Efficient Techniques for High-Speed Elliptic Curve Cryptography

Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010)

Patrick Longa
University of Waterloo

Joint work with C. Gebotys

Outline

- Elliptic Curve Cryptography (ECC):
  - Basics and recent developments
- x86-64 based Processors
- Approach
- Optimizations
  - Scalar, point and field arithmetic levels
- Optimizations with the GLS Method
- Implementation Results
- Conclusions and References

ECC: Basics

- An elliptic curve E over a prime field $\mathbb{F}_p$, $p > 3$, in (short) Weierstrass form is given by:

$$E: \; y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ ($a = -3$ for efficiency purposes)
Given a point $P \in E(\mathbb{F}_p)$ of order r and an integer $k \in [1, r-1]$, we define scalar multiplication as:

$$Q = [k]P = P + P + \ldots + P \;\; (k \text{ times})$$

- Scalar multiplication is the central/most time-consuming operation in ECC
- Security is based on the ECDLP problem: given points P and Q, find k
- Only exponential attacks are known for solving ECDLP

ECC: Recent developments

- **Curve forms with faster arithmetic**
  An elliptic curve E over a prime field $\mathbb{F}_p$, $p > 3$, in Twisted Edwards form is given by, Bernstein et al. (2008):

$$E: \ ax^2 + y^2 = 1 + dx^2 y^2$$

  where $a, d \in \mathbb{F}_p^*$, $a \neq d$ ($a = -1$ for efficiency purposes)

- **The Galbraith-Lin-Scott (GLS) method**, Galbraith et al. (Eurocrypt 2009)
  Let E be an elliptic curve over $\mathbb{F}_p$, s.t. the quadratic twist E' of $E(\mathbb{F}_{p^2})$ has an efficiently computable homomorphism $\psi(x,y) \rightarrow (\alpha x, \alpha y)$, $\psi(P) = \lambda P$

  Then: $$[k]P = [k_0]P + [k_1](\lambda P)$$

  where $\log k_0 \approx \log k_1 \approx$ ½ $\log k$

x86-64 based Processors

Computers from laptop/desktop/server classes are rapidly adopting x86-64 ISA (wordlength $w = 64$)

Main features:

- 64-bit GPRs and operations with powerful multiplier $\Rightarrow$ favours $\mathbb{F}_p$ arithmetic
- Deeply pipelined architectures (e.g., Intel Core 2 Duo: 15 stages)
- Aggressive out-of-order scheduling to exploit *Instruction Level Parallelism* (ILP)
- Sophisticated branch predictors

Key observation:
As $w \uparrow$, $\lceil (\log p)/w \rceil \downarrow$, number of stages in pipeline gets larger and scheduling gets more "aggressive", "negligible" operations/issues get significant: addition, subtraction, division/ multiplication by constants, pipeline stalls (by data dependencies) and branch mispredictions

Approach

- Bottom-up optimization of each layer of ECC computation taking into account architectural features of x86-64 based processors

- Best ECC algorithms (to our knowledge) for each layer are identified and optimized

- *Three* representative 64-bit processors for our analysis and tests:

  - 1.66GHz Intel Atom N450 (netbook/notebook class) New!
  - 2.66GHz Intel Core 2 Duo E6750 (desktop class)
  - 2.6GHz AMD Opteron 252 (server/workstation class)

Field Arithmetic

Incomplete Reduction (IR), Yanik et al. (2002):

Given $a, b \in [0, p-1]$, allow the result to stay in the range $[0, 2^s - 1]$ instead of performing a complete reduction, where $p < 2^s < 2p - 1$, $s = n.w$ (n: number of words, w: wordlength)

- For maximal efficiency, select a pseudo-Mersenne prime $p = 2^m - c$, where $m = s$, c small (i.e., $c << 2^w$):
  - Reduction after addition $a + b$: discard carry bit in most significant word and then add c
  - Subtraction does not require IR (already optimal!)
- However, other operations may benefit from IR: addition between *completely reduced* and *incompletely reduced* numbers, multiplication by constant, division by constant,…

Field Arithmetic

Conditional branches

- Modular operations are traditionally implemented with conditional branches
- Example: addition

Given $a, b \in [0, p-1]$, execute $a+b$. If $a+b>p$, then $a+b-p$

- Condition is true ~50% in a random pattern $\Rightarrow$ worst "nightmare" of predictors
- We'd better eliminate conditional branches in modular reduction.
  Two alternatives:
  - Using predicated move instructions (e.g., `cmov` in x86)
  - Using look-up tables and indexed indirect addressing
- Basic idea: perform reduction with 0 when it is not actually required

Field Arithmetic

Incomplete Reduction and Conditional branches

Cost (in cycles) of 256-bit modular operations, $p = 2^{256} - 189$

Modular operation	Intel Core 2 Duo w/o CB	Intel Core 2 Duo with CB	Intel Core 2 Duo Cost reduction (%)	AMD Opteron w/o CB	AMD Opteron with CB	AMD Opteron Cost reduction (%)
Sub	**21**	37	43%	**16**	23	30%
Add with IR	**20**	37	46%	**13**	21	38%
Add	25	39	36%	20	23	13%
Mult2 with IR	**19**	38	50%	**10**	19	47%
Mult2	24	38	37%	17	20	15%
Div2 with IR	**20**	36	44%	**11**	18	39%
Div2	25	39	36%	18	27	33%

⇒ Cost reductions using IR in the range 7% - 41%

⇒ Cost reductions by eliminating conditional branches as high as 50%

⇒ Operations using IR are more benefited

Field Arithmetic

"Contiguous" dependencies: RAW dependencies between successive field operations

Field Operations

```
⋮
> Add(op1,op2,res1)
> Add(res1,op3,res2)
⋮
```

Assembly instructions

```
⋮
> addq %rcx,%r8
> movq %r8, 8(%rdx)
> adcq $0,%r9
> movq %r9, 16(%rdx)
> adcq $0,%r10
> movq %r10,24(%rdx)
> adcq $0,%r11
> movq %r11,32(%rdx)
> xorq %rax,%rax
> movq $0xBD,%rcx
> movq 8(%rdi) ,%r8
> addq 8(%rsi) ,%r8
> movq 16(%rdi),%r9
> adcq 16(%rsi),%r9
> movq 24(%rdi),%r10
> adcq 24(%rsi),%r10
> movq 32(%rdi),%r11
> adcq 32(%rsi),%r11
⋮
```

ρ : "distance" between instructions

"Ideal" non-superscalar CPU:

Pipeline stalls for ~$(\delta_{write} - \rho)$ cycles

δ_{write} : pipeline latency of write instruction

Field Arithmetic

"Contiguous" dependencies (*Cont'd*)

We propose *three* solutions:

1. Field arithmetic scheduling $\Rightarrow$ execute other field operations while previous memory writings complete their pipeline latencies

2. Merging point operations $\Rightarrow$ more possibilities for field operation rescheduling (it additionally reduces number of function calls)

3. Merging field operations $\Rightarrow$ direct elimination of "contiguous" dependencies (it additionally reduces memory reads/writes)

   E.g., $a-b-c \pmod{p}$, $a+a+a \pmod{p}$ (as in other crypto libraries, MIRACL)
   $a-2b \pmod{p}$, merging of $a-b \pmod{p}$ and $(a-b)-2c \pmod{p}$

Field Arithmetic

"Contiguous" dependencies *(Cont'd)* $(X_1,Y_1,Z_1) \leftarrow 2(X_1,Y_1,Z_1)$

> Sqr(Z1,t3)		> Sqr(Z1,t3)		> Sqr(Z1,t3)	
> Add(X1,t3,t1)	**D**	> Sqr(Y1,t2)		> Sqr(Y1,t2)	
> Sub(X1,t3,t3)		> Add(X1,t3,t1)		> Add(X1,t3,t1)	
> Mult(t3,t1,t2)	**D**	> Sub(X1,t3,t3)		> Sub(X1,t3,t3)	
> Mult3(t2,t1)	**D**	> Mult3(t3,t0)	**D**	> Mult3(t3,t0)	**D**
> Div2(t1,t1)	**D**	> Mult(X1,t2,t4)		> Mult(X1,t2,t4)	
> Mult(Y1,Z1,t3)		> Mult(t1,t0,t3)		> Mult(t1,t0,t3)	
> Sqr(Y1,t2)		> Sqr(t2,t0)		> Sqr(t2,t0)	
> Mult(t2,X1,t4)	**D**	> Div2(t3,t1)		> Div2(t3,t1)	
> Sqr(t1,t3)		> Mult(Y1,Z1,Z1)		> Mult(Y1,Z1,Z1)	
> Sub(t3,t4,X1)	**D**	> Sqr(t1,t2)		> Sqr(t1,t2)	
> Sub(X1,t4,X1)	**D**	> DblSub(t2,t4,X1)	**D**	**> Sqr(Z1,t3)**	
> Sub(t4,X1,t3)	**D**	> Sub(t4,X1,t2)	**D**	> DblSub(t2,t4,X1)	
> Mult(t3,t1,t4)	**D**	> Mult(t1,t2,t4)	**D**	> Sub(t4,X1,t2)	**D**
> Sqr(t2,t0)		> Sub(t4,t0,Y1)	**D**	**> Add(X1,t3,t5)**	
> Sub(t4,t0,Y1)	**D**			> Mult(t1,t2,t4)	
				> Sub(X1,t3,t3)	
				> Sub(t4,t0,Y1)	
				> Mult3(t3,t1)	
				> Sqr(Y1,t2)	
				> …	
"Unscheduled"		Scheduled		Scheduled and merged DBL-DBL	

Field Arithmetic

"Contiguous" dependencies (*Cont'd*)

Cost (in cycles) of point doubling, $p = 2^{256} - 189$

	Intel Atom		Intel Core 2 Duo		AMD Opteron	
Point operation	**"Unscheduled"**	**Scheduled and merged**	**"Unscheduled"**	**Scheduled and merged**	**"Unscheduled"**	**Scheduled and merged**
DBL	3390	3332	1115	979	786	726
Relative reduction	-	2%	-	12%	-	8%
Estimated reduction for $[k]P$	-	1%	-	9%	-	5%

⇒ Estimated reduction of 5% and 9% on AMD Opteron and Intel Core 2 Duo, respect.

⇒ Less "aggressive" architectures are not greatly affected by "contiguous" dependencies

Point Arithmetic

Our choice of formulas:

- Jacobian coordinates: $(x, y) \mapsto (X/Z^2, Y/Z^3, 1)$, $(X : Y : Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z) : \lambda \in \mathbb{F}_p^{*}\}$

$$\begin{array}{ll} \text{DBL } (a = -3) & \Rightarrow\ 4\text{M} + 4\text{S} \\ \text{mDBLADD } (Z_2 = 1) & \Rightarrow\ 13\text{M} + 5\text{S} \\ \text{DBLADD } (Z_2^{\,2}, Z_2^{\,3} \text{ cached}) & \Rightarrow\ 16\text{M} + 5\text{S} \end{array}$$

Longa 2007

- Extended Twisted Edwards coordinates: $(x, y) \mapsto (X/Z, Y/Z, 1, T/Z)$, $T = XY/Z$
$(X : Y : Z : T) = \{(\lambda X, \lambda Y, \lambda Z , \lambda Z) : \lambda \in \mathbb{F}_p^{*}\}$

$$\begin{array}{ll} \text{DBL } (a = -1) & \Rightarrow\ 4\text{M} + 3\text{S} \\ \text{mDBLADD } (Z_2 = 1) & \Rightarrow\ 11\text{M} + 3\text{S} \\ \text{DBLADD} & \Rightarrow\ 12\text{M} + 3\text{S} \end{array}$$

Hisil et al. 2008

Point Arithmetic

Minimizing costs:

- Trade additions for subtractions (or vice versa) by applying $\lambda = -1 \in \mathbb{F}_p{}^*$
- Minimize constants and additions/subtractions by applying $\lambda = 2^{-1} \in \mathbb{F}_p{}^*$

E.g., $(X_2,Y_2,Z_2) \leftarrow 2(X_1,Y_1,Z_1)$ using Jacobian coord.

$$\left\{\begin{array}{l} A = 3(X_1 + Z_1{}^2)(X_1 - Z_1{}^2),\ \ B = \cancel{4}X_1Y_1{}^2 \\ X_2 = A^2 - 2B \\ Y_2 = A(B - X_2) - \cancel{8}Y_1{}^4 \\ Z_2 = \cancel{2}Y_1Z_1 \end{array}\right. \quad \Longrightarrow \quad \left\{\begin{array}{l} A = 3(X_1 + Z_1{}^2)(X_1 - Z_1{}^2)/2,\ \ B = X_1Y_1{}^2 \\ X_2 = A^2 - 2B \\ Y_2 = A(B - X_2) - Y_1{}^4 \\ Z_2 = Y_1Z_1 \end{array}\right.$$

- Most constants are eliminated
- If 1Mult > 1Sqr + 3"Add", replace Y_1Z_1 by $[(Y_1+Z_1)^2 - Y_1{}^2 - Z_1{}^2]/2$
- See our database of formulas using Jacobian coordinates:

http://patricklonga.bravehost.com/jacobian.html

Scalar Arithmetic

1. Convert k to an efficient "window-based" representation, say $k = \sum_{i=0}^{N-1} k_i 2^i$,
   where $k_i \in \{0,\ 1,\ 3,\ 5, \ldots,\ m\}$

   In particular, we use width-w non-adjacent form (wNAF) that insert $(w-1)$ "0"-digits between nonzero digits:

   - If $m = 2^{w-1} - 1$, $w \geq 2 \in \mathbb{Z}$ $\Rightarrow$ traditional integral window, nonzero density $(w+1)^{-1}$

   On-the-fly conversion algorithms that save memory are not good candidates here (too many function calls, and memory is not constrained)

   $\Rightarrow$ we'd better convert k first and then execute evaluation stage

Scalar Arithmetic

2. Precompute $L = (m - 1)/2$ non-trivial points $\{P, [3]P, [5]P, \ldots, [m]P\}$
   Inversion is relatively expensive, 1I = 175M

   - For Jacobian coord., use LM method without inversions, Longa and Gebotys (2009):

     $$\text{Cost} = (5L+2)\text{M} + (2L+4)\text{S},$$

     which is the lowest cost in the literature

   - For Twisted Edwards, compute $P + 2P + 2P + \ldots + 2P$ using general additions

3. Evaluate $[k]P$ using a *double-and-doubleadd* algorithm

   - For both systems, $w = 5$ ($L = 7$) is optimal for $bitlength(k) = 256$ bits
     Two main functions: merged 4DBL and DBLADD

GLS Method

Field and Point Arithmetic:

- Similar techniques apply to $\mathbb{F}_{p^2}$ arithmetic
- Conditional branches can be avoided by clever choice of p (e.g., $p = 2^{127} - 1$)

- "Contiguous" dependencies are more expensive ($n = 2$ words), but more easily avoided by rescheduling $\Rightarrow$ scheduling at $\mathbb{F}_{p^2}$ and $\mathbb{F}_p$ levels

- More opportunities for merging field operations because of $\mathbb{F}_{p^2} / \mathbb{F}_p$ interaction and reduced operand size (more GPRs are available for intermediate computations)

E.g., $a - 2b \pmod{p}$, $(a + a + a)/2 \pmod{p}$, $a + b - c \pmod{p}$,
merging of $a + b \pmod{p}$ and $a - b \pmod{p}$, merging of $a - b \pmod{p}$ and $c - d \pmod{p}$, and merging of $a + a \pmod{p}$ and $a + a + a \pmod{p}$

GLS Method

Scalar Arithmetic:

- Recall that $[k]P = [k_0]P + [k_1](\lambda P)$
  Use (fractional) wNAF to convert k_0 and k_1:

  $\Rightarrow$ Again, it is better to convert k_0 and k_1 first and then execute evaluation stage

- Precompute $L = (m - 1)/2$ non-trivial points $\{P, [3]P, [5]P, \ldots, [m]P\}$
  Inversion is not so expensive, 1I = 59M

  - For Jacobian coord., use LM method with *one* inversion, Longa and Miri (PKC 2008):

    $$\text{Cost} = 1\text{I} + (9L+1)\text{M} + (2L+5)\text{S},$$

    which is the lowest cost in the literature

  - For Twisted Edwards, compute $P + 2P + 2P + \ldots + 2P$ using general additions (general addition is only 1M more expensive than mixed addition)

GLS Method

Scalar Arithmetic: (*Cont'd*)

- Evaluate $[k]P = [k_0]P + [k_1](\lambda P)$ using *interleaving*, Gallant et al. (Crypto 2001) and Möller (SAC 2001)
  - For Jacobian coord., a fractional window $L = 6$ is optimal (*bitlength*(k) = 256 bits)
  - For Twisted Edwards, an integral window $w = 5$ ($L = 7$) is optimal (*bitlength*(k) = 256 bits)
- *Three* main functions: DBL, DBLADD and DBLADDADD

Implementation Results

- Implementation of variable-scalar-variable-point $[k]P$ with $\sim$128-bit security
- Mostly in C with underlying field arithmetic in assembly
- Plugged to MIRACL library [Scott]
- *Four* versions:

  - Jacobian coordinates, $p = 2^{256}-189$: ***jac256189***
    $E/\mathbb{F}_p: \ y^2 = x^3 - 3x + b$, with b = 0×fd63c3319814da55e88e9328e96273c483dca6cc84df53ec8d91b1b3e0237064
    $\#E(\mathbb{F}_p) = p + 1 - t = 10r$, r prime

  - (Extended) Twisted Edwards coord., $p = 2^{256} - 189$: ***ted256189***
    $E/\mathbb{F}_p: \ -x^2 + y^2 = 1 + 358x^2y^2$, $\#E(\mathbb{F}_p) = p + 1 - t = 4r$, r prime

  - GLS method using Jacobian coordinates, $p = 2^{127}-1$: ***jac1271gls***
    $E'/\mathbb{F}_{p^2}: \ y^2 = x^3 - 3\mu^2 x + 44\mu^3$, $\mu = 2 + i \in \mathbb{F}_{p^2}$, $\#E'(\mathbb{F}_{p^2}) = (p + 1 - t)(p + 1 + t)$ is prime

  - GLS method using (Extended) Twisted Edwards coord., $p = 2^{127}-1$: ***ted1271gls***
    $E'/\mathbb{F}_{p^2}: \ -\mu x^2 + y^2 = 1 + 109\mu x^2 y^2$, $\mu = 2 + i \in \mathbb{F}_{p^2}$, $\#E'(\mathbb{F}_{p^2}) = (p + 1 - t)(p + 1 + t) = 4r$, r prime

- We ran each implementation 10^4 times on targeted processors and averaged the timings

Implementation Results

Standard curve (256 bits): cost of $[k]P$ in cycles

	Intel Core 2 Duo		AMD Opteron	
Method	**Cost**	**Relative reduction (%)**	**Cost**	**Cost reduction (%)**
Hisil et al. [HWC09]	468000	-	-	-
***Jac256189* (this work)**	**337000**	**28% / 13%**	**274000**	**- / 11%**
Curve25519 [GT07]	386000	-	307000	-

Twisted Edwards curve (256 bits):

	Intel Core 2 Duo		AMD Opteron	
Method	**Cost**	**Relative reduction (%)**	**Cost**	**Cost reduction (%)**
Hisil et al. [HWC09]	362000	-	-	-
***Ted256189* (this work)**	**281000**	**22% / 27%**	**232000**	**- / 24%**
Curve25519 [GT07]	386000	-	307000	-

Implementation Results

Standard curve using GLS: cost of $[k_0]P + [k_1](\lambda P)$ in cycles

Method	Intel Atom Cost	Intel Atom Relative reduction (%)	Intel Core 2 Duo Cost	Intel Core 2 Duo Relative reduction (%)	AMD Opteron Cost	AMD Opteron Cost reduction (%)
Galbraith et al. [GLS09] *	832000	-	332000	-	341000	-
***Jac1271gls* (this work)**	**644000**	**23% / -**	**252000**	**24% / 35%**	**238000**	**30% / 22%**
Curve25519 [GT07]	-	-	386000	-	307000	-

Twisted Edwards curve using GLS:

Method	Intel Atom Cost	Intel Atom Relative reduction (%)	Intel Core 2 Duo Cost	Intel Core 2 Duo Relative reduction (%)	AMD Opteron Cost	AMD Opteron Cost reduction (%)
Galbraith et al. [GLS08] *	732000	-	295000	-	295000	-
***Ted1271gls* (this work)**	**588000**	**20% / -**	**229000**	**22% / 41%**	**211000**	**28% / 31%**
* **Curve25519 [GT07]**	-	-	386000	-	307000	-

Implementation Results

Recent improvements!! New!

Intel Core 2 Duo E6750

Galbraith et al. [GLS09]	295000 (1)
Ted1271gls	210000

29%

AMD Opteron 275

Galbraith et al. [GLS09]	284000 (1)
Ted1271gls	200000

30%

Intel Xeon 5130

Galbraith et al. [GLS09]	323000 (2)
Ted1271gls	213000

34%

AMD Phenom II X4 940 / 955

Galbraith et al. [GLS09]	255000 (1) / 262000 (2)
Ted1271gls	181000

29% / 31%

(1) Our own measurements, same platform, same compiler

(2) eBACS, accessed 08/2010 (http://bench.cr.yp.to/results-dh.html)

Conclusions

- Thorough bottom-up optimization process (field/point/scalar arithmetic levels)
- Proposed several optimizations taking into account architectural features
- New implementations are (at least) **30%** faster than state-of-the-art implementations on all x86-64 CPUs tested
- Optimizations can be easily extended to other implementations using fixed point *P*, digital signatures and different coordinate systems/curve forms/underlying fields

References

More details can be found in:

- P. Longa, “Speed Benchmarks for Elliptic Curve Scalar Multiplication”, 07/2010. Available at:

  http://patricklonga.bravehost.com/speed_ecc.html

- P. Longa and C. Gebotys, “Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors”, in *Cryptology ePrint Archive*, Report 2010/335, 2010.

References

[GLS08] S. Galbraith, X. Lin and M. Scott, "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves," in *Cryptology ePrint Archive*, Report 2008/194, 2008.

[GLS09] S. Galbraith, X. Lin and M. Scott, "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves," in *EUROCRYPT 2009*.

[GLS01] R. Gallant, R. Lambert and S. Vanstone, "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms," in *CRYPTO 2001*.

[GT07] P. Gaudry and E. Thomé, "The mpFq Library and Implementing Curve-Based Key Exchanges," in *SPEED 2007*.

[HWC08] H. Hisil, K. Wong, G. Carter and E. Dawson, "Twisted Edwards Curves Revisited," in *ASIACRYPT 2008*.

[HWC09] H. Hisil, K. Wong, G. Carter and E. Dawson, "Jacobi Quartic Curves Revisited," in *Cryptology ePrint Archive*, Report 2009/312, 2009.

[LG08] P. Longa and C. Gebotys, "Setting Speed Records with the (Fractional) Multibase Non-Adjacent Form Method for Efficient Elliptic Curve Scalar Multiplication," *CACR technical report*, CACR 2008-06, 2008.

[LM08] P. Longa and A. Miri, "New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields," in *PKC 2008*.

[Möl01] B. Möller, "Algorithms for Multi-Exponentiation," in *SAC 2001*.

[Scott] M. Scott, "MIRACL – Multiprecision Integer and Rational Arithmetic C/C++ Library," 1988–2007.

Efficient Techniques for High-Speed Elliptic Curve Cryptography

Q & A

Patrick Longa
University of Waterloo
http://patricklonga.bravehost.com

UNIVERSITY OF Waterloo