A Faster way to do ECC

Mike Scott

Dublin City University

Joint work with

Steven Galbraith

Royal Holloway, University of London

Xibin Lin

Sun Yat-Sen University

Elliptic Curve based Crypto

- People like to use ECC because...
- 1. Smaller Key sizes
- 2. Faster implementation $\longleftarrow$
- 3. Solid number theoretic based security

Elliptic Curve based Crypto

- For security, field size needs to be ≥ 160 bits.
- We can do it over $\mathbb{F}_p$, and $\mathbb{F}_{p^m}$ with small p and large prime m.
- For $\mathbb{F}_{p^m}$ with large p and small $m > 2$, we need to be careful - Weil descent attacks apply.
- Which leaves a largely unexplored "window of opportunity" for elliptic curves over $\mathbb{F}_{p^2}$ (but see early work by Nogami and Iijima et al. 2002/2003).

Elliptic curves over $\mathbb{F}_{p^2}$

- No really compelling reason to go there just for the sake of it..
- ..unless some new trick applies that makes it more efficient that $E(\mathbb{F}_p)$, in particular which speeds up variable point multiplication.

Lets back-up..

- In 2000 Gallant, Lambert and Vanstone (GLV) come up with a very nice idea..
- Consider an elliptic curve $E(\mathbb{F}_p)$ on which, when presented with a random point P, we somehow automagically know a non-trivial multiple of P, say λP.

GLV method - 1

- Then when asked to calculate kP, we can always break it down into $kP = k_0P + k_1.(\lambda P)$.
- where k_0 and k_1 have half the number of bits of k.
- Then we can apply a fast double-multiplication algorithm (aka multi-exponentiation), which is much faster than calculating kP directly.
- In many contexts where a random multiplier k is required, k_0 and k_1 can instead be chosen directly at random.

GLV method - 2

- Its not quite as simple as I made it sound.

GLV method - 3

- But how to get λP?
- On curves with low CM discriminant, its easy!
- Let $p = 1 \bmod 3$, and consider the curve $E(\mathbb{F}_p) : y^2 = x^3 + B$ of prime order r.
- Then if $P(x, y)$ is a point on the curve, then so is $Q(\beta x, y)$, where β is a non-trivial cube root of unity mod p.

GLV method - 4

- Furthermore $Q = \lambda P$, where λ is a solution of $\lambda^2 + \lambda + 1 \equiv 0 \bmod r$.
- β is in $\mathbb{F}_p$, λ in $\mathbb{F}_r$. Both can be easily pre-calculated.
- So in this case the fast method applies, because we have a suitable homomorphism $\psi(x, y) \to (\beta x, y)$, $\psi(P) = \lambda P$.

GLV method - 5

- There is also the Frobenius endomorphism
- Let E be an elliptic curve defined over $\mathbb{F}_q$, where $q = p^m$. Then the map defined by $\psi(x, y) \rightarrow (x^q, y^q)$ is an endomorphism.
- Not useful if $m = 1$ and $q = p$.

GLV method - 6

- In fact GLV method not much used..
- In choosing regular elliptic curves we can pre-select a really nice prime p, and then search for an elliptic curve $y^2 = x^3 - 3x + B$ of prime order r, by iterating on B.
- This gives us a huge search space..

GLV Method - 7

- For the GLV-friendly curve $y^2 = x^3 + B$, over $\mathbb{F}_p$ there are only 6 possible curves for any particular choice of p! So, sadly, the odds are very much against the order being prime....
- So what is gained on the swings, may be lost on the roundabouts, as we may have to settle for a less than ideal form of p, which will make ECC slower.
- Also, there is a superstitious distrust of low CM discriminant curves.

Elliptic curves over $\mathbb{F}_{p^2}$

- Consider now the elliptic curve $E : y^2 = x^3 - 3x + B$ defined over $\mathbb{F}_p$.
- This has $p + 1 - t$ points on it.
- Now consider the same curve over $\mathbb{F}_{p^2}$. This has $(p + 1 - t)(p + 1 + t)$ points on it $= p^2 + 1 - (t^2 - 2p)$.
- Next consider the quadratic twist of this curve. This will have $p^2 + 1 + (t^2 - 2p)$ points on it, **which can be a prime**.
- This is where we propose to do our ECC.

The twisted curve

- The formula for the twisted curve is $E' : y^2 = x^3 - 3u^2x + u^3B$, where u is a quadratic non-residue in $\mathbb{F}_{p^2}$.
- So this curve is defined over $\mathbb{F}_{p^2}$, and is of prime order - a viable place to do ECC.
- Note that from the method of construction these are not completely general curves over $\mathbb{F}_{p^2}$.
- But there are a lot of them!
- If $p = 3 \bmod 4$, then an element x in $\mathbb{F}_{p^2}$ can be represented as $x = (a + ib)$, where $i = \sqrt{-1}$. Sometimes we write this as $[a, b]$.
- The conjugate of x is represented as $\bar{x} = a - ib$.

The bonus

- On this curve we have a nice homomorphism!
- $\psi(x, y) \rightarrow ((u/u^p).\bar{x}, \sqrt{(u^3/u^{3p})}.\bar{y})$.
- Basically we "lift" (x, y) up to the curve $E(\mathbb{F}_{p^4})$, apply the Frobenius endomorphism, and then "drop" it back down to $E'(\mathbb{F}_{p^2})$.
- $\lambda = t^{-1}(p-1) \bmod r$.
- The GLV method applies.

Multi-exponentiation – $\sum_{i=0}^{i<m} k_i P_i$

- There is a large and rather confusing literature on the subject.
- Basic idea - a precomputation based on P_i, exponents k_i expressed in NAF format, then a double-and-add loop.
- Two methods explored – Solinas's Joint Sparse Form (JSF) and the interleaving algorithm (see Hankerson, Menezes and Vanstone "Guide to Elliptic Curve Cryptography").
- Former method good for $m = 2$ and if little or no space available for precomputation. But interleaving seems to be faster, and generalises easier to $m > 2$.
- For now consider only double-exponentiation, $m = 2$ case, $R = aP + bQ$.

Interleaving algorithm – 1

- The idea here is precompute $\{P, 3P, 5P, .., [(2w-1)/2]P\}$ and $\{Q, 3Q, 5Q, .., [(2w-1)/2]Q\}$, for some choice of (fractional) window w. (In practise different values for w can be used for P and Q if desired).
- Convert a and b into NAF format.
- For example if $a = 11_{10} = 001011_2$, then $3a = 33_{10} = 100001_2$. Now calculate $a = (3a - a)/2$, doing the subtraction bit-by-bit, $a = 10\bar{1}0\bar{1}$, where $\bar{1} = -1$. This is the NAF form of a.

Interleaving algorithm – 2

- Initialise a point R to the point-at-infinity.
- We then scan the NAFs for a and b together from left to right. As each bit is processed, double the value of R. While scanning pick out sub-sections of the corresponding NAF to get the largest multiple of P or Q which is in the precomputed tables. Add this precomputed multiple of P or Q to R.

Interleaving algorithm – 3

- For the case $m = 1$ this is just the normal sliding-windows algorithm for exponentiation.
- The bigger w, the more time required for precomputation, but the less additions in the main double-and-add loop. So there is an optimal value for w. In practise there would be some consideration to keep w small, to conserve memory.
- We will want to use some form of projective coordinate (x, y, z) representation for the points, as affine coordinates (x,y) will be far too slow – each addition/doubling requiring a modular inversion.

The precomputation problem – 1

- Rather overlooked in the literature. Given P in affine coordinates, find $\{P, 3P, 5P, .., [(2w-1)/2]P\}$, also in affine coordinates (as ideally we would like the additions in the main loop to be "mixed" additions)
- So calculate $2P$ in affine coordinates, and keep adding it to P in affine coordinates. Too slow.
- Calculate $2P$ in affine coordinates, then keep adding it to P in projective coordinates. Then convert $\{3P, 5P, .., [(2w-1)/2]P\}$ to affine coordinates all together using Montgomery's trick. Two inversions in total.
- Montgomery's trick – Given $1/(z_1.z_2)$ then $1/z_1 = z_2/(z_1.z_2)$ and $1/z_2 = z_1/(z_1.z_2)$

The precomputation problem – 2

- Dahmen, Okeya and Schepers (DOS), and recently Longa and Miri, have come up with clever fast techniques requiring only one inversion. See also recent review paper by Bernstein and Lange (2008)
- New idea (?)
- From P, calculate $3P$, and then double it to get $6P$. Then calculate $6P - P$ and $6P + P$ together (which can share most of the calculation) to get $5P$ and $7P$. Then double $5P$ to get $10P$, and calculate $10P + P$ and $10P - P$, to get $9P$ and $11P$, etc. Note that $W + P$ and $W - P$ have the same z coordinates, so less values to be inverted via Montgomery. All additions are mixed.
- Idea works well over any field, any projective representation. However not quite as fast as DOS.

Multi-exponentiation with a homomorphism

- On our proposed curves a variable point multiplication can be calculated as $kP = k_0P + k_1Q$, where $Q = \psi(P)$.
- So having precomputed the table $\{P, 3P, 5P, .., [(2w - 1)/2]P\}$, the second table can be quickly calculated from this one by simply applying ψ to each of its elements.

Finding a curve

- For AES-128 level of security, it makes sense to choose $p = 2^{127} - 1$ (which God surely supplied for this very purpose...). Observe that $p = 7 \bmod 8$, and $p = 2 \bmod 5$.
- We use a modified Schoof algorithm to find an elliptic curve such that $E(\mathbb{F}_p) : p^2 + 1 + (t^2 - 2p)$ is prime. Note that point counting on a 127-bit curve like this is very fast.
- The first suitable curve we find (by incrementing the B parameter in the Weierstrass form) is $E : y^2 = x^3 - 3x + 44$, for which $t = \texttt{3204F5AE088C39A7}$.
- Choose as a quadratic non-residue $u = 2 + i$.

The homomorphism

- The homomorphism is $\psi(x, y) = (\omega_x \bar{x}, \omega_y \bar{y})$, where
- $\omega_x = [(p+3)/5, (3p+4)/4]$
- $\omega_y =$ $[12B04E814703D49C1AFAC10F88821962, 426B94A2AD451F296F755142FE73FB62]$
- $\lambda =$ $B6F12BDE99042C16290B3B18FD545035402B0743BC131F5B775D928BCFBCD7A$
- $\psi(P) = \lambda P$.

The implementation

- We choose regular Jacobian coordinates, a reasonably efficient projective form supported by many standards.
- For the double-exponentiation, we choose $w = 5$, which is close to optimal.
- Assume a field multiplication over $\mathbb{F}_p$ has a cost of m.
- A field multiplication over $\mathbb{F}_{p^2}$ requires 3 multiplications over $\mathbb{F}_p$, using Karatsuba.
- A field squaring over $\mathbb{F}_{p^2}$ requires 2 multiplications over $\mathbb{F}_p$.
- The theoretical cost of a variable point multiplication, using the homomorphism, is 4147m.
- (plus cost of modular additions/subtractions, plus 2 inversions)

The competition

- What to compare with?
- Initially consider an elliptic curve $E(\mathbb{F}_p)$, for p a pseudo mersenne 256-bit prime. Again we use standard Jacobian coordinates
- Assume a field multiplication over $\mathbb{F}_p$ in this case has a cost of M.
- The theoretical cost of a variable point multiplication using Jacobian coordinates is 2614M

The comparison

- We also count the number of operations required for implementing $E'(\mathbb{F}_{p^2})$ **without** using the homomorphism
- Important note – there are two effects which impact the comparison – the effect of moving from $E(\mathbb{F}_p)$ (256-bit prime) to $E'(\mathbb{F}_{p^2})$ (128-bit prime) – and the effect of exploiting the homomorphism.
- We want to be able to distinguish between these two effects.

Actual counts

Table: Point multiplication operation counts

	Method	$\mathbb{F}_p$ muls	$\mathbb{F}_p$ adds/subs
$E(\mathbb{F}_p)$, 256-bit p	SSW	2600	3775
$E'(\mathbb{F}_{p^2})$, 127-bit p	SSW	6641	16997
$E'(\mathbb{F}_{p^2})$, 127-bit p	GLV+JSF	4423	10785
$E'(\mathbb{F}_{p^2})$, 127-bit p	GLV+INT	4109	10112

What can be concluded?

- The theoretical and actual results are very close.
- But how to compare 2600M against 4109m?
- It clearly depends on the m/M ratio.
- What about all those extra modular additions in the $E'(\mathbb{F}_{p^2})$ case?
- In all cases just 2 modular inversions required.

Lets get real..

- Comparisons like this only get us so far...
- We need real implementations to compare against.
- Idea: Set up a straw-man implementation and beat the hell out of it. Hmm...

An 8-bit processor

- The Atmel Atmega 1281 is a nice 8-bit RISC architecture.
- 32 registers, and an 8x8 bit multiply instruction.
- Popular choice for Wireless Sensor Networks.
- Free cycle accurate simulator is available – works with GCC tool chain.
- Only 8Kbytes of RAM...

8-bit implementation

- We have tools to automatically generate unlooped assembly language code for modular multiplication.
- Modular multiplication/squaring dominates the execution time.
- Compared head-to-head with 256-bit prime $E(\mathbb{F}_p)$ implementation. (In what follows $E'(\mathbb{F}_{p^2})$ refers to a 128-bit prime p, $E(\mathbb{F}_p)$ refers to a 256-bit prime p.)
- $\mathbb{F}_{p^2}$ modmul takes $2327\mu S$, modsqr takes $1529\mu S$, modadd takes $174\mu S$
- $\mathbb{F}_p$ modmul takes $1995\mu S$, modsqr takes $1616\mu S$, modadd takes $124\mu S$

Whats going on?

- It appears that simply moving to a quadratic extension is not going to be beneficial.
- A 128-bit $\mathbb{F}_{p^2}$ modmul using Karatsuba requires 3 multiplication, followed by 3 reductions modulo p, plus 5 modular additions/subtractions. A 256-bit $\mathbb{F}_p$ modmul requires just one (albeit larger) multiplication, and one reduction.
- Hopefully using the homomorphism will overcome this initial disadvantage....

8-bit results

Table: Point multiplication timings – 8-bit processor

Atmel Atmega1281 processor	Method	Time (s)
$E(\mathbb{F}_p)$, (256-bit p)	SSW	5.49
$E'(\mathbb{F}_{p^2})$ (127-bit p)	SSW	6.20
$E'(\mathbb{F}_{p^2})$, (127-bit p)	GLV+JSF	4.21
$E'(\mathbb{F}_{p^2})$, (127-bit p)	GLV+INT	3.87

A 64-bit processor

- Almost all new desktop and laptop computers use a 64-bit Intel Core 2, or AMD equivalent.
- 64-bit computing has arrived (as has multi-core computing, but thats another story..)
- On a 64-bit processor an element of $\mathbb{F}_p$ for p a 127-bit mersenne prime, can be stored in just 2 registers! This is not multi-precision, its double precision!
- Writing an assembly language module to handle field arithmetic is very easy (1 day to write, 1 day to optimize/debug).

64-bit issues

- Now the field additions/subtractions cannot be ignored – as n becomes smaller (here $n = 2$) $O(n)$ and $O(1)$ contributions become significant, and are no longer completely dominated by the $O(n^2)$ operations like multiplication and squaring.
- General purpose multi-precision techniques become very inefficient (Avanzi)
- Field specific code will be much faster (see MPFQ, Gaudry-Thomé)
- As we ruthlessly optimize the code, components that one would think are utterly negligible (like computing and scanning the NAF), become significant.

64-bit issues

- "Squishing" – the cycle of profiling, identifying "hotspots", and optimizing the code, to squeeze down timings.

64-bit issues

- "Squantum effects" – the strange tendency of apparently insignificant operations, to become significant while squishing.

Modular addition

- Since this is now significant, perhaps its time to look at its implementation..
- Let $x = a + b$, if $x > p$ then $x = x - p$. Return x.
- This necessitates a horribly unpredictable branch – which deeply pipelined processors hate, and (if mispredicted) punish severely with wasted cycles as it flushes and re-initialises the pipeline.
- Can be avoided using a mersenne (or pseudo-mersenne) modulus (details left as exercise for the reader...)

Profiling

- Profiling our code shows that it spends 49% of its time doing $\mathbb{F}_p$ modmuls and modsqrs. It spends 15% of its times doing $\mathbb{F}_p$ modadds and modsubs. Its spends 6% of its time doing the few modular inversions.
- The remaining 30% of the time is spent on (what ought to be) minor tasks, like NAF calculation, memory initialisation and "glue" code that calls the significant functions.

The competition

- Strategy – identify the fastest implementation out there for ECC at the AES-128 level, and try to beat it.
- The current record is held by Gaudry and Thomé (SPEED 2007), for an implementation of Bernstein's `curve25519`
- This curve is ideal for an elliptic curve using Montgomery coordinates, and is designed specifically for a very fast Diffie-Hellman implementation. It has other advantages (side-channel attack resistance for example).

How to fairly compare?

- It would be useful to have an independent external facility to do the comparisions.
- Ideally this facility would have access to numerous different models of computers.
- Such a facility exists – `eBats`, a.k.a SuperCop. – and should be more widely used.
- We have made our code available in the form of a fast DH key-exchange implementation `eBat`.

Results

Table: Point multiplication timings – 64-bit processor

Intel Core 2 processor	Method	Clock cycles
$E(\mathbb{F}_p)$, 255-bit p	Montgomery (Gaudry-Thomé)	386,000
$E'(\mathbb{F}_{p^2})$, 127-bit p	SSW	490,000
$E'(\mathbb{F}_{p^2})$, 127-bit p	GLV+JSF	359,000
$E'(\mathbb{F}_{p^2})$, 127-bit p	GLV+INT	326,000

Signature Verification

- This would require the calculation of $R = a_0 P + a_1 \psi(P) + b_0 Q + b_1 \psi(Q)$.
- That is a 4-dim multi-exponentiation.
- Since P and therefore $\psi(P)$ are fixed, precomputed tables can be calculated offline using a much bigger fractional window size w.
- Fortunately the interleaving algorithm allows this.
- Antipa et al. have a nice trick also using multi-exponentiation for ECDSA verification on regular $E(\mathbb{F}_p)$ curves.

Signature Verification - Timings

Table: Signature Verification timings – 64-bit processor

Intel Core 2 processor	Method	$\mathbb{F}_p$ muls	$\mathbb{F}_p$ adds/subs	Clock cycles
$E'(\mathbb{F}_{p^2})$, 127-bit p	INT	7638	19046	581,000
$E'(\mathbb{F}_{p^2})$, 127-bit p	GLV+INT	5174	12352	425,000

Future work

- Consider implementation over binary fields

Future work

- .. too late! Already done by Hankerson et al. (2008)

Future work

- Our implementation would certainly benefit from a better parameterisation than standard Jacobian. (Edwards, Jacobi Quartic, Inverse Edwards etc.). Note that on the 64-bit processor, over $\mathbb{F}_{p^2}$, the I/M ratio is about 20.
- Maybe a 10% improvement is possible.
- Montgomery and multi-exponentiation do not work well together – so this is not really an option.
- Hard to extend to, for example, Hyperelliptic curves, as Weil descent becomes viable again.
- Implement on an FPGA. Lots of low level parallelism to be exploited....

Question Time

- Thank you for your attention