit section. We implement these final steps in 64-bit mode to avoid expensive shifts in 128-bit registers.

2.4 Multiplication

Two different strategies are commonly considered for the implementation of multiplication in $\mathbb{F}_{2^m}$. The first one consists in applying the Karatsuba algorithm [9] to divide the multiplication in sub-problems and solve each problem independently [7] (for $a(z) = A_1 z^{\lceil m/2 \rceil} + A_0$ and $b(z) = B_1 z^{\lceil m/2 \rceil} + B_0$):

$$c(z) = a(z) \cdot b(z) = A_1 B_1 z^m + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^{\lceil m/2 \rceil} + A_0 B_0.$$

Algorithm 2. Proposed implementation of square root in $\mathbb{F}_{2^m}$.

Input: $a(z) = a[0..n-1]$, exponents m and t of trinomial $f(z)$.
Output: $c(z) = c[0..n-1] = a(z)^{\frac{1}{2}} \bmod f(z)$.

▷ Permutation mask to divide a 128-bit value in bytes with odd and even indexes.
$perm \leftarrow$ `0x0F0D0B0907050301,0x0E0C0A0806040200`
▷ Tables to divide a low/high nibble in bits with odd and even indexes.
$sqrt_L \leftarrow$ `0x3332232231302120,0x1312030211100100`
▷ Table to divide a high nibble in bits with odd and even indexes ($sqrt_L \ll 2$).
$sqrt_H \leftarrow$ `0xCCC88C88C4C08480,0x4C480C0844400400`
▷ Bit masks to isolate bytes in lower or higher nibbles.
$mask_L \leftarrow$ `0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F`
$mask_H \leftarrow$ `0xF0F0F0F0F0F0F0F0,0xF0F0F0F0F0F0F0F0`
$c[0 \ldots n-1] \leftarrow 0$, $h \leftarrow \frac{n+1}{2}, l \leftarrow \frac{t+1}{128}, s_1 \leftarrow \frac{m+1}{2} \bmod 64, s_2 \leftarrow \frac{t+1}{2} \bmod 64$
for $i \leftarrow 0$ **to** $\frac{n}{2} - 1$ **do**
  $a_0 \leftarrow load(a[2i]), \quad a_0 \leftarrow shuffle(a_0, perm)$
  $a_L \leftarrow a_0 \wedge mask_L, \; a_L \leftarrow lookup(sqrt_L, a_L),$
  $a_H \leftarrow a_0 \wedge mask_H, \; a_H \leftarrow a_H \gg_{\dagger 8} 4, \quad a_H \leftarrow lookup(sqrt_H, a_H)$
  $a_0 \leftarrow a_L \vee a_H, \quad a_L \leftarrow a_0 \wedge mask_L, \; a_H \leftarrow a_0 \wedge mask_H$
  $u \leftarrow store(a_L), \; v \leftarrow store(a_H)$
  ▷ From now on, operate in 64-bit registers.
  $a_{even} \leftarrow u[0] \vee u[1] \ll 4, \; a_{odd} \leftarrow v[1] \vee v[0] \gg 4$
  $c[i] \leftarrow c[i] \oplus a_{even}$
  $c[i+h-1] \leftarrow c[h+i-1] \oplus (a_{odd} \ll s_1)$
  $c[i+h] \leftarrow c[h+i] \oplus (a_{odd} \gg (64 - s_1))$
  $c[i+l] \leftarrow c[i+l] \oplus (a_{odd} \ll s_2)$
  $c[i+l+1] \leftarrow c[i+l+1] \oplus (a_{odd} \gg (64 - s_2))$
end for
return c

The second one consists in applying a direct algorithm like the *comb* method proposed by López and Dahab in [10]. Conventionally, the series of additions involved in this method are implemented through additions over sub parts of a double-precision vector. In order to reduce the number of memory accesses during these additions, we employ n registers. These registers simulate the series of memory additions by accumulating consecutive writes, allowing the implementation to reach maximum `XOR` throughput. We also employ an additional table T_1 analogue to T_0 which stores $u(z) \cdot (b(z) \ll 4)$ to eliminate shifts by 4, as discussed in [10]. Recall that shifts by multiples of 8 bits are faster in the target platform. We assume that the length of operand $b[0..n-1]$ is at most $64n-7$ bits; if necessary, terms of higher degree can be processed separately at relatively low cost. The implemented LD multiplication algorithm is shown as Algorithm 3. The element $a(z)$ is processed in groups of 8 bits separated by intervals of 128 bits. This avoids shifts of the register vector since a 128-bit shift can be emulated by referencing m_{i+1} instead of m_i. The multiple precision shift by 8 bits of the register vector ($\lhd 8$) is implemented with 15-byte shifts with carry propagation ($\lhd$) of register pairs.

Algorithm 3. LD multiplication implemented with n 128-bit registers.

Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..n-1]$.
Note: m_i denotes the vector of $\frac{n}{2}$ 128-bit registers $(r_{(i-1+n/2)}, \ldots, r_i)$.

1: Compute $T_0(u) = u(z) \cdot b(z), T_1(u) = u(z) \cdot (b(z) \ll 4)$ for all $u(z)$ of degree < 4.
2: $(r_{n-1} \ldots, r_0) \leftarrow 0$
3: **for** $k \leftarrow 56$ **downto** 0 **by** 8 **do**
4: **for** $j \leftarrow 1$ **to** $n-1$ **by** 2 **do**
5: Let $u = (u_3, u_2, u_1, u_0)$, where u_t is bit $(k+t)$ of $a[j]$.
6: Let $v = (v_3, v_2, v_1, v_0)$, where v_t is bit $(k+t+4)$ of $a[j]$.
7: $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_0(u)$
8: $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_1(v)$
9: **end for**
10: $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
11: **end for**
12: **for** $k \leftarrow 56$ **downto** 0 **by** 8 **do**
13: **for** $j \leftarrow 0$ **to** $n-2$ **by** 2 **do**
14: Let $u = (u_3, u_2, u_1, u_0)$, where u_t is bit $(k+t)$ of $a[j]$.
15: Let $v = (v_3, v_2, v_1, v_0)$, where v_t is bit $(k+t+4)$ of $a[j]$.
16: $m_{j/2} \leftarrow m_{j/2} \oplus T_0(u)$
17: $m_{j/2} \leftarrow m_{j/2} \oplus T_1(v)$
18: **end for**
19: **if** $k > 0$ **then** $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
20: **end for**
21: **return** $c = (r_{n-1} \ldots, r_0) \bmod f(z)$

2.5 Modular Reduction

Efficient modular reduction depends on the format of the trinomial or pentanomial $f(z)$. In general, it's better to choose $f(z)$ such that bitwise shifts amounts are multiples of 8 bits. If the non-null coefficients of $f(z)$ are located in the lower words of the array representation of $f(z)$, consecutive writes into memory can also be accumulated into registers to avoid redundant memory writes. We illustrate these optimizations with modular reduction by $f(z) = z^{1223} + z^{255} + 1$ in Algorithm 4. The algorithm receives as input a vector of n 128-bit elements and reduces this vector by accumulating four memory writes at a time in registers. Note also that shifts by multiples of 8 bits are used whenever possible.

2.6 Inversion

For inversion in $\mathbb{F}_{2^m}$ we implemented a variant of the Extended Euclidean Algorithm for polynomials [7] where the length of each temporary vector is tracked. Since this algorithm requires flexible left shifts by arbitrary amounts, we implemented the full algorithm in 64-bit mode. Some Assembly in the form of a compiler intrinsic was used to efficiently count the number of leading 0 bits to determine the highest set bit.

Algorithm 4. Proposed modular reduction by $f(z) = z^{1223} + z^{255} + 1$.

Input: $t(z) = t[0..n-1]$ (vector of 128-bit elements).
Output: $c(z) \bmod f(z) = c[0..n-1]$.
Note: The accumulate function $R(r_3, r_2, r_1, r_0, t)$ executes:

$s \leftarrow t \gg_{\dagger 8} 7, r_3 \leftarrow t \ll_{\dagger 8} 57$
$r_3 \leftarrow r_3 \oplus (s \ll_8 64)$
$r_2 \leftarrow r_2 \oplus (s \gg_8 64)$
$r_1 \leftarrow r_1 \oplus (t \ll_8 56)$
$r_0 \leftarrow r_0 \oplus (t \gg_8 72)$

1: $r_0, r_1, r_2, r_3 \leftarrow 0$
2: **for** $i \leftarrow 19$ **downto** 15 **by** 4 **do**
3: $R(r_3, r_2, r_1, r_0, t[i])$, $t[i-7] \leftarrow t[i-7] \oplus r_0$
4: $R(r_0, r_3, r_2, r_1, t[i-1])$, $t[i-8] \leftarrow t[i-8] \oplus r_1$
5: $R(r_1, r_0, r_3, r_2, t[i-2])$, $t[i-9] \leftarrow t[i-9] \oplus r_2$
6: $R(r_2, r_1, r_0, r_3, t[i-3])$, $t[i-10] \leftarrow t[i-10] \oplus r_3$
7: **end for**
8: $R(r_3, r_2, r_1, r_0, t[11])$, $t[4] \leftarrow t[4] \oplus r_0$
9: $R(r_0, r_3, r_2, r_1, t[10])$, $t[3] \leftarrow t[3] \oplus r_1$
10: $t[2] \leftarrow t[2] \oplus r_2$, $t[1] \leftarrow t[1] \oplus r_3$, $t[0] \leftarrow t[0] \oplus r_0$
11: $r_0 \leftarrow m[9] \gg_8 64$, $r_0 \leftarrow r_0 \gg_{\dagger 8} 7$, $t[0] \leftarrow t[0] \oplus r_0$
12: $r_1 \leftarrow r_0 \ll_8 64$, $r_1 \leftarrow r_1 \ll_{\dagger 8} 63$, $t[1] \leftarrow t[1] \oplus r_1$
13: $r_1 \leftarrow r_0 \gg_{\dagger 8} 1$, $t[2] \leftarrow t[2] \oplus r_1$
14: **for** $i \leftarrow 0$ **to** 9 **do** $c[2i] \leftarrow store(t[i])$
15: $c[19] \leftarrow c[19] \wedge$ `0x7F`
16: **return** c

2.7 Implementation Timings

In this section, we present our timings for finite field arithmetic. We implemented arithmetic in $\mathbb{F}_{2^{1223}}$ with irreducible trinomial $f(z) = z^{1223} + z^{255} + 1$. This field is suitable for instantiations of the η_T pairing over supersingular binary curves at the 128-bit security level [4]. The C programming language was used in conjunction with compiler intrinsics for accessing vector instructions. The chosen compiler was GCC version 4.1.2 because it generated the fastest code from vector intrinsics, as already observed by [4]. The differences between our implementations in the 65nm and 45nm processors can be explained by the lower cost of the `PSLLDQ` and `PSHUFB` instructions in the newer generation after the introduction of the *Super Shuffle Engine* by Intel.

Field multiplication was implemented by a combination of one instance of Karatsuba and the LD method depicted as Algorithm 3. Karatsuba's splitting point was at 632 bits and the divide-and-conquer steps were also implemented with vector instructions. Note that our binary field multiplier precomputes two tables of 16 rows, while the multiplier implemented in [4] precomputes a single table. This increase in memory consumption is negligible when compared to the total memory capacity of the target platform.

Table 1. Comparison of different software implementations of finite field arithmetic in two Intel Core 2 platforms. All timings are reported in cycles. Improvements are computed in comparison with the previous fastest result in a 65nm platform, since the related works do not present timings for field operations in a 45nm platform.

	Operation			
Implementation	$a^2 \bmod f$	$a^{\frac{1}{2}} \bmod f$	$a \cdot b \bmod f$	$a^{-1} \bmod f$
Hankerson et al. [4]	600	500	8200	162000
Beuchat et al. [11]	480	749	5438	–
This work (Core 2 65nm)	160	166	4030	149763
Improvement	66.7%	66.8%	25.9%	7.6%
This work (Core 2 45nm)	108	140	3785	149589

3 Pairing Computation

Miller's Algorithm for pairing computation requires a rich mathematical framework. We briefly present some definitions and point the reader to more complete treatments of the subject presented in [12,13].

3.1 Preliminary Definitions

An *admissible bilinear pairing* is an efficiently computable map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are additive groups of points in an elliptic curve E and $\mathbb{G}_T$ is a related multiplicative group. Let P, Q be r-torsion points. The computation of a bilinear pairing $e(P, Q)$ requires the construction and evaluation of a function $f_{r,P}$ such that $div(f_{r,P}) = r(P) - r(\mathcal{O})$ at a divisor $\mathcal{D}$ which is equivalent to $(Q)-(\mathcal{O})$. Miller constructs $f_{r,P}$ in stages by using a double-and-add method [14]. Let $g_{U,V} : E(\mathbb{F}_{q^k}) \rightarrow \mathbb{F}_{q^k}$ be the line equation through points U and V. If $U = V$, the line $g_{U,V}$ is the tangent to the curve at U. If $V = -U$, the line g_U is the shorthand for $g_{U,-U}$. A *Miller function* is any function $f_{c,P}$ with divisor $div(f_{c,P}) = c(P) - (cP) - (c-1)(\mathcal{O})$, $c \in \mathbb{Z}$. The following property is true for all integers $a, b \in \mathbb{Z}$ [13, Theorem 2]:

$$f_{a+b,P}(\mathcal{D}) = f_{a,P}(\mathcal{D}) \cdot f_{b,P}(\mathcal{D}) \cdot \frac{g_{aP,bP}(\mathcal{D})}{g_{(a+b)P}(\mathcal{D})}. \tag{1}$$

Direct corollaries are:

(i) $f_{1,P}(\mathcal{D}) = 1$.
(ii) $f_{a,P}(\mathcal{D}) = f_{a-1,P}(\mathcal{D}) \cdot \frac{g_{(a-1)P,P}(\mathcal{D})}{g_{aP}(\mathcal{D})}$.
(iii) $f_{2a,P}(\mathcal{D}) = f_{a,P}(\mathcal{D})^2 \cdot \frac{g_{aP,aP}(\mathcal{D})}{g_{2aP}(\mathcal{D})}$.

Miller's Algorithm is depicted in Algorithm 5. The work by Barreto et al. [13] later showed how to use the final exponentiation of the Tate pairing to eliminate the denominators involved in the algorithm and to evaluate $f_{r,P}$ at Q instead of the divisor $\mathcal{D}$. Additional optimizations published in the literature focus on minimizing the latency of the Miller loop, that is, reduce the length of r while keeping its low Hamming weight [1,15,16].

Algorithm 5. Miller's Algorithm [14].

Input: $r = \sum_{i=0}^{\log_2(r)} r_i 2^i$, P, $\mathcal{D} = (Q + R) - (R)$
Output: $f_{r,P}(\mathcal{D})$.

1: $T \leftarrow P$, $f \leftarrow 1$
2: **for** $i = \lfloor \log_2(r) \rfloor - 1$ **downto** 0 **do**
3: $\quad f \leftarrow f^2 \cdot \frac{g_{T,T}(Q+R)g_{2T}(R)}{g_{2T}(Q+R)g_{T,T}(R)}$
4: $\quad T \leftarrow 2T$
5: $\quad$ **if** $r_i = 1$ **then**
6: $\quad\quad f \leftarrow f \cdot \frac{g_{T,P}(Q+R)g_{T+P}(R)}{g_{T+P}(Q+R)g_{T,P}(R)}$
7: $\quad\quad T \leftarrow T + P$
8: $\quad$ **end if**
9: **end for**
10: **return** f

3.2 Related Work

In this work, we are interested in parallel algorithms for pairing computation with no static limits on scalability, or more precisely, algorithms in which the scalability is not restricted by the mathematical definition of the pairing. Practical limits will always exist when: (i) the communication cost is dominant; (ii) the cost of parallelization is higher than the cost of computation.

Several works already developed parallel strategies for the computation of pairings achieving mixed results. Grabher et al. [3] analyzes two approaches: parallel extension field arithmetic, which gives good results but has a clear limit on scalability; a parallel Miller loop strategy for two processors, where lines 3-4 for all iterations in Miller's Algorithm are precomputed by one processor and both processors compute in parallel the iterations where $r_i = 1$. Because r frequently has a low Hamming weight, this strategy results in performance losses due to unbalanced computational costs between the processors.

Mitsunari [17] observes that the different iterations of the algorithm can be computed in parallel if the points T of different iterations are available and proposes a specialized version of the η_T pairing over $\mathbb{F}_{3^m}$ for parallel execution in 2 processors. In this version, all the values $(x_P{}^{\frac{1}{3}{}^i}, y_P{}^{\frac{1}{3}{}^i}, x_Q{}^{3^i}, y_Q{}^{3^i})$ used for line evaluation in the i-th iteration of the algorithm are precomputed and the Miller loop iterations are divided in sets of the same size. Hence load balancing is trivially achieved. Since the cost of cubing and cube root computation is small, this approach achieves good speedups ranging from 1.61 to 1.76 at two different security levels. However, it requires significant storage overhead, since $4 \cdot (\frac{m+1}{2})$ field elements must be precomputed and stored. This approach is generalized and extended in the work by Beuchat et al. [11], where results are presented for fields of characteristic 2 and 3 at the 128-bit security level. For characteristic 2, the speedups achieved by parallel execution reach 1.75, 2.53 and 2.57 for 2, 4, and 8 processors, respectively. For characteristic 3, the speedups reach 1.65, 2.26 and 2.79, respectively. This parallelization represents the current state-of-the-art in parallel implementations of cryptographic pairings.

Cesena and Avanzi [18,19] propose a technique to compute pairings over trace zero varieties constructed from supersingular elliptic curves and extensions with degrees $a = 3$ or $a = 5$. This approach allows a pairing computation to be packed in a short parallel Miller loops by the action of the a-th power of Frobenius. The problem with this approach is again the scalability limit (restricted by the extension degree a). The speedup achieved with parallel execution in 3 processors is 1.11 over a serial implementation of the η_T pairing at the same security level [19].

3.3 Parallelization

In this section, a parallelization of Miller's Algorithm is derived. This parallelization can be used to accelerate serial pairing implementations or improve the scalability of parallel approaches restricted by the pairing definition. This formulation is similar to the parallelization presented by [17] and [11], but our method focuses on minimizing the number of points needed for parallel executions of different iterations of the algorithm. This allows us to eliminate the overhead of storing $4(\frac{m+1}{2})$ precomputed field elements.

Miller's Algorithm computes $f_{r,P}$ in $\log_2(r)$ iterations. For a parallel algorithm, we must divide these $\log_2(r)$ iterations between some number π of processors. To achieve this, first we need a simple property of Miller functions [16,20].

Lemma 1. *Let P, Q be points on $E(\mathbb{F}_q)$, $\mathcal{D} \sim (Q) - (\infty)$ and $f_{c,P}$ denote a Miller function. For all integers $a, b \in \mathbb{Z}$, $f_{a \cdot b,P}(\mathcal{D}) = f_{b,P}(\mathcal{D})^a \cdot f_{a,bP}(\mathcal{D})$.*

We need this property because Equation (1) just divides a Miller's Algorithm instance computed in $\log_2(r)$ iterations in two instances computed in at least $\log_2(r) - 1$ iterations. If we could represent r as a product $r_0 \cdot r_1$, it would be possible to compute $f_{r,P}$ in two instances of $\frac{\log_2(r)}{2}$ iterations. Since for some pairing instantiations, r is a prime group order, we write r in the simple and flexible form $2^w r_1 + r_0$, with $w \sim \frac{\log_2(r)}{2}$. This way, we can compute:

$$f_{r,P}(\mathcal{D}) = f_{2^w r_1 + r_0,P}(\mathcal{D}) = f_{2^w r_1,P}(\mathcal{D}) \cdot f_{r_0,P}(\mathcal{D}) \cdot \frac{g_{(2^w r_1)P, r_0 P}(\mathcal{D})}{g_{rP}(\mathcal{D})}. \quad (2)$$

The previous Lemma provides two choices to further develop $f_{2^w r_1,P}(\mathcal{D})$:

(i) $f_{2^w r_1,P}(\mathcal{D}) = f_{r_1,P}(\mathcal{D})^{2^w} \cdot f_{2^w, r_1 P}(\mathcal{D})$.
(ii) $f_{2^w r_1,P}(\mathcal{D}) = f_{2^w,P}(\mathcal{D})^{r_1} \cdot f_{r_1, 2^w P}(\mathcal{D})$.

The choice can be made based on efficiency: (i) compute w squarings in the extension field $\mathbb{F}_{q^k}^*$ and a point multiplication by r_1; (ii) compute an exponentiation to r_1 in the extension field and a point multiplication by 2^w (or w repeated point doublings). In the general case, the most efficient strategy will depend on the curve and embedding degree. The higher the embedding degree, the higher the cost of exponentiation in the extension field in comparison with point multiplication in the elliptic curve. If r has low Hamming weight, the two strategies should have similar costs. We adopt the first strategy:

$$f_{r,P}(\mathcal{D}) = f_{r_1,P}(\mathcal{D})^{2^w} \cdot f_{2^w, r_1 P}(\mathcal{D}) \cdot f_{r_0,P}(\mathcal{D}) \cdot \frac{g_{(2^w r_1)P, r_0 P}(\mathcal{D})}{g_{rP}(\mathcal{D})}. \quad (3)$$

This formula is clearly suitable for parallel execution in $\pi = 3$ processors, since each Miller function can be computed in $\frac{\log_2(r)}{2}$ iterations. For our purposes, however, r will have low Hamming weight and r_0 will be very small. In this case, $f_{r,P}$ can be computed by two Miller functions of approximately $\frac{\log_2(r)}{2}$ iterations. The parameter w can be adjusted to balance the costs in both processors (w extension field squarings with a point multiplication by r_1).

This formula can also be applied recursively for $f_{r_1,P}$ and f_{2^w,r_1P} to develop a parallelization suitable for any number of processors. Observe that π also does not have to be a power of 2, because of the flexible way we write r to exploit parallelism. An important detail is that a parallel implementation will only have significant speedups if the cost of the Miller loop is dominant over the communication overhead or the parallelization overhead. It is also important to note that the higher the number of processors, the higher the number of squarings and the smaller the constants r_i involved in point multiplication. However, applying the formula recursively can increase the size of the integers which multiply P, because they will be a product of r_i constants. Thus, the scalability of this algorithm for π processors depends on the cost of squarings in the extension field, the cost of point multiplications by r_i in the elliptic curve and the actual length of the Miller loop. Fortunately, these parameters are constant and can be statically determined. If P is fixed (a private key, for example), the multiples r_iP can also be precomputed and stored with low storage overhead.

3.4 Parallel η_T Pairing

In this section, the performance gain of a parallel implementation of the η_T pairing over a serial implementation is investigated following the analysis by [4].

Let E be a supersingular curve with embedding degree $k = 4$ defined over $\mathbb{F}_{2^m}$ with equation $E/\mathbb{F}_{2^m} : y^2 + y = x^3 + x + b$. The order of E is $2^m + 1 \pm 2^{\frac{m+1}{2}}$. A quartic extension is built over $\mathbb{F}_{2^m}$ with basis $\{1, s, t, st\}$, where $s^2 = s + 1$ and $t^2 = t + s$. Let $P, Q \in E(\mathbb{F}_{2^m})$ be r-torsion points. An associated distortion map ψ from $E(\mathbb{F}_{2^m})[r]$ to $E(\mathbb{F}_{2^{4m}})$ is defined by $\psi : (x, y) \to (x + s^2, y + sx + t)$. For this family of curves, Barreto et al. [1] defined the optimized η_T pairing:

$$\begin{aligned} \eta_T &: E(\mathbb{F}_{2^m})[r] \times E(\mathbb{F}_{2^{4m}})[r] \to \mathbb{F}^*_{2^{4m}}, \\ \eta_T(P, Q) &= f_{T',P'}(Q')^M, \end{aligned} \tag{4}$$

with $Q' = \psi(Q)$, $T' = (-v)(2^m - \#E(\mathbb{F}_{2^m}))$, $P' = (-v)P$, $M = (2^{2m} - 1)(2^m + 1 \pm 2^{\frac{m+1}{2}})$ for a curve-dependent parameter $v \in \{-1, 1\}$.

At the 128-bit security level, the base field must have $m = 1223$ bits [4]. Let E_1 be the supersingular curve with embedding degree $k = 4$ defined over $\mathbb{F}_{2^{1223}}$ with equation $E_1(\mathbb{F}_{2^{1223}}) : y^2 + y = x^3 + x$. The order of E_1 is $5r = 2^{1223} + 2^{612} + 1$, where r is a 1221-bit prime number. Applying the parallel form developed in Section 3.3, the pairing computation can be decomposed in:

$$f_{T',P'}(Q')^M = \left(f_{2^{612-w},P'}(Q')^{2^w} \cdot f_{2^w,2^{612-w}P'}(Q') \cdot \frac{g_{2^{612-w}P',P'}(Q')}{g_{T'P'}(Q')} \right)^M .$$

Since squarings in $\mathbb{F}_{2^{4m}}$ and point duplication in supersingular curves require only binary field squarings and these can be efficiently computed, the cost of parallelization is low, but further improvements are possible. Barreto *et al.* [1] proposed a closed formula for this pairing based on a reversed-loop approach with square roots which eliminates the extension field squarings in Miller's Algorithm. Beuchat et al. [21] encountered further algorithmic improvements and proposed a slightly faster formula for the η_T pairing computation. We can obtain a parallel algorithm directly from the parallel formula derived above by excluding the involved extension field squarings and simply dividing the loop iterations between the processors. This algorithm is shown as Algorithm 6. In this algorithm, each processor i starts the loop from the w_i counter, computing w_i squarings/square roots of overhead. Without extension field squarings to offset these operations, it makes sense to assign processor 1 the first line evaluation and to increase the loop parts executed by processors with small w_i. The total overhead is smaller because extension field squarings are not needed and point arithmetic in binary supersingular curves can be computed with inexpensive squarings and square roots. Observe that the combining step can be implemented in at least two different ways: (i) serial combining of results with $(\pi - 1)$ serial extension field multiplications executed in one processor; (ii) parallel logarithmic combining of results with latency of $\lceil \log_2(\pi) \rceil$ extension field multiplications. We adopt the parallel strategy for efficiency.

3.5 Performance Analysis

Now we proceed with performance analysis of Algorithm 6. Processor 1 has an initialization cost of 3 multiplications and 2 squarings. Processor i has a parallelization cost of $2w_i$ squarings and $2w_i$ square roots. Additional parallelization overhead is $\lceil \log_2(\pi) \rceil$ extension field multiplications to combine the results. A full extension field multiplication costs 9 field multiplications. Each iteration of the algorithm executes 2 square roots, 2 squarings, 1 field multiplication and 1 extension field multiplication. Exploring the sparsity of G_i, this extension field multiplication costs 6 field multiplications. The final exponentiation has a cost of 26 multiplications, 7 finite field squarings, 612 extension field squarings and 1 inversion. Each extension field squaring costs 4 finite field squarings [21].

Let $\widetilde{m}, \widetilde{s}, \widetilde{r}, \widetilde{i}$ be the cost of finite field operations: multiplication, squaring, square root and inversion, respectively. For our efficient implementation of finite field $\mathbb{F}_{2^{1223}}$ in an Intel Core 2 65nm processor, we have $\widetilde{r} \approx \widetilde{s}$, $\widetilde{m} \approx 25\widetilde{s}$ and $\widetilde{i} \approx 37\widetilde{m}$. From these ratios, we will illustrate how to compute the optimal w_i values which balance the computational cost between processors. Let $c_\pi(i)$ be the computational cost of a processor $0 < i \leq \pi$ while executing its portion of the parallel algorithm. For $\pi = 2$ processors:

$$c_2(1) = (3\widetilde{m} + 2\widetilde{s}) + (7\widetilde{m} + 4\widetilde{s})w_2 = 80\widetilde{s} + (186\widetilde{s})w_2$$
$$c_2(2) = (4\widetilde{s})w_2 + (7\widetilde{m} + 4\widetilde{s})\,(611 - w_2)\,.$$

Naturally, we always have $w_1 = 0$ and $w_{\pi+1} = 611$. Solving $c_2(1) = c_2(2)$ for w_2, we can obtain the optimal $w_2 = 309$. For $\pi = 4$ processors, we solve

Algorithm 6. Proposed parallelization of the η_T pairing (π processors).

Input: $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{2^m}[r])$, starting point w_i for processor i.
Output: $\eta_T(P, Q) \in \mathbb{F}^*_{2^{4m}}$.

$y_P \leftarrow y_P + 1 - \delta$
parallel section(processor i)
if $i = 0$ **then**
  $u_i \leftarrow x_P + \alpha, v_i \leftarrow x_Q + \alpha$
  $g_{0_i} \leftarrow u_i \cdot v_i + y_P + y_Q + \beta$
  $g_{1_i} \leftarrow u_i + x_Q, g_{2_i} \leftarrow v_i + x_P^2$
  $G_i \leftarrow g_{0_i} + g_{1_i} s + t$
  $L_i \leftarrow (g_{0_i} + g_{2_i}) + (g_{1_i} + 1)s + t$
  $F_i \leftarrow L_i \cdot G_i$
else
  $F_i \leftarrow 1$
end if
$x_{Q_i} \leftarrow (x_Q)^{2^{w_i}}$, $y_{Q_i} \leftarrow (y_Q)^{2^{w_i}}$
$x_{P_i} \leftarrow (x_P)^{\frac{1}{2^{w_i}}}$, $y_{P_i} \leftarrow (y_P)^{\frac{1}{2^{w_i}}}$
for $j \leftarrow w_i$ **to** $w_{i+1} - 1$ **do**
  $x_{P_i} \leftarrow \sqrt{x_{P_i}}, y_{P_i} \leftarrow \sqrt{y_{P_i}}, x_{Q_i} \leftarrow x_{Q_i}^2, y_{Q_i} \leftarrow y_{Q_i}^2$
  $u_i \leftarrow x_{P_i} + \alpha, v_i \leftarrow x_{Q_i} + \alpha$
  $g_{0_i} \leftarrow u_i \cdot v_i + y_{P_i} + y_{Q_i} + \beta$
  $g_{1_i} \leftarrow u_i + x_{Q_i}$
  $G_i \leftarrow g_{0_i} + g_{1_i} s + t$
  $F_i \leftarrow F_i \cdot G_i$
end for
$F \leftarrow \prod_{i=0}^{\pi} F_i$
end parallel
return F^M

$c_4(1) = c_4(2) = c_4(3) = c_4(4)$ to obtain $w_2 = 158, w_3 = 312, w_4 = 463$. Observe that by solving a simple system of equations it is always possible to balance the computational cost between the processors. Furthermore, the latency of the Miller loop will always be equal to $c_\pi(1)$. Let $c_1(1)$ be the cost of a serial implementation of the main loop, *par* be the parallelization overhead and *exp* be the cost of final exponentiation. Considering the additional $\lceil \log_2(\pi) \rceil$ extension field multiplications as parallelization overhead and $26\widetilde{m} + (7 + 2446)\widetilde{s} + \widetilde{i}$ as the cost of final exponentiation, the speedup for π processors is the ratio between the cost of the serial implementation over the cost of the parallel implementation:

$$s(\pi) = \frac{c_1(1) + exp}{c_\pi(1) + par + exp} = \frac{77 + 179 \cdot 611 + 3978}{c_\pi(1) + 225\lceil \log_2(\pi) \rceil + 3978}.$$

Table 2 presents speedups estimated by our performance analysis. Note that our efficient implementation of binary field arithmetic in a 45nm processor has a bigger multiplication-to-squaring ratio, concentrating higher computational costs in the main loop of the algorithm. This explains why the speedups should be higher in the 45nm processor.

Table 2. Estimated speedups for our parallelization of the η_T pairing over supersingular binary curves at the 128-bit security level. The optimal partitions were computed by a Sage[1] script.

	Number π of processors					
Estimated speedup $s(\pi)$	**1**	**2**	**4**	**8**	**16**	**32**
Core 2 65nm	1.00	1.90	3.45	5.83	8.69	11.48
Core 2 45nm	1.00	1.92	3.54	6.11	9.34	12.66

4 Experimental Results

We implemented the parallel algorithm for the η_T pairing over our efficient binary field arithmetic in two Intel Core platforms: an Intel Core 2 Quad 65nm platform running at 2.4GHz (Platform 1) and a dual quad-core Intel Xeon 45nm processor running at 2.0GHz (Platform 2). The parallel sections were implemented with OpenMP[2] constructs. OpenMP is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran. We used a special version of the GCC 4.1.2 compiler included in Fedora Linux 8 with OpenMP support backported from GCC 4.2 and SSSE3 support backported from GCC 4.3. This way, we could use both multiprocessing support and fast code generation for SSE intrinsics.

The timings and speedups presented in Table 3 were measured on 10^4 executions of each algorithm. We present timings in millions of cycles to ignore differences in clock frequency between the target platforms. From the table, we can observe that real implementations can obtain speedups close to the estimated speedups derived in the previous section. We verified that threading creation and synchronization overhead stayed in the order of microseconds, being negligible compared to the pairing computation time. Timings for $\pi > 4$ processors in Platform 1 and $\pi > 8$ processors in Platform 2 were measured through a high-precision per-thread counter measured by the main thread. These timings might be an accurate approximation of future real implementations, but memory effects (such as cache locality) or scheduling influence may impose penalties.

Table 3 shows that the proposed parallelization presents good scalability. We improve the state-of-the-art serial and parallel execution times significantly. The fastest timing for computing the η_T pairing obtained by our implementation was 1.51 milliseconds using all 8 cores of Platform 2. The work by Beuchat *et al.* [11] reports a timing of 3.08 milliseconds in a Intel Core i7 45nm processor clocked at 2.9GHz. Note that we obtain a much faster timing with a lower clock frequency and without requiring the storage overhead of $4 \cdot (\frac{m+1}{2})$ field elements present in [11], which may reach 365KB for these parameters and be prohibitive in resource-constrained embedded devices.

[1] SAGE: Software for Algebra and Geometry Experimentation, `http://www.sagemath.org`

[2] Open Multi-Processing, `http://www.openmp.org`

Table 3. Experimental results for serial/parallel executions of the η_T pairing. Times are presented in millions of cycles and the speedups are computed by the ratio between execution times of serial implementations over execution times of parallel implementations. The columns marked with (*) present estimates based on per-thread data.

	Number of threads					
Platform 1 – Intel Core 2 65nm	**1**	**2**	**4**	**8***	**16***	**32***
Hankerson et al. [4] – latency	39	–	–	–	–	–
Beuchat et al. [11] – latency	26.86	16.13	10.13	–	–	–
Beuchat et al. [11] – speedup	1.00	1.67	2.65	–	–	–
This work – latency	18.76	10.08	5.72	3.55	2.51	2.14
This work – speedup	1.00	1.86	3.28	5.28	7.47	8.76
Improvement	30.2%	32.9%	39.9%	–	–	–
Platform 2 – Intel Core 2 45nm	**1**	**2**	**4**	**8**	**16***	**32***
Beuchat et al. [11] – latency	23.03	13.14	9.08	8.93	–	–
Beuchat et al. [11] – speedup	1.00	1.77	2.54	2.58	–	–
This work – latency	17.40	9.34	5.08	3.02	2.03	1.62
This work – speedup	1.00	1.86	3.42	5.76	8.57	10.74
Improvement	24.4%	28.9%	44.0%	66.2%	–	–

5 Conclusion and Future Work

In this work, we proposed novel techniques for exploring parallelism during the implementation of the η_T pairing over supersingular binary curves in modern multi-core computers. Powerful vector instructions of the SSE family were shown to accelerate considerably the arithmetic in binary fields. We obtained significant performance in computing the η_T pairing, using an efficient implementation of field multiplication, squaring and square root computation. The optimizations improved the state-of-the-art timings of this pairing instantiation at the 128-bit security level by 24% and 30% in two different Intel Core processors.

We also derived a parallelization of Miller's Algorithm to compute pairings. This parallelization is generic and can be applied to any pairing algorithm or instantiation. The construction also achieves good scalability in the symmetric case and this scalability is not restricted by the definition of the pairing. We illustrated the formulation when applied to the η_T pairing over supersingular binary curves and validated our performance analysis with a real implementation. The experimental results show that the actual implementation could sustain performance gains close to the estimated speedups. Parallel execution of the η_T pairing improved the state-of-the-art timings by at least 28%, 44% and 66% in 2, 4 and 8 cores respectively. This parallelization is suitable for embedded platforms and can be applied to reduce computation latency when response time is critical.

Future work can adapt the introduced techniques for the case $\mathbb{F}_{3^m}$. Improvements to the parallelization should focus on minimizing the serial region and parallelization cost. The proposed parallelization should also be applied to an optimal asymmetric pairing setting, where parallelization costs are clearly higher. Preliminary data for the R-ate pairing [16] over Barreto-Naehrig curves at the 128-bit security level points to a 10% speedup using 2 processor cores.

References

1. Barreto, P.S.L.M., Gailbraith, S., Ó hÉigeartaigh, C., Scott, M.: Efficient Pairing Computation on Supersingular Abelian Varieties. Design, Codes and Cryptography 42(3), 239–271 (2007)
2. Wechsler, O.: Inside Intel Core Microarchitecture: Setting new standards for energy-efficient performance. Technology@Intel Magazine (2006)
3. Grabher, P., Groszschaedl, J., Page, D.: On Software Parallel Implementation of Cryptographic Pairings. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) Selected Areas in Cryptography. LNCS, vol. 5381, pp. 34–49. Springer, Heidelberg (2009)
4. Hankerson, D., Menezes, A., Scott, M.: Identity-Based Cryptography, ch. 12, pp. 188–206. IOS Press, Amsterdam (2008)
5. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, `http://www.intel.com/Assets/PDF/manual/253666.pdf`
6. Gueron, S., Kounavis, M.E.: Carry-Less Multiplication and Its Usage for Computing The GCM Mode. White paper, `http://software.intel.com/`
7. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, Secaucus (2003)
8. Fong, K., Hankerson, D., López, J., Menezes, A.: Field inversion and point halving revisited. IEEE Transactions on Computers 53(8), 1047–1059 (2004)
9. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers (in Russian). Doklady Akad. Nauk SSSR 145, 293–294 (1962)
10. López, J., Dahab, R.: High-speed software multiplication in GF(2^m). In: Roy, B., Okamoto, E. (eds.) INDOCRYPT 2000. LNCS, vol. 1977, pp. 203–212. Springer, Heidelberg (2000)
11. Beuchat, J., López-Trejo, E., Martínez-Ramos, L., Mitsunari, S., Rodríguez-Henríquez, F.: Multi-core implementation of the Tate pairing over supersingular elliptic curves. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 413–432. Springer, Heidelberg (2009)
12. Barreto, P.S.L.M., Lynn, B., Scott, M.: Efficient Implementation of Pairing-Based Cryptosystems. Journal of Cryptology 17(4), 321–334 (2004)
13. Barreto, P.S.L.M., Kim, H.Y., Lynn, B., Scott, M.: Efficient algorithms for pairing-based cryptosystems. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 354–368. Springer, Heidelberg (2002)
14. Miller, V.S.: The Weil Pairing, and Its Efficient Calculation. Journal of Cryptology 17(4), 235–261 (2004)
15. Hess, F., Smart, N.P., Vercauteren, F.: The eta pairing revisited. IEEE Trans. on Information Theory 52, 4595–4602 (2006)
16. Lee, H., Lee, E., Park, C.: Efficient and Generalized Pairing Computation on Abelian Varieties. IEEE Trans. on Information Theory 55(4), 1793–1803 (2009)
17. Mitsunari, S.: A Fast Implementation of η_T Pairing in Characteristic Three on Intel Core 2 Duo Processor. Cryptology ePrint Archive, Report 2009/032 (2009)
18. Cesena, E.: Pairing with Supersingular Trace Zero Varieties Revisited. Cryptology ePrint Archive, Report 2008/404 (2008)
19. Cesena, E., Avanzi, R.: Trace Zero Varieties in Pairing-based Cryptography. In: Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc. (2009), `http://inst-mat.utalca.cl/chile2009/Slides/Roberto_Avanzi_2.pdf`
20. Vercauteren, F.: Optimal pairings. Cryptology ePrint Archive, Report 2008/096 (2008)
21. Beuchat, J., Brisebarre, N., Detrey, J., Okamoto, E., Rodríguez-Henríquez, F.: A Comparison Between Hardware Accelerators for the Modified Tate Pairing over $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$. In: Galbraith, S.D., Paterson, K.G. (eds.) Pairing 2008. LNCS, vol. 5209, pp. 297–315. Springer, Heidelberg (2008)