A note on the calculation of some functions in finite fields: Tricks of the Trade

Michael Scott

Cryptography Research Centre
Technical Innovation Institute
michael.scott@tii.ae

Abstract. Optimization of finite field arithmetic is important for the deployment of public key cryptography, particularly in the context of elliptic curve cryptography. Until now the primary concern has been operations over the prime field $\mathbb{F}_p$, where p is a prime. With the advent of pairing-based cryptography there arises a need to also look at optimal arithmetic over extension fields $\mathbb{F}_{p^n}$ for small values of n. Here we focus on the determination of quadratic residuosity and the calculation of inverses and square roots over these fields, operations often carried out in conjunction with one another. We demonstrate with a minor improvement in a hash-to-curve algorithm, and a major speed-up in the calculation of square roots in quadratic extensions.
Keywords: Elliptic Curves, Pairing-based cryptography, Implementations

1 Introduction

Finite field arithmetic is a well established topic in number theory. However when moving from theory to practise such that implementations are optimal, an implementor must take into account the fact that the theoretically asymptotically fastest method is not necessarily the best (or indeed even the fastest) in the context of cryptography, where the range of interest is typically restricted to numbers of a few hundreds or thousands of bits. The need to bake in side channel resistance in the form of constant time implementation, is also a major consideration, as some algorithms are more constant-time-friendly than others. Typically implementors avoid extreme methods that are theoretically faster for very large numbers, and consider that they are working in a context where school-boy (or at a stretch Karatsuba-based) methods of basic arithmetic are optimal, or close to optimal. Even then some extra insight is required, which often manifests in the deployment of what are sometimes called "tricks". We start by reviewing some of these tricks as they apply over the base field $\mathbb{F}_p$, where p is a large odd prime modulus.

1.1 Modular inversion

Modular inversion is a good place to start. As is well known the fastest method for modular inversion is based on the extended Euclidean algorithm (algorithm

2.107 [12]). If we assume the modulus is of size m bits then the complexity of the method is $\mathcal{O}(m^2)$. The inverse is found indirectly by applying the extended Euclidean algorithm given x and p to find a and b such that $ax + bp = \gcd(x, p)$. As x and p are co-prime then $ax + bp = 1$ and $ax = 1 \bmod p$, and therefore $a = 1/x \bmod p$. This method is fast, but difficult, although not impossible [5], to implement in constant time (although as an alternative to a constant time implementation, random blinding might also be used to protect against side-channel attacks, see [5] footnote 1).

An alternative method based on Fermat's Little Theorem (fact 2.127 [12]) calculates the inverse of a non-zero field element directly as $a = 1/x = x^{p-2} \bmod p$. This method, based as it is on modular exponentiation, has a complexity of $\mathcal{O}(m^3)$. Not only is it slower in theory, it is slower in practice as well. However the Fermat based method is commonly used as it is much easier to implement in constant time. And as we will see it has other advantages. Another consideration is that it is often possible to implement cryptographic algorithms in such a way that modular inversions are only rarely required, for example by using projective coordinates in the context of elliptic curve cryptography, in which case performance may not be an issue and the slower but simpler and safer Fermat approach may be preferred.

1.2 Quadratic residuosity

Here we seek to determine whether or not a non-zero field element x is, or is not, a quadratic residue (QR) with respect to a prime modulus p. If it is, it has two square roots, if it is not, it does not have any square roots. In the context of a prime modulus this requires the calculation of the Legendre symbol. Again there are two approaches. The fastest way is to exploit the law of quadratic reciprocity in a Euclid-like algorithm (algorithm 2.149 [12]), but again this is difficult to implement in constant time. But as with modular inversion there is also a simpler, but slower, method based on modular exponentiation which calculates quadratic residuosity as the value of $x^{(p-1)/2} \bmod p$. If this evaluates as $+1$ then x is a quadratic residue and one can go on to calculate its square roots. Alternatively if it evaluates as $p-1$, that is $-1 \bmod p$, then x is a quadratic non-residue (QNR), and has no square roots.

1.3 Modular square roots

If an element x is a quadratic residue then we can go on to find a square root. In this case there is no avoidance of modular exponentiation. Furthermore the solution becomes dependent on the exact form of p. Often an assumption is made that $p = 3 \bmod 4$, which allows square roots to be evaluated as $\pm x^{(p+1)/4}$, and also conveniently guarantees that -1 is a quadratic non-residue. Here we avoid specialising in this way, as in fact cases where $p \neq 3 \bmod 4$ are quite common ($p = 5 \bmod 8$ often arises [6], and see also the recently proposed JubJub [17] and Tweedledum/Tweedledee [7] elliptic curves). To this end we introduce

the Tonelli-Shanks method (algorithm 3.34 [12]) for calculating modular square roots.

First we categorise the prime moduli according to the value of e, where e is the maximum integer such that $2^e | p-1$. We will assume that e is relatively small. Also needed will be a precomputed 2^e-th root of unity, that is $z = d^{(p-1)/2^e}$, where d is any small quadratic non-residue. Then the square root is found by first calculating

$$y = x^{(p-2^e-1)/2^{e+1}} \bmod p$$

using modular exponentiation. For reasons that will become clear we call this y value the *progenitor*. Then the algorithm to find the square root of x, the Tonelli-Shanks algorithm, proceeds as follows (in a Pythoneque pseudo-code, where the `cmov` function moves the second parameter into its first parameter if the condition specified in its third parameter is true. Such a function is a staple of constant-time implementation).

```
s=y*x
t=s*y
for k in range(e,1,-1) :
    b=t
    for i in range(1,k-1) :
        b*=b
    cmov(s,s*z,b!=1)
    z*=z
    cmov(t,t*z,b!=1)
```

The square root will be the final value of s. Observe that in the case where $e = 1$ the for loop is not executed, and the calculation collapses into that described above. For small values of e there is a minor amount of extra work, still dominated by the initial calculation of y. Note that this calculation can be carried out even if x is not a quadratic residue, although of course it will not return a correct answer. Note also that outside of this square root calculation, we are no longer concerned with the form of p, and all of the complications that can arise from special case handling are thus avoided.

2 Some useful tricks

Consider the calculation of multiple inverses of a diverse set of field elements. In theory of course this is not interesting, simply calculate them all individually. However there is a much better way in practice. The idea, known as "Montgomery's trick", due to Peter Montgomery, is for example to calculate $1/x$ and $1/y$ modulo p as $1/x = y/(xy)$ and $1/y = x/(xy)$. Now only one expensive modular inversion of $1/(xy)$ is required, and indeed it is easy to see that the same basic idea extends to n modular inversions which can be found at the cost of a single inversion. This trick has surprisingly widespread application in a number

of different contexts. For example multiple elliptic curve points can be translated from projective to affine form at the major cost of just a single modular inversion.

In some situations inverses and square roots are needed in combination. First observe that from a precalculated progenitor y, we can quickly calculate the modular inverse after a few more multiplications and squarings as

$$1/x = x^{p-2} = x^{2^e-1}.y^{2^{e+1}} = x^{2^{e-1}-1}.(xy^4)^{2^{e-1}} \bmod p$$

The inverse square root trick [6] finds $\sqrt{u/v}$ as

$$\sqrt{u/v} = u^2.\frac{1}{u^3v}.\sqrt{u^3v}$$

Clearly the same progenitor value for $x = u^3.v$ can be used for both the inversion and the square root. As claimed in [6] the overall calculation "takes just a few multiplications more than a single exponentiation". Obviously setting $u = 1$ finds the inverse of the square root.

Hamburg takes this idea further [11], and demonstrates that an inversion and a square root of two distinct elements can be found with a single exponentiation, that required to calculate a single progenitor, this time for $x = u^2.v$.

$$1/u = uv/(u^2.v)$$
$$\sqrt{v} = (1/u).\sqrt{u^2.v}$$

This idea can be considered as an extension of Montgomery's trick: It is now possible to calculate multiple inverses and a single square root modulo a prime p with just one exponentiation. However there remains an issue - what if the square root does not exist, that is if the x value is a quadratic non-residue? Clearly the inverse square root trick breaks down completely. The Hamburg trick still returns the correct inverse, but a false square root.

Fortunately quadratic residuosity can also be determined from the same progenitor, as the value of $(xy^2)^{2^{e-1}} = x^{(p-1)/2}$. So now at little extra cost we can easily flag if the inverse square root and Hamburg tricks have in fact returned valid square roots. Alternatively one can simply go ahead and apply the Tonelli-Shanks method, and test its output for correctness by squaring it.

2.1 Tweaking the Progenitor

It may be that in some circumstances a calculation may involve consideration of two inputs, one of which differs from x by a small fixed multiplier c. For example if our attention should switch from x to cx, then the progenitor for cx would be $c^{(p-2^e-1)/2^{e+1}}.y$, where y is the progenitor derived from x. In these cases $c^{(p-2^e-1)/2^{e+1}}$ can be stored as a precalculated constant, and the progenitor for cx obtained for the extra cost of a single multiplication. This might occur in a context where such a tweak may result in generating a cx which is a quadratic residue.

3 Application – Point Decompression

It is common for an elliptic curve point to be transmitted in compressed form, typically as just the x coordinate plus a single bit to indicate the correct sign of y. Decompression involves inserting x into the curve equation and solving for y, using the single bit to indicate the correct square root. For curves in Weierstrass form, this is quite straightforward. However for a twisted Edwards curve in the form $ax^2 + y^2 = 1 + dx^2y^2$, at first glance this appears more complicated.

$$y = \pm\sqrt{(ax^2 - 1)/dx^2 - 1)}$$

But of course as pointed out by [6], this requires a simple application of the inverse square root trick to reduce the cost again to just one exponentiation.

4 Application – Hash to Curve

Constant time hashing of a string to a point on an elliptic curve is one of the more complex requirements for many applications of elliptic curve cryptography. A randomly generated x coordinate will not necessarily map to an (x, y) point on an elliptic curve in short Weierstrass form, $y^2 = g(x) = x^3 + Ax + B$, as the right-hand side has only a 50% chance of being a quadratic residue for a randomly generated x. Therefore schemes have been proposed to ensure that an x value generated from a string can be pre-processed in such a way as to ensure that it does indeed coincide with a curve point. As always, we would like the operation to be fast, as well as for it to be constant time.

In the context of Edwards and Montgomery curves, the Elligator 2 method [4] exploits the inverse square root trick so that the cost of constant time hashing is essentially that of a single exponentiation. However the situation for Weierstrass curves has not been so clear, until the recent work of Wahby and Boneh [16] who demonstrated that in most cases a single exponentiation is again sufficient. For Weierstrass curves the simplified SWU method appears to be the method of choice [15], [8], [16]. However it only applies where $A \neq 0$ and $B \neq 0$, and unfortunately curves that violate this condition are quite commonly used, particularly in the context of pairing-based cryptography [13]. The Wahby-Boneh solution is to find an isogeneous curve with non-zero A and B, hash to that curve, then use the isogeny to return the point to the original curve. This process, while complicated, is computationally fast as long as a suitable isogeny can be found.

Here we review the Wahby-Boneh method, and by careful use of our bag of tricks show how it may be implemented at the cost of a single exponentiation, plus some negligible amount of work including an optional application of an isogeny. In the process we make a small improvement to the scheme as originally described.

Concentrating on the implementational aspects of the method, we start by generating a carefully crafted x_1 value derived from the input u in a rational form $x_1 = n/d$. Substituting this into the curve equation we find $g(x_1) = (n^3 + And^2 + Bd^3)/d^3 = f/d^3$. By construction we also have a second candidate $x_2 =$

$(Z.u^2).x_1$, such that $g(x_2) = (Z.u^2)^3.g(x_1) = Z.(Z.u^3)^2.g(x_1)$, where Z is a small QNR. Either $g(x_1)$ or $g(x_2)$ will be a QR, as the product of two QNRs is a QR. We start by calculating a progenitor value based on fd, this being our one and only exponentiation. Clearly the quadratic residuosity of $g(x_1)$ can be determined by testing whether or not fd is a QR. To find the candidate x coordinates, we also use the progenitor to calculate $1/fd$ and hence $1/d$, and then x_1 and x_2 as non-rational field elements.

If $g(x_1)$ is not a quadratic residue, then $Z.g(x_1)$ will be, so next we tweak the progenitor for this latter case by multiplying it by a precomputed constant derived from Z. We select one or the other of this pair along with x_1 or x_2 in constant time depending on the original QR test, calculate the square root, and finally obtain y by multiplying by a correction factor of either $1/d^2$ or $Z.u^3/d^2$, again selected depending on the QR test. See the appendix for a pseudo-code implementation.

Since our resulting point (x, y) is in affine co-ordinates, it is now a simple matter to apply an isogeny if such should be necessary, producing a final result in projective coordinates. As originally described by Wahby-Boneh the method required the isogeny to handle input points in projective coordinates, which required more complex processing, see [16] section 4.3.

5 Extension fields

First we recap on the *field norm* of a finite extension field element $\in \mathbb{F}_{q^m}$, where q is a prime power. If $K = \mathbb{F}_q$ is the base field and $L = \mathbb{F}_{q^m}$ is the extension, then the field norm of an element α in the extension field L can be calculated as $N_{L/K}(\alpha) = \alpha.\alpha^q...\alpha^{q^{m-1}}$. Field norms have a nice multiplicative property, such that $N_{L/K}(ab) = N_{L/K}(a).N_{L/K}(b)$. Calculation of the norm is simplified by exploiting the Frobenius map, that is the exponentiation of a field element to the power of q, which is a cheap operation.

Next we consider the quadratic extension $L = \mathbb{F}_{p^2}$ over $K = \mathbb{F}_p$ where p is a prime, as it arises in the context of BN [3] and BLS12 [2] pairing friendly curves, common targets of standardisation attention due to their ability to reach the AES-128 level of security, and their relatively good performance. Recall that a pairing is a mapping $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ [13] and that for both the BN and BLS12 curves the group $\mathbb{G}_2$ is of elliptic curve points over the quadratic extension field $\mathbb{F}_{p^2}$.

Elements in the quadratic extension field can be represented as $a + ib$, were $a, b \in \mathbb{F}_p$, and $i^2 - \beta = 0$, where β is a small quadratic non-residue. In the case that $p = 3 \bmod 4$, then $\beta = -1$ and $i = \sqrt{-1}$ and the analogy with complex arithmetic can be helpful. In this case the *norm* of the field element is $N_{L/K}(a + ib) = \delta = a^2 - \beta b^2 \in \mathbb{F}_p$. When the context is obvious we will drop the norm subscript notation.

An important observation is that while the cost of quadratic residuosity testing and multiplicative inverses do not rise above the cost of a single exponentiation over the base field $\mathbb{F}_p$, the cost of square rooting will increase.

5.1 Inversion

The inverse of an element in $\mathbb{F}_{p^2}$ can be descended to an exponentiation in $\mathbb{F}_p$ using the inverse of the norm, and a multiplication by the conjugate.

$$1/(a+ib) = (a-ib)/\delta$$

5.2 Quadratic Residuosity

The determination of the quadratic character of $a+ib$ can similarly be descended to an exponentiation in $\mathbb{F}_p$ by exploiting the Frobenius operator, as shown by [1]. It is simply the quadratic residuosity of the norm, $\delta^{(p-1)/2}$ in $\mathbb{F}_p$.

5.3 Square roots

The cost of square roots over quadratic extensions was considered in detail by Adj and Rodriguez-Henriquez [1]. An old idea from complex arithmetic [14], [10] provides us with a simple "complex" formula for calculating square roots in this extension field.

$$\sqrt{a+ib} = \pm(\sqrt{(a \pm \sqrt{a^2-\beta b^2})/2} + i.(b/2)/(\sqrt{(a \pm \sqrt{a^2-\beta b^2})/2}))$$

It should be clear from the above that a progenitor calculated from the $\mathbb{F}_p$ norm of $\delta = a^2 - \beta b^2$ allows the full determination of the quadratic character of $a+ib$ and a partial determination of the square root of $a+ib$ in the form of $\sqrt{\delta}$. However if $\delta = N(a+ib)$ is a quadratic residue in $\mathbb{F}_p$ we can be assured that $a+ib$ is a quadratic residue, and hence that its square roots exist.

If $a+ib$ is a quadratic residue, then one of $(a+\sqrt{a^2-\beta b^2})/2$ or $(a-\sqrt{a^2-\beta b^2})/2$ will be a quadratic residue in $\mathbb{F}_p$. And indeed this must be the case since their product is equal to $\beta b^2/4$. This is a QNR times a perfect square, and hence a QNR. If a QNR is the product of two other field elements then one of them must be a QR, and the other a QNR. Therefore substituting δ for $a^2 - \beta b^2$

$$(a-\sqrt{\delta})/2 = (\beta b^2/4)/((a+\sqrt{\delta})/2)$$

and

$$\sqrt{(a-\sqrt{\delta})/2} = \pm(b/2).\sqrt{\beta/((a+\sqrt{\delta})/2)}$$

So now the two candidates for the correct square roots are

$$\sqrt{a+ib} = \pm(\sqrt{(a+\sqrt{\delta})/2} + i.(b/2)/(\sqrt{(a+\sqrt{\delta})/2}))$$

and

$$\sqrt{a+ib} = \pm(b/2)/\sqrt{((a+\sqrt{\delta})/2)/\beta)} + i.\sqrt{((a+\sqrt{\delta})/2)/\beta))}$$

Now the point of this development becomes clear: A single progenitor for $(a+\sqrt{\delta})/2$ will now suffice for all required square roots and inversions. In the case that the second pair of square roots are the correct ones, the progenitor can be tweaked by multiplication by a precalculated constant $N(\beta^{-1})^{(p-2^e-1)/2^{e+1}}$. For a constant time implementation both should be calculated, with the correct one selected based on the quadratic residuosity of $(a+\sqrt{\delta})/2$.

Adj and Rodriguez-Henriquez [1] cost this complex method of square root calculation at two residuosity tests, two square roots and one inversion. However it is now apparent that the residuosity tests can be absorbed into the square root costs, and that the inverse square root trick implies that the cost of the inversion can also be eliminated. It will also be apparent that an existing progenitor, probably from a prior residuosity test, can also be used to calculate $\sqrt{\delta}$, and hence the cost of the full square root can be reduced to just one extra exponentiation in $\mathbb{F}_p$, associated with the calculation of the square root of either $(a+\sqrt{\delta})/2$ or $((a+\sqrt{\delta})/2)/\beta$.

As a consequence we can conclude that hash-to-curve in the group $\mathbb{G}_2$ for a BN or BLS12 curve, using the same basic method as described above, now requires a total cost of just two $\mathbb{F}_p$ exponentiations. We note that a similar idea in the context of point decompression on the FourQ elliptic curve is described in [9].

5.4 Higher extensions

For standards currently under consideration the extension n in $\mathbb{G}_2$ would be $n = 2$, 4 and 8 respectively for BN/BLS12, BLS24 and BLS48 pairing friendly curves [13]. By applying the above method for the quadratic extension recursively, the cost of hash-to-curve will be n exponentiations in the base field $\mathbb{F}_p$. Whereas the quadratic character and inverse calculations descend again to a single $\mathbb{F}_p$ exponentiation, the cost of a final square root increases linearly with the extension.

As pointed out by Adj and Rodriguez-Henriquez [1], the case for odd extensions like $\mathbb{F}_{p^3}$ must be handled differently. Inverses and the test for quadratic residuosity of a field element $\alpha \in \mathbb{F}_{p^m}$ can again be descended to a multiplication by the inverse of the $\mathbb{F}_p$ norm $\delta = N(\alpha)$, and the quadratic residuosity of that same norm. For the square root the complex method does not appear to generalise easily, but the Tonelli-Shanks method can be used directly, based on a progenitor derived from the computation of α^d where $d = (p-2^e-1)/2^{e+1}$, where the bulk of the work has been reduced to this smaller calculation by suitable application of the Frobenius as described in [1], [14], using the identity

$$(p^m - 2^e - 1)/2^{e+1} = d + p[pd + (2^e+1)d + 2^{e-1} + 1] \sum_{i=0}^{(m-3)/2} p^{2i}$$

6 Conclusion

While non-exponentiation methods for calculating modular inverses and for determining quadratic residuosity may be faster for one-off calculations, in many cases exponentiation-based methods are to be preferred on the grounds of simplicity, constant-time-friendliness, and the fact that an initial progenitor calculation may in fact find multiple uses and re-uses in the context of more complex algorithms. Implementation of the methods described here can be found (in a range of popular programming languages) in the code at `https://github.com/miracl/core`.

7 Acknowledgements

Thanks to Rene Struik for feedback and for drawing our attention to reference [9].

References

1. G. Adj and F. Rodriguez-Henriquez. Square root computation over even extension fields. *IEEE Transactions on Computers*, 63(11):2829–2841, 2014.
2. P. S. L. M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks – SCN'2002*, volume 2576 of *Lecture Notes in Computer Science*, pages 263–273. Springer-Verlag, 2002.
3. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC'2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 2006.
4. D. Bernstein, M. Hamburg, M. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on computer and communications security*, pages 967–980, 2013.
5. D. Bernstein and B-Y. Yang. Fast constant-time gcd computation and modular inversion. Cryptology ePrint Archive, Report 2019/266, 2019. `http://eprint.iacr.org/2019/226`.
6. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77–89, 2012. `http://eprint.iacr.org/2011/368`.
7. S. Bowe, J. Grigg, and D. Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. `http://eprint.iacr.org/2019/1021`.
8. E. Brier, J-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *CRYPTO 2010*, pages 237–254, 2010.
9. C. Costello and P. Longa. SchnorrQ: Schnorr signatures on fourQ, 2016. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/SchnorrQ.pdf`.

10. P. Friedland. Algorithm 312: Absolute value and square root of a complex number. *Commun. ACM*, 10(10), 1967.
11. M. Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. `http://eprint.iacr.org/2012/309`.
12. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, Boca Raton, Florida, 1996. URL: `http://cacr.math.uwaterloo.ca/hac`.
13. N. El Mrabet and M. Joye, editors. *Guide to Pairing-Based Cryptography*. Chapman and Hall/CRC, 2016. `https://www.crcpress.com/Guide-to-Pairing-Based-Cryptography/El-Mrabet-Joye/p/book/9781498729505`.
14. M. Scott. Implementing cryptographic pairings. In *Pairing-based Cryptography – Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 177–196, 2007.
15. A. Shallue and C. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *Algorithmic Number Theory*, pages 510–524, 2006.
16. R. S. Wahby and D. Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. In *IACR Transactions on Cryptographic Hardware and Embedded Systems CHES 2019*, pages 154–179, 2019.
17. B. WhiteHat, J. Baylina, and Marta Belles. Generation of twisted edwards elliptic curves for circuit use, 2019. `https://docs.zkproof.org/pages/standards/accepted-workshop2/proposal--elliptic-curve-generation.pdf`.

Simplified SWU, with affine output

This code implements the simplified SWU method, and hashes a random field element u to a point (x, y) on an elliptic curve $y^2 = x^3 + Ax + B$ defined over $\mathbb{F}_p$, with $A \neq 0$ and $B \neq 0$. The prime modulus p can be of any form. The Z value is a small non-residue in $\mathbb{F}_p$ (which may need to meet some extra conditions). For clarity we omit exceptional case handling. If e is the largest integer to divide $p-1$, then the precomputed tweak $w = Z^{(p-2^e-1)/2^{e+1}}$. The code is constant time, and requires only one large modular exponentiation. The function `qres(x,g)` returns a boolean to indicate whether or not its first argument x is a quadratic residue, and uses the precalculated progenitor value g. Similarly `inverse(x,g)` calculates a modular inverse, and `sqrt(x,g)` calculates a modular square root using the Tonelli-Shanks method (see above). The function `cmov`$(\mathtt{x}, \mathtt{y}, \mathtt{b})$ returns the first argument if the boolean b is true, otherwise it returns the second argument, in constant time.

```
Input Z,e,w,u,A,B
Output x,y

s=sgn0(u)

j=Z*u^2
k=j^2+j

d=A*k

x=-B*(k+1)
x2=x*j

gx=x^3+A*d^2*x+B*d^3

t=d*gx
g=t^[(p-2^e-1)/2^(e+1)]
qres=qr(t,g)
d=inverse(t,g)

d=d*gx
x=x*d
x2=x2*d
d=d^2

d2=d*j*u
t2=t*Z
g2=g*w

x=cmov(x,x2,!qres)
d=cmov(d,d2,!qres)
t=cmov(t,t2,!qres)
g=cmov(g,g2,!qres)

y=sqrt(t,g)
y=y*d
y=cmov(y,-y,sgn0(y)!=s)
```