# New software speed records for cryptographic pairings

Michael Naehrig, Ruben Niederhagen, Peter Schwabe



August 9, 2010

Latincrypt 2010, Puebla, México

# Apologies



- Mistake in the paper as appeared in the proceedings
- Wrong choice of curve parameters
- Corrected in current version online
- Software (of course) also corrected

Thanks to Francisco for pointing this out.



- Let  $G_1, G_2$ , and  $G_3$  be finite abelian groups.
- A pairing is a bilinear, nondegenerate map

 $e:G_1\times G_2\to G_3$ 



- Let  $G_1, G_2$ , and  $G_3$  be finite abelian groups.
- A pairing is a bilinear, nondegenerate map

 $e:G_1 \times G_2 \to G_3$ 

Different pairings derived from the Tate pairing



- Let  $G_1, G_2$ , and  $G_3$  be finite abelian groups.
- A pairing is a bilinear, nondegenerate map

 $e:G_1 \times G_2 \to G_3$ 

- Different pairings derived from the Tate pairing
- ► For practical applications: based on elliptic-curve arithmetic
- Need "special" curves



- Let  $G_1, G_2$ , and  $G_3$  be finite abelian groups.
- A pairing is a bilinear, nondegenerate map

 $e:G_1\times G_2\to G_3$ 

- Different pairings derived from the Tate pairing
- ► For practical applications: based on elliptic-curve arithmetic
- Need "special" curves
- ► For 128-bit security level: Barreto-Naehrig curves (BN curves)



- ▶ Let G<sub>1</sub>, G<sub>2</sub>, and G<sub>3</sub> be finite abelian groups.
- A pairing is a bilinear, nondegenerate map

 $e:G_1\times G_2\to G_3$ 

- Different pairings derived from the Tate pairing
- ► For practical applications: based on elliptic-curve arithmetic
- Need "special" curves
- ► For 128-bit security level: Barreto-Naehrig curves (BN curves)
- Currently fastest: optimal ate pairing, r-ate pairing

TU/e Technische Universiteit Eindhoven University of Technology

"Definition" for this talk

The ate pairing over a BN curve is a sequence of operations in a field  $\mathbb{F}_{p^2}$ 

"Definition" for this talk

The ate pairing over a BN curve is a sequence of operations in a field  $\mathbb{F}_{p^2}$ 

 $\blacktriangleright$  BN-curve construction: Find u such that

$$p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$$

$$n = n(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$$

are prime

 $\blacktriangleright$  For 128-bit security level: n should have 256 bits, p will also have 256 bits

TU

#### "Definition" for this talk

The ate pairing over a BN curve is a sequence of operations in a field  $\mathbb{F}_{p^2}$ 

 $\blacktriangleright$  BN-curve construction: Find u such that

$$p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$$

$$n = n(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$$

are prime

- $\blacktriangleright$  For 128-bit security level: n should have 256 bits, p will also have 256 bits
- The choice of u influences the sequence of operations in  $\mathbb{F}_{p^2}$
- Details on high-level algorithms in the paper

TU

#### "Definition" for this talk

The ate pairing over a BN curve is a sequence of operations in a field  $\mathbb{F}_{p^2}$ 

 $\blacktriangleright$  BN-curve construction: Find u such that

$$p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$$

$$n = n(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$$

are prime

- $\blacktriangleright$  For 128-bit security level: n should have 256 bits, p will also have 256 bits
- The choice of u influences the sequence of operations in  $\mathbb{F}_{p^2}$
- Details on high-level algorithms in the paper

Question: Can we exploit the special shape of p for faster arithmetic in  $\mathbb{F}_p$  or  $\mathbb{F}_{p^2}?$ 

TU

# Exploiting the shape of p



#### Answer 1, Scott, May 2009

"exploit the form of p(x) for a faster modular arithmetic [...] Not really demonstrated successfully, (except recently in hardware?)."

# Exploiting the shape of p



#### Answer 1, Scott, May 2009

"exploit the form of p(x) for a faster modular arithmetic [...] Not really demonstrated successfully, (except recently in hardware?)."

## Answer 2, Fan, Vercauteren, Verbauwhede, Sep. 2009

Faster  $\mathbb{F}_p$ -arithmetic for Cryptographic Pairings on Barreto-Naehrig Curves, presented at CHES 2009, exploiting the special structure of p in hardware



#### Answer 1, Scott, May 2009

"exploit the form of p(x) for a faster modular arithmetic [...] Not really demonstrated successfully, (except recently in hardware?)."

## Answer 2, Fan, Vercauteren, Verbauwhede, Sep. 2009

Faster  $\mathbb{F}_p$ -arithmetic for Cryptographic Pairings on Barreto-Naehrig Curves, presented at CHES 2009, exploiting the special structure of p in hardware

How about software?

#### Polynomial representation (Inspired by Bernstein's curve25519 paper)



Consider the ring  $R = \mathbb{Z}[x] \cap \overline{\mathbb{Z}}[\sqrt{6}ux]$  and the element

$$P = 36u^4x^4 + 36u^3x^3 + 24u^2x^2 + 6ux + 1$$
  
=  $(\sqrt{6}ux)^4 + \sqrt{6}(\sqrt{6}ux)^3 + 4(\sqrt{6}ux)^2 + \sqrt{6}(\sqrt{6}ux) + 1.$ 

Then P(1) = p.

#### Polynomial representation (Inspired by Bernstein's curve25519 paper)



Consider the ring  $R = \mathbb{Z}[x] \cap \overline{\mathbb{Z}}[\sqrt{6}ux]$  and the element

$$P = 36u^4x^4 + 36u^3x^3 + 24u^2x^2 + 6ux + 1$$
  
=  $(\sqrt{6}ux)^4 + \sqrt{6}(\sqrt{6}ux)^3 + 4(\sqrt{6}ux)^2 + \sqrt{6}(\sqrt{6}ux) + 1.$ 

Then P(1) = p.

• Represent  $f \in \mathbb{F}_p$  as polynomial in R:

$$F = f_0 + f_1 \cdot \sqrt{6}(\sqrt{6}ux) + f_2 \cdot (\sqrt{6}ux)^2 + f_3 \cdot \sqrt{6}(\sqrt{6}ux)^3$$
  
=  $f_0 + f_1 \cdot (6u)x + f_2 \cdot (6u^2)x^2 + f_3 \cdot (36u^3)x^3$ 

▶ Then: f = F(1)

• For implementation needs to store 4 coefficients  $f_0, f_1, f_2, f_3$ .

Polynomial multiplication of f and g yields 7 coefficients  $t_0, \ldots, t_6$ Reduction mod p to  $r_0, \ldots, r_3$ :

$$r_{0} \leftarrow t_{0} - t_{4} + 6t_{5} - 2t_{6}$$
  

$$r_{1} \leftarrow t_{1} - t_{4} + 5t_{5} - t_{6}$$
  

$$r_{2} \leftarrow t_{2} - 4t_{4} + 18t_{5} - 3t_{6}$$
  

$$r_{3} \leftarrow t_{3} - t_{4} + 2t_{5} + 3t_{6}$$

TU

## Four coefficients are not enough



- > 256-bit numbers in 4 coefficients: Each coefficient 64 bits
- Coefficients do not have exactly the same size
- Small multiples in the reduction are larger than 128 bits
- Easy to realize in hardware, not in software
- For software we need more coefficients

## Four coefficients are not enough



- Coefficients do not have exactly the same size
- Small multiples in the reduction are larger than 128 bits
- Easy to realize in hardware, not in software
- For software we need more coefficients
- ▶ Idea: Consider  $u = v^3$ , use 12 coefficients  $f_0, \ldots, f_{11}$

$$\begin{split} f = & f_0 + 6vf_1 + 6v^2f_2 + 6v^3f_3 + 6v^4f_4 + 6v^5f_5 + 6v^6f_6 + \\ & 36v^7f_7 + 36v^8f_8 + 36v^9f_9 + 36v_{10}f_{10} + 36v^{11}f_{11} \end{split}$$

TU

## Four coefficients are not enough



- Coefficients do not have exactly the same size
- Small multiples in the reduction are larger than 128 bits
- Easy to realize in hardware, not in software
- For software we need more coefficients
- ▶ Idea: Consider  $u = v^3$ , use 12 coefficients  $f_0, \ldots, f_{11}$

$$f = f_0 + 6vf_1 + 6v^2f_2 + 6v^3f_3 + 6v^4f_4 + 6v^5f_5 + 6v^6f_6 + 36v^7f_7 + 36v^8f_8 + 36v^9f_9 + 36v_{10}f_{10} + 36v^{11}f_{11}$$

- v has about 21 bits, products have about 42 bits
- Double-precision floats have 53-bit mantissa
- Use double-precision floats, still some space to add up coefficients and compute small multiples

TU

Technische Universiteit

# Reducing coefficients



- At some point the coefficients will *overflow* (become larger than 53 bits)
- Need to do coefficient reduction (carry)

## Reducing coefficients



- At some point the coefficients will overflow (become larger than 53 bits)
- Need to do coefficient reduction (carry)
- ► Carry from  $f_0$  to  $f_1$   $c \leftarrow \mathsf{round}(f_0/6v)$   $f_0 \leftarrow f_0 - c \cdot 6v$  $f_1 \leftarrow f_1 + c$
- ► Carry from  $f_1$  to  $f_2$   $c \leftarrow \text{round}(f_1/v)$   $f_1 \leftarrow f_1 - c \cdot v$  $f_2 \leftarrow f_2 + c$

# Reducing coefficients



- At some point the coefficients will overflow (become larger than 53 bits)
- Need to do coefficient reduction (carry)
- ► Carry from  $f_0$  to  $f_1$   $c \leftarrow \mathsf{round}(f_0/6v)$   $f_0 \leftarrow f_0 - c \cdot 6v$  $f_1 \leftarrow f_1 + c$
- ► Carry from  $f_1$  to  $f_2$   $c \leftarrow \text{round}(f_1/v)$   $f_1 \leftarrow f_1 - c \cdot v$  $f_2 \leftarrow f_2 + c$
- $f_0 \in [-3v, 3v], f_1 \in [-v/2, v/2]$
- Carry from  $f_{11}$  goes to  $f_0, f_3, f_6$ , and  $f_9$



- Use fast vector instructions mulpd and addpd
- 2 multiplications/ 2 additions in one instruction
- 1 mulpd and 1 addpd (and one mov) per cycle



- Use fast vector instructions mulpd and addpd
- 2 multiplications/ 2 additions in one instruction
- 1 mulpd and 1 addpd (and one mov) per cycle
- Problem:  $\mathbb{F}_p$  arithmetic requires a lot of shuffeling, combining etc.

- Use fast vector instructions mulpd and addpd
- 2 multiplications/ 2 additions in one instruction
- 1 mulpd and 1 addpd (and one mov) per cycle
- Problem:  $\mathbb{F}_p$  arithmetic requires a lot of shuffeling, combining etc.
- Solution: Implement arithmetic in  $\mathbb{F}_{p^2}$
- Use schoolbook multiplication in  $\mathbb{F}_{p^2}$  yielding 4 multiplications in  $\mathbb{F}_p$
- $\blacktriangleright$  For squaring in  $\mathbb{F}_{p^2}$  use complex method: 2 multiplications in  $\mathbb{F}_p$
- Perform 2  $\mathbb{F}_p$  multiplications in parallel using vector instructions

- Use fast vector instructions mulpd and addpd
- 2 multiplications/ 2 additions in one instruction
- 1 mulpd and 1 addpd (and one mov) per cycle
- Problem:  $\mathbb{F}_p$  arithmetic requires a lot of shuffeling, combining etc.
- Solution: Implement arithmetic in  $\mathbb{F}_{p^2}$
- Use schoolbook multiplication in  $\mathbb{F}_{p^2}$  yielding 4 multiplications in  $\mathbb{F}_p$
- For squaring in  $\mathbb{F}_{p^2}$  use complex method: 2 multiplications in  $\mathbb{F}_p$
- Perform 2  $\mathbb{F}_p$  multiplications in parallel using vector instructions
- $\mathbb{F}_p$  polynomial reduction after  $\mathbb{F}_{p^2}$  polynomial reduction
- ▶ Only two  $\mathbb{F}_p$  polynomial reductions and two coefficient reductions per multiplication in  $\mathbb{F}_{p^2}$
- Those reductions also done in SIMD way

## Detecting and avoiding overflows



- After each multiplication we need to reduce coefficients
- Sometimes also before a multiplication after several additions
- Problem: How to detect where?
- Need to detect overflow in the worst case

## Detecting and avoiding overflows



- After each multiplication we need to reduce coefficients
- Sometimes also before a multiplication after several additions
- Problem: How to detect where?
- Need to detect overflow in the worst case
- Implement software in C
- Replace double with C++ class CheckDouble
- Perform arithmetic on values and in parallel on worst-case values
- Abort at overflow (allows backtrace in debugger)

## Detecting and avoiding overflows



- After each multiplication we need to reduce coefficients
- Sometimes also before a multiplication after several additions
- Problem: How to detect where?
- Need to detect overflow in the worst case
- Implement software in C
- Replace double with C++ class CheckDouble
- > Perform arithmetic on values and in parallel on worst-case values
- Abort at overflow (allows backtrace in debugger)
- Re-implement algorithms in assembly (qhasm)
- Would be good to have overflow checks in assembly

Results



#### Performance of dclxvi software

- Cycles on an Intel Core 2 Quad Q6600 (65 nm): 4,134,643 cycles
- ► Similar for other 64-bit Intel processors

Results



#### Performance of dclxvi software

- Cycles on an Intel Core 2 Quad Q6600 (65 nm): 4,134,643 cycles
- Similar for other 64-bit Intel processors
- Comparison: Fastest published pairing benchmark (on one core) before: 10,000,000 cycles on a Core 2 by Hankerson, Menezes, Scott, 2008
- Unpublished: 7,850,000 cycles on a Core 2 T5500 (Scott 2010)

# Even faster pairings



New paper by Beuchat, González Díaz, Mitsunari, Okamoto, Rodríguez-Henríquez, and Teruya: *"High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves"* Claims: 2,490,000 cycles on a Core i7, 3,140,000 cycles on a Core 2 with Visual Studio 2008

# Even faster pairings



New paper by Beuchat, González Díaz, Mitsunari, Okamoto, Rodríguez-Henríquez, and Teruya: *"High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves"* Claims: 2,490,000 cycles on a Core i7, 3,140,000 cycles on a Core 2 with Visual Studio 2008

Cycle counts on a Core 2 Q6600 with gcc-4.3.3

	dclxvi	[BGM+10]
multiplication in $\mathbb{F}_{p^2}$	$\sim 585$	$\sim 588$
squaring in $\mathbb{F}_{p^2}$	$\sim 359$	$\sim 487$
optimal ate pairing	$\sim 4, 135,000$	$\sim 3,269,000$



Three reasons why we are slower



Three reasons why we are slower

1. Restricted choice of u: Need more operations in  $\mathbb{F}_{p^2}$ 



#### Three reasons why we are slower

- 1. Restricted choice of u: Need more operations in  $\mathbb{F}_{p^2}$
- 2. Additional coefficient reductions take quite a bit of time (> 400,000 cycles)



#### Three reasons why we are slower

- 1. Restricted choice of u: Need more operations in  $\mathbb{F}_{p^2}$
- 2. Additional coefficient reductions take quite a bit of time (> 400,000 cycles)
- 3. Multiplication is not (much) faster

# Why is our multiplication not faster?

- Fast multiplication (and squaring) was the target of our implementation
- Always need to perform even number of  $\mathbb{F}_p$  multiplications
- Have to use schoolbook instead of Karatsuba in  $\mathbb{F}_{p^2}$
- 4 instead of 3 multiplications in  $\mathbb{F}_p$

TU

# Why is our multiplication not faster?

- Fast multiplication (and squaring) was the target of our implementation
- Always need to perform even number of  $\mathbb{F}_p$  multiplications
- Have to use schoolbook instead of Karatsuba in  $\mathbb{F}_{p^2}$
- 4 instead of 3 multiplications in  $\mathbb{F}_p$
- Using vector instructions still requires quite some shuffeling
- Overhead: 60 cycles per  $\mathbb{F}_{p^2}$  multiplication

TU

Technische Universiteit

Ihoven versity of Technology

# Conclusion



- Fastest (current) implementation based on double-precision floating-point arithmetic exploits special p
- On Intel (and AMD) processors: integer-based approach (with Montgomery arithmetic) is faster
- But: several architectures have much faster double-precision floating-point than integer arithmetic





Paper: http://cryptojedi.org/users/peter/#dclxvi
Software: http://cryptojedi.org/crypto/#dclxvi
(public domain)