GLV+HWCD for 2y^2=x^3+x/GF(8^91+5)

Dan Brown[*]

March 22, 2021

Abstract

This report considers combining three well-known optimization methods for elliptic curve scalar multiplication: Gallant–Lambert–Vanstone (GLV) for complex multiplication endomorphisms $[i]$ and $[i+1]$; 3-bit fixed windows (signed base 8); and Hisil–Wong–Carter–Dawson (HWCD) curve arithmetic for twisted Edwards curves.

An x-only Diffie–Hellman scalar multiplication for curve $2y^2 = x^3+x$ over field size $8^{91}+5$ has arithmetic cost $947\mathbf{M}+1086\mathbf{S}$, where $\mathbf{M}$ is a field multiplication and $\mathbf{S}$ is a field squaring. This is approximately $(3.55\mathbf{M}+4.07\mathbf{S})$/bit, with $1\mathbf{S}$/bit for input decompression and $1\mathbf{S}$/bit for output normalization. Optimizing speed by allowing uncompressed input points leads to an estimate $(3.38\mathbf{M}+2.95\mathbf{S})$/bit.

To mitigate some side-channel attacks, the secret scalar is only used to copy curve points from one array to another: the field operations used are fixed and independent of the secret scalar. The method is likely vulnerable to cache-timing attacks, nonetheless.

1 Review of HWCD elliptic curve arithmetic

This section reviews how Hisil–Wong–Carter–Dawson (HWCD) arithmetic for twisted Edwards elliptic curve can be applied to Montgomery (Miller?) curve $2y^2 = x^3 + x$.

[*]`danibrown@blackberry.com`

1.1 Twisted Edwards in extended coordinates

This report considers a special twisted Edwards elliptic curve E defined by equations:

$$-X^2 + Y^2 = T^2 + Z^2, \tag{1.1}$$
$$XY = TZ, \tag{1.2}$$

with points in projective coordinates $(X : Y : T : Z)$, known as HWCD extended coordinates. The curve is the intersection of two quadric equations in projective 3-space. The zero point of the group of points is chosen to be $O = (0 : 1 : 0 : 1)$.

(The usual parameters (a, d) for a general twisted Edwards elliptic curve have been set to $(a, d) = (-1, 1)$ in the case of this curve E.)

1.2 Conversion between Montgomery and Edwards

The curve E is isomorphic to Montgomery curve, $M : 2y^2 = x^3 + x$. The isomorphism is $\varepsilon : M \to E$:

$$\varepsilon : P \mapsto \begin{cases} (0 : 1 : 0 : 1) & \text{if } P = \infty, \\ (0 : -1 : 0 : 1) & \text{if } P = (0, 0), \\ (x(x-1) : y(x+1) : x(x+1) : y(x-1)) & \text{if } P = (x, y) \neq (0, 0). \end{cases}$$

An inverse isomorphism $\varepsilon^{-1} : E \to M$ is

$$\varepsilon^{-1} : P \mapsto \begin{cases} \infty & \text{if } P = (0 : 1 : 0 : 1), \\ (0, 0) & \text{if } P = (0 : -1 : 0 : 1), \\ \left(\frac{T+X}{T-X}, \frac{Y+Z}{T-X}\right) & \text{if } P = (X : Y : T : Z) \neq (0 : \pm 1 : 0 : 1). \end{cases}$$

Computing the isomorphism ε from an uncompressed input

$$\begin{array}{cccc} a = x^2 & b = x & c = xy & d = y \\ X = a - b & Y = c + d & T = a + b & Z = c - d \end{array}$$

has cost $1\mathbf{M} + 1\mathbf{S}$, plus four field additions.

1.3 Endomorphisms

The curves E and M are special with complex multiplication (CM) by i.

1.3.1 Multiplication by i

The map $[i] : E \to E$ can be computed as:

$$[i](X : Y : T : Z) = (iT : Z : iX : Y), \tag{1.3}$$

with arithmetic cost $2\mathbf{M}$. The two multiplication are by a constant field element i (which perhaps permits some further optimization, which was not attempted in this report).

The shuffling of coordinates should also has a cost, but it should be much lower, so we do not include it here, or in other arithmetic costs. Such details are accounted for in the final runtime cost of the implementation.

1.3.2 Multiplication by 1+i

The map $[1 + i] : E \to E$ can be computed as:

$$[1 + i](x : y : t : z) = (ixz + yt : yz + ixt : xz + iyt : yz - ixt), \tag{1.4}$$

which can be computed with the intermediate steps

$$\begin{array}{cccc} E = x & F = z & G = y & H = it \\ a = EF & b = GH & c = HE & d = FG \\ X = i(a - b) & Y = d + c & T = a + b & Z = d - c \end{array}$$

for an arithmetic cost of approximately $6\mathbf{M}$. In more detail, it is four general field multiplication, plus two multiplications by the constant i, and four field additions, which tend to have relative low cost.

1.4 Hisil–Wong–Carter–Dawson double and add

The HWCD formulas for doubling and adding points are reviewed for the special case of our curve E.

1.4.1 Point addition

Given two input points $(X_j : Y_j : T_j : Z_j)$ for $j \in \{1, 2\}$, we can compute a point $(X : Y : T : Z) = (X_1 : Y_1 : T_1 : Z_1) + (X_2 : Y_2 : T_2 : Z_2)$ follows:

$$e = Y_1 - X_1 \qquad f = Y_2 - X_2 \qquad g = Y_1 + X_1 \qquad h = Y_2 + X_2 \tag{1.5}$$
$$a = ef \qquad b = gh \qquad c = 2T_1T_2 \qquad d = 2Z_1Z_2 \tag{1.6}$$
$$E = b - a \qquad F = d - c \qquad G = d + c \qquad H = b + a \tag{1.7}$$
$$X = EF \qquad Y = GH \qquad T = HE \qquad Z = FG. \tag{1.8}$$

The arithmetic cost is $8\mathbf{M}$. In detail, eight field general multiplication, and ten field additions.

1.4.2 Point doubling

Given a point $(X_1 : Y_1 : T_1 : Z_1)$, we can compute a point $(X : Y : T : Z) = 2(X_1 : Y_1 : Z_1)$ as follows:

$$a = X_1^2 \qquad b = Y_1^2 \qquad c = Z_1^2 \qquad d = (X_1 + Y_1)^2 \tag{1.9}$$
$$E = d - b - a \qquad F = -2c - a + b \qquad G = b - a \qquad H = b + a \tag{1.10}$$
$$X = EF \qquad Y = GH \qquad T = HE \qquad Z = FG. \tag{1.11}$$

The arithmetic cost is $4\mathbf{M} + 4\mathbf{S}$. In more detail, there four field squarings, four general field multiplication, and six field additions.

The step of computing $T = HE$ should usually be considered as part of a point addition, which we explain in the next section. This means a point doubling has arithmetic cost $3\mathbf{M} + 4\mathbf{S}$

1.4.3 Tracking the cost of T

In point doubling, the input coordinate T_1 is not used. The previous point operation, whether addition or doubling can skip the step of computing T_1. In the general, the step of computing this T coordinate took the form $T = HE$. Skipping it saves $1\mathbf{M}$.

In a sequence of repeated doublings, no intermediate T coordinates are needed, so each doubling costs $3\mathbf{M} + 4\mathbf{S}$.

The last addition addition before a sequence of doublings, can save $1\mathbf{M}$, but then the last doubling in the sequence will need to compute T for the next point addition, at a cost $1\mathbf{M}$.

By attributing the extra cost of computing T in the final doubling to the addition before the first doubling, we can attribute $8\mathbf{M}$ to each point addition, and $3\mathbf{M} + 4\mathbf{S}$ to each point doubling.

2 Precomputing a table

Given a point P, we pre-compute 64 fixed multiples of P. Dropping the square bracket notation, and abbreviating $\tau = 1 + i$, the points are:

$P_{0,0} = O$	$P_{1,0} = P$	$P_{1,1} = \tau P$	$P_{0,1} = iP$
$P_{-1,1} = iP_{1,1}$	$P_{-1,0} = -P$	$P_{-1,-1} = -P_{1,1}$	$P_{0,-1} = -P_{0,1}$
$P_{1,-1} = -P_{-1,1},$	$P_{2,-1} = P + P_{1,-1}$	$P_{2,0} = \tau P_{1,-1}$	$P_{2,1} = P + P_{1,1}$
$P_{2,2} = \tau P_{2,0}$	$P_{1,2} = iP_{2,-1}$	$P_{0,2} = iP_{2,0}$	$P_{-1,2} = iP_{2,1}$
$P_{-2,2} = iP_{2,2}$	$P_{-2,1} = -P_{2,-1}$	$P_{-2,0} = -P_{2,0}$	$P_{-2,-1} = -P_{2,1}$
$P_{-2,-2} = -P_{2,2}$	$P_{-1,-2} = -P_{1,2}$	$P_{0,-2} = -P_{0,2}$	$P_{1,-2} = -P_{-1,2}$
$P_{2,-2} = -P_{-2,1}$	$P_{3,-2} = P + P_{2,-3}$	$P_{3,-1} = \tau P_{1,-2}$	$P_{3,0} = P + P_{2,0}$
$P_{3,1} = \tau P_{-1,2}$	$P_{3,2} = P + P_{2,2}$	$P_{3,3} = \tau P_{3,0}$	$P_{2,3} = iP_{3,-2}$
$P_{1,3} = iP_{3,-1}$	$P_{0,3} = iP_{3,0}$	$P_{-1,3} = iP_{3,1}$	$P_{-2,3} = iP_{3,2}$
$P_{-3,3} = iP_{3,3}$	$P_{-3,2} = -P_{3,-2}$	$P_{-3,1} = -P_{3,-1}$	$P_{-3,0} = -P_{3,0}$
$P_{-3,-1} = -P_{3,1},$	$P_{-3,-2} = -P_{3,2}$	$P_{-3,-3} = -P_{3,3}$	$P_{-2,-3} = -P_{3,2}$
$P_{-1,-3} = -P_{1,3}$	$P_{0,-3} = -P_{0,3}$	$P_{1,-3} = -P_{-1,3}$	$P_{2,-3} = -P_{-2,3}$
$P_{3,-3} = -P_{-3,3}$	$P_{4,-3} = P + P_{3,-3}$	$P_{4,-2} = \tau P_{1,-3}$	$P_{4,-1} = P + P_{3,-1}$
$P_{4,0} = \tau P_{2,-2}$	$P_{4,1} = P + P_{3,1}$	$P_{4,2} = \tau P_{3,-1}$	$P_{4,3} = P + P_{3,3}$
$P_{4,4} = \tau P_{4,0}$	$P_{3,4} = iP_{4,-3}$	$P_{2,4} = iP_{4,-2}$	$P_{1,4} = iP_{4,-1}$
$P_{0,4} = iP_{4,0}$	$P_{-1,4} = iP_{4,1}$	$P_{-2,4} = iP_{4,2}$	$P_{3,4} = iP_{4,-3}$

A case-by-case formula for $P_{j,k}$ is this:

$$P_{j,k} = \begin{cases} O & \text{if } (j,k) = (0,0) \\ P & \text{if } (j,k) = (1,0) \\ -P_{-j,-k} & \text{if } j + k < 0 \text{ or } (-k) = j > 0 \\ iP_{k,-j} & \text{if } k > 0 \text{ and } -k \leq j < k \\ P + P_{j-1,k} & \text{if } j > 1 \text{ and } j > |k| \text{ and } j \not\equiv k \bmod 2 \\ \tau P_{\frac{j+k}{2},\frac{k-j}{2}} & \text{if } j > 0 \text{ and } j > |k-1| \text{ and } j \equiv k \bmod 2 \end{cases} \tag{2.1}$$

The computation above can be visualized as a spiral, with the computation starting from 0, moving to 1, then moving up, and so on, the in the following array.

$$\begin{array}{|cccccccc|}
\hline
i & i & i & i & i & i & i & \tau \\
i & i & i & i & i & i & \tau & + \\
- & i & i & i & i & \tau & + & \tau \\
- & - & i & i & \tau & + & \tau & + \\
- & - & - & 0 & 1 & \tau & + & \tau \\
- & - & - & - & - & + & \tau & + \\
- & - & - & - & - & - & + & \tau \\
- & - & - & - & - & - & - & + \\
\hline
\end{array}$$

An iteration rule for the next pair (j', k') in the spiral after (j, k) is:

$$(j', k') = \begin{cases} (j+1, k) & \text{if } k \leq j \leq -k \\ (j, k+1) & \text{if } -j < k < j \\ (j-1, k) & \text{if } -k < j \leq k \\ (j, k-1) & \text{if } j < k \leq -j \end{cases} \tag{2.2}$$

Once we arrive at $(j, k) = (-4, 4)$ we should stop.

The total number of general field multiplications (not counting field multiplications by constant i), is thus:

$$10 \times 4 + 9 \times 8 = 112, \tag{2.3}$$

because each of the ten τ operations requires four general field multiplications, and each of the nine point additions requires eight general field multiplications. We should also count field multiplications by i, which we probably should, then there is an extra

$$10 \times 2 + 19 \times 2 = 58 \tag{2.4}$$

field multiplications. So, the total arithmetic cost of the pre-computation is approximately 170**M**.

2.1 Decompression

If input point P is received in compressed form, then the step of decompressing P will be necessary in the GLV+HWCD scalar multiplication. Decompression is part of the pre-computation phase, because it does not depend on the (secret) scalar.

This report has not focused on optimizing decompression. Instead, it has considered that main benefit of GLV+HWCD scalar multiplication will be speed-prioritized settings, where uncompressed points P will be transmitted.

2.2 Cofactor multiplication

Once a point P is validated for being on the curve, it often makes sense to either check that it has the correct prime order, or to modify it by using cofactor multiplication.

The curve M has a cofactor of 72, meaning that the $72n$ points on M, where n is a large prime. Actually, for any point P on M, the point $[12]P$ belongs to the order n subgroup. So, $[12]P$ has prime order n or 1. The latter case, means $[12]P = O$, and that P had order dividing 12.

In this case, when using Diffie–Hellman, if P is the received point of the peer, usually the peer's ephemeral public key, then we may wish to replace it by $[12]P$. Then we check that $[12]P \neq O$.

Alternatively, we can opt to not use cofactor multiplication. In this case, we might risk an attacker sending P with order other than n. The attack might be able to use such a point P of invalid order to learn $s \bmod 12$, where s is the secret scalar. Diffie–Hellman can probably tolerate an attacker learning $s \bmod 12$. For signatures, the attacker does not get to choose which P gets combined with a secret scalar. So, in typical applications, cofactor multiplication is perhaps not that helpful.

2.3 Validation

Given an uncompressed point P in the affine plane to which M belongs, the GLV+HWCD only works properly if point P lies on the curve M.

Checking that $P \in M$ can be tested at arithmetic cost of $1\mathbf{M} + 2\mathrm{S}$, along with two field additions.

There is slight reason to hope that GLV+HWCD is not vulnerable to attacks based on point $P \notin M$. Perhaps, complex multiplication or the HWCD formula themselves, mean that when secret scalar applied to the invalid point, nothing predictable is leaked to the attacker. However, the cost of validation is so small, that it is better to validate the point than to rely on this hope.

3 Scalar multiplication

Suppose that we have two polynomials t and u of degree 45, with coefficients in the set the $\{-3, -2, -1, 0, 1, 2, 3, 4\}$.

Let $s = t(8) + iu(8) \bmod n$, where n is the order of the point P, and i is an integer such that $i^2 = -1 \bmod n$. We wish to compute $[s]P$.

(Finding polynomials t and u given arbitrary integer s, is called re-coding, but is not the focus of this report.)

The coefficients of t and u give indices into the pre-computed table. The resulting is multiplied by, with three consecutive point doublings. Then another point from the pre-computed is added. This takes 44 point additions, and 44 point doublings. The total arithmetic cost is:

$$44 \times (8\mathbf{M} + 3 \times (3\mathbf{M} + 4\mathbf{S})) \qquad = \qquad 748\mathbf{M} + 528\mathbf{S}. \tag{3.1}$$

3.1 Forming a scalar-dependent array

Mitigations against cache-timing side channel attacks include using very little memory, and avoiding secret-dependent table lookups. These mitigations seem not to work for GLV+HWCD.

Instead, an alternative method has been tried (but not tested). Do all the secret table-lookups by copying between the pre-computed table to a table of coefficients. No elliptic curve or field arithmetic is done between any the point copying steps. Hopefully, copying is fast enough to reduce the time window of opportunity for a cache miss to a short enough time to render the attacker's task too difficult.

3.2 Normalization

In most situations, after computing sP, the next step is to convert sP to a normalized form, such as affine Montgomery form.

For the field of size $8^{91}+5$, a conversion from extended Edward coordinate to affine Montgomery coordinates can be done with a cost of $4\mathbf{M} + 273\mathbf{S}$.

4 Arithmetic cost per bit

Assuming an uncompressed point P on curve M, and s given by the polynomials t and u, the arithmetic cost of computing $x([s]P)$ by the GLV+HWCD method is

$$(2\mathbf{M} + 3\mathbf{S}) + (170\mathbf{M}) + (748\mathbf{M} + 528\mathbf{S}) + (4\mathbf{M} + 273\mathbf{S}) = 924\mathbf{M} + 804\mathbf{S}$$

This ignored cofactor multiplication.

Considering the scalars as 273 bits, then the cost per is approximately:

$$3.38\mathbf{M} + 2.95\mathbf{M}.$$

A Speculative questions

Does GLV+HWCD under-utilize the specialness of curve $2y^2 = x^3 + x$? Both optimization methods, GLV and HWCD, work for curves more generic than $2y^2 = x^3 + x$.

The GLV method optimizes quite a few less special curves. The curve needs an efficient endomorphism, but it does not need complex multiplication by i. It is believe the GLV offers an advantage if the endomorphism has a speed competitive with a few doublings. Also, the version of GLV described in this report uses the endomorphisms $[i]$ and $[i + 1]$, many times, not just once as was the case in GLV. Perhaps, straying from the original way that GLV used the endomorphism resulted less of an acceleration.

The HWCD method optimizes many far less special curves. Up to a quarter of all randomly chosen curves can be optimized using HWCD. Any curve with order divisible by four is isomorphic to an twisted Edwards curve, and thus optimizable with HWCD. Maybe elliptic curve experts as clever as Montgomery, Hisil, Wong, Carter or Dawson, could, if they felt like it, find new point addition formulas that are specialized to curves with three-term equations, such as $2y^2 = x^3 + x$. Besides being just too greedy, why not ask them to find a point addition formula with arithmetic cost 6**M**, instead of 8**M** formula for four-term curves?

B Sample code

The following sample code is an edited excerpt pasted from an preliminary experimental implementation of some ideas in this report. The code is very incomplete, and has not been tested for quality. It is likely vulnerable to side channel attacks, particular cache-timing attacks.

```
#include "8^91+5.c"
#define _2(E) (1<<(0 E))
#define LEF(a,B)     (mal(a,1,B))
typedef struct ed_s { f X,Y,T,Z;} ed;
ed o = {{0},{1},{0},{1}};
int ed_on_curve (ed p, int t, int debugging)
{
  int r;   f a,b;
  if (t) {
mul(a,p.X,p.Y);
mul(b,p.T,p.Z);
r = eq(a,b);
squ(p.X,p.X);
squ(p.Y,p.Y);
squ(p.T,p.T);
squ(p.Z,p.Z);
sub(a,p.Y,p.X);
add(b,p.T,p.Z);
r *= eq(a,b);
r *= 1- eq(p.X,_0)*eq(p.Y,_0)*eq(p.T,_0)*eq(p.Z,_0);
  } else {
squ(p.X,p.X);
squ(p.Y,p.Y);
squ(p.Z,p.Z);
mul(p.T,p.X,p.Y);
mul(p.X,p.X,p.Z);
mul(p.Y,p.Y,p.Z);
squ(p.Z,p.Z);
sub(a,p.Y,p.X);
add(b,p.T,p.Z);
r = eq(a,b);
r *= 1- eq(p.X,_0)*eq(p.Y,_0)*eq(p.Z,_0);
  }
  return r;
}
inline FUN ed_n (ed*q, ed p)
{
  *q=p;
  mal(q->X, -1, p.X);
  mal(q->T, -1, p.T);
}
inline FUN ed_i (ed*q, ed p)
{
  mul(q->X, I, p.T);
  LEF(q->Y,    p.Z);
  mul(q->T, I, p.X);
  LEF(q->Z,    p.Y);
}
inline FUN ed_EFGH (ed *q, ed p, int t)
{
  mul(q->X, p.X, p.Y); // EF
  mul(q->Y, p.T, p.Z); // GH
  if(t)
  mul(q->T, p.Z, p.X); // HE (optional)
  mul(q->Z, p.Y, p.T); // FG
}
inline FUN ed_tau (ed*q, ed p)
{
  LEF(q->X,  p.X);
  LEF(q->Y,  p.Z);
  LEF(q->T,  p.Y);
  mul(q->Z,I,p.T);
  ed_EFGH(&p,*q,1);
  sub(q->X, p.X, p.Y);
  add(q->Y, p.Z, p.T);
  add(q->T, p.X, p.Y);
  sub(q->Z, p.Z, p.T);
  mul(q->X,I,q->X);
}
FUN ed_add (ed*r, ed p, ed q, int t)
{
  f a,b,c,d,e,f,g,h ;
  sub(e,p.Y,p.X), sub(f,q.Y,q.X);
  add(g,p.Y,p.X), add(h,q.Y,q.X);
  mul(a,  e,  f);
  mul(b,  g,  h);
  mul(c,p.T,q.T);
  mul(d,p.Z,q.Z);
  mal(c,2,c), mal(d,2,d);
  sub(p.X, b, a);
  sub(p.Y, d, c);
  add(p.T, d, c);
  add(p.Z, b, a);
  ed_EFGH(r,p,t);
}
inline FUN ed_dub (ed *q, ed p, int t)
{
  add(q->Z, p.X,p.Y);
  squ(p.X, p.X);
  squ(p.Y, p.Y);
  squ(p.T, p.Z);
  squ(p.Z,q->Z);
  add(q->Z,   p.Y,  p.X);  // H = b+a
  sub(q->T,   p.Y,  p.X);  // G = b-a
  mal(q->Y,    -2,  p.T);  // F = -2c
  add(q->Y,  q->Y, q->T);  //      + b-a
  sub(q->X,   p.Z, q->Z);  // E = d - b-a
  ed_EFGH(q,*q,t);
}
FUN ed_dbs(ed *p, int e)
{
  int i=e;
  for (;i--;) ed_dub (p, *p, !i);
}
FUN ed_3(ed *q, ed p)
{
ed_dub(q,p,1);
ed_add(q,*q,p,0);
}
FUN ed_12 (ed *q, ed p)
{
  ed_3(q,p);
  ed_dbs(q, 2);
}
```

```
#define W 3 // 2
#define P(j,k)  (w[(j)+M-1][(k)+M-1])
FUN glv_win_spiral ( ed w[_2(^W)][_2(^W)], ed p,  int cofactor)
{
  int j,k,t,M=_2(^(W-1));
  if (cofactor) ed_12(&p,p);
  for (j=k=0;  j>-M;
 t = j + (k <=j && j<=-k) - (-k<j && j<= k),
 k = k + (-j< k && k<  j) - ( j<k && k<=-j),
 j = t
   ) {
if ( j+k<0 || (-k == j && j > 0))
  ed_n(&P(j,k),
   P(-j,-k));
else if (k>0 && -k<=j && j <k)
  ed_i(&P(j,k),
   P(k,-j));
else if (j>0 && j>-(k-1) && j >(k-1) && (j-k)%2==0)
  ed_tau(&P(j,k),
 P((j+k)/2, (k-j)/2));
    else if (j>1 && j>-k && j>k  && (j-k)%2!=0)
  ed_add(&P(j,k),
 p, P(j-1,k), 1 );
else if (j==1 && k==0)
  P(j,k) = p ;
else if (j==0 && k==0)
  P(j,k) = o ;
  }
}
#undef P
typedef struct {
  int degree ;
  int coeff[300];
} poly;
FUN glv_fill (ed q[], ed w[_2(^W)][_2(^W)], poly *s)
{
  int c,k,M=_2(^W);
  for (k=0;k<s->degree;k+=1) {
```

```
c=s->coeff[k];
if ((W > 2)  || 1) {
  q[k]=w[c/M][c%M];
}
  }
}
FUN glv_mul (ed*sp, poly *s, ed p, int cofactor)
{
  unsigned i ;
  ed w[_2(^W)][_2(^W)], q[s->degree] ;
  glv_win_spiral(w,p,cofactor);
  glv_fill(q,w,s);
  for(i=0;i<s->degree-1;i+=1){
if (0==i)
  *sp=q[i];
else
  ed_add(sp, *sp, q[i], W==1);
ed_dbs(sp, W);
  }
  ed_add(sp, *sp, q[i], 1);
}
FUN to_eddie (ed*p,  f x, f y)
{
  f xx,xy;
  mul(xx,x,x),mul(xy,x,y);
  sub(p->X,xx,x);
  add(p->Y,xy,y);
  add(p->T,xx,x);
  sub(p->Z,xy,y);
}
FUN to_monty (f x, f y, ed p)
{
  /* To do: check if T=X */
  f tsx,tax,yaz ;
  add(tax,p.T,p.X), sub(tsx,p.T,p.X), add(yaz,p.Y,p.Z);
  inv(tsx,tsx);
  mul(x,tax,tsx), mul(y,yaz,tsx);
  fix(x),fix(y);
```

```
}
FUN decompress(f x, f y)
{
  /* To do: public key validation */
  f t;
  squ(t,x)  ;
  mul(t,x,t);
  add(t,x,t);
  mul(t,t,(f){3,0,0,0,(1LL<<52)});
  root(y,t);
}
```