Faster ECC over $\mathbb{F}_{2^{521}-1}$

Robert Granger[1] and Michael Scott[2]

[1] Laboratory for Cryptologic Algorithms
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne, CH-1015, Switzerland
robbiegranger@gmail.com

[2] Certivox Labs
mike.scott@certivox.com

Abstract. In this paper we present a new multiplication algorithm for residues modulo the Mersenne prime $2^{521}-1$. Using this approach, on an Intel Haswell Core i7-4770, constant-time variable-base scalar multiplication on NIST's (and SECG's) curve P-521 requires 989,000 cycles, while on the recently proposed Edwards curve E-521 it requires just 779,000 cycles. As a comparison, on the same architecture openSSL's ECDH speed test for curve P-521 requires 1,319,000 cycles. Furthermore, our code was written entirely in C with no non-compiler optimisations and so is robust across different platforms. The basic observation behind these speedups is that the form of the modulus allows one to multiply residues with as few word-by-word multiplications as is needed for squaring, while incurring very little overhead from extra additions, in contrast to the usual Karatsuba methods.

Keywords: elliptic curve cryptography, performance, P-521, E-521, Edwards curves, generalised repunit primes, Crandall numbers, Karatsuba

1 Introduction

Nearly all research on elliptic curve cryptography (ECC) focuses on improving efficiency, the bedrock of which is efficient field arithmetic. Amongst prime fields the multiply-then-reduce paradigm suggests that arithmetic modulo the Mersenne primes $M_k = 2^k - 1$ should be optimal, since modular reduction can be effected by a single modular addition, as is well known. Within this paradigm, research on fast modular multiplication has naturally tended to focus on reducing the cost of the reduction step. This rationale led Solinas in 1999 to introduce *Generalised Mersenne Numbers* (GMNs), five of which feature in the NIST (FIPS 186-2) [17] and SECG [18] standards for use in ECC, ranging in size from 192 to 521 bits, all with fast modular reduction. Solinas' reduction method regards both the output of an integer by integer residue multiplication and the modulus as polynomials in a base $t = 2^w$, with w the word size of the underlying architecture. Reducing the former polynomial modulo the latter then gives an algebraic modular reduction that requires only a few modular additions and/or subtractions and possibly a few final subtractions of the modulus.

However, more recent approaches to modular multiplication at bitlengths relevant to ECC do not adhere to the multiply-then-reduce paradigm. In particular, Chung and Hasan [10] proposed a slight modification of Solinas' method, viewing it as a two stage process, with residues also regarded as polynomials in base t. For generic residue coefficients, the product of two residues modulo the modulus polynomial is precomputed. Then in the first stage these coefficient expressions are evaluated on the input coefficients; in the second stage the resulting coefficients are renormalised by expanding them in base t and performing the required carries. This small shift in perspective somewhat interleaves the multiplication and reduction steps, and allows one to use a smaller base than before. Using a smaller base is useful since it firstly allows more primes to be represented, and secondly it means that the coefficient evaluations need not overflow beyond double precision on the underlying architecture. The latter property has been applied numerous times [2,3,6,9,11,12,16,19] and is now standard.

Another example of when it is advantageous to allow the form of the modulus to influence how one multiplies residues is for the Mersenne numbers M_k, for $k \to \infty$. In particular, modular

multiplication can be carried out using a cyclic convolution effected by an irrational-base discrete weighted transform (IBDWT) [16, §6], whereas for integer multiplication a linear convolution is required, as each multiplicand must be padded with k zeros before a cyclic convolution of length $2k$ can be performed. Hence multiplication modulo a Mersenne number is asymptotically approximately twice as fast as integer multiplication.

Unfortunately, Fast Fourier Transform techniques are not effective at ECC bitlengths and hence can not be applied to M_{521}. However, using a more natural alternative generalisation of the Mersenne numbers than Solinas proposed – already known in the literature as *Generalised Repunit Primes* (GRPs) – Granger and Moss described a modular multiplication algorithm that is surprisingly fast, being about three times faster than Montgomery multiplication for primes of around 600 bits, on a 2.2GHz Intel Core 2 Duo [19]. GRPs are those of the form $\sum_{i=0}^{p-1} t^i$ for prime p and integer $t > 1$. Although they have many attractive features, the only example in the standards is M_{521} with $t = 2$, but in this case t is too small to take advantage of these features.

In this paper we show that one can use an analogue of the multiplication technique of [19] to speed up multiplication modulo M_{521}. For demonstration purposes this operation – as well as constant-time variable-base scalar multiplication – has been implemented in C on a 3.4GHz Intel Haswell Core i7-4770, which has yielded interesting speedups. In particular, on NIST's (and SECG's) curve P-521 this requires $989,000$ cycles, while on the recently proposed Edwards curve E-521 it requires just $779,000$ cycles. As a comparison, on the same architecture openSSL's ECDH speed test for curve P-521 requires $1,319,000$ cycles. We note that the Edwards curve E-521 is a particularly attractive target for implementors; according to the safecurves website [5] this curve has been proposed independently three times, by Bernstein-Lange, Hamburg, and Aranha *et al.* [1]. It addresses all of the recent concerns regarding the security of NIST curves (as well as others), having been generated in a deterministic pseudorandom manner while being twist-secure, complete and permitting point representations which are indistinguishable from uniform random strings [4].

Although not described as such, the technique of [19] which forms the basis of our multiplication speedup can be viewed as a 'twisted' version of Karatsuba's trick [20], albeit one in which the form of the modulus lends itself very favourably. Its effectiveness therefore runs counter to conventional wisdom which posits that Karatsuba-like techniques are not efficient at bitlengths relevant to ECC (at least on 64-bit architectures), due to the high number of extra additions required. A recent proposal by Bernstein, Chuengsatiansup and Lange also uses a variation of Karatsuba for fast multiplication modulo $2^{414} - 17$ [3], with efficiencies being extracted at a much lower level than we use in the present paper; indeed *two* Karatsuba levels are used, as well as a clever method to reduce inputs to the required multiplications, rather than outputs (cf. [3, Section 4.6]). Their implementation on the 32-bit ARM Cortex-A8 also exploits vectorisation to great effect. While not directly comparable to Bernstein's *et al.*'s implementation, one advantage of our implementation is that it is written entirely in C with no non-compiler optimisations; it therefore has robust performance characteristics across a range of 64 bit platforms. Having said that, we naturally expect that further optimisations are possible, both on the demonstration and similar architectures, and especially on ARM processors, due to their higher multiplication-to-addition cost ratio. We therefore encourage others to explore these possibilities; our software is freely downloadable.

To a lesser extent, our same basic observation may be applied to so-called Crandall numbers, i.e., those of the form $2^k - c$ for c usually much smaller than the word size of the underlying architecture, and we provide two examples of how this may be done.

The sequel is organised as follows. In Section 2 we explain our basic observation, while in Section 3 we show how it may be applied to M_{521}, giving squaring and inversion routines as well. In Section 4 we provide details of the target curves and our implementation results, while in Section 5 we describe how our basic observation may be applied to Crandall numbers. We conclude in Section 6.

2 The basic observation

The best way to describe our basic observation is via an example. For an integer t let the modulus be $t^9 - 1$ and let the base t expansion of residues x and y be $\sum_{i=0}^{8} x_i t^i$ and $\sum_{i=0}^{8} y_i t^i$ respectively. For convenience, we also denote x by $\overline{\mathbf{x}} = [x_0, \ldots, x_8]$ and y by $\overline{\mathbf{y}} = [y_0, \ldots, y_8]$.

The multiplication of $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ mod $t^9 - 1$ is just their cyclic convolution; in particular, if $\overline{\mathbf{z}} \equiv \overline{\mathbf{x}}\,\overline{\mathbf{y}} \pmod{t^9 - 1}$ then $\overline{\mathbf{z}} = [z_0, \ldots, z_8] =$

$$\begin{aligned}
&[x_0y_0 + x_1y_8 + x_2y_7 + x_3y_6 + x_4y_5 + x_5y_4 + x_6y_3 + x_7y_2 + x_8y_1,\\
&x_0y_1 + x_1y_0 + x_2y_8 + x_3y_7 + x_4y_6 + x_5y_5 + x_6y_4 + x_7y_3 + x_8y_2,\\
&x_0y_2 + x_1y_1 + x_2y_0 + x_3y_8 + x_4y_7 + x_5y_6 + x_6y_5 + x_7y_4 + x_8y_3,\\
&x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + x_4y_8 + x_5y_7 + x_6y_6 + x_7y_5 + x_8y_4,\\
&x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0 + x_5y_8 + x_6y_7 + x_7y_6 + x_8y_5,\\
&x_0y_5 + x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1 + x_5y_0 + x_6y_8 + x_7y_7 + x_8y_6,\\
&x_0y_6 + x_1y_5 + x_2y_4 + x_3y_3 + x_4y_2 + x_5y_1 + x_6y_0 + x_7y_8 + x_8y_7,\\
&x_0y_7 + x_1y_6 + x_2y_5 + x_3y_4 + x_4y_3 + x_5y_2 + x_6y_1 + x_7y_0 + x_8y_8,\\
&x_0y_8 + x_1y_7 + x_2y_6 + x_3y_5 + x_4y_4 + x_5y_3 + x_6y_2 + x_7y_1 + x_8y_0].
\end{aligned} \tag{2.1}$$

The coefficients in this expression are about twice the size of t; we assume that t is selected in such a way that there is no overflow beyond double the initial precision of the coefficients. The arithmetic cost of a direct evaluation of each coefficient is then nine coefficient multiplications and eight double length additions, which we count as (and denote by) $9M + 16A$. This gives a total cost of $81M + 144A$.

For bitlengths relevant to ECC it is not beneficial to use asymptotically efficient FFT-based (cyclic) convolutions, as suggested in [14] and as are used for the Lucas-Lehmer primality test for Mersenne numbers, see [15, 16]. However, one can exploit some of the symmetry of (2.1) as follows. Let $s = \sum_{i=0}^{8} x_i y_i$. Then $\overline{\mathbf{z}}$ may also be expressed as

$$\begin{aligned}
&[s - (x_1 - x_8)(y_1 - y_8) - (x_2 - x_7)(y_2 - y_7) - (x_3 - x_6)(y_3 - y_6) - (x_4 - x_5)(y_4 - y_5),\\
&s - (x_1 - x_0)(y_1 - y_0) - (x_2 - x_8)(y_2 - y_8) - (x_3 - x_7)(y_3 - y_7) - (x_4 - x_6)(y_4 - y_6),\\
&s - (x_5 - x_6)(y_5 - y_6) - (x_2 - x_0)(y_2 - y_0) - (x_3 - x_8)(y_3 - y_8) - (x_4 - x_7)(y_4 - y_7),\\
&s - (x_5 - x_7)(y_5 - y_7) - (x_2 - x_1)(y_2 - y_1) - (x_3 - x_0)(y_3 - y_0) - (x_4 - x_8)(y_4 - y_8),\\
&s - (x_5 - x_8)(y_5 - y_8) - (x_6 - x_7)(y_6 - y_7) - (x_3 - x_1)(y_3 - y_1) - (x_4 - x_0)(y_4 - y_0),\\
&s - (x_5 - x_0)(y_5 - y_0) - (x_6 - x_8)(y_6 - y_8) - (x_3 - x_2)(y_3 - y_2) - (x_4 - x_1)(y_4 - y_1),\\
&s - (x_5 - x_1)(y_5 - y_1) - (x_6 - x_0)(y_6 - y_0) - (x_7 - x_8)(y_7 - y_8) - (x_4 - x_2)(y_4 - y_2),\\
&s - (x_5 - x_2)(y_5 - y_2) - (x_6 - x_1)(y_6 - y_1) - (x_7 - x_0)(y_7 - y_0) - (x_4 - x_3)(y_4 - y_3),\\
&s - (x_5 - x_3)(y_5 - y_3) - (x_6 - x_2)(y_6 - y_2) - (x_7 - x_1)(y_7 - y_1) - (x_8 - x_0)(y_8 - y_0)].
\end{aligned} \tag{2.2}$$

The cost of computing s is $9M+16A$. The subsequent cost of each coefficient evaluation is $4M+16A$, since each term costs two single length additions and one double length addition, and there are four terms. Hence the total cost is now $45M + 160A$, giving a saving of $36M$ at the cost of $16A$. This new expression for $\overline{\mathbf{z}}$ is of course very reminiscent of Karatsuba's method [20]. Indeed, it only (repeatedly) uses the identity $x_iy_j + x_jy_i = x_iy_i + x_jy_j - (x_i - x_j)(y_i - y_j)$, which is a slight twist of the more common version $x_iy_j + x_jy_i = (x_i + x_j)(y_i + y_j) - x_iy_i - x_jy_j$. Using the 'twisted' version means that precisely the same s term appears in each coefficient, thus saving several additions.

In general, for the modulus $t^n - 1$, evaluating the cyclic convolution using the schoolbook method costs $n^2M + 2n(n-1)A$, whereas using our basic observation it costs $\frac{1}{2}n(n+1)M + 2(n^2-1)A$. This is the same number of multiplications required for squaring with the schoolbook method; this is because the same symmetry is being exploited, however for squaring one can simply express

$x_i x_j + x_j x_i$ as $2x_i x_j$. One thus saves nearly half the number of multiplications while incurring very little overhead from extra additions. Hence even at small bitlengths for which the schoolbook method for integer multiplication is faster than Karatsuba techniques, one expects this method to give a speedup for multiplication modulo $t^n - 1$.

Note that if one instead uses the modulus $p = \sum_{i=0}^{n-1} t^i$ then s need not even be computed, since the first term of the i-th coefficient contributes st^i, which altogether gives $s \sum_{i=0}^{n-1} t^i \equiv 0 \pmod{p}$. This was the rationale behind the proposal of Granger and Moss to use GRPs for ECC [19].

3 Application to $M_{521} = 2^{521} - 1$

In this section we show how our basic observation may be applied to the Mersenne prime $M_{521} = 2^{521} - 1$. For the interested reader, we point out that Crandall and Pomerance used this prime to demonstrate Crandall's asymptotically fast algorithm for multiplication modulo Mersenne numbers which uses an IBDWT [13, Alg. 9.5.19]. This algorithm provides a method to obtain integer coefficients when mimicking an irrational-base expansion of residues and of course exploits the cyclic convolution far more cleverly than we do here, but it is not efficient for such small bitlengths.

In order to use the basic observation, first observe that one should not set $t = 2$ and $n = 521$, as this would involve far too much redundancy and too many multiplications. On a 64-bit architecture residues mod p require $\lceil 521/64 \rceil = 9$ words, so what one would like to do is set $n = 9$ and use the irrational base $t = 2^{521/9}$ while using integer coefficients only, either à la Crandall or à la Bernstein's method for performing arithmetic modulo $2^{255} - 19$ [2], which uses the irrational base $2^{25.5}$. It turns out that for our prime of interest, the analogue of Bernstein's method is nothing but operand scaling [21, 22]. Since this is easier to explain, we do so here.

Observe that $521 \equiv 8 \pmod 9$. Hence one can work modulo $2p = t^9 - 2$ instead of p, with $t = 2^{58}$. This representation was used by Langley in OpenSSL 1.0.0e in September 2011, which greatly improved efficiency relative to the base 2^{64} approach[3]. The multiplication formulae are now slightly different; if $\overline{\mathbf{z}} \equiv \overline{\mathbf{x}}\,\overline{\mathbf{y}} \pmod{t^9 - 2}$ then $\overline{\mathbf{z}} = [z_0, \ldots, z_8] =$

$$
\begin{aligned}
&[x_0y_0+2x_1y_8+2x_2y_7+2x_3y_6+2x_4y_5+2x_5y_4+2x_6y_3+2x_7y_2+2x_8y_1,\\
&x_0y_1+\ x_1y_0+2x_2y_8+2x_3y_7+2x_4y_6+2x_5y_5+2x_6y_4+2x_7y_3+2x_8y_2,\\
&x_0y_2+\ x_1y_1+\ x_2y_0+2x_3y_8+2x_4y_7+2x_5y_6+2x_6y_5+2x_7y_4+2x_8y_3,\\
&x_0y_3+\ x_1y_2+\ x_2y_1+\ x_3y_0+2x_4y_8+2x_5y_7+2x_6y_6+2x_7y_5+2x_8y_4,\\
&x_0y_4+\ x_1y_3+\ x_2y_2+\ x_3y_1+\ x_4y_0+2x_5y_8+2x_6y_7+2x_7y_6+2x_8y_5,\\
&x_0y_5+\ x_1y_4+\ x_2y_3+\ x_3y_2+\ x_4y_1+\ x_5y_0+2x_6y_8+2x_7y_7+2x_8y_6,\\
&x_0y_6+\ x_1y_5+\ x_2y_4+\ x_3y_3+\ x_4y_2+\ x_5y_1+\ x_6y_0+2x_7y_8+2x_8y_7,\\
&x_0y_7+\ x_1y_6+\ x_2y_5+\ x_3y_4+\ x_4y_3+\ x_5y_2+\ x_6y_1+\ x_7y_0+2x_8y_8,\\
&x_0y_8+\ x_1y_7+\ x_2y_6+\ x_3y_5+\ x_4y_4+\ x_5y_3+\ x_6y_2+\ x_7y_1+\ x_8y_0].
\end{aligned}
\tag{3.1}
$$

There are several possible approaches to applying the basic observation to (3.1), all of which incur a slight overhead relative to (2.2) due to the presence of the factors of 2. For the target architecture, the most efficient one we found is presented in Algorithm 1, which computes and reduces each component of $\overline{\mathbf{z}}$ sequentially. We first detail how we represent residues.

Residue representation: It is a simple matter to represent a mod p residue x in the form $\overline{\mathbf{x}} = [x_0, \ldots, x_8]$ by taking the base $t = 2^{58}$ expansion. Since we wish to allow negative coefficients, we use signed integers. Due to our choice of reduction method, we stipulate that the first component x_0 in our *reduced format* is in $[-2^{59}, 2^{59} - 1]$ while the remaining components are in $[0, 2^{58} - 1]$, although these bounds are not enforced except for the output coordinates of point addition and doubling.

[3] We independently implemented our multiplication for curve P-521 using this representation in the summer of 2011, but have decided to publish only now due to the recent interest in alternative elliptic curves over M_{521}.

Multiplication and squaring: Algorithms 1 and 2 detail pseudocode for our multiplication and squaring routines respectively. Observe that in both algorithms the wrap-around from z_8 to z_0 is computed twice. This is to ensure that there is at most one bit of overflow beyond 58 bits for z_0. One could instead rotate the order in which terms are computed so that the term that is computed twice is $(t_i >> 58)$ rather than $2(t_i >> 58)$, which would save one shift operation. However, we chose to keep the components $z_1, \ldots, z_8 \geq 0$ with only z_0 possibly being negative, since this allows one to check whether a given residue is zero or one without mapping back to the integer residue representation.

Algorithm 1: **MUL**

INPUT: $\overline{\mathbf{x}} = [x_0, \ldots, x_8], \overline{\mathbf{y}} = [y_0, \ldots, y_8] \in [-2^{59}, 2^{59} - 1] \times [0, 2^{58} - 1]^8$
OUTPUT: $\overline{\mathbf{z}} \in [-2^{59}, 2^{59} - 1] \times [0, 2^{58} - 1]^8$ where $\overline{\mathbf{z}} \equiv \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \pmod{t^9 - 2}$

1. $t_0 \leftarrow x_0y_0 + x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$
2. $t_5 \leftarrow x_5y_5$, $t_6 \leftarrow x_6y_6$, $t_7 \leftarrow x_7y_7$, $t_8 \leftarrow x_8y_8$
3. $t_1 \leftarrow t_5 + t_6 + t_7 + t_8$
4. $t_2 \leftarrow t_0 + t_1 - (x_0 - x_8)(y_0 - y_8) - (x_1 - x_7)(y_1 - y_7)$
   $-(x_2 - x_6)(y_2 - y_6) - (x_3 - x_5)(y_3 - y_5)$
5. $t_0 \leftarrow t_0 + 4t_1$
6. $t_1 \leftarrow t_2 \bmod t$
7. $t_3 \leftarrow t_0 - (x_4 - 2x_5)(y_4 - 2y_5) - (x_3 - 2x_6)(y_3 - 2y_6)$
   $-(x_2 - 2x_7)(y_2 - 2y_7) - (x_1 - 2x_8)(y_1 - 2y_8) + 2(t_2 >> 58)$
8. $z_0 \leftarrow t_3 \bmod t$
9. $t_0 \leftarrow t_0 - 2t_5$
10. $t_2 \leftarrow t_0 - (x_0 - x_1)(y_0 - y_1) - (x_4 - 2x_6)(y_4 - 2y_6)$
    $-(x_2 - 2x_8)(y_2 - 2y_8) - (x_3 - 2x_7)(y_3 - 2y_7) + (t_3 >> 58)$
11. $z_1 \leftarrow t_2 \bmod t$
12. $t_0 \leftarrow t_0 - t_5$
13. $t_3 \leftarrow t_0 - (x_0 - x_2)(y_0 - y_2) - (x_5 - 2x_6)(y_5 - 2y_6)$
    $-(x_3 - 2x_8)(y_3 - 2y_8) - (x_4 - 2x_7)(y_4 - 2y_7) + (t_2 >> 58)$
14. $z_2 \leftarrow t_3 \bmod t$
15. $t_0 \leftarrow t_0 - 2t_6$
16. $t_2 \leftarrow t_0 - (x_0 - x_3)(y_0 - y_3) - (x_1 - x_2)(y_1 - y_2)$
    $-(x_4 - 2x_8)(y_4 - 2y_8) - (x_5 - 2x_7)(y_5 - 2y_7) + (t_3 >> 58)$
17. $z_3 \leftarrow t_2 \bmod t$
18. $t_0 \leftarrow t_0 - t_6$
19. $t_3 \leftarrow t_0 - (x_0 - x_4)(y_0 - y_4) - (x_1 - x_3)(y_1 - y_3)$
    $-(x_5 - 2x_8)(y_5 - 2y_8) - (x_6 - 2x_7)(y_6 - 2y_7) + (t_2 >> 58)$
20. $z_4 \leftarrow t_3 \bmod t$
21. $t_0 \leftarrow t_0 - 2t_7$
22. $t_2 \leftarrow t_0 - (x_0 - x_5)(y_0 - y_5) - (x_1 - x_4)(y_1 - y_4)$
    $-(x_2 - x_3)(y_2 - y_3) - (x_6 - 2x_8)(y_6 - 2y_8) + (t_3 >> 58)$
23. $z_5 \leftarrow t_2 \bmod t$
24. $t_0 \leftarrow t_0 - t_7$
25. $t_3 \leftarrow t_0 - (x_0 - x_6)(y_0 - y_6) - (x_1 - x_5)(y_1 - y_5)$
    $-(x_2 - x_4)(y_2 - y_4) - (x_7 - 2x_8)(y_7 - 2y_8) + (t_2 >> 58)$
26. $z_6 \leftarrow t_3 \bmod t$
27. $t_0 \leftarrow t_0 - 2t_8$
28. $t_2 \leftarrow t_0 - (x_0 - x_7)(y_0 - y_7) - (x_1 - x_6)(y_1 - y_6)$
    $-(x_2 - x_5)(y_2 - y_5) - (x_3 - x_4)(y_3 - y_4) + (t_3 >> 58)$
29. $z_7 \leftarrow t_2 \bmod t$
30. $t_3 \leftarrow t_1 + (t_2 >> 58)$
31. $z_8 \leftarrow t_3 \bmod t$
32. $z_0 \leftarrow z_0 + 2(t_3 >> 58)$
33. Return $\overline{\mathbf{z}}$

Algorithm 2: **SQR**

`INPUT:` $\overline{\mathbf{x}} = [x_0, \ldots, x_8] \in [-2^{59}, 2^{59}-1] \times [0, 2^{58}-1]^8$
`OUTPUT:` $\overline{\mathbf{z}} \in [-2^{59}, 2^{59}-1] \times [0, 2^{58}-1]^8$ `where` $\overline{\mathbf{z}} \equiv \overline{\mathbf{x}}^2 \pmod{t^9 - 2}$

1. $t_1 = 2(x_0x_8 + x_1x_7 + x_2x_6 + x_3x_5) + x_4^2$
2. $t_0 = t_1 \bmod t$
3. $t_2 = 4(x_1x_8 + x_2x_7 + x_3x_6 + x_4x_5) + x_0^2 + 2(t_1 >> 58)$
4. $z_0 = t_2 \bmod t$
5. $t_1 = 4(x_2x_8 + x_3x_7 + x_4x_6) + 2(x_0x_1 + x_5^2) + (t_2 >> 58)$
6. $z_1 = t_1 \bmod t$
7. $t_2 = 4(x_3x_8 + x_4x_7 + x_5x_6) + 2x_0x_2 + x_1^2 + (t_1 >> 58)$
8. $z_2 = t_2 \bmod t$
9. $t_1 = 4(x_4x_8 + x_5x_7) + 2(x_0x_3 + x_1x_2 + x_6^2) + (t_2 >> 58)$
10. $z_3 = t_1 \bmod t$
11. $t_2 = 4(x_5x_8 + x_6x_7) + 2(x_0x_4 + x_1x_3) + x_2^2 + (t_1 >> 58)$
12. $z_4 = t_2 \bmod t$
13. $t_1 = 4x_6x_8 + 2(x_0x_5 + x_1x_4 + x_2x_3 + x_7^2) + (t_2 >> 58)$
14. $z_5 = t_1 \bmod t$
15. $t_2 = 4x_7x_8 + 2(x_0x_6 + x_1x_5 + x_2x_4) + x_3^2 + (t_1 >> 58)$
16. $z_6 = t_2 \bmod t$
17. $t_1 = 2(x_0x_7 + x_1x_6 + x_2x_5 + x_3x_4 + x_8^2) + (t_2 >> 58)$
18. $z_7 = t_1 \bmod t$
19. $t_2 = t_0 + (t_1 >> 58)$
20. $z_8 = t_2 \bmod t$
21. $z_0 = z_0 + 2(t_2 >> 58)$
22. `Return` $\overline{\mathbf{z}}$

Addition, subtraction, inversion and multiplication by small constants: Addition and subtraction are performed component-wise and need not be reduced if the result is used as input to a multiplication or squaring. Constant-time inversion is performed by powering by $M_{521} - 2 = 2^{521} - 3$. Let x be the element to be inverted and denote x^{2^n-1} by α_n, so $\alpha_1 = x$. Then the inverse of x can be computed at a cost of $520S + 13M$, as follows:

$$\begin{aligned}
\alpha_2 &\leftarrow \alpha_1^2 \cdot \alpha_1 \\
\alpha_3 &\leftarrow \alpha_2^2 \cdot \alpha_1 \\
\alpha_6 &\leftarrow \alpha_3^{2^3} \cdot \alpha_3 \\
\alpha_7 &\leftarrow \alpha_6^2 \cdot \alpha_1 \\
\alpha_8 &\leftarrow \alpha_7^2 \cdot \alpha_1 \\
\alpha_{16} &\leftarrow \alpha_8^{2^8} \cdot \alpha_8 \\
\alpha_{32} &\leftarrow \alpha_{16}^{2^{16}} \cdot \alpha_{16} \\
\alpha_{64} &\leftarrow \alpha_{32}^{2^{32}} \cdot \alpha_{32} \\
\alpha_{128} &\leftarrow \alpha_{64}^{2^{64}} \cdot \alpha_{64} \\
\alpha_{256} &\leftarrow \alpha_{128}^{2^{128}} \cdot \alpha_{128} \\
\alpha_{512} &\leftarrow \alpha_{256}^{2^{256}} \cdot \alpha_{256} \\
\alpha_{519} &\leftarrow \alpha_{512}^{2^7} \cdot \alpha_7 \\
x^{2^{521}-3} &\leftarrow \alpha_{519}^{2^2} \cdot \alpha_1
\end{aligned}$$

This inversion technique is an analogue of the one used by Bernstein for `curve25519` [2]; it may be possible to reduce the number of multiplications slightly, however this would only have a marginal

impact on the efficiency. One could alternatively use Bos' technique [7], but since inversion is required only once during a point multiplication we did not explore this option on the present architecture.

Multiplication by small constants (such as by d for Edwards curves, see Section 4.2) are computed per component and are reduced in-place. We also employ a *short coefficient reduction* (SCR) routine which takes as input a residue $\overline{\mathbf{x}} = [x_0, \ldots, x_8] \in [-2^{63}, 2^{63}-1] \times [-2^{62}, 2^{62}-1]^8$ and outputs one in reduced form. For both multiplication by small constants and SCR, the wrap-around from z_8 to z_0 is computed twice, as with multiplication and squaring.

4 Curves and implementation results

In this section we detail our target curves and implementation results for constant-time variable-base scalar multiplication.

4.1 NIST curve P-521

The Weierstrass form NIST curve P-521 as standardised in [17, 18] has the form $y^2 = x^3 - 3x + b$, with

$$b = 1093849038073734274511112390766805569936207598951683748994586394495953116150735$$
$$0160137087375737596232485921322967063133094384525315910129121423274884789859 84,$$

and group order

$$r_P = 6864797660130609714981900799081393217269435300143305409394463459185543183397655$$
$$3942450577463332171975329639963713633211138647686124403803403728089270700544 9.$$

Using Jacobian projective coordinates, for $P_1 = (X_1, Y_1, Z_1)$ the point $2P_1 = (X_3, Y_3, Z_3)$ is computed as follows:

$$R_0 = Z_1^2, \quad R_1 = Y_1^2, \quad R_2 = X_1 \cdot R_1, \quad R_3 = 3(X_1 + R_0)(X_1 - R_0),$$
$$X_3 = R_3^2 - 8R_2, \quad Z_3 = (Y_1 + Z_1)^2 - R_0 - R_1, \quad Y_3 = R_3 \cdot (4R_2 - X_3) - 8R_1^2.$$

For a point $P_2 = (X_2, Y_2, 1)$ written in affine form which is not equal to P_1, let $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$. Then P_3 is computed as follows:

$$R_0 = Z_1^2, \quad R_1 = X_2 \cdot R_0, \quad R_2 = Y_2 \cdot Z_1 \cdot R_0, \quad R_3 = R_1 - X_1, \quad R_4 = R_3^2$$
$$R_5 = 4R_4, \quad R_6 = R_3 \cdot R_5, \quad R_7 = 2(R_2 - Y_1), \quad R_8 = X_1 \cdot R_5$$
$$X_3 = R_7^2 - R_6 - 2R_8, \quad Y_3 = R_7(R_8 - X_3) - 2Y_1R_6, \quad Z_3 = (Z_1 + R_3)^2 - R_0 - R_4.$$

For efficiency we fused several of the required arithmetic operations. As these are standard techniques we do not detail them here, but they may be found in our freely downloadable software.

In order to achieve constant-time variable-base point multiplication, we used Algorithm 1 of [8] with fixed windows of width 6. Note that since the cost of inversion is about $365M$ (as may be deduced from Table 1 and the formula $I = 520S + 13M$), one should not convert all the precomputed points to affine coordinates as this only saves 5M per addition, which does not outweigh the cost of doing so. One method of precomputation which uses Chudnovsky coordinates is detailed in Algorithm 4 of [8]; we instead use Jacobian projective coordinates as above. For a scalar in $[0, \ldots, 2^{521} - 1]$, in terms of multiplications and squarings only, the cost for a constant-time variable-base scalar multiplication is: $344M + 160S$ for precomputation, plus $2523M + 3045S$ for the windowing plus $16M + 521S$ for the final map to affine coordinates, giving a total cost of $2883M + 3726S$.

4.2 Edwards curve E-521

The Edwards curve E-521 is defined by $x^2 + y^2 = 1 - 376014x^2y^2$ and has group order $4r_E$ where

$$r_E = 1716199415032652428745475199770348304317358825035826352348615864796385795849413$$
$$675475876651663657849636693659065234142604319282948702542317993421293670108523.$$

Point addition, doubling and constant-time point multiplication proceed using exactly the same coordinate systems and formulae described in [3] for `curve41417`, the only differences being that we use fixed windows of width 6 rather than 5 and multipliers of bitlength 519 rather than 414. We therefore do not reproduce these here and refer the reader to Section 3 and Appendix A of [3] for the relevant details.

For a scalar in $[0, \ldots, 2^{519} - 1]$, the cost for a constant-time variable-base scalar multiplication is: $183M + 63S$ for precomputation, plus $2322M + 2064S$ for the windowing plus $16M + 521S$ again for the final map to affine coordinates, giving a total cost of $2522M + 2648S$.

4.3 Timings

We implemented multiplication and squaring as per Algorithms 1 and 2, as well as constant-time variable-base scalar multiplication for curves P-521 and E-521[4], as per Sections 4.1 and 4.2. Our results are detailed in Table 1.

`openSSL`	P-521	E-521	M	S
$1,319,000$	$989,000$	$779,000$	155	105

Table 1. Cycle counts for openSSL version 1.0.2-beta2, P-521 and E-521 variable-base scalar multiplication on a 3.4GHz Intel Haswell Core i7-4770 and compiled with gcc 4.7 on Ubuntu 12.04. The counts are given to the nearest thousand and were obtained by taking the minimum over 10^3 data points, where each data point was the average of 10^4 point multiplications. Cycle counts for multiplication and squaring modulo M_{521} are also included and were the minimum of 10^6 such operations.

Regarding comparisons with previous benchmarks, there are two obvious candidates. For curve P-521, one can test Langley's openSSL implementation (which first featured in version 1.0.0e) using the command `openssl speed ecdh`. On the same architecture, version 1.0.2-beta2 reports 2578.1 operations per second, which implies a count of approximately $1,319,000$ cycles per scalar multiplication. We timed the actual M, S, DBL and ADD functions using several different compilers and options, and found the code to be rather fragile, in that nearly all of them reported a two-fold or more slow down relative to our cycle counts of $155, 105, 1175$ and 1728 for the above operations respectively. However, using gcc 4.7 we obtained cycle counts of $173, 112, 1312$ and 2010 respectively. For a scalar multiplication this implies that our code requires about 88% to 90% of the time required by Langley's in the worst case and less than 50% in the best case.

For E-521, the closest benchmark in the literature is due to Bos *et al.* [8], which reports a cycle count of 1,552,000 for a constant-time variable-base scalar multiplication on the twisted Edwards curve $E/\mathbb{F}_p : -x^2 + y^2 = 1 + dx^2y^2$ with $-1/(d + 1) = 550440$, on a 3.4GHz Intel Core i7-2600 Sandy Bridge processor (albeit with Intel's Turbo Boost and Hyper-threading disabled). This curve form allows a saving of $1M$ per point addition relative to ordinary Edwards curves, but one can not map E-521 to this form since -1 is not a square modulo M_{521}. Our implementation is therefore nearly twice as fast as this one. Apart from the exploitation of our basic observation, much of this difference in performance can be explained by the use of base 2^{58} arithmetic, rather than the base 2^{64} used by Bos *et al.*

[4] For our code see `indigo.ie/~mscott/ed521.cpp` and `indigo.ie/~mscott/ws521.cpp` resp.

Lastly, we note that based on the multiplication and squaring cycle counts in Table 1, the operation counts given in Sections 4.1 and 4.2 for constant-time scalar multiplication on P-521 and E-521 account for about 84.7% and 85.9% of the total cycles respectively. Hence the operation counts give a fair indicator of the relative performance of the two curves.

5 Application to Crandall numbers

Let the modulus be $t^n - c$ and let residues x and y be represented in base t as $\sum_{i=0}^{n-1} x_i t^i$ and $\sum_{i=0}^{n-1} y_i t^i$ respectively. The multiplication of x and y modulo $t^n - c$ is $z = \sum_{i=0}^{n-1} z_i t^i$ where

$$z_i = \sum_{j=0}^{n-1} d(i,j)\, x_{\langle i-j\rangle}\, y_{\langle j\rangle},$$

where the subscripts of the coefficients of $x_{\langle i-j\rangle}$ and $y_{\langle j\rangle}$ are taken modulo n and

$$d(i,j) = \begin{cases} 1 & \text{if } \langle i-j\rangle + \langle j\rangle < n \\ c & \text{otherwise.} \end{cases} \tag{5.1}$$

In particular, by symmetry (or by (5.1)) the terms $x_{\langle i-j\rangle}\, y_{\langle j\rangle}$ and $x_{\langle j\rangle}\, y_{\langle i-j\rangle}$ occurring in the expression for z_i both have the same $d(i,j)$. If $d(i,j) = 1$ then the expression $x_{\langle i-j\rangle}\, y_{\langle j\rangle} + x_{\langle j\rangle}\, y_{\langle i-j\rangle}$ may be rewritten just as before as

$$x_{\langle i-j\rangle}\, y_{\langle i-j\rangle} + x_{\langle j\rangle}\, y_{\langle j\rangle} - (x_{\langle i-j\rangle} - x_{\langle j\rangle})\,(y_{\langle i-j\rangle} - y_{\langle j\rangle}).$$

Similarly, the expression $c\,(x_{\langle i-j\rangle}\, y_{\langle j\rangle} + x_{\langle j\rangle}\, y_{\langle i-j\rangle})$ may of course be rewritten as

$$c\,\left(x_{\langle i-j\rangle}\, y_{\langle i-j\rangle} + x_{\langle j\rangle}\, y_{\langle j\rangle} - (x_{\langle i-j\rangle} - x_{\langle j\rangle})\,(y_{\langle i-j\rangle} - y_{\langle j\rangle})\right),$$

or even as

$$x_{\langle i-j\rangle}\, y_{\langle i-j\rangle} + c^2\, x_{\langle j\rangle}\, y_{\langle j\rangle} - (x_{\langle i-j\rangle} - c\, x_{\langle j\rangle})\,(y_{\langle i-j\rangle} - c\, y_{\langle j\rangle}),$$

as was used in Algorithm 1 for M_{521}. Therefore, by precomputing the terms $x_i y_i$ for $i = 0, \ldots, n-1$, the paired terms in the expression for each z_i may be computed as the products above, which again nearly halves the total number of coefficient multiplications required, at the expense of a few additions and multiplications by c.

5.1 Two examples

In order to use the above observations, it will often be necessary to first multiply a given Crandall number $p = 2^k - c$ by 2^i so that $k+i$ is a multiple of a suitable n, such that all coefficient arithmetic does not overflow beyond double precision. In this case the base $t = 2^{(k+i)/n}$ is a power of two and coefficient renormalisation – which ensures I/O stabilty – can be effected via simple operations (including shifts) only. This will be clear from the following two examples.

5.2 Application to $p = 2^{221} - 3$

This prime was proposed in [1]. For use on a 64-bit architecture we choose $t = 2^{56}$ and use the scaled modulus $8p = t^4 - 24$. The multiplication algorithm is as follows.

Algorithm 3: **MUL2213**

INPUT: $\overline{\mathbf{x}} = [x_0, \ldots, x_3], \overline{\mathbf{y}} = [y_0, \ldots, y_3] \in [-2^{57}, 2^{57} - 1] \times [0, 2^{56} - 1]^3$
OUTPUT: $\overline{\mathbf{z}} \in [-2^{57}, 2^{57} - 1] \times [0, 2^{56} - 1]^3$ where $\overline{\mathbf{z}} \equiv \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \pmod{t^4 - 24}$

1. $a_0 \leftarrow x_0 y_0$, $a_1 \leftarrow x_1 y_1$, $a_2 \leftarrow x_2 y_2$, $a_3 \leftarrow x_3 y_3$, $b_2 \leftarrow a_3 + a_2$, $b_1 \leftarrow b_2 + a_1$, $b_0 \leftarrow b_1 + a_0$
2. $t_0 \leftarrow b_0 - (x_0 - x_3)(y_0 - y_3) - (x_1 - x_2)(y_1 - y_2)$
3. $z_3 \leftarrow t_0 \bmod t$
4. $t_1 \leftarrow a_0 + 24(b_1 - (x_1 - x_3)(y_1 - y_3) + (t_0 >> 56))$
5. $z_0 \leftarrow t_1 \bmod t$
6. $a_0 \leftarrow a_0 + a_1$
7. $t_0 \leftarrow a_0 - (x_0 - x_1)(y_0 - y_1) + 24(b_3 - (x_2 - x_3)(y_2 - y_3)) + (t_1 >> 56)$
8. $z_1 \leftarrow t_0 \bmod t$
9. $t_1 \leftarrow a_0 + a_2 + 24a_3 - (x_0 - x_2)(y_0 - y_2) + (t_0 >> 56)$
10. $z_2 \leftarrow t_1 \bmod t$
11. $t_0 \leftarrow z_3 + (t_1 >> 56)$
12. $z_3 \leftarrow t_0 \bmod t$
13. $z_0 \leftarrow z_0 + 24(t_0 >> 56)$
14. Return $\overline{\mathbf{z}}$

Note that as with Algorithm 1 we compute the wrap around from the highest coefficient to the lowest twice, in order to maintain I/O stability for the chosen reduced format for residues.

5.3 Application to $p = 2^{255} - 19$

This prime was proposed in [2] and later developed with base $t = 2^{51}$ arithmetic [6], which we use here. We hence use the modulus $p = t^5 - 19$. The multiplication algorithm is as follows.

Algorithm 4: **MUL25519**

INPUT: $\overline{\mathbf{x}} = [x_0, \ldots, x_4], \overline{\mathbf{y}} = [y_0, \ldots, y_4] \in [-2^{52}, 2^{52} - 1] \times [0, 2^{51} - 1]^4$
OUTPUT: $\overline{\mathbf{z}} \in [-2^{52}, 2^{52} - 1] \times [0, 2^{51} - 1]^4$ where $\overline{\mathbf{z}} \equiv \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \pmod{t^5 - 19}$

1. $a_0 \leftarrow x_0 y_0$, $a_1 \leftarrow x_1 y_1$, $a_2 \leftarrow x_2 y_2$, $a_3 \leftarrow x_3 y_3$, $a_4 \leftarrow x_4 y_4$
   $b_3 \leftarrow a_4 + a_3$, $b_2 \leftarrow a_3 + a_2$, $b_1 \leftarrow b_2 + a_1$, $b_0 \leftarrow b_1 + a_0$
2. $t_0 \leftarrow b_0 - (x_0 - x_4)(y_0 - y_4) - (x_1 - x_3)(y_1 - y_3)$
3. $z_4 \leftarrow t_0 \bmod t$
4. $t_1 \leftarrow a_0 + 19(b_1 - (x_1 - x_4)(y_1 - y_4) - (x_2 - x_3)(y_2 - y_3) + (t_0 >> 51))$
5. $z_0 \leftarrow t_1 \bmod t$
6. $a_0 \leftarrow a_0 + a_1$
7. $t_0 \leftarrow a_0 - (x_0 - x_1)(y_0 - y_1) + 19(b_2 - (x_2 - x_4)(y_2 - y_4)) + (t_1 >> 51)$
8. $z_1 \leftarrow t_0 \bmod t$
9. $a_0 \leftarrow a_0 + a_2$
10. $t_1 \leftarrow a_0 - (x_0 - x_2)(y_0 - y_2) + 19(b_3 - (x_3 - x_4)(y_3 - y_4)) + (t_0 >> 51)$
11. $z_2 \leftarrow t_1 \bmod t$
12. $t_0 \leftarrow a_0 + a_3 - (x_0 - x_3)(y_0 - y_3) - (x_1 - x_2)(y_1 - x_2) + 19a_4 + (t_1 >> 51)$
13. $z_3 \leftarrow t_0 \bmod t$
14. $t_1 \leftarrow z_4 + (t_0 >> 51)$
15. $z_4 \leftarrow t_1 \bmod t$
16. $z_0 \leftarrow z_0 + 19(t_1 >> 51)$
17. Return $\overline{\mathbf{z}}$

Note that the multiplications by 24 in Algorithm 3 and by 19 in Algorithm 4 are generally on double precision integers, so are sometimes more costly than the usual evaluation of coefficients in which one can sometimes achieve a saving by first multiplying one the inputs to a multiplication by c. Since there are also more additions required than for Algorithm 1's analogue of the basic observation, these formulae may only be interesting when optimised, or when implemented on ARM processors, for instance. As our focus is primarily on M_{521}, P-521 and E-521, we leave such options as open research.

6 Conclusion

We have proposed a very simple way to improve multiplication efficiency over the prime field $\mathbb{F}_{2^{521}-1}$, which requires as few word-by-word multiplications as is needed for squaring, while incurring very little overhead from extra additions. With optimised code our timings may be reduced even further, with potentially interesting results on ARM processors, for which the multiplication-to-addition cost ratio is higher than on the Haswell and for which there are numerous similar methods to represent and multiply residues using 32 bit words. It remains to be seen whether the same basic observation improves the efficiency of multiplication modulo Crandall numbers as well.

Acknowledgements

We thank Dan Bernstein for answering our questions regarding his irrational base modular multiplication method.

References

1. Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, and Jefferson Ricardini. A note on high-security general-purpose elliptic curves, 2013. `http://eprint.iacr.org/2013/647`.
2. Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pages 207–228, 2006.
3. Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. Cryptology ePrint Archive, Report 2014/526, 2014. `http://eprint.iacr.org/`.
4. Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980, 2013.
5. Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to`, accessed 11 September 2014.
6. DanielJ. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.
7. Joppe W. Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, pages 1–7, 2014.
8. Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130, 2014. `http://eprint.iacr.org/`.
9. Joppe W. Bos, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Efficient simd arithmetic modulo a mersenne number. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*, ARITH '11, pages 213–221, Washington, DC, USA, 2011. IEEE Computer Society.
10. Jaewook Chung and Anwar Hasan. More generalized mersenne numbers. In Mitsuru Matsui and RobertJ. Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 335–347. Springer Berlin Heidelberg, 2004.
11. Jaewook Chung and M. Anwar Hasan. Montgomery reduction algorithm for modular multiplication using low-weight polynomial form integers. In ARITH 18, pages 230–239, 2007.
12. Jaewook Chung and M.A Hasan. Low-weight polynomial form integers for efficient modular multiplication. *Computers, IEEE Transactions on*, 56(1):44–57, Jan 2007.
13. R. Crandall and C.B. Pomerance. *Prime Numbers: A Computational Perspective*. Lecture notes in statistics. Springer, 2006.
14. R.E. Crandall. Method and apparatus for public key exchange in a cryptographic system, October 27 1992. US Patent 5,159,632.
15. R.E. Crandall. *Topics in Advanced Scientific Computation*. Electronic Library of Science. Springer-Telos, 1996.
16. Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Math. Comput.*, 62(205):305–324, 1994.
17. US Department of Commerce/N.I.S.T. 2000. Federal Information Processing Standards Publication 186-2. Fips 186-2. digital signature standard.
18. Standards for Efficient Cryptography Group. Recommended elliptic curve domain parameters, 2000. Available from `www.secg.org/collateral/sec2.pdf`.
19. Robert Granger and Andrew Moss. Generalised Mersenne numbers revisited. *Math. Comp.*, 82(284):2389–2420, 2013.
20. A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595–596, January 1963.

21. E. Öztürk, B. Sunar, and E. Sava. Low-power elliptic curve cryptography using scaled modular arithmetic. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin Heidelberg, 2004.
22. Colin D. Walter. Faster modular multiplication by operand scaling. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 313–323, London, UK, UK, 1992. Springer-Verlag.