Slothful reduction

Michael Scott

MIRACL.com
mike.scott@miracl.com

Abstract. In the implementation of many public key schemes, there is a need to perform modular arithmetic. Typically this consists of addition, subtraction, multiplication and (occasionally) division with respect to a prime modulus. To resist certain side-channel attacks it helps if implementations are "constant time". But as the calculations proceed there is potentially a need to reduce the result of an operation to its remainder modulo the prime modulus. At first glance whether or not a reduction is required, seems dependent on the data being processed, and hence a constant time implementation would be difficult to achieve. However often this reduction can be delayed, a process known as "lazy reduction". The idea is that results do not have to be fully reduced at each step, and that full reduction takes place only occasionally. Here we extend this idea to determine the circumstances under which full reduction can be delayed to the very end of a particular public key operation, and the entire operation executed in constant time.

1 Introduction

It is generally accepted that a good first line of defense against side-channel attacks is to ensure that our cryptographic code is "constant time", that is its execution profile is the same irrespective of the data it is processing. One way to achieve this is to ensure that a single execution path is followed in all cases, which implies as a bonus that testing will be simpler, as only one execution path needs to be debugged.

Although our general technique has application wherever modular arithmetic is required, here we describe it mostly in the context of elliptic curve cryptography. Indeed it has often been demonstrated in the past that it is perfectly possible to write constant time code to implement elliptic curve cryptography [3]. For a quick illustration inspect the code here [1], which in a few hundred lines implements a point multiplication on an Edwards curve over the field $\mathbb{F}_p$, where $p = 2^{521} - 1$, a Mersenne prime. It is striking that the code has no conditional branching instructions. All of the functions consist of "straight-line" compiler-friendly code. All loops could be fully unrolled by an optimizing compiler. However this implementation (and many others in the literature) is for a particular case, and does not suggest a general strategy that would work in all cases. It is the novel contribution of this paper to suggest such a general

[1] indigo.ie/~mscott/ed521.cpp

strategy, that leads to a cook-book approach that can be used by any competent programmer, who may not have any particular cryptographic expertise.

What often hinders the delivery of constant time code is (a) the way in which the finite field arithmetic is implemented, and (b) the elliptic curve arithmetic may involve “exceptions”, which are special actions which may be required for particular inputs. This latter issue can be dealt with by always implementing elliptic curve crypto using “exception-free” formulae for point addition and doubling. Thankfully efficient exception-free formulations are now available [4], [12], and so there is really no excuse not to always use them. Therefore here we concentrate on the finite field arithmetic.

Say we were tasked with implementing elliptic curves over a 256 bit field. In the past a plausible starting point would have been to represent 256 bit numbers as 16x16 bit words, or 8x32 bit words, or 4x64 bit words depending on our processor (and its registers) being 16, 32 or 64 bits. Then implement the field arithmetic (addition, subtraction, multiplication and occasional division) on top of that. Multiplication would be implemented using the standard long multiplication algorithm, followed by reduction modulo the field prime p, which may be of an exploitable form which would facilitate fast reduction of a 512 bit product modulo the 256 bit prime. Addition would require a simple digit-by-digit integer addition (with carry processing), and then maybe a correction to ensure that the result was less than p. Subtraction might involve a digit-by-digit subtraction, again maybe followed by a correction to ensure a result in the range $0 \ldots p-1$.

But we would contend that it is impossible to make such an implementation constant time. The modular reduction, and modular addition and subtraction often require corrections, based on the data being processed. This will typically take the form of a conditional subtraction by the modulus, to get the result into the range $0 \ldots p-1$.

A potential solution to this problem is to delay the corrections. This technique is called “lazy reduction”, that is we allow results to drift out of their range, and only correct them sometime later. Taking this idea to the extreme, we would ideally like to delay any correction right to the very end of an at-risk calculation like an elliptic curve point multiplication.

Here we suggest a general strategy to achieve this, independent of the curve, its size, or its parameterization. The basic idea is to tailor the finite field representation to the particular curve, rather than try and force-fit the curve to a predetermined format.

2 A basic observation

Consider a typical modular squaring operation. We require the square of $x \in \mathbb{F}_p$, that is $x \leftarrow x^2 \bmod p$. Assume that x is such that $x^2 < pR$, where R is some value greater than p. Assume that the modular reduction algorithm used is such that after completion $x < 2p$. In fact this is exactly how the well known Montgomery reduction algorithm works [11] (unless a conditional final subtraction is performed, and we do not want to do that).

Now if x is not fully reduced prior to the squaring, say if we can only be sure that $x < 8p$, then as long as $R > 64p$, the output x is such that $x < 2p$. Therefore we contend that a calculation that consists of a combination of modular additions, subtractions and multiplications may never need explicit reduction, due to the tendency of the squaring/multiplication code to force an implicit (albeit incomplete) reduction.

While hardly novel (see for example [10], section 4.4), this simple observation is key to our subsequent analysis.

We will refer to any modular reduction algorithm which results in an output less than $2p$, as a relaxed reduction. In general whereas it can be quite challenging to perform a full reduction in constant time, it seems to be quite easy to perform a constant time relaxed reduction.

3 Finite field element representation

It is commonly assumed that a “packed-radix” representation is used for field elements as described above, where numbers are represented to a base equivalent to the word-length of the computer. But clearly to support any kind of lazy reduction some kind of redundant representation must be used which allows elements to increase beyond the bit length of the modulus. Indeed it is now common for moduli to be proposed which are a few bits shy of the maximum possible for a fixed length representation in computer words. For a justification in the context of elliptic curve cryptography see [6]. For example rather than a 256 bit modulus, a 254 bit modulus may be preferred for the two bits of redundancy it allows. A disadvantage is the small loss of security that arises from using a smaller modulus.

The required redundancy arises more naturally if a “reduced radix” representation is used. Here elements are represented to a base somewhat less than the full word-length. So for example on a 64 bit architecture a base of 2^{60} might be used. This brings its own advantages as carry propagation can be delayed as well, a process which might be referred to as “lazy carrying”, which is essentially a form of lazy reduction applied to individual digits. For more details see the appendix to this paper. Also a faster method for modular multiplication applies [13] when a reduced radix representation is used.

Our main point is that using a reduced radix representation, common modulus sizes will no longer have a tight fit into a fixed number of computer words. So for example an element in $\mathbb{F}_p$, where p is 256 bits, may be represented as four 60 bit least-significant digits, and a most significant digit of only 16 bits, leaving a large amount of redundancy in the top word.

Note that our idea applies to any representation with sufficient redundancy. However the method works best with a reduced radix representation.

We refer to the extent that an element x may have crept beyond its modulus as its worst-case “excess” the integer E, where $x < E_x.p$. Knowledge of this worst-case excess, either explicitly stored with the field element, or implicitly known, is central to our proposed technique. Indeed we could use it immediately to carry out a "pseudo-constant-time" modular reduction – see algorithm 1. However this is quite inefficient, and we would prefer to use it sparingly, if at all.

Algorithm 1 Explicit constant time modular reduction

INPUT: An unreduced field element $w < E_w.p$, and the modulus p
OUTPUT: $w < p$

```
function CTRED($w$)
    $t \leftarrow \lceil \lg E_w \rceil$
    $m \leftarrow 2^t p$
    while $t > 0$ do
        $m \leftarrow m/2$
        $r \leftarrow w - m$
        $w \leftarrow w \oplus ((w \oplus r) \wedge (w > m))$
        $t \leftarrow t - 1$
    end while
end function
```

A key observation is that decisions based on these worst-case excesses that may arise at certain points in a calculation (like the number of times the loop is executed in algorithm 1), are not decisions based on the actual data being processed. Indeed assuming exception-free formulae are being used, they will be the same for every run of a particular protocol. In short the number of times around the loop depends only on the context in which the function is called, not on the data being processed.

The next issue is to fix the value of R. For multi-precision Montgomery reduction the ideal choice is 2^n where $p < 2^n$, and n is a multiple of the base of the representation. So for our 64 bit reduced radix example above, a natural choice would be 2^{300}, where $300 = 5.60$. For the packed-radix case the natural choice would be 2^{256}.

If Montgomery modular reduction is to work correctly then it is clearly sufficient when multiplying x by y that $E_x.E_y.p < R$. Observe that numbers less than R are representable using the same number of words as elements in $\mathbb{F}_p$.

3.1 Special moduli

For elliptic curve cryptography often a prime of special form is used, for which a faster reduction method applies. A common choice is a pseudo-Mersenne prime of the form $2^m - c$. A product to be reduced is split into a lower and higher part, with the lower part of length m bits. The top part is multiplied by c and added to the lower part. Finally the small excess beyond m bits is extracted, multiplied by c and added to the total. It is easy to see that the output may not be fully reduced, but will certainly be less than $2p$.

Now when the product is split, as long as it is less than pR, then the top part will be less than R, and hence representable. So basically the same rule applies as for Montgomery's method: As long as the input is less than pR, then the reduced output will be less than $2p$. Therefore this method constitutes another example of relaxed reduction.

4 Modular arithmetic

Multiplication has already been discussed. Addition is carried out without reduction. Therefore if calculating $z = x + y$, then (assuming the worst case) $E_z = E_x + E_y$.

Subtraction is negation, followed by addition. To find $-y$ we subtract y from a suitable multiple r of p and then $-y = r.p - y$. Clearly a suitable choice for r would be $r = E_y$. If calculating $z = x - y$, then again $E_z = E_x + E_y$. Note that suitable multiples of the modulus could be precalculated and stored, although we will avoid this in the sequel.

5 Edwards curves

As an example we will consider an implemention of point multiplication on the Edwards curve [4],

$$x^2 + y^2 = 1 + dx^2y^2$$

using some kind of constant time double-and-add algorithm (which typically would be a windowing method with fixed window width [5]). Using projective coordinates the calculations for point doubling and point addition are a mixture of modular additions, subtractions, squarings and multiplications. We want to determine the circumstances under which explicit reduction (as in algorithm 1) can be delayed until the very end of the point multiplication (assuming that a fully reduced output is desired). Modular inversion, if required, is assumed to be calculated using Fermats little theorem, $1/x = x^{p-2} \bmod p$, which given its fixed exponent is well known to have a constant time implementation using a simple square-and-multiply method.

From the explicit formula database [8] we find formulae for point doubling of an input point in projective coordinates (X, Y, Z). A common trick used when deriving these formulae is to use the identity $2XY = (X + Y)^2 - X^2 - Y^2$, which swaps a multiplication for a squaring if X^2 and Y^2 are already known. However if using a reduced radix representation, multiplications and squarings have essentially the same complexity [13]. So we will not use this trick here, as it has the unfortunate side-effect of artifically inflating the excesses.

The formula is broken down into atomic operations, and simplified for our purposes. See Table 1. On each line the excess of the output value is recorded.

We can assume that the curve constant d is fully reduced. For reasons that will become clear we assume that the initial excesses of X, Y and Z are all equal to 2.

Operation	Excess	Note
$A = X$	2	
$B = Y$	2	
$C = Z$	2	
$D = A^2$	2	$M = 4$
$B = B^2$	2	$M = 4$
$C = C^2$	2	$M = 4$
$C = C + C$	4	
$D = B + D$	4	
$B = B + B$	4	
$B = D - B$	8	$r = 4$
$A = X.Y$	2	$M = 4$
$A = A + A$	4	
$C = D - C$	8	$r = 4$
$X = A.C$	2	$M = 32$
$Y = B.D$	2	$M = 32$
$Z = C.D$	2	$M = 32$

Table 1. Edwards Point doubling

Some observations: This operation is stable, in that the excesses of the outputs are the same as those of the inputs. Therefore multiple doubling operations can follow one another without affecting the worst case excesses recorded here. In the Notes column of the table, for subtractions we note r the multiple of the modulus required when negating, that is the excess of the subtrahend. Here $4p$ will work in all cases. After each multiplication or squaring we note M the product of the excesses of the inputs. The maximum value for M is significant here – it will determine the minimum value for R.

Next is the point addition formula (Table 2). Again the overall operation is stable, and again we see that any combination of additions and doublings will not change these values. In this case a precomputed $2p$ is sufficient for all negations.

Overall the maximum value for M is 32, or 2^5. Now if $R = 2^n$ and $p < 2^m$, then we can conclude that as long as $n - m$ is greater than or equal to 5 bits, then these functions when combined to calculate a point multiplication, will work correctly. For a packed-radix representation, a 251 bit modulus is therefore a better choice than 256 bits or 254 bits. For a reduced radix representation it is not difficult to meet this constraint for any size of modulus.

It might be regarded as inconvenient to have to precompute both $2p$ and $4p$. In fact $4p$ can be used for all the negations that arise in the point addition formula without increasing the maximum value of M.

To be concrete, to implement any Edwards curve modulo a 256 bit prime on a 32 bit processor, choose a base of 2^{29} in which case nine digits are required

Operation	Excess	Note
$A = X_1$	2	
$B = Y_1$	2	
$C = Z_1$	2	
$D = X_2$	2	
$E = Y_2$	2	
$F = Z_2$	2	
$C = C.F$	2	$M = 4$
$G = A + B$	4	
$H = D + E$	4	
$A = A.D$	2	$M = 4$
$B = B.E$	2	$M = 4$
$G = G.H$	2	$M = 16$
$G = G - A$	4	$r = 2$
$G = G - B$	6	$r = 2$
$G = G.C$	2	$M = 12$
$H = A.B$	2	$M = 4$
$H = d.H$	2	$M = 2$
$B = B - A$	4	$r = 2$
$B = B.C$	2	$M = 8$
$C = C^2$	2	$M = 4$
$A = C - H$	4	$r = 2$
$C = C + H$	4	
$X_3 = A.G$	2	$M = 8$
$Y_3 = B.C$	2	$M = 8$
$Z_3 = C.A$	2	$M = 16$

Table 2. Edwards Point addition

to represent field elements. Setting $R = 2^{9.29} = 2^{261}$ our conditions are met, as $32 = 2^{261-256}$. In all cases modular subtraction of $x - y$ can be calculated as $4p - y + x$. Since explicit reduction is never required, reduction is not merely lazy, it is slothful.

6 RFC7748

This RFC specifies a key exchange protocol that takes place using a Montgomery curve [9]. This protocol uses x-only projective coordinates. Applying the same methodology as above we derive Table 3. We follow the notation of [9], section 5, and neglect the conditional swaps that occcur in the main program loop as they have no impact on our analysis. Iterating on this loop will result in a stable calculation, as the excesses of both inputs and outputs are equal to 2.

Operation	Excess	Note
$A = x_2 + z_2$	4	
$AA = A^2$	2	$M = 16$
$B = x_2 - z_2$	4	$r = 2$
$BB = B^2$	2	$M = 16$
$E = AA - BB$	4	$r = 2$
$C = x_3 + z_3$	4	
$D = x_3 - z_3$	4	$r = 2$
$DA = D.A$	2	$M = 16$
$CB = C.B$	2	$M = 16$
$F = DA + CB$	4	
$G = DA - CB$	4	$r = 2$
$x_3 = F^2$	2	$M = 16$
$G = G^2$	2	$M = 16$
$z_3 = x_1.G$	2	$M = 4$
$x_2 = AA.BB$	2	$M = 4$
$F = a24.E$	2	$M = 4$
$G = AA + F$	4	
$z_3 = E.G$	2	$M = 16$

Table 3. RFC7748

In all cases modular addition can take place without reduction, and modular subtraction of $x - y$ can be calculated as $2p - y + x$. For Montgomery reduction after multiplication or squaring, given that the maximum $M = 16$, it is required that R be at least 4 bits longer than p.

7 Weierstrass curves

For a more challenging example we consider the exception-free formulae for arithmetic on a prime-order Weierstrass curve

$$y^2 = x^3 - 3x + b$$

as recently described by Renes, Costello and Batina [12], [10]. We start with point doubling (Table 4). In this case the output values (X_3, Y_3, Z_3) are not the result of modular multiplications, and so we cannot assume that the calculation is necessarily stable. However for an input point (X, Y, Z) with input excesses of 4, 4 and 4 respectively, the outputs have the same excesses (or less), and therefore we conclude that this function is in fact stable.

Operation	Excess	Note	Operation	Excess	Note
1. $t_0 = X^2$	2	$M = 16$	2. $t_1 = Y^2$	2	$M = 16$
3. $t_2 = Z^2$	2	$M = 16$	4. $t_3 = X.Y$	2	$M = 16$
5. $t_3 = t_3 + t_3$	4		6. $Z_3 = X.Z$	2	$M = 16$
7. $Z_3 = Z_3 + Z_3$	4		8. $Y_3 = b.t_2$	2	$M = 2$
9. $Y_3 = Y_3 - Z_3$	6	$r = 4$	10. $X_3 = Y_3 + Y_3$	12	
11. $Y_3 = X_3 + Y_3$	18		12. $X_3 = t_1 - Y_3$	20	$r = 18$
13. $Y_3 = t_1 + Y_3$	20		14. $Y_3 = X_3.Y_3$	2	$M = 400$
15. $X_3 = X_3.t_3$	2	$M = 80$	16. $t_3 = t_2 + t_2$	4	
17. $t_2 = t_2 + t_3$	6		18. $Z_3 = b.Z_3$	2	$M = 4$
19. $Z_3 = Z_3 - t_2$	8	$r = 6$	20. $Z_3 = Z_3 - t_0$	10	$r = 2$
21. $t_3 = Z_3 + Z_3$	20		22. $Z_3 = Z_3 + t_3$	30	
23. $t_3 = t_0 + t_0$	4		24. $t_0 = t_3 + t_0$	6	
25. $t_0 = t_0 - t_2$	12	$r = 6$	26. $t_0 = t_0.Z_3$	2	$M = 360$
27. $Y_3 = Y_3 + t_0$	**4**		28. $t_0 = Y.Z$	2	$M = 16$
29. $t_0 = t_0 + t_0$	4		30. $Z_3 = t_0.Z_3$	2	$M = 120$
31. $X_3 = X_3 - Z_3$	**4**	$r = 2$	32. $t_0 = t_0 + t_0$	8	
33. $t_1 = t_1 + t_1$	4		34. $Z_3 = t_0.t_1$	**2**	$M = 32$

Table 4. Weierstrass Point Doubling

For point addition (Table 5) on the same curve, we assume the same input excesses for (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) of 4, 4 and 4 respectively, as the inputs may be results of previous doubling operations.

In this case the maximum $M = 676$ and so R would need to be 10 bits greater than the prime modulus to permit slothful reduction. Using a reduced radix representation this is an easy condition to meet on a 64 bit processor. On a 32 bit computer 256 bit field elements could be represented as ten digits to a base of 2^{27}, which allows $R = 2^{270}$ with a comfortable safety margin. There is a small cost here given that the field element representation has increased from nine to ten digits. But this may be regarded as a worthwhile trade-off for a constant time implementation.

For this example more multiples of the modulus are required to support modular negation in all cases. Again with care some r multiples can be adjusted upwards to coincide with others, without increasing the maximum value of M, so that less multiples may be needed. Experimentally we determined that by

Operation	Excess	Note	Operation	Excess	Note
1. $t_0 = X_1.X_2$	2	$M = 16$	2. $t_1 = Y_1.Y_2$	2	$M = 16$
3. $t_2 = Z_1.Z_2$	2	$M = 16$	4. $t_3 = X_1 + Y_1$	8	
5. $t_4 = X_2 + Y_2$	8		6. $t_3 = t_3.t_4$	2	$M = 64$
7. $t_4 = t_0 + t_1$	4		8. $t_3 = t_3 - t_4$	6	$r = 4$
9. $t_4 = Y_1 + Z_1$	8		10. $X_3 = Y_2 + Z_2$	8	
11. $t_4 = t_4.X_3$	2	$M = 64$	12. $X_3 = t_1 + t_2$	4	
13. $t_4 = t_4 - X_3$	6	$r = 4$	14. $X_3 = X_1 + Z_1$	8	
15. $Y_3 = X_2 + Z_2$	8		16. $X_3 = X_3.Y_3$	2	$M = 64$
17. $Y_3 = t_0 + t_2$	4		18. $Y_3 = X_3 - Y_3$	6	$r = 4$
19. $Z_3 = b.t_2$	2	$M = 2$	20. $X_3 = Y_3 - Z_3$	8	$r = 2$
21. $Z_3 = X_3 + X_3$	16		22. $X_3 = X_3 + Z_3$	24	
23. $Z_3 = t_1 - X_3$	26	$r = 24$	24. $X_3 = t_1 + X_3$	26	
25. $Y_3 = b.Y_3$	2	$M = 6$	26. $t_1 = t_2 + t_2$	4	
27. $t_2 = t_1 + t_2$	6		28. $Y_3 = Y_3 - t_2$	8	$r = 6$
29. $Y_3 = Y_3 - t_0$	10	$r = 2$	30. $t_1 = Y_3 + Y_3$	20	
31. $Y_3 = t_1 + Y_3$	30		32. $t_1 = t_0 + t_0$	4	
33. $t_0 = t_1 + t_0$	6		34. $t_0 = t_0 - t_2$	12	$r = 6$
35. $t_1 = t_4.Y_3$	2	$M = 180$	36. $t_2 = t_0.Y_3$	2	$M = 360$
37. $Y_3 = X_3.Z_3$	2	$M = 676$	38. $Y_3 = Y_3 + t_2$	**4**	
39. $X_3 = t_3.X_3$	2	$M = 156$	40. $X_3 = X_3 - t_1$	**4**	$r = 2$
41. $Z_3 = t_4.Z_3$	2	$M = 156$	42. $t_1 = t_3.t_0$	2	$M = 72$
43. $Z_3 = Z_3 + t_1$	**4**				

Table 5. Weierstrass Point addition

rounding up some of the r multiples, values from the set $2, 4, 8, 32$ are sufficient, and increase the maximum M to only 884, so no extra bits are required for the representation. Since the members of the set are all powers of 2, the required multiples of p can be formed on the fly by simple shift operations.

8 Extension field arithmetic

Slothful reduction may also find application in extension field arithmetic, as required in pairing-based cryptography. Consider for example powering in the extension field $\mathbb{F}_{p^2}$ using some square-and-multiply algorithm. Assuming p is congruent to 3 mod 4, then -1 will always be a quadratic non-residue, and elements can be represented as $u + iv$, where $u, v \in \mathbb{F}_p$, and $i = \sqrt{-1}$.

To find the square of an element, we calculate $u \leftarrow (u+v)(u-v)$ and $v \leftarrow 2uv$.

To multiply two elements $u_1 + iv_1$ and $u_2 + iv_2$ a Karatsuba method applies, and $u_3 = u_1.u_2 - v_1.v_2$ and $v_3 \leftarrow (u_1 + v_1)(u_2 + v_2) - (u_1.u_2 + v_1.v_2)$.

First we take the same approach as above. To achieve stability we must it seems assume initial excesses for u and v of 4 and 6 respectively. See Tables 6 and 7.

But this is not the best approach. Aranha et al. [1] have described in detail a potentially faster way in which to do multiplication in $\mathbb{F}_{p^2}$, which involves the use of lazy reduction in a different sense to which we have used it here. The idea

Operation	Excess	Note
$A = u$	4	
$B = v$	6	
$C = A.B$	2	$M = 24$
$D = A + B$	10	
$E = A - B$	10	$r = 6$
$u = D.E$	2	$M = 100$
$v = C + C$	4	

Table 6. Squaring in $\mathbb{F}_{p^2}$

Operation	Excess	Note
$A = u_1.u_2$	2	$M = 16$
$B = v_1.v_2$	2	$M = 36$
$C = u_1 + v_1$	10	
$D = u_2 + v_2$	10	
$E = C.D$	2	$M = 100$
$F = A + B$	4	
$u_3 = A - B$	4	$r = 2$
$v_3 = E - F$	6	$r = 4$

Table 7. Multiplication in $\mathbb{F}_{p^2}$

is that if required to calculate say $ab+cd \bmod p$, then it makes sense to delay the reduction until after the addition, so that only one reduction is required rather than two. This is particularly true if reduction following multiplication is slow, as would be the case for a p of no exploitable special form, as for example arises in pairing-based cryptography.

First we re-describe squaring with smaller initial excesses for u and v of 2 and 2 respectively, but for which the Aranha et al. idea does not apply. See Table 8. We also re-order the instructions to our advantage.

Operation	Excess	Note
$A = u$	2	
$B = v$	2	
$C = A + A$	4	
$C = C.B$	2	$M = 8$
$D = A + B$	4	
$E = A - B$	4	$r = 2$
$u = D.E$	2	$M = 16$
$v = C$	2	

Table 8. Improved squaring in $\mathbb{F}_{p^2}$

Next we apply our methodology to the Aranha et al. method for multiplication (Table 9).

Operation	Excess	Note
1. $\mathbf{A} = u_1.u_2$	4	
2. $\mathbf{B} = v_1.v_2$	4	
3. $C = u_1 + v_1$	4	
4. $D = u_2 + v_2$	4	
5. $\mathbf{E} = C.D$	16	
6. $\mathbf{F} = \mathbf{A} + \mathbf{B}$	8	
7. $\mathbf{A} = \mathbf{A} - \mathbf{B}$	8	$r = 4$
8. $\mathbf{E} = \mathbf{E} - \mathbf{F}$	16	
9. $u_3 = \mathbf{A}$	2	$M = 8$
10. $v_3 = \mathbf{E}$	2	$M = 16$

Table 9. Faster Multiplication in $\mathbb{F}_{p^2}$

In this table double precision numbers are represented in bold-face. The product of two values x and y with excesses of E_x and E_y respectively results in a double precision product $z = x.y$, where $z < E_x.E_y.p^2$, and so the double precision excess is $E_z = E_x.E_y$.

Modular subtraction in line 7 is calculated as $A - B = 4p^2 - B + A$ (and so $4p^2$ should be precalculated), but the modular subtraction in line 8 clearly requires only simple non-modular subtraction (as $E > F$). Note that the maximum M value is now reduced to 16. The delayed double-precision reduction takes place in lines 9 and 10.

8.1 Application to pairings

When implementing pairing based crypto, multiplication in $\mathbb{F}_{p^2}$ can arise in a number of different contexts. It may for example arise at the bottom of a tower of extensions, or it may arise in the context of point multiplication in the group $\mathbb{G}_2$ on, for example, a BN curve [2]. Therefore the input excesses may not be fixed in advance. In this case for the modular subtraction in line 7 of Table 9 it may be difficult to determine which multiple of p^2 is appropriate.

So we suggest a different approach for the calculation in line 7. Simply calculate $T = A - B = A + (pR - B)$. Now in this case for $A, B < pR$, then clearly $T < 2pR$ which may be outside of the correct range for certain reduction methods, as it would be for Montgomery reduction [11]. However Montgomery reduction does not fail in this case, rather the output value is no longer guaranteed to be less than $2p$, in this case it will be less than $3p$, so the only impact is that the output worst case excess is slightly increased.

The objection may be raised that a number $< 2pR$ may not be representable. Luckily, using a reduced radix representation, there will always be a few bits "to spare" at the top of the most significant word, and so this is not a problem in practice.

9 Unstable calculations

Unfortunately not all algorithms are stable in the sense used here. Consider for example the application of our original methodology to the fast algorithm for squaring in the cyclotomic subgroup of sixth degree extensions, as proposed by Granger and Scott [7]. Here $a, b, c \in \mathbb{F}_{q^2}$ are the three components of an element in $\mathbb{F}_{q^6}$.

$$a \leftarrow 3a^2 - 2\bar{a}$$

$$b \leftarrow 3\sqrt{i}.c^2 - 2\bar{b}$$

$$c \leftarrow 3b^2 - 2\bar{c}$$

Consider now a sequence of squarings of some initial value. Observe that $2\bar{a}$ contributes to the new value of a after each squaring. So clearly after each sequential squaring, the excess of a increases monotonically. Therefore in this case explicit reduction (by application of algorithm 1) may be required to curb the excesses that will arise if a sequence of such squarings should occur in a program. So in this case fully slothful reduction may not be possible.

10 Conclusion

We have described a novel but simple cook-book methodology that could be used to assist a programmer in implementing certain cryptographic functions that rely on modular arithmetic, efficiently, and in constant time. We are not claiming that this is the only way to achieve this outcome. Basically we tailor the representation of finite field elements to fit the needs of the cryptographic algorithm, rather than force an algorithm unto a predetermined form of representation.

The method applies also to other moduli types, in particular to generalised Mersenne primes. The bound on the output of a product, that it be less than $2p$, can be significantly tightened in particular cases. For example for Montgomery reduction if $E_x.E_y.p$ is much less than R, then $2p$ can be reduced to $\delta.p$ for some $\delta < 2$.

Finally we reiterate that the worst case field excesses can at any time be fully reduced by application of algorithm 1. An alternate but more expensive response would be to identify field elements with excessive excesses, and simply multiply them by the appropriate representation of unity. By targeting such reductions at “excess hotspots”, maximum excesses can be significantly reduced.

References

1. D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. Lopez. Faster explicit formulae for computing pairings over ordinary curves. In *Eurocrypt – 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 48–68. Springer-Verlag, 2011.

2. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptology – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 2006.
3. D. Bernstein. Curve25519: new Diffie-Hellman speed records. In *PKC – 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag, 2006.
4. D. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Asiacrypt – 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer-Verlag, 2007.
5. Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In *CHES –2014*, volume 8731 of *Lecture Notes in Computer Science*. Springer-Verlag, 2014.
6. J. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, 6(4):259–296, 2016.
7. R. Granger and M. Scott. Faster squaring in the cyclotomic subgroup of sixth degree extensions. In *PKC – 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 2010.
8. T. Lange. Explicit formula database. `http://hyperelliptic.org/EFD/`.
9. Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, 2016.
10. P. Massolino, J. Renes, and L. Batina. Implementing complete formulas on Weierstrass curves in hardware. In *SPACE – 2016*, volume 10076 of *Lecture Notes in Computer Science*, pages 89–108. Springer-Verlag, 2015.
11. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
12. J. Renes, C. Costello, and L. Batina. Complete addition formulas for prime order elliptic curves. In *Eurocrypt – 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer-Verlag, 2016.
13. M. Scott. Missing a trick: Karatsuba variations. *Cryptography and Communications*, 2017. `https://link.springer.com/article/10.1007%2Fs12095-017-0217-x`.

Lazy carrying

Basically by lazy carrying we mean lazy reduction applied to individual digits of a big number, when using a reduced radix representation. Assume a number base of 2^b, where b is a little less that the wordlength w of the processor, and further assume that each digit is stored as a signed integer. Then to add two such numbers we can simply add digit-by-digit and delay the carry propagation. This is quite commonly done, and will be very fast on a modern superscalar architecture and in hardware, as all digits can be added in parallel. However the extent to which we can do this without causing overflow is limited, so we must proceed with caution.

We can assume that when subtraction is required in the context of modular arithmetic as described above, that a smaller number is being subtracted from a bigger number. Nevertheless individual digits may become negative. Eventually a simple carry-propagation (or normalisation) process will restore the number to its proper base 2^b representation, if required, in constant time. But for performance purposes we would like to avoid normalisation where possible.

It is not hard to see that up to 2^{w-b-1} normalised numbers can be added without causing additive overflow, and the same applies to subtraction. When multiplying two such numbers a stability criteria, as for example described in [13], may be impacted by the larger digits that arise as a result of lazy carrying. The impact will depend on the particulars of the method used for multiplication (for example whether or not a product-scanning or operand-scanning algorithm is used). This may require that arguments are normalised prior to a modular multiplication. We do assume that the output of a modular multiplication is fully normalised, so where the output of a multiplication feeds into the input of another, normalisation would not be required.

For 256 bit Edwards and Weierstrass curves implemented on a 32 bit processor and using the exception free formulae as described above, and the recommended number bases of 2^{29} and 2^{27} respectively, we observe that all modular additions and subtractions can proceed without explicit normalisation (or reduction). The same outcome is easy to achieve on a 64 bit processor, using for example a base of 2^{56}.