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 $\psi(x,y) \to (\alpha x, \alpha y)$, $\psi(P) = \lambda P$

  Then:

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

  where $\log k_0 \approx \log k_1 \approx \frac{1}{2} \log k$
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)
  - 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 ⇒ 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, $p = 2^{256} - 189$

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

⇒ 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

 Assembly instructions

\[
\begin{align*}
&> \text{addq } %rcx, %r8 \\
&> \text{movq } %r8, 8(\%rdx) \\
&> \text{adcq } 0, %r9 \\
&> \text{movq } %r9, 16(\%rdx) \\
&> \text{adcq } 0, %r10 \\
&> \text{movq } %r10, 24(\%rdx) \\
&> \text{adcq } 0, %r11 \\
&> \text{movq } %r11, 32(\%rdx) \\
&> \text{xorq } %rax, %rax \\
&> \text{movq } %0xbd, %rcx \\
&> \text{movq } 8(\%rdi), %r8 \\
&> \text{addq } 8(\%rsi), %r8 \\
&> \text{movq } 16(\%rdi), %r9 \\
&> \text{adcq } 16(\%rsi), %r9 \\
&> \text{movq } 24(\%rdi), %r10 \\
&> \text{adcq } 24(\%rsi), %r10 \\
&> \text{movq } 32(\%rdi), %r11 \\
&> \text{adcq } 32(\%rsi), %r11 \\
\end{align*}
\]

\( \rho \): “distance” between instructions

“Ideal” non-superscalar CPU:

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

\( \delta_{write} \): pipeline latency of write instruction
Field Arithmetic

“Contiguous” dependencies (Cont’d)

We propose three solutions:

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

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

3. Merging field operations ⇒ 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 $(a - b) - 2c \pmod{p}$
### “Contiguous” dependencies (Cont’d) \( (X_1,Y_1,Z_1) \leftarrow 2(X_1,Y_1,Z_1) \)

<table>
<thead>
<tr>
<th>“Unscheduled”</th>
<th>Scheduled</th>
<th>Scheduled and merged DBL-DBL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Sqr(21,t3) )</td>
<td>( Sqr(Z1,t3) )</td>
<td>( Sqr(Z1,t3) )</td>
</tr>
<tr>
<td>( Add(X1,t3,t1) )</td>
<td>( Add(Z1,Y1,t2) )</td>
<td>( Add(Z1,Y1,t2) )</td>
</tr>
<tr>
<td>( Sub(X1,t3,t3) )</td>
<td>( Add(X1,t3,t1) )</td>
<td>( Add(X1,t3,t1) )</td>
</tr>
<tr>
<td>( Mult(t3,t1,t2) )</td>
<td>( Sub(X1,t3,t3) )</td>
<td>( Sub(X1,t3,t3) )</td>
</tr>
<tr>
<td>( Mult3(t2,t1) )</td>
<td>( Mult3(t3,t0) )</td>
<td>( Mult3(t3,t0) )</td>
</tr>
<tr>
<td>( Div2(t1,t1) )</td>
<td>( Mult(X1,t2,t4) )</td>
<td>( Mult(X1,t2,t4) )</td>
</tr>
<tr>
<td>( Mult(Y1,Z1,t3) )</td>
<td>( Mult(t1,t0,t3) )</td>
<td>( Mult(t1,t0,t3) )</td>
</tr>
<tr>
<td>( Sqr(Y1,t2) )</td>
<td>( Sqr(t2,t0) )</td>
<td>( Sqr(t2,t0) )</td>
</tr>
<tr>
<td>( Mult(t2,X1,t4) )</td>
<td>( Div2(t3,t1) )</td>
<td>( Div2(t3,t1) )</td>
</tr>
<tr>
<td>( Sqr(t1,t3) )</td>
<td>( Mult(Y1,Z1,Z1) )</td>
<td>( Mult(Y1,Z1,Z1) )</td>
</tr>
<tr>
<td>( Sub(t3,t4,X1) )</td>
<td>( Sqr(t1,t2) )</td>
<td>( Sqr(t1,t2) )</td>
</tr>
<tr>
<td>( Sub(X1,t4,X1) )</td>
<td>( DblSub(t2,t4,X1) )</td>
<td>( DblSub(t2,t4,X1) )</td>
</tr>
<tr>
<td>( Sub(t4,X1,t3) )</td>
<td>( Sub(t4,X1,t2) )</td>
<td>( Sub(t4,X1,t2) )</td>
</tr>
<tr>
<td>( Mult(t3,t1,t4) )</td>
<td>( Mult(t1,t2,t4) )</td>
<td>( Mult(t1,t2,t4) )</td>
</tr>
<tr>
<td>( Sqr(t2,t0) )</td>
<td>( Sub(t4,t0,Y1) )</td>
<td>( Sub(t4,t0,Y1) )</td>
</tr>
<tr>
<td>( Sub(t4,t0,Y1) )</td>
<td>( ... )</td>
<td>( ... )</td>
</tr>
</tbody>
</table>
Field Arithmetic

“Contiguous” dependencies (Cont’d)

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

<table>
<thead>
<tr>
<th>Point operation</th>
<th>Intel Atom “Unscheduled”</th>
<th>“Scheduled and merged”</th>
<th>Intel Core 2 Duo “Unscheduled”</th>
<th>“Scheduled and merged”</th>
<th>AMD Opteron “Unscheduled”</th>
<th>“Scheduled and merged”</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL</td>
<td>3390</td>
<td>3332</td>
<td>1115</td>
<td>979</td>
<td>786</td>
<td>726</td>
</tr>
<tr>
<td>Relative reduction</td>
<td>-</td>
<td>2%</td>
<td>-</td>
<td>12%</td>
<td>-</td>
<td>8%</td>
</tr>
<tr>
<td>Estimated reduction for ([k]P)</td>
<td>-</td>
<td>1%</td>
<td>-</td>
<td>9%</td>
<td>-</td>
<td>5%</td>
</tr>
</tbody>
</table>

⇒ Estimated reduction of 5% and 9% on AMD Opteron and Intel Core 2 Duo, respectively.
⇒ Less “aggressive” architectures are not greatly affected by “contiguous” dependencies.
Point Arithmetic

Our choice of formulas:

- Jacobian coordinates: \((x, y) \mapsto (X/Z^2, Y/Z^3, 1), \ (X : Y : Z) = \{ (\lambda^2 X, \lambda^3 Y, \lambda Z) : \lambda \in \mathbb{F}_p^* \}\)
  - `DBL (a = -3)` ⇒ 4M + 4S
  - `mDBLADD (Z_2 = 1)` ⇒ 13M + 5S
  - `DBLADD (Z_2^2, Z_2^3 cached)` ⇒ 16M + 5S
  
  Longa 2007

- Extended Twisted Edwards coordinates: \((x, y) \mapsto (X/Z, Y/Z, 1, T/Z), \ T = XY/Z\)
  \((X : Y : Z : T) = \{(\lambda X, \lambda Y, \lambda Z, \lambda Z) : \lambda \in \mathbb{F}_p^* \}\)
  - `DBL (a = -1)` ⇒ 4M + 3S
  - `mDBLADD (Z_2 = 1)` ⇒ 11M + 3S
  - `DBLADD` ⇒ 12M + 3S
  
  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.

\[
\begin{align*}
A &= 3(X_1 + Z_1^2)(X_1 - Z_1^2), \quad B = 4X_1Y_1^2 \\
X_2 &= A^2 - 2B \\
Y_2 &= A(B - X_2) - 8Y_1^4 \\
Z_2 &= 2Y_1Z_1
\end{align*}
\]

\[
\begin{align*}
A &= 3(X_1 + Z_1^2)(X_1 - Z_1^2)/2, \quad B = X_1Y_1^2 \\
X_2 &= A^2 - 2B \\
Y_2 &= A(B - X_2) - Y_1^4 \\
Z_2 &= Y_1Z_1
\end{align*}
\]

- Most constants are eliminated
- If $1\text{Mult} > 1\text{Sqr} + 3\text{“Add”}$, 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

   Inversion is relatively expensive, \( 1I = 175M \)
   - For Jacobian coord., use LM method without inversions, Longa and Gebotys (2009):
     \[
     \text{Cost} = (5L+2)M + (2L+4)S,
     \]
     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 \( \text{bitlength}(k) = 256 \) 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 ⇒ 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 $a + a + a \pmod{p}$
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

  Inversion is not so expensive, \(1I = 59M\)
  - For Jacobian coord., use LM method with one inversion, Longa and Miri (PKC 2008):
    \[
    \text{Cost} = 1I + (9L+1)M + (2L+5)S, \]
    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 (\( \text{bitlength}(k) = 256 \) bits)
  - For Twisted Edwards, an integral window \( w = 5 \) (\( L = 7 \)) is optimal (\( \text{bitlength}(k) = 256 \) bits)

- Three main functions: DBL, DBLADD and DBLADDDADD
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, \(p = 2^{256} - 189\): \textbf{jac256189}
    \[
    E / \mathbb{F}_p : \ y^2 = x^3 - 3x + b , \quad \text{with } b = 0x\text{fd63c3319814da55e88e9328e96273c483dca6cc84df53ec8d91b1b3e0237064} \\
    \#E(\mathbb{F}_p) = p + 1 - t = 10r, \quad r \text{ prime}
    \]
  - (Extended) Twisted Edwards coord., \(p = 2^{256} - 189\): \textbf{ted256189}
    \[
    E / \mathbb{F}_p : \ -x^2 + y^2 = 1 + 358x^2y^2 , \quad \#E(\mathbb{F}_p) = p + 1 - t = 4r, \quad r \text{ prime}
    \]
  - GLS method using Jacobian coordinates, \(p = 2^{127} - 1\): \textbf{jac1271gls}
    \[
    E' / \mathbb{F}_{p^2} : \ y^2 = x^3 - 3\mu^2x + 44\mu^3 , \quad \mu = 2 + i \in \mathbb{F}_{p^2}, \quad \#E'(\mathbb{F}_{p^2}) = (p + 1 - t)(p + 1 + t) \text{ is prime}
    \]
  - GLS method using (Extended) Twisted Edwards coord., \(p = 2^{127} - 1\): \textbf{ted1271gls}
    \[
    E' / \mathbb{F}_{p^2} : \ -\mu x^2 + y^2 = 1 + 109\mu x^2y^2 , \quad \mu = 2 + i \in \mathbb{F}_{p^2}, \quad \#E'(\mathbb{F}_{p^2}) = (p + 1 - t)(p + 1 + t) = 4r, \quad r \text{ prime}
    \]
- 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Intel Core 2 Duo</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Relative reduction (%)</td>
</tr>
<tr>
<td>Hisil et al. [HWC09]</td>
<td>468000</td>
<td>-</td>
</tr>
<tr>
<td>\textit{Jac256189} (this work)</td>
<td>337000</td>
<td>28% / 13%</td>
</tr>
<tr>
<td>Curve25519 [GT07]</td>
<td>386000</td>
<td>-</td>
</tr>
</tbody>
</table>

### Twisted Edwards curve (256 bits):

<table>
<thead>
<tr>
<th>Method</th>
<th>Intel Core 2 Duo</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Relative reduction (%)</td>
</tr>
<tr>
<td>Hisil et al. [HWC09]</td>
<td>362000</td>
<td>-</td>
</tr>
<tr>
<td>\textit{Ted256189} (this work)</td>
<td>281000</td>
<td>22% / 27%</td>
</tr>
<tr>
<td>Curve25519 [GT07]</td>
<td>386000</td>
<td>-</td>
</tr>
</tbody>
</table>
Implementation Results

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Intel Atom</th>
<th>Intel Core 2 Duo</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Relative reduction (%)</td>
<td>Cost</td>
</tr>
<tr>
<td>Galbraith et al. [GLS09] *</td>
<td>832000</td>
<td>-</td>
<td>332000</td>
</tr>
<tr>
<td>Jac1271gls (this work)</td>
<td>644000</td>
<td>23% / -</td>
<td>252000</td>
</tr>
<tr>
<td>Curve25519 [GT07]</td>
<td>-</td>
<td>-</td>
<td>386000</td>
</tr>
</tbody>
</table>

Twisted Edwards curve using GLS:

<table>
<thead>
<tr>
<th>Method</th>
<th>Intel Atom</th>
<th>Intel Core 2 Duo</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Relative reduction (%)</td>
<td>Cost</td>
</tr>
<tr>
<td>Galbraith et al. [GLS08] *</td>
<td>732000</td>
<td>-</td>
<td>295000</td>
</tr>
<tr>
<td>Ted1271gls (this work)</td>
<td>588000</td>
<td>20% / -</td>
<td>229000</td>
</tr>
<tr>
<td>* Curve25519 [GT07]</td>
<td>-</td>
<td>-</td>
<td>386000</td>
</tr>
</tbody>
</table>
## Implementation Results

### Recent improvements!!

**Intel Core 2 Duo E6750**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Galbraith et al. [GLS09]</td>
<td>295000 (1)</td>
<td></td>
</tr>
<tr>
<td>Ted1271gls</td>
<td>210000</td>
<td></td>
</tr>
</tbody>
</table>

29%

**AMD Opteron 275**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Galbraith et al. [GLS09]</td>
<td>284000 (1)</td>
<td></td>
</tr>
<tr>
<td>Ted1271gls</td>
<td>200000</td>
<td></td>
</tr>
</tbody>
</table>

30%

**Intel Xeon 5130**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Galbraith et al. [GLS09]</td>
<td>323000 (2)</td>
<td></td>
</tr>
<tr>
<td>Ted1271gls</td>
<td>213000</td>
<td></td>
</tr>
</tbody>
</table>

34%

**AMD Phenom II X4 940 / 955**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Galbraith et al. [GLS09]</td>
<td>255000 (1) / 262000 (2)</td>
<td></td>
</tr>
<tr>
<td>Ted1271gls</td>
<td>181000</td>
<td></td>
</tr>
</tbody>
</table>

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
More details can be found in:


References


Efficient Techniques for High-Speed Elliptic Curve Cryptography

Q & A

Patrick Longa
University of Waterloo

http://patricklonga.bravehost.com