Fast Scalar Multiplication for ECC over GF(p) Using Division Chains

Colin D. Walter*

Information Security Group, Royal Holloway, University of London,
Egham, Surrey, TW20 0EX, United Kingdom
Colin.Walter@rhul.ac.uk

Abstract. There have been many recent developments in formulae for efficient composite elliptic curve operations of the form $dP+Q$ for a small integer d and points P and Q where the underlying field is a prime field. To make best use of these in a scalar multiplication kP, it is necessary to generate an efficient "division chain" for the scalar where divisions of k are by the values of d available through composite operations.

An algorithm-generating algorithm for this is presented that takes into account the different costs of using various representations for curve points. This extends the applicability of methods presented by Longa & Gebotys at PKC 2009 to using specific characteristics of the target device. It also enables the transfer of some scalar recoding computation details to design time. An improved cost function also provides better evaluation of alternatives in the relevant addition chain.

One result of these more general and improved methods includes a slight increase over the scalar multiplication speeds reported at PKC. Furthermore, by the straightforward removal of rules for unusual cases, some particularly concise yet efficient presentations can be given for algorithms in the target device.

Keywords: Scalar multiplication, multibase representation, addition chain, division chain, exponentiation, DBNS, elliptic curve cryptography.

1 Introduction

Exponentiation has been the subject of much study over the years. Classic summaries of the state of the art a decade ago are Knuth and Gordon [14,12]. The aim of these methods is almost always minimising the operation count in an addition chain for the exponent. This is an NP-hard problem. More recent work has concentrated on wider optimisation issues, such as those imposed by constraints on the size of any pre-computed table of values, the working space required, the relative costs of squaring and multiplication, and reducing the side channel leakage generated by a particular operation sequence.

* The work described in this paper has been supported [in part] by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

Recently, efficient composite elliptic curve operations of the form $dP{+}Q$ have been developed for double-and-add ($d = 2$), triple-and-add ($d = 3$) and quintupling ($d = 5$) etc. over $\mathbb{F}_p$ [6,18,16]. By careful choice of curve formulae and affine coordinates for the table entries, these become faster than equivalent combinations of the normal point addition and doubling formulae, and so should lead to faster scalar multiplication. With each application of the operation $dP{+}Q$ to the accumulating scalar multiple, the part of the key which has yet to be processed is reduced by a factor d using integer division. This is exactly the process for generating a division chain, and so division chains are the natural setting for optimising scalar multiplication when these composite operations are available.

Division chains were first defined in [19] and [20] where the author uses them to derive efficient addition chains in the context of limited table and working space. The resulting chains are significantly shorter than those for binary exponentiation when using just one more register. Without cheap inversion (as is the case for RSA), exponentiation by k can be performed in a right to left direction using an average of about $1.4\log_2(k)$ operations using divisors $d = 2$ and $d = 3$ (*op. cit.* §5). This improves on the usual $\frac{3}{2}\log_2(k)$ operations using only the divisor $d = 2$ but is less efficient than the $\frac{4}{3}\log_2(k)$ for sliding 2-bit windows[1] which also uses space equivalent to one more register than binary methods. Although with more divisors the speed can be improved to under $\frac{5}{4}\log_2(k)$ operations using the same space, this requires much more pre-processing of the exponent and applies only to the right to left direction for which the composite elliptic curve operations $dP{+}Q$ do not have more efficient forms. Figures are not provided in [19] for the addition-subtraction chains relevant for elliptic curves, nor are they provided for the left-to-right direction which is marginally less efficient for space constrained implementations but which is of relevance here where the composite operations provide increased efficiency. One aim of the present work is to fill this gap.

For application to elliptic curves, it is the number of field operations which particularly determines the time efficiency of scalar point multiplication. This number depends to a large extent on the form of the curve equation and the coordinate representations chosen for the table entries and intermediate points during processing[2] [13]. The resulting cost of point doublings, additions, triplings etc., rather than the length of the addition sub-chain, is the natural input to the cost evaluation function from [19] which is used to compare competing alternative divisors. This explicit function enables the design time derivation of the algorithm for choosing divisors at run time in a more refined manner than in [17]. Moreover, it transfers *all* of the search for the best divisor to design time.

[1] k is partitioned from right to left into "digit" subsequences 0, 01 and 11. Exponentiation is performed left to right using table entries for 01 and 11. Even digits 0 occur with probability $\frac{1}{2}$ whereas the odd digits 01 and 11 occur with probability $\frac{1}{4}$. So an exponent bit has probability $\frac{1}{3}$ of being associated with digit 0, and $\frac{2}{3}$ with digit 01 or 11. Thus the average cost per exponent bit is asymptotically $\frac{1}{3}\cdot\frac{1}{1}+\frac{2}{3}\cdot\frac{3}{2} = \frac{4}{3}$ operations.

[2] Some speed-ups require additional space for four or more coordinates per point.

The less structured double base and multibase representations of Dimitrov *et al.* [4,5,6] represent the scalar k as a signed sum of powers of two (or more) numbers. The extra freedom there allows the use of a slower, greedy algorithm to obtain a very compact representation. However, compactness is not the aim here and extra conditions are required to provide the structure required here for an efficient conversion to a fast addition sequence. These extra conditions are those of a *randomary* representation [22]. Similar representations have been created more recently by Ciet, Longa *et al.* [3,15,17] but restricted to cases in which all digits are zero except those associated with divisor 2 (or its powers).

Here those restrictions are lifted to give a very general setting which encompasses all this previous work on the more structured multibase forms. A side effect of the generality is a modest increase in the speed records achieved in [17]. But the results are also applicable in wider contexts than the particular devices targetted there. For example, the relatively cheap cost of tripling on DIK3 curves [10] makes use of the new availability of non-zero digits associated with base 3.

The main body of the paper describes the above in more detail: division chains and their notation are covered initially; then a discussion on selecting divisor and digit sets followed by details of the cost function for determining the recoding process, and results to evaluate a variety of parameter choices. Finally, Appendix B contains an explicit and very straightforward example algorithm for generating an efficient division chain in the case of divisor set {2,3}.

2 Addition Chains

The concept of addition chains and addition-subtraction chains needs extending to the context here. Let $\mathcal{OP}$ be the set of "distinguished" operations which we wish to use in exponentiations schemes. For convenience only unary or binary operations are considered. For the (additive) group G of interest, they combine one or two multiples of a given $g \in G$ into a higher multiple of g. So, for each $p \in \mathcal{OP}$ there will be a pair $(\lambda, \mu) \in \mathbb{Z}^2$ such that $p(g,h) = \lambda g+\mu h$ for all $g, h \in G$. As special cases, $(1,1)$ yields $g+h$ (an addition), $(2,0)$ yields $2g$ (a doubling) and $(-1,0)$ provides $-g$ (an inversion). Thus $\mathcal{OP}$ can be viewed as a subset of $\mathbb{Z}^2$. Recent research has provided double-and-add, triple-and-add and even quintuple operations [18,16] to help populate $\mathcal{OP}$. These are of the type $p = (2,d)$ or $p = (3,d)$ etc. where the d-th multiple of the initial point g is in the pre-computed table. These operations lead to a more general addition chain:

Definition 1. *For a set $\mathcal{OP}$ of linear binary operators on an additive group, an $\mathcal{OP}$-*addition chain *for $k \in \mathbb{Z}$ of length n is a sequence of quadruples $(a_i, b_i, k_i, p_i) \in \mathbb{Z}^3 \times \mathcal{OP}$, $0 \le i \le n$, such that, for all $i > 0$,*

- $k_i = \lambda_i a_i + \mu_i b_i$ *for* $p_i = (\lambda_i, \mu_i) \in \mathcal{OP}$
- $a_i = k_{s_i}$ *and* $b_i = k_{t_i}$ *for some* s_i *and* t_i *with* $0 \le s_i < i$ *and* $0 \le t_i < i$
- $(a_0, b_0, k_0, p_0) = (1, 0, 1, (1, 0))$
- $k_n = k$.

3 Representations from Division Chains

Addition chains are too general for use in devices with limited memory. The standard left-to-right binary exponentiation algorithm overwrites all previous calculations with the latest value and so its operations can only access the most recent value and the initial input. Hence its addition chain can only contain triples of the form $(k_{i-1}, k_{i-1}, 2k_{i-1})$ (a squaring) or $(k_{i-1}, 1, k_{i-1}{+}1)$ (a multiplication). Schemes with pre-computed tables are similarly restricted.

The natural extension of this restriction allows only quadruples of the form $(k_{i-1}, 1, \lambda_i k_{i-1}{+}\mu_i 1, (\lambda_i, \mu_i))$ in an $\mathcal{OP}$-addition chain, i.e. $k_i = \lambda_i k_{i-1}{+}\mu_i$. Consequently, just as the sequence of squares and multiplications completely determines the exponent k in binary exponentiation, so the sequence of operations (λ_i, μ_i) also determines k. Given k, we can obtain such a sequence (in reverse order) by iteratively choosing $(\lambda_i, \mu_i) \in \mathcal{OP}$ such that μ_i lies in the residue class of k_i mod λ_i, and then performing the division $k_{i-1} = (k_i{-}\mu_i)/\lambda_i$. The process starts with $k = k_n$ and finishes with $k_0 = 0$. This defines a *division chain* for an integer k, and it is specified by the list of pairs (λ, μ) corresponding to the constituent operations. This is clear from the presentation of k as

$$k = (((\mu_1\lambda_2 + \mu_2)\lambda_3 + ... + \mu_{n-2})\lambda_{n-1} + \mu_{n-1})\lambda_n + \mu_n \qquad (1)$$

Subscripts for addition chains are used in the opposite direction from those in digit representations of numbers. In order to obtain the customary notation for the latter, let us re-number the pairs (λ, μ) by defining $(r_i, d_i) = (\lambda_{n-i}, \mu_{n-i})$. Then the division chain determines a *randomary* representation[3] of k [19]:

$$k = (((d_{n-1}r_{n-2} + d_{n-2})r_{n-3} + ... + d_2)r_1 + d_1)r_0 + d_0 \qquad (2)$$

in which the r_i are called *bases* and the d_i are called *digits*. If all the divisors r_i were equal, then this would simply be a representation in radix $r = r_i$. Digits would be chosen in $[0..r-1]$ to yield the standard representation. Here the bases will belong to some small set $\mathcal{B} \subset \mathbb{N}$ such as $\{2,3,5\}$ and the digits will lie in a subset $\mathcal{D} \subset \mathbb{Z}$ which may contain negative values and may depend on the base.

An example from [22] is

$$235_{10} = (((((1)3 + 0)2 + 1)5 + 4)2 + 0)3 + 1.$$

Following standard subscript notation for specifying the base, this is written $235_{10} = 1_2 0_3 1_2 4_5 0_2 1_3$. Digit/base pairs are obtained using the usual *change of base* algorithm except that the base may be varied at each step: for $i = 0, 1, ...$ extract digit d_i by reducing modulo r_i, and divide by r_i. In the example of 235_{10},

[3] Developed from the words *binary*, *ternary*, *quaternary* etc. to indicate the apparent random nature of base choices as used in, for example, [21]. This is a not a multi-base (or mixed base) representation in the original sense of that term, and so "randomary" may be useful to distinguish the two forms. A double base (DBNS) example is writing $k = \sum_i \pm 2^{b_i} 3^{t_i}$, which has a different structure.

the first base is 3 and 235 mod 3 = 1 is the first digit (that of index 0). The algorithm continues with input $(235 - 1)/3 = 78$ to determine the next digit.

Such representations determine a left-to-right exponentiation scheme using Horner's rule (which follows the bracketing above from the inside outwards). In an additive group G the iterative step is

$$k_i g = r_{n-i}(k_{i-1}g) + (d_{n-i}g) \qquad (3)$$

using table entry $d_{n-i}g$ and starting at the most significant, i.e. left, end.

4 Choosing the Base Set $\mathcal{B}$

This section considers the choice of base set when the scalar k is used just once and the full cost of creating the multibase representation must be counted. If a cryptographic token were to use the same key throughout its lifetime then more effort could be spent on optimising the representation offline before initialisation, and so different rules would apply. Such a situation is not covered here.

Application of base r reduces the key by a factor of almost exactly r and so shortens k by close to $\log_2 r$ bits. If this requires c_r group operations when used, then its approximate cost for comparing with other bases is $c_r/\log_2 r$ (when neglecting other benefits). Apart from powers of 2 this ratio is lowest for numbers of the form $r' = 2^{n'} \pm 1$ as these consume very close to n' scalar bits for a cost of only $n'+1$ or $n'+2$ operations according to whether the digit is zero or represented in the pre-computed table. For large n' this is going to be better than the average cn for the whole algorithm on a key of n bits, where c is typically in the region of 1.2 to 1.25. With a table of T entries such bases can be used in about $2T+1$ out of r' cases. k mod r' must be computed every time to check the usability of r' and, for k in binary, the effort involved in this is proportional to $n' \log(k)$. However, the saving to the scalar multiplication is less than $(c-1)n'$ group operations per use, giving an average of $(2T+1)(c-1)n'2^{-n'}$ per calculation of k mod r'. Hence, as n' increases the re-coding cost will quickly outweigh any benefit obtained from its use. The same reasoning applies to any large non-2-power radix. Hence $\mathcal{B}$ should only contain small numbers.

Generation of addition-subtraction chains for all small integers shows that, with few exceptions, exponentiation by most bases r requires close to $1.25 \log_2 r$ operations[4] without counting the extra one for any point addition. With an aim of achieving an average of fewer than 1.25 operations per bit, such base choices are not going to be of much use unless the associated digit is zero or there is foreseeable benefit, such as a zero digit in the immediate future.

An investigation of small bases was carried out. It was noted first that no base r below 2^9 required more than two registers for executing a shortest addition

[4] The distribution is much more uniform than for NAF, and the 1.25 operations per bit slowly decreases. NAF gives a chain with an asymptotic average of $\frac{4}{3}\log_2 r$ operations. 27 is the first number with an add/sub chain shorter than the NAF, and 427 is the first number with an add/sub chain two operations shorter than the NAF. 31 is the first that needs a subtraction for its shortest chain.

chain for r – the minimum possible for numbers other than powers of 2. So space is not an issue. For the most part, only prime candidates need be considered for the following reason. Suppose radix $r = st$ is composite and it is to be used on k with digit d. Then k would be reduced to $(k{-}d)/st$. However, application of the base/digit pairs $(s, 0)$, (t, d) has the same effect. Consequently, the value of including r as a divisor depends on whether or not a minimal addition chain for r is shorter than the sum of the chain lengths for any factorisation of r. As an example, no powers of 2 above 2 itself need be included. Thus, the pair $(4, 3)$ can be replaced without cost by $(2, 0)$ followed by $(2, 3)$. However, 33 should be kept in as it has a chain of length 6 which is less than the combined chain lengths for 3 and 11. The investigation revealed $\{33, 49, 63, 65, 77, 95, 121, 129, 133, 143, \ldots\}$ to be the sequence of composite numbers having shorter addition/subtraction chains (by sequence length) than the sum of those of any factorisation.

In fact, *shorter* needs to be interpreted at a finer level of detail. Powers of 3 then become more interesting: two applications of base 3 requires two point doublings and two point additions but one application of base 9 can be achieved with three doublings and one addition. This is the same number of point operations, but should be cheaper since a doubling ought to have a lower cost than a point addition. Similarly, 15 is cheaper than 3 and 5 separately.[5]

After omissions for the above reasons and because of poor ratios $c_r/\log_2 r$, there remains only the initial part of the sequence

$$\{2, 3, 5, 9, 15, 17, 31, 33, 47, 63, 65, 77, 97, 127, 129, \ldots\}.$$

In particular, 7, 11 and 13 have been removed. They have among the very highest costs of any base, with $c_r/\log_2 r$ well above $\frac{4}{3}$. This high cost holds even when the relative cost of a doubling is accounted for and has a value anywhere between 0.5 and 1.0 times the cost of an addition. As the ratio approaches 0.5 the popular base 3 also becomes extremely expensive under this measure, whereas bases 5 and 9 are consistently good. 3 is better than 5 only once the ratio is above 0.87.

An exhaustive search was made for shortest divisor chains using choices from the above sequence to determine which bases were the most useful since any resource-constrained device can only allow a few possibilities. Shortest was measured assuming point doubling to have a cost between half and one times the cost of an addition (*see* [17], Table 1). The digit set was allowed to be any subset of $\{-7, ..., +7\}$. In all cases, the frequency of use was close to, in descending order, $\{2; 3; 5; 9, 7; 17, 13, 11; 33, 31, 23, 19; 65, 63, ...; 129, 127, ...\}$ which is 2 concatenated with the sequences $2^{n+1}{+}1, ..., 2^n{+}3$ for $n = 0, 1, 2, ...$[6] As expected, the frequencies of bases r decrease as r increases. Consequently, it is reasonable to consider only bases in the set $\{2, 3, 5, 9, 17\}$[7] since, with such low frequencies, more bases can only give negligible speed-up to a scalar multiplication. Furthermore, extra bases would require more code and more pre-computation. In fact,

[5] The finer detail of gains from using composite operations is treated below.

[6] This listing assumes that bases r of interest have minimal addition chains of length $1{+}\lceil\log_2(r{-}2)\rceil$, which is true for the range of interest and the above restrictions.

[7] In order of decreasing frequency when doubling is half the cost of addition.

the decrease in frequencies is much greater when the cost of doubling approaches that of point additions. In this case, it is hardly worth considering more than just $\{2, 3, 5, 17\}$.[8] However, when doubling is just half the cost of adding, note that 7, 11 and 13 are still possible options. Despite the heavy average cost per bit by which these bases reduce k, they can lead to future values of k which have cheaper than average reductions and enable the extra cost to be recovered. The search for optimal chains also revealed that odd bases rarely make use of non-zero digits. This was the case even when the addition of a table element was assigned just half the cost of the additions used in multiplying by a base. Thus almost no extra speed is likely to be achieved if non-zero digits are allowed for odd bases when composite curve operations are introduced.

This leads to the most sensible base choices for embedded cryptosystems being $\mathcal{B} = \{2, 3\}$ or $\mathcal{B} = \{2, 3, 5\}$. They were first considered in [21,22] and more recently by [18,17] where non-zero digits are only allowed for base 2. Larger sets provide virtually no reasonable advantages and might only be considered out of vanity for attaining a rather marginal increase in speed.

5 Choosing the Digit Set $\mathcal{D}$

Using the very rough cost function given by addition chain length in the previous section it became clear that the fastest scalar multiplication methods would use non-zero digits almost exclusively with the base 2. Consequently, digits which are multiples of 2 can be eliminated as follows. Applying the base/digit pair $(r, 0)$ followed by $(2, 2d)$ is equivalent to applying the (r, d) followed by $(2, 0)$. So the table need not contain even digit multiples of the initial point to cover this case. Similarly, $(2, d')$ followed by $(2, 2d)$ is equivalent to $(2, d{+}d')$ followed by $(2, 0)$, but the new expression saves one point addition. If $d{+}d'$ is not an acceptable digit, $(2, d{+}d')$ could be implemented as a double and add (of multiple d) followed by an extra addition (of multiple d'), thereby again eliminating the need for any even digit multiples in the table or any further adjustment to the representation. Alternatively, $d{+}d'$ could be split into an acceptable digit and a carry up to the preceding base/digit pair as in the constant base case, with this process repeated until the digit problem is resolved at no additional cost.

As in the case of NAF, and confirmed by the search in §4, the choice of a non-zero digit is almost always going to be made to make the next one or more digits zero. By taking the digit set $\mathcal{D} = \{0, \pm1, \pm2, \ldots, \pm(2T{-}1)\}$ for given table size T, it is possible to maximise the power of 2 achievable as a factor of the next value of k during recoding. Because the composite operators work equally well for any table entry, the desired operation set now has the form $\mathcal{OP} = \mathcal{B}{\times}\mathcal{D}$ where the two factors are now known.

6 Optimising the Representation

As just noted, a primary aim in the multibase algorithm is to select bases for which the digit is zero as this saves the point addition. When this is not possible,

[8] In order of decreasing frequency when doubling has the same cost as addition.

the recoding strategy should make a base/digit choice to ensure it happens next time or as soon as possible thereafter. This requires knowing the key modulo a power of each available base. Let π be a common multiple of the elements in the base set $\mathcal{B}$ which contains only primes dividing available bases and in which it is helpful to have a high power of two because of the frequency with which 2 will be chosen as a base. Decisions will be based on $k_i \bmod \pi$. Indeed, for arbitrary p prime to all base choices, the value of $k_i \bmod p$ can clearly have no influence on the choice of base. However, if π were picked large enough (e.g. larger than k) we would have complete knowledge of k and so there would be enough information to determine an optimal division chain.

Divisor choices are not independent. The choice of the first divisor is partly affected by the best choice for the next, and less so by subsequent ones. After a chain of four or more, simulations show that the effect is minimal. Hence we want to pick the first divisor to minimise the average cost of the addition sub-chain formed by the next four or so divisors. So let λ be the length of sub-chain to be investigated. This would be the window size if the base were fixed. Initially, let π be the least common multiple of the λ^{th} powers of each base in $\mathcal{B}$. If the pre-computed table has T elements and δ is minimal such that $T \leq 2^{\delta}$ then a base 2 digit can sometimes be chosen to guarantee divisibility by at least $2^{\delta+2}$. (For example, with $T = 3$ and table entries for $\{\pm1, \pm3, \pm5\}$, we have $\delta = 2$ and, when $k_i{\equiv}5 \bmod 8$, either of the digits $-3, 5$ might be used, but one will make $k_i{-}d_i{\equiv}0 \bmod 16$.) In such cases, in effect, we have a base of $2^{\delta+2}$. Thus the base 2 digits cannot be chosen accurately towards the end of the window unless π is augmented by another factor of at least $2^{\delta+1}$. Moreover, this example shows that the cost benefits of digit choices continue beyond the end of the window.

For each residue mod π, all possible sub-chains of length λ are generated. The first pair (r, d) is then chosen from the "best" sub-chain and assigned to be the choice whenever k_i has that value modulo π. Explicit code for this is presented in the first algorithm of Appendix A. Since the search takes a total of $O(\pi\{T\sum_{r\in\mathcal{B}}\frac{1}{r}\}^{\lambda})$ time, $\lambda = 3$ is easily achievable with reasonable resources.

In the next section, the device-specific definition of "best" is given properly. However, the above association of pairs (r, d) with residues mod π is obtained from an off-line search and yields a simple algorithm for programming into the target device, such as the second algorithm in Appendix A. Most divisor choices are determined by very simple rules such as selecting the pairs $(2, 0)$ and $(3, 0)$ respectively when the residue mod π is divisible by 2 or 3. So the table of data can generally be converted to a much more compact set of simple rules. The full set of rules mod π may contain many exceptional cases. They become less critical to the performance of the algorithm as the number of residues affected decreases, and so some very small cases can be absorbed into larger ones with very little loss in performance, thereby reducing code size. Appendix B contains two examples of the recoding algorithm's iterative step showing just how compact the resulting code can be and yet still provide a very efficient scheme.

6.1 The Detailed Cost Function

In §4 the cost per bit of applying base r is given as $c_r/\log_2 r$ where c_r was initially taken as the length of the addition chain used for r in order to ascertain which were the best values to put in $\mathcal{B}$. From now on, costs $c_{r,d}$ are taken at the accuracy required to differentiate successfully between competing choices for (r, d). If reading and writing operations, field additions and scalar field multiplications are relatively cheap, $c_{r,d}$ is given by the appropriately weighted sum of field multiplication and squaring counts. They depend on the form of the equation and the coordinate representation chosen for the elliptic curve and its points. The reader is referred to [17], Table 1, for examples covering the specialised efficient doubling, tripling and quintupling operations of interest here.

A sequence of λ iterative steps of the type (3) can be composed into a single operation of the same type, represented by a base/digit pair (r, d) where the new base r is the product of those of the constituent pairs. The cost function $c_{r,d}$ can be extended to this by adding the costs of the component pairs. However, the per bit cost $c_{r,d}/\log_2 r$ is no longer a good estimate of the relative value of the choice of either (r, d) or the first pair in the window. The reduction achieved by any sub-chain is typically followed by an average reduction. Suppose c is the average cost per bit and an alternative sub-chain has cost $c_{r',d'}$ to achieve a reduction by (r', d'). Removing a further factor of r/r' from $(k_i{-}d')/r'$ costs an average of $c\log_2(r/r')$, giving a total of $c_{r',d'}{+}c\log_2(r/r')$ for reducing k_i by a factor of r. So the first choice is better on average if $c_{r,d} < c_{r',d'}{+}c\log_2(r/r')$, i.e. if $c_{r,d}{-}c\log_2 r < c_{r',d'}{-}c\log_2 r'$. Thus the "best" sub-chain is taken to be the one minimising

$$c_{r,d} - c\log_2 r \tag{4}$$

This is simply the extra work required above the average expected. Although use of (4) requires advance knowledge of the average cost per bit c, various values can be tried in practice, of course, in order to converge on the smallest.

Finally, no choice of base/digit sequence for the window makes full use of the value $k_i \bmod \pi$. For a sub-chain equivalent to (r, d), $c_{r,d}$ does not take into account properties of $(k_i{-}d)/r \bmod \pi/r$. As noted above, the final digit in the window is usually chosen, like the others, to give exact divisibility in the following digit when that possible. So the sub-chain pair (r, d) in (4) should be augmented to include pairs $(r', 0)$ representing any remaining divisibility in $k_i{-}d \bmod \pi$.

6.2 A Markov Process

An interesting consequence of specifying the divisor/digit pair to choose given the residue of k_i modulo π is that these residues become non-uniform after the first iteration and, after about half a dozen choices, have settled close to their asymptotic probabilities. This is a Markov process which, in the limit, leads to a precise value for the work c per bit required when evaluating (4).

The process is described by a $\pi{\times}\pi$ transition probability matrix $P = (p_{ij})$ where p_{ij} is the probability that an input of $i \bmod \pi$ will generate an output of $j \bmod \pi$. If the pair (r_i, d_i) is associated with $i \bmod \pi$ then the output j

will be one of the r_i values $(i{-}d_i{+}t\pi)/r_i \bmod \pi$ for $0 \le t \le r_i{-}1$. As these are all equally likely, each of these p_{ij} has the value ${r_i}^{-1}$ and for other values of j, p_{ij} is zero. The matrix P^m contains entries $p_{ij}^{(m)}$ which are the probabilities of input $i \bmod \pi$ resulting in an output of $j \bmod \pi$ after m divisors are applied to k using the algorithm. Averaging over all i gives the probability of residue $j \bmod \pi$ after m iterations. This converges very quickly to a steady state in which the probabilities typically lie in a range between $\frac{3}{4}\pi^{-1}$ and $\frac{5}{4}\pi^{-1}$.

If p_j is the asymptotic probability for residue class $j \bmod \pi$ and c_j the cost of the pair (r_j, d_j) associated with that class, then the average cost per bit is

$$c = \frac{\sum_{j=0}^{\pi-1} p_j c_j}{\sum_{j=0}^{\pi-1} p_j \log_2(r_j)} \tag{5}$$

In fact, using a slightly more aggressive (smaller) value for c in (4) seems to yield the best value for c in (5). Our overall objective is, of course, to minimise this.

The transition matrix is usually too big for practical calculations. Instead, a Monte Carlo simulation can be performed. A large number of random keys k is generated, written in base π for convenience, and recoded to find the probabilities. Since (4) may have failed to capture all the useful information in the associated residue modulo π, this process can be repeated on neighbouring schemes to identify the local minimum for c: for each residue $i \bmod \pi$ alternative pairs (r_i, d_i) can be tried to improve c. The same process can also be used to evaluate potential simplifications to the scheme.

7 Test Results and Evaluation

Table 1 shows figures comparing the above algorithm with the best earlier results as provided by the refined NAF method of Longa and Gebotys in [17], Table 4. It illustrates the speed-ups for three sets of projective coordinate systems. These are standard Jacobian coordinates (with $a = -3$), inverted Edwards coordinates [7,1] and extended coordinates on the Jacobi Quartic form [2,13]. The cost per operator columns give the number of field multiplications or equivalents (counting 0.8 for each squaring) which are used for doubling (D), tripling (T), quintupling (Q) and the addition of a table entry (A) within a double-add, triple-add etc. operation in the given coordinate system. The stored points were the first few of 1, 3, 5, 7,... times the initial point. The number of them is recorded in the third column group, and the cost of initialising them is included in the operator count totals. In order to make scaling to other key lengths easier, an initialisation optimisation [17,8,16] which saves 10 to 12 field multiplications has not been applied to either set of results. The final column is the asymptotic number of field multiplications (or equivalents) per bit of k, calculated as in §6.2. This is the value of c in (5).

The methodology here performs a systematic search at design time to determine the best run-time choice. This choice is wider than that in [17] because, for example, 2 is not automatically the divisor when k is even (see Appendix B for an example), bases which divide k exactly need not be chosen as divisors,

Table 1. Comparative Point Multiplication Speeds for 160-bit Scalar (*cf* [17], Table 4)

Method	Coords	Mult Cost/Op				$\mathcal{B}$	#Stored	π	Op. Count	Cost/Bit
		D	T	Q	A		Pts		Sq = 0.8M	c
[17]	JQuart	6.0	11.2		8.4	{2,3}	8		1229.2	7.50
”	”	”	”		”	”	1		1288.4	8.10
”	InvEdw	6.2	12.2		8.8	”	8		1277.1	7.79
”	Jacobian	7.0	12.6		10.2	”	8		1445.1	8.91
Here	JQuart	6.0	11.2		8.4	{2,3}	8	$2^{13}3^3$	1207.1	7.32
”	”	”	”		”	”	4	$2^{13}3^3$	1212.0	7.52
”	”	”	”		”	”	2	$2^{17}3^4$	1253.8	7.84
”	”	”	”		”	”	1	$2^{17}3^4$	1287.6	8.10
”	InvEd	6.2	12.2		8.8	”	8	$2^{13}3^3$	1242.5	7.55
”	Jacobian	7.0	12.6		10.2	”	8	$2^{13}3^3$	1414.9	8.57
[17]	JQuart	6.0	11.2	17.2	8.4	{2,3,5}	8		1226.5	7.48
”	Jacobian	7.0	12.6	19.6	10.2	”	8		1435.2	8.84
Here	JQuart	6.0	11.2	17.2	8.4	{2,3,5}	8	$2^9 3^2 5^2$	1203.7	7.30
”	Jacobian	7.0	12.6	19.6	10.2	”	8	$2^9 3^2 5^2$	1409.8	8.53

and divisors other than 2 can have non-zero digits associated with them. All this automatically leads to the possibility of faster scalar multiplication. However, the tabulated results here represent a less deep search – a larger value for π is necessary to uncover all the conditions listed in [17], Tables 2 and 3. Further computation could have been used to obtain them. There is no clear cost function in [17] but an explicit cost function has been given above in (4). This is not the obvious one (which is sub-optimal) but arguments have been presented to justify the conclusion that it is better. This difference probably also contributes to the greater speeds here, which are asymptotically 2.5% faster than [17].

The best value for money seems to occur with tables of 4 points, being under 1% slower than their 8 point counterparts. For the fastest times, observe that the Jacobi Quartic coordinate representation [13] can be used to perform 160-bit scalar multiplication with divisors 2 and 3 and a table of 8 pre-computed points using, on average, the equivalent of fewer than 1200 field multiplications when the optimisations of [8,16] are applied. This can be achieved with very straightforward code, which is given in the first example of Appendix B.

8 Conclusion

A detailed methodology has been presented for deriving compact, device-specific algorithms which perform very efficient scalar multiplication for elliptic curves in modestly resource constrained environments when the scalar is an unseen fresh, random, cryptographic key. The method is based on multibase representations of the scalar and careful construction of a cost function for evaluating alternatives. A valuable output demonstrating the efficacy of the approach is the fastest known algorithm in certain contexts, such as when composite curve operations are available. Another output is a couple of examples which illustrate the compactness of resulting code for the iterated recoding step.

With an initial recoding cost equivalent to two or three field multiplications, the exponentiation algorithm is efficient and avoids an expensive run-time search by resolving the search questions at design time. Part of the speed-up is derived from a much fuller choice of base/digit pairs at each step than prior algorithms and part from a new, explicit device-specific cost function for evaluating different options in the search for the most efficient addition chain. That function can, and should moreover, use detailed knowledge of the times for various field operations. Exactly the same methods can be applied with more extensive preparatory computation to generate still faster or compact algorithms than tabulated here.

References

1. Bernstein, D., Lange, T.: Analysis and Optimization of Elliptic-Curve Single-Scalar Multiplication. Cryptology ePrint Archive, Report 2007/455, IACR 2007 (2007)
2. Billet, O., Joye, M.: The Jacobi Model of an Elliptic Curve and Side-Channel Analysis. In: Fossorier, M.P.C., Høholdt, T., Poli, A. (eds.) AAECC 2003. LNCS, vol. 2643, pp. 34–42. Springer, Heidelberg (2003)
3. Ciet, M., Joye, M., Lauter, K., Montgomery, P.: Trading Inversions for Multiplications in Elliptic Curve Cryptography. Designs, Codes and Cryptography 39(2), 189–206 (2006)
4. Dimitrov, V., Cooklev, T.: Two Algorithms for Modular Exponentiation using Non-Standard Arithmetics. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E78-A(1), 82–87 (1995)
5. Dimitrov, V.S., Jullien, G.A., Miller, W.C.: Theory and Applications for a Double-Base Number System. In: Proc. 13th IEEE Symposium on Computer Arithmetic, Monterey, July 6-9, pp. 44–51. IEEE, Los Alamitos (1997)
6. Dimitrov, V.S., Imbert, L., Mishra, P.K.: Efficient and Secure Elliptic Curve Point Multiplication using Double-Base Chains. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 59–78. Springer, Heidelberg (2005)
7. Edwards, H.: A Normal Form for Elliptic Curves. Bull. Amer. Math. Soc. 44, 393–422 (2007)
8. Elmegaard-Fessel, L.: Efficient Scalar Multiplication and Security against Power Analysis in Cryptosystems based on the NIST Elliptic Curves over Prime Fields, Masters Thesis, University of Copenhagen (2006)
9. Fouque, P.-A., Valette, F.: The Doubling Attack – Why upwards is better than downwards. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 269–280. Springer, Heidelberg (2003)
10. Doche, C., Icart, T., Kohel, D.R.: Efficient Scalar Multiplication by Isogeny Decompositions. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T.G. (eds.) PKC 2006. LNCS, vol. 3958, pp. 191–206. Springer, Heidelberg (2006)
11. Giessmann, E.-G.: Ein schneller Algorithmus zur Punktevervielfachung, der gegen Seitenanalattacken resistent ist. In: Workshop über Theoretische und praktische Aspekte von Kryptographie mit Elliptischen Kurven, Berlin (2001)
12. Gordon, D.M.: A Survey of Fast Exponentiation Algorithms. Journal of Algorithms 27, 129–146 (1998)
13. Hisil, H., Wong, K., Carter, G., Dawson, E.: Faster Group Operations on Elliptic Curves. Cryptology ePrint Archive, Report 2007/441, IACR (2007)

14. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Seminumerical Algorithms, vol. 2, §4.6.3, pp. 441–466. Addison-Wesley, Reading (1981)
15. Longa, P.: Accelerating the Scalar Multiplication on Elliptic Curve Cryptosystems over Prime Fields, Masters Thesis, University of Ottawa (2007)
16. Longa, P., Miri, A.: New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields. In: Cramer, R. (ed.) PKC 2008. LNCS, vol. 4939, pp. 229–247. Springer, Heidelberg (2008)
17. Longa, P., Gebotys, C.: Fast Multibase Methods and Other Several Optimizations for Elliptic Curve Scalar Multiplication. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 443–462. Springer, Heidelberg (2009)
18. Mishra, P.K., Dimitrov, V.: Efficient Quintuple Formulas for Elliptic Curves and Efficient Scalar Multiplication using Multibase Number Representation. In: Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) ISC 2007. LNCS, vol. 4779, pp. 390–406. Springer, Heidelberg (2007)
19. Walter, C.D.: Exponentiation using Division Chains. In: Proc. 13th IEEE Symposium on Computer Arithmetic, Monterey, CA, July 6-9, pp. 92–98. IEEE, Los Alamitos (1997)
20. Walter, C.D.: Exponentiation using Division Chains. IEEE Transactions on Computers 47(7), 757–765 (1998)
21. Walter, C.D.: MIST: An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 53–66. Springer, Heidelberg (2002)
22. Walter, C.D.: Some Security Aspects of the MIST Randomized Exponentiation Algorithm. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 276–290. Springer, Heidelberg (2003)
23. Yao, A.C.-C.: On the Evaluation of Powers. SIAM J. Comput. 5(1), 100–103 (1976)

Appendix A. Pseudo-Code for the Recoding Scheme

The algorithm in Fig. 1 determines the base/digit pairs to associate with each residue i mod π of the key k during generation of the multibase representation. Its inputs include a choice of base set $\mathcal{B}$ and digit set $\mathcal{D}$ suitable for the target device, the window width λ and the value of π (the λth power of the lcm of bases in $\mathcal{B}$ times a power of 2 determined by the table size). The output is deposited in two arrays, `Base` and `Digit`. The function *Cost* is that in (4), and it is described in detail in §6.1.

The algorithm in Fig. 2 uses the arrays `Base` and `Digit` to recode the key k, storing the resulting multibase representation in an array `MB`. Usually the array accesses in the loop body will be replaced by coded rules as in Appendix B.

Appendix B. Code for the Iterative Step

The very simple code of Figure 3 provides mod $2^6 3^2$ rules for selecting the next divisor/ residue pair (r, d) for k with base set $\mathcal{B} = \{2, 3\}$ and a precomputed table of 8 points containing 1, 3, 5,..., 15 times the input point. Once the pair (r, d) is chosen, k is then reduced to $(k-d)/r$ for the next application of the rules. The rules were obtained using a sub-chain length of $\lambda = 2$ with $c = 7.328$

```
For i: 0 ≤ i < π do
{  BestCost ← MaxCost
   For all r1 in 𝓑 and all d1 in 𝓓 do
   If (i-d1) mod r1 = 0 then
   {  i1 = (i-d1)/r1
      For all r2 in 𝓑 and all d2 in 𝓓 do
      If (i1-d2) mod r2 = 0 then
      {  ...
         If Cost((r1,d1),(r2,d2),...,(rλ,dλ)) < BestCost then
         {  BestCost ←  Cost((r1,d1),(r2,d2),...,(rλ,dλ))
            Base[i] ← r1
            Digit[i] ← d1
         }
      }
   }
}
```

Fig. 1. Algorithm to determine Base/Digit pairs for each Residue mod π

```
n ← 0
While k ≠ 0 do
{  i ← k mod π
   MB[n].r ← Base[i]
   MB[n].d ← Digit[i]
   k ← (k-Digit[i])/Base[i]
   n ← n+1
}
MB.len ← n
```

Fig. 2. Algorithm to recode k to a Division Chain Representation

```
If k = 0 mod 9 and k ≠ 0 mod 4 then
         r ← 3, d ← 0
else if k = 0 mod 2 then
         r ← 2, d ← 0
else if k = 0 mod 3 and 18 < (k mod 64) < 46
     and ((k mod 64)-32) ≠ 0 mod 3 then
         r ← 3, d ← 0
else     r ← 2, d ← ((k+16) mod 32) - 16.
```

Fig. 3. A Choice of Mod $2^6 3^2$ Recoding Rules for a Table Size of 8

in (4) and the Jacobi Quartic form costs of the composite operations as listed in Table 1. This generated the same value for c in (5). Increasing λ by 1 only saves 1 more multiplication per 160-bit scalar.

Similar conditions are given in [17], Tables 2 and 3. The main difference here is that some multiples of 2 are assigned the divisor 3 rather than the 2 forced by [17]. This enables the next residue to be modified by a large table entry to make it much more highly divisible by 2. Note, however, that divisor 3 is not

```
If k = 0 mod 9 and k ≠ 0 mod 4
and (16 < (k mod 256) < 240) then
        r ← 3, d ← 0
else if k = 0 mod 2 then
        r ← 2, d ← 0
else if k = 0 mod 3 and 8 < (k mod 32) < 24
     and ((k mod 32)-16) ≠ 0 mod 3 then
        r ← 3, d ← 0
else    r ← 2, d ← ((k+8) mod 16) - 8.
```

Fig. 4. A Choice of Mod $2^8 3^2$ Recoding Rules for a Table Size of 4

used with a non-zero digit although the generating algorithm could have allowed it. Including the pre-computations for table entries, the total field multiplication count for 160-bit scalars is just under 1208 (at 1 squaring = 0.8 multiplications). This was obtained by a Monte Carlo simulation. The optimisations referred to in [17,16] enable a further 10-12 multiplications to be saved during initialisation using [8], §4.2.2, thereby taking the total to under 1200.

In the same context as the above except for having a pre-computed table of four instead of eight points, the code in Fig. 4 works with $\pi = 2^8 3^2$ and generates a recoding which requires 1214.2 field multiplications on average. This is over 22.7 multiplications better than code with no triplings.