A Duality in Space Usage between Left-to-Right and Right-to-Left Exponentiation

Colin D. Walter

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

Abstract. Most exponentiation algorithms are categorised as being left-to-right or right-to-left because of the order in which they use the digits of the exponent. There is clear value in having a canonical way of transforming an algorithm in one direction into an algorithm in the opposite direction: it may lead to new algorithms, different implementations of existing algorithms, improved side-channel resistance, greater insights. There is already an historic duality between left-to-right and right-to-left exponentiation algorithms which shows they take essentially the same time, but it does not treat the space issues that are always so critical in resource constrained embedded crypto-systems. To address this, here is presented a canonical duality which preserves both time and space. As an example, this is applied to derive a new, fast yet compact, left-to-right algorithm which makes optimal use of recently developed composite elliptic curve operations.

Key Words. Scalar multiplication, multi-base representation, addition chain, division chain, dual chain, exponentiation, elliptic curve cryptography.

1 Introduction

Exponentiation is the highest level arithmetic operation in all the most popular public key crypto-systems, and in Diffie-Hellman, RSA and ECC in particular. There are a number of different algorithms for performing exponentiation [8, 7] which have various properties that allow some control over their time efficiency, their use of space resources and their susceptibility to side channel analysis.

The ability to choose between processing an exponent from left to right or from right to left enables implementers to improve side channel resistance (e.g. by avoiding pre-computed tables [13]) or to make use of more efficient composite group operations such as double-and-add, triple-and-add and quintuple-and-add elliptic curve operations [6, 10, 9]. The direction of treating the exponent bytes may also be determined by the order in which those bytes become available.

These reasons make it of interest to find a canonical way of restructuring an exponentiation algorithm so that it can process the exponent in the opposite direction. An example of what we would like to do in practice is given

[0] O. Dunkelman (Ed.): CT-RSA 2012, LNCS 7178, pp. 84–97, 2012.

by comparing the usual left-to-right m-ary algorithm due to Brauer [3] with Yao's right-to-left method [14]. These use the same time and space resources. Moreover, it is clear how to extend both to sliding window versions that make the same use of resources. In general, it would be useful to be able to take any exponentiation algorithm processing the bits in one direction and deduce immediately a corresponding algorithm for processing the bits in the opposite direction. Knuth in his well-known *Semi-numerical Algorithms* [8] describes the *transposition method*, [2] §5, which enables one to reverse the order of processing the exponent and compute the required power using the same time. With care, the same number of squarings and non-squarings occur in the two directions. However, this method does not show how to preserve the space requirements, nor does it provide a canonical re-ordering. Yet the preservation of space usage is of critical importance to achieve when memory is limited, as on a smart card or an embedded cryptographic device, as well as on SSL servers with systolic arrays for processing many exponentiations in parallel. Nevertheless, these space issues do not seem to have been treated satisfactorily in the literature.

The aim of the present work is too provide a canonical duality between the two directions which not only preserves the time usage of an exponentiation algorithm but also makes identical use of memory. This is done by starting with an addition chain which is annotated with the register locations for the inputs and output of each operation. A careful restriction on the allowable operations makes the chain reversible, so that the use of space is clearly the same in both directions. Some additional conditions are required to ensure that the numbers of squaring and non-squaring operations are also the same for the addition chain and its dual. The restriction on allowed operations is essentially just a requirement on the way the chain is presented, and so does not confine the applicability of the method. The additional conditions are natural ones in an efficient system and so are normally satisfied in a practical environment. The main novel contributions here are the establishment of this correct set of allowable operations to make duality possible, identification of the right conditions to preserve the time cost of an exponentiation when the dual is applied, and a proof of this property.

Application of the duality process shows that Brauer's m-ary method has Yao's method as its dual, and *vice versa*. Also, application to Walter's division chain method [11] yields a new compact left-to-right algorithm which can take maximal advantage of recently developed composite elliptic curve operations [6, 10, 9] because the recoding of the exponent can be tailored to the different relative costs of any desired combinations of squaring and non-squaring operations on the underlying group, namely the elliptic curve in this case.

Finally, having established that the main space and time requirements are the same for an exponentiation algorithm and its dual, there are some secondary space issues to tidy up. When the duality is applied to an addition chain derived from a recoding of the exponent, extra space may be required to store the complete recoding when processed in one direction, but for the other direction the recoding may be generated on-the-fly. One may also require the initial inputs (the base and/or the exponent) to remain at the end of the exponentiation. This may happen in one direction, whereas they may be overwritten in the other.

2 Notation & Addition Chains

The duality defined here applies to exponentiation schemes which are defined in terms of *addition* or *addition-subtraction chains* [8]. Most exponentiation algorithms first perform a re-coding of the exponent D, and then convert the re-coding into an addition chain which is applied to an element M of some group G to yield the element $C = M^D$. With cryptographic applications in mind, M will be called the *plaintext*, D the *(secret) key*, and C the *ciphertext*. G might be the group of points on an elliptic curve. It will be written multiplicatively so that the operation of interest is $C \leftarrow M^D$, which is reasonably called an *exponentiation*.

In order to obtain a good measure of the computational time for exponentiating, we will assume there are two (probably distinct) algorithms for performing the group operation. The first computes M^2 for any $M \in G$ and is called a *squaring*. When the two arguments of the group operation are known to be identical this algorithm will be used. The other algorithm computes $M_1 \times M_2$ for any $M_1, M_2 \in G$ and is called a (non-squaring) *multiplication*. This will be used whenever it is not possible to guarantee that $M_1 = M_2$. Normally $M_1 \neq M_2$ when this algorithm is applied, but it is possible that $M_1 = M_2$ could occur by chance. However, the same computational cost will be assumed for all applications of it. Lastly, there may also be a unary operation for computing the *inverse* M^{-1} of M, or, more generally, several unary operations $M \to M^s, s \in S$, for a small subset $S \subset \mathbb{Z}$ of integers. This enables us to deal with a Frobenius map as well as inversion. It may be convenient to include squaring in this category. The following definition picks up these distinctions:

Definition 1.
i) An addition chain *of* length n *for* D *is a sequence* $D_0, D_1, D_2, \ldots, D_n$ *of integers such that*

a) $D_0 = 1$ *and* $D_n = D$*;*
b) for all $k, 0<k\leq n$*, either there are* $i, j < k, i \neq j$*, such that* $D_i + D_j = D_k$ *or there is an* $i < k$ *such that* $2D_i = D_k$*.*

ii) A (generalised) addition-subtraction chain *of* length n *for* D *is a sequence* $D_0, D_1, D_2, \ldots, D_n$ *of integers such that*

a) $D_0 = 1$ *and* $D_n = D$*;*
b) for all $k, 0<k\leq n$*, either there are* $i, j < k, i \neq j$*, such that* $D_i + D_j = D_k$ *or there are* $i < k$ *and* $s \in S$ *such that* $sD_i = D_k$*.*

These translate into exponentiation schemes for D in the obvious way. The kth step in obtaining $M^D \in G$ is to compute $M^{D_k} = M^{D_i + D_j} = M^{D_i} \times M^{D_j}$ or $M^{D_k} = M^{sD_i} = (M^{D_i})^s$.

Memory locations for holding elements of G will, for convenience, be called *registers* and denoted $R_i, i \in I$, for some small index set I. i (or R_i) will be called a *location* of $g \in G$ if R_i stores the value of g. In practice, R_i could be any form of memory, perhaps different for each i so that the cost of reading from or writing to R_i may very well depend on the value of i. Such costs generally result in minor differences in execution times between an algorithm and its dual.

In general, for $i, j, k \in I$ the *mu*ltiplicative operation which writes the product of the contents of R_i and R_j into R_k is denoted μ_{ijk}, and the powering operation writing the sth power of the content of R_i into R_k is denoted $\iota_{ik}^{(s)}$ (choosing "ι" for *inverse* because often $s = -1$). It is clear that, once a location for each D_k in an addition or addition-subtraction chain is known, then the chain can be expressed as a sequence of operations of type μ_{ijk} or $\iota_{ik}^{(s)}$. However, to define the dual chain, only the following restricted sets of operators are allowed:

Definition 2. *For $i, j \in I$ with $i \neq j$ and $s \in S$, six sets of operators are defined:*
i) Copying from R_i to R_j is denoted γ_{ij}.
ii) Copying from R_i to R_j combined with initialising R_i to the group identity 1_G is denoted $\gamma_{ij}^{(0)}$.
iii) The multiplicative operation which writes the product of the contents of R_i and R_j into R_j is denoted μ_{ij}.
iv) The multiplicative operation which writes the product of R_i and R_j into R_j and initialises R_i to 1_G is denoted $\mu_{ij}^{(0)}$.
v) The operation which raises the contents of R_i to the power s is denoted $\iota_i^{(s)}$.
vi) The operation swapping the contents of registers R_i and R_j is denoted σ_{ij}.
A location-aware chain *is a finite sequence of such operations.* □

Location-aware chains will also be called *space-aware* chains, especially where the overall space usage rather than individual data movements are of concern.

Any μ_{ijk} or $\iota_{ij}^{(s)}$ can be expressed using a sequence of either one or two of the above operations with no increase in the number or type of multiplicative operations. For example, if $i \neq k \neq j$ then R_j can be first copied to R_k using γ_{jk} and then μ_{ik} completes the process of computing μ_{ijk}. Similarly, any squaring μ_{iik} can be expressed by first using the copy γ_{ik} if $i \neq k$ and then the powering operation $\iota_k^{(2)}$, thereby making all squaring explicit. If required at execution time, the two operations from such splittings can always be recombined into one when deciding the code to execute. Also at execution time many of the initialisations to 1_G in $\gamma_{ij}^{(0)}$ and $\mu_{ij}^{(0)}$ might be skipped as they are mostly redundant.

The swapping operation enables it to be made explicit when data is moved around. It is included for completeness as it may be needed in implementations to put data in a particular location without loosing the data which is already in that location. Later conditions require this (for the *symmetric* property), but data must also be moved around if only certain locations (such as actual registers) can be used for the I/O of an operation. However, from here onwards, and without loss of generality, *swapping* will be ignored in any proofs since it is irrelevant to them and simply complicates the description of where data is.

3 The Dual of a Location-Aware Chain

The operations in Defn. 2 can be represented using matrices, indexed by I. For example, if $A = (a_{st})$ were the matrix for $\mu_{ij}, i \neq j$, then $a_{ss} = 1$ for $s \in I$,

$a_{ji} = 1$, and $a_{st} = 0$ otherwise. It is the identity matrix except for an extra non-zero entry at (j, i). This acts from the left on a column vector containing the exponents of the powers of the input M which are in each register, adding the values with indices i and j into the location with index j. In other words, the matrix performs the same addition as an element of an addition chain.

For a device with two memory locations, i.e. $|I| = 2$, matrix examples of each class are, respectively,

$$\gamma_{12} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \quad \gamma_{12}^{(0)} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad \mu_{21} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad \mu_{21}^{(0)} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix},$$
$$\iota_1^{(s)} = \begin{bmatrix} s & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{and} \quad \sigma_{12} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

This view enables the *transpose* of each operator to be defined to coincide with the transpose of its matrix:

Definition 3. *The* transposes *of the operators in Definition 2 are as follows:*

$$\gamma_{ij}^{\mathsf{T}} = \mu_{ji}^{(0)},\ \gamma_{ij}^{(0)\mathsf{T}} = \gamma_{ji}^{(0)},\ \mu_{ij}^{\mathsf{T}} = \mu_{ji},\ \mu_{ij}^{(0)\mathsf{T}} = \gamma_{ji},\ \iota_i^{(s)\mathsf{T}} = \iota_i^{(s)} \text{ and } \sigma_{ij}^{\mathsf{T}} = \sigma_{ij}.$$

Clearly the transpose operator $^{\mathsf{T}}$ is a bijection of order two on the set of operations listed in Definition 2. In greater detail, it is the identity on elements listed in parts (v) and (vi), a bijection on the subsets of parts (ii) and (iii), and a bijection between the elements of parts (i) and (iv). Hence the transpose of a list of such operations will also be a list of such operations, so that the following concept of a *dual chain* is well-defined:

Definition 4. *The* dual *of a location-aware chain* $\rho = (\rho_1, \rho_2, \rho_3, \ldots, \rho_n)$ *is the location-aware chain* $\rho^{\mathsf{T}} = (\rho_n^{\mathsf{T}}, \ldots, \rho_3^{\mathsf{T}}, \rho_2^{\mathsf{T}}, \rho_1^{\mathsf{T}})$.

The foregoing observations imply that the number and type of the powering operations, such as squarings and inversions, is the same for a chain and its dual. Also, the number of multiplications without initialisation, the number of copyings with initialisation, and the number of swappings are all preserved under application of the dual map. However, the number of multiplications with initialisation and the number of copyings without initialisation are interchanged by the dual. Additional conditions are required to make these numbers equal so that the cost of a space aware chain, in terms of the counts of each type of operation, is unchanged when the dual is taken.

Before tackling these conditions, let us reflect on the choice of operations in Definition 2. The general operations $\mu_{ijk}, i \neq k \neq j$, were omitted because their transposes are too complicated for a sensible definition of a dual chain. However, the remaining cases of multiplicative operations, namely $\mu_{ij}, i \neq j$, are not powerful enough to enable all the required operations to be done. As a result, the copying operations γ_{ij} need to be included. The need for closure under transpose results in the inclusion of multiplications with initialisation. Lastly, the copying with initialisation arises naturally from the conditions in §4.

3.1 Example

In this example, the notation is illustrated by computing M^{15} using two registers, and starting with the addition chain $(1, 2, 3, 6, 12, 15)$. The construction of the chain under Defn. 1 is usually given explicitly in the form

$$1+1=2,\ 1+2=3,\ 3+3=6,\ 6+6=12,\ 12+3=15.$$

but the three doublings can be exhibited by writing it as

$$2{\times}1=2,\ 1+2=3,\ 2{\times}3=6,\ 2{\times}6=12,\ 12+3=15.$$

The corresponding computation with M is

$$(M^1)^2=M^2;\ M^1{\times}M^2=M^3;\ (M^3)^2=M^6;\ (M^6)^2=M^{12};\ M^{12}{\times}M^3=M^{15}$$

which contains 3 squarings and 2 multiplications.

A minimum of 2 storage locations is required for this, say R_1 and R_2. Suppose that only R_1 may be used for input and output. So it is assumed to hold M after initialisation, and should contain the final value M^{15} at the end of the calculation. Using a vector to give the values in the registers, the computation starts with $(M, \perp)$ in (R_1, R_2) where $\perp$ denotes an undefined or unknown value. As M is still needed after it is squared, it must first be copied: γ_{12} yields values (M, M). Then application of $\iota_2^{(2)}$ creates (M, M^2) and $\mu_{21}^{(0)}$ yields $(M^3, 1_G)$. Here the 1_G is created by the superscript $^{(0)}$, and used to overwrite the M^2 as it is no longer required. This is a feature of the chains of interest that is introduced in the next section in order to obtain a dual of equal computational effort. Of course, the computationally unnecessary, and essentially free, initialisation to 1_G would probably be skipped in practice. Using γ_{12} to create (M^3, M^3) means that M^3 is not lost when the next squaring, $\iota_2^{(2)}$ generates (M^3, M^6). Repeating $\iota_2^{(2)}$ produces (M^3, M^{12}) so that the final multiplication $\mu_{21}^{(0)}$ achieves $(M^{15}, 1_G)$. This has the desired power M^{15} in the desired location R_1 at the end of the calculation. It has also eliminated the unwanted data from R_2, which is again a requirement described in the next section for the chains of interest. Summarising, the sequence of operations and register contents is thus

$$\binom{M}{\perp} \xrightarrow{\gamma_{12}} \binom{M}{M} \xrightarrow{\iota_2^{(2)}} \binom{M}{M^2} \xrightarrow{\mu_{21}^{(0)}} \binom{M^3}{1_G} \xrightarrow{\gamma_{12}} \binom{M^3}{M^3} \xrightarrow{\iota_2^{(2)}} \binom{M^3}{M^6} \xrightarrow{\iota_2^{(2)}} \binom{M^3}{M^{12}} \xrightarrow{\mu_{21}^{(0)}} \binom{M^{15}}{1_G}$$

The transposes of the operations $\gamma_{12}, \iota_2^{(2)}, \mu_{21}^{(0)}, \gamma_{12}, \iota_2^{(2)}, \iota_2^{(2)}, \mu_{21}^{(0)}$ are, in order, $\mu_{21}^{(0)}, \iota_2^{(2)}, \gamma_{12}, \mu_{21}^{(0)}, \iota_2^{(2)}, \iota_2^{(2)}, \gamma_{12}$. Reversing the order yields the dual chain, which acts on the registers thus:

$$\binom{M}{\perp} \xrightarrow{\gamma_{12}} \binom{M}{M} \xrightarrow{\iota_2^{(2)}} \binom{M}{M^2} \xrightarrow{\iota_2^{(2)}} \binom{M}{M^4} \xrightarrow{\mu_{21}^{(0)}} \binom{M^5}{1_G} \xrightarrow{\gamma_{12}} \binom{M^5}{M^5} \xrightarrow{\iota_2^{(2)}} \binom{M^5}{M^{10}} \xrightarrow{\mu_{21}^{(0)}} \binom{M^{15}}{1_G}$$

It corresponds to the different addition chain $(1, 2, 4, 5, 10, 15)$. In fact, at a higher level, the dual of the computation $M \to M^3 \to M^{3\times5}$ is $M \to M^5 \to M^{3\times5}$.

4 Preserving the Number of Multiplications

A standard measure of the time taken by an exponentiation is given by the following *cost* associated with the underlying addition chain. Once the execution time for each type of operation is known, the corresponding weighted sum of the entries in the cost tuple will yield the total time for exponentiation.

Definition 5. *The* cost *of a location-aware chain is the tuple consisting of the numbers of each type of operation in the chain, as classified in Definition 2 and refined to separate the counts of the unary operations in part (v) according to the value of $s \in S$.*

Thus, in particular, the cost of a chain yields separately the numbers of copyings, non-squaring multiplications, squarings and inversions, the first two being divided into two parts according to whether the operation includes an initialisation to 1_G or not. (Finer time measurements may be required [1].) In order to preserve cost when taking the dual of a chain, some extra conditions are required:

Definition 6. *A location-aware chain is said to be* normalised *if it satisfies the following criteria:*
i) There is a prescribed subset of registers, indexed by $I_{IO} \subseteq I$, say, which is used for I/O[1]*.*
ii) The inputs to every operation and the final value in any output register must be defined, i.e. no operation output or final output depends on the initial value of any non-input register.
iii) The initial value of an input register and the output from every operation in the chain must be used, i.e. every operation output is the input to a subsequent multiplicative operation or is the final value in an output register.
iv) 1_G is never explicitly the input to any operation nor explicitly the final value of an output register.
v) If an operation involving two registers does not include an initialisation to 1_G then the value remaining in the non-result register must be used by a subsequent multiplicative operation or be the final value in an output register.

The conditions (*ii*)–(*v*) actually specify which registers are for input and output:

- Registers R_J with $J \in I_{IO}$ will have their initial values used by the chain and their final values must be defined and not explicitly set to 1_G.
- Initial values in registers $R_J, J \in I \setminus I_{IO}$, must not be used, and final values in these registers must be removed by initialising them to 1_G.

Thus all the I/O registers will both import values and export values, but none of the non-I/O registers will either import values or export values. Adding copying operations, with initialisation if necessary, at the end of a chain enables any outputs of the chain to be via the same set of registers as is used for inputs. So this condition mainly imposes a requirement for there to be the same number of

[1] All that is needed is to have the same *number* of input registers as output registers rather than the same subset for both. The restriction here is reasonable for hardware.

outputs as inputs. There need not be just one input – the chain could perform a multi-exponentiation.

Part (iii) means there are no redundant operations. This can be achieved from any space-aware chain simply by deleting operations whose values are not used, i.e. those operations whose output is neither an input to a subsequent operation nor an output of the chain. Although redundant, it is allowed to have registers which do not figure in any operations.

Property (iv) means, for example, that neither of the inputs to any multiplication has been set to 1_G as a result of a previous operation with an initialisation of one of the two named registers. Similarly, the input to a copy or powering operation should not have been set to 1_G by a preceding operation. This does not impose any real restriction on allowable chains. The unary operations with 1_G as an input have 1_G as an output and so can be deleted from the chain without affecting the final output(s). A multiplication with 1_G as an input has an output equal to the other input and so it can be replaced by a copy or deleted entirely according to whether the output is to the register that contains 1_G or not. A copy of 1_G can be removed, and an initialisation attached instead to the previous operation that used the value in the register that needs to be set to 1_G.

Condition (v) is easily achieved in any chain by modifying every operation to include an initialisation whenever it makes no difference to the computations performed or the values exported. For any operation involving two registers but with no initialisation, it means that the values in both the named registers will be used by subsequent operations or exported. Unary operations, i.e. powering operations, do not have registers affected by this rule.

Finally, the definition really assumes there are no swapping operations involved. If there is any swapping, then, in the obvious way, the old and new locations of the value in a register need to be taken into account when deciding whether a value is used or exported or has been initialised to 1_G or etc.

The example in section §3.1 is of a normalised chain, as is easily checked. The I/O subset of $I = \{1, 2\}$ is $I_{IO} = \{1\}$. It is clear that all non-trivial intermediate register values are used; the unused intermediate values have all been deliberately set to 1_G and are eventually overwritten. The dual chain is also clearly a normalised chain. Both chains in this example have the same costs: there are three squarings, two copyings, and two multiplications with initialisations.

Theorem 1. *The cost of a normalised location-aware chain is unchanged by taking the dual.*

Proof. For simplicity, and without loss of generality, assume there are no unary operations (i.e. those covered by Defn. 2(v)) and no swapping operations. It has already been observed that the numbers of such operations are not changed by taking the dual, nor are the numbers of copyings with initialisation and multiplications without initialisation.

The proof works by counting the number of instances of 1_G or $\perp$ (undefined) occurring in registers, and equating this 1) to the number of operations that create them and 2) to the number of operations that destroy them. So let γ

be the number of copyings without initialisation, γ_I the number of copyings with initialisation to 1_G and μ_I the number of multiplications with initialisation to 1_G. It will be shown that $\gamma = \mu_I$, from which the theorem follows almost immediately. (In the example of §3.1, there are 2 instances of 1_G, 1 of $\perp$, and $\gamma_I = 0$, $\gamma = 2$, $\mu_I = 2$. Equating the numbers yields $1+\mu_I+\gamma_I = 3 = 1+\gamma+\gamma_I$, so that $\gamma = \mu_I$ $(= 2)$.)

Suppose an arbitrary, fixed register R contains 1_G or $\perp$ at a given time. Then, because the chain is normalised,

- The previous operation naming R, if any, initialised it to 1_G;
- The next operation naming R, if any, must be a copy into R;
- If R is for I/O, there is always a previous and a next such operation;
- If R is not for I/O, the first such value has no preceding operation naming R and the final one no such subsequent operation.

Of course, the 1_Gs which occur are in one-to-one correspondence with the chain operations which include an initialisation, each being associated with the operation which created it. Copyings can only overwrite 1_G or $\perp$, and so there is a one-to-one correspondence between copyings (with or without an initialisation) and any instance of $\perp$ or non-final instances of 1_G, each being associated with the copying which destroys it. There is only an instance of $\perp$ if R is not for I/O, and then only one. A final instance of 1_G means one which remains in the register at the end of executing the chain. There can only be one such instance, and it only occurs for a non-IO register. So there is the same number of instances of $\perp$ as number of instances of a final 1_G. Thus the number of times register R is explicitly initialised to 1_G is equal to the number of occurrences of 1_G in R during the execution of the chain, and this in turn equals the number of copyings (with or without initialisation) into R. Summing over all R, $\mu_I+\gamma_I = \gamma+\gamma_I$. So $\mu_I = \gamma$. Since copyings without initialisation become multiplications with initialisation and *vice versa* when the dual is taken, and these numbers are the same, the numbers of them are not changed when the dual is applied. □

5 Preserving the Chain Output under Duality

The action of a space-aware chain $\rho = (\rho_1, \rho_2, \rho_3, \ldots, \rho_n)$ is given by the composition $\nu(\rho) = \rho_n \circ \ldots \circ \rho_3 \circ \rho_2 \circ \rho_1$ of its elements. For a specific chain, this would be calculated as the matrix product, say M_ρ, of the representatives for each operation. The dual chain has action given by the transpose $\nu(\rho^{\mathsf{T}}) = \rho_1^{\mathsf{T}} \circ \rho_2^{\mathsf{T}} \circ \rho_3^{\mathsf{T}} \circ \ldots \circ \rho_n^{\mathsf{T}}$.

Definition 7. *A location-aware chain ρ is* symmetric *if $\nu(\rho^{\mathsf{T}}) = \nu(\rho)$.*

In other words, a symmetric chain is one such that the dual computes the same output. Its matrix is symmetric because the dual is represented by the transpose matrix.

Lemma 1.
i) With the above notation for a location-aware chain ρ, $M_\rho^{\mathsf{T}} = M_{\rho^{\mathsf{T}}}$.
ii) The chain ρ is symmetric if, and only if, its matrix M_ρ is symmetric.

This gives a criterion for checking whether or not the dual chain will compute the same value(s): it must be symmetric. In the case of a normalised chain, $M_\rho = (m_{ij})$ has $m_{ij} = 0$ if i is the index of a non-I/O register. This is because 1_G is the final value left in register R_i by ρ. Similarly, $m_{ij} = 0$ if j is the index of a non-I/O register. This is because the initial value in register R_i prior to applying ρ is not used by ρ. Hence the action of ρ is entirely described by the sub-matrix of elements indexed by I_{IO}. Consequently,

Theorem 2. *If a normalised, location-aware chain has only one I/O register, then its dual computes the same value.*

Thus, unless we are performing multi-exponentiations, a normalised chain and its dual will certainly output the same values from a given input.

6 Mixed Base Representations

Most exponentiation algorithms start by performing a recoding of the exponent D (normally from binary) into some variety of the *mixed base* form [4, 11]:

$$D = ((d_{n-1}r_{n-2}+d_{n-2})r_{n-3}+\ldots+d_1)r_0+d_0 \quad \text{with} \quad (r_i, d_i) \in \mathcal{R}\times\mathcal{D} \qquad (1)$$

where $\mathcal{R}$ is a set of allowed *radices*, e.g. $\mathcal{R} = \{2, 4\}$ or $\mathcal{R} = \{2, 3, 5\}$; $\mathcal{D}$ is a set of possible *digits*, such as $\mathcal{D} = \{0, 1, 2, 3, 4\}$, $\mathcal{D} = \{0, 1, 3\}$ or $\mathcal{D} = \{0, \pm1, \pm2\}$; and there are some *rules* on the allowable choices for radix/digit pairs (r_i, d_i), such as a pair of consecutive digits having to include at least one 0 (as in NAF). In general, these representations can be generated by the usual change-of-base algorithm modified to vary the base choice as necessary at each step. As an example, $235_{10} = (((((1)3 + 0)2 + 1)5 + 4)2 + 0)3 + 1 = 1_2 0_3 1_2 4_5 0_2 1_3$. A typical step in generating this is to choose base 3 for 235, obtain the (least significant) digit 1_3 as 235 mod 3 and repeat the process on $(235 - 1)/3 = 78$.

Inputs: $M \in G$, $D = ((d_{n-1}r_{n-2}+d_{n-2})r_{n-3}+\ldots+d_1)r_0+d_0 \in \mathbb{N}$ where $d_i \in \mathcal{D}$
Output: $M^D \in G$

---

read $P \leftarrow M$
Initialisation: $T[d] \leftarrow P^d$ **for all** $d \neq 0$
$P \leftarrow 1_G$
for $i \leftarrow n{-}1$ **downto** 0 **do** {
  if $i \neq n{-}1$ **then** $P \leftarrow P^{r_i}$
  if $d_i \neq 0$ **then** $P \leftarrow P{\times}T[d_i]$ }
Finalisation: $T[d] \leftarrow 1_G$ **for all** $d \neq 0$
return P

read $P \leftarrow M$
Initialisation: $T[d] \leftarrow 1_G$ **for all** $d \neq 0$
for $i \leftarrow 0$ **to** $n{-}1$ **do** {
  if $d_i \neq 0$ **then** $T[d_i] \leftarrow T[d_i]{\times}P$
  if $i \neq n{-}1$ **then** $P \leftarrow P^{r_i}$ }
Finalisation: $P \leftarrow \prod_{d\neq0} T[d]^d$
$T[d] \leftarrow 1_G$ **for all** $d \neq 0$
return P

Fig. 1. Left-to-Right (left) and Right-to-Left (right) Table-based Exponentiation.

The recoding enables the exponentiation to be simplified into a sequence of easy steps which process the digits from left to right or right to left. Those steps are converted into a space-aware addition chain when implemented. Normalised, but high level, versions of Brauer's m-ary scheme [3] and the scheme of Yao [14] are illustrated in Figure 1. The first line reads the plaintext input M into the only I/O register, namely P, and the last line writes the resulting ciphertext M^D from that register. The non-I/O registers named $T[d], d \in \mathcal{D} \setminus \{0\}$, are initialised before use in the second line, and reduced to a final 1_G in the second last line. The second last line is included to meet the I/O conditions of being normalised; it could be omitted, but is good for security.

Considering just the two registers P and $T[d_i]$ for a fixed i, the matrix corresponding to the addition chain which performs the loop iteration of index i is $\begin{bmatrix} r_i & 1 \\ 0 & 1 \end{bmatrix}$ for the left-to-right version and its transpose, $\begin{bmatrix} r_i & 0 \\ 1 & 1 \end{bmatrix}$ for the right-to-left version. Using Defn. 4, this shows that the composite operations corresponding to the loop bodies are duals of each other if they are written out in corresponding ways using the atomic operations of Defn. 2 — the sequence for one composite operation is transposed to give the sequence for the other.

The combination of the initialisation of table T and setting of P to 1_G in the left-to-right case has a matrix representation indexed by P and the $T[d]$, and it is entirely zero except that the Pth column contains 0 in row P and d in the row for $T[d]$. Its transpose is a matrix with only one non-zero row, namely that of index P and with d in the column for $T[d]$, which is clearly the matrix required to achieve the product which is assigned to P after the loop in the right-to-left case. Thus these two parts of the algorithms are also dual when defined suitably in terms of the atomic operations. Lastly, the finalisation line of the left-to-right case and the initialisation line of the right-to-left case are dual, because both are given by the symmetric matrix, indexed by P and the $T[d]$, which is all zeros except for a 1 as the diagonal entry of index P. This completes a proof that the formulations of the algorithms given in Fig. 1 are, in fact, dual because, in the correspondence, the totality of composite operations in one is reversed and transposed to give the operations of the other.

Strictly speaking, the definition of duality has been extended above to algorithms which are described at the level of composite operations on registers rather than the atomic ones of Defn. 2. However, as in Fig. 1, algorithms are often presented at such a level. This presentation usually has to satisfy the I/O requirements of normalised form in order that the transpose of the initialisation stage of one algorithm yields the finalisation step of the other. This was done for Fig. 1. As composite operations can always be decomposed into segements of a normalised location-aware chain, this extended definition of duality between algorithms means there is an underlying duality in the original sense of Defn. 4.

7 A New Compact Exponentiation Algorithm

In a typical resource-constrained embedded system, there is normally only room for a very small table. This was the motivation for the division chain method

of Walter [11], given as the right-to-left algorithm in Figure 2. Typically it uses only three registers: explicit T for the accumulating product and P providing the right power of the plaintext input for each digit, and implicit working space P' for temporary values. The idea is that pairs (r_i, d_i) in the representation of D have efficient addition chains for r_i which include d_i as an intermediate value so that P^{d_i} can be computed cheaply *en route* to P^{r_i}. As in the table-based algorithms of Fig. 1, the dual left-to-right algorithm given in Fig. 2 is derived simply by reversing and transposing each step. Consequently, a duality proof would follow the same pattern as for the table-based algorithms.

Inputs: $M \in G$, $D = ((d_{n-1}r_{n-2}+d_{n-2})r_{n-3}+\ldots+d_1)r_0+d_0 \in \mathbb{N}$ where $d_i \in \mathcal{D}$
Output: $M^D \in G$

---

read $P \leftarrow M$
Initialisation: $T \leftarrow P$
$P \leftarrow 1_G$
for $i \leftarrow n{-}1$ **downto** 0 **do**
if $i \neq n{-}1$ **then** $P \leftarrow P^{r_i} \times T^{d_i}$
else $P \leftarrow T^{d_i}$
Finalisation: $T \leftarrow 1_G$
return P

read $P \leftarrow M$
Initialisation: $T \leftarrow 1_G$
for $i \leftarrow 0$ **to** $n{-}1$ **do in parallel** {
$T \leftarrow T \times P^{d_i}$
if $i \neq n{-}1$ **then** $P \leftarrow P^{r_i}$ }
Finalisation: $P \leftarrow T$
$T \leftarrow 1_G$
return P

Fig. 2. Left-to-Right (left) and Right-to-Left (right) Compact Exponentiation.

The dual space-aware chain of atomic operations is still needed for each loop iteration. The right-to-left loop iteration is achieved by a matrix indexed by P and T, namely $\begin{bmatrix} r_i & 0 \\ d_i & 1 \end{bmatrix}$. Its transpose, $\begin{bmatrix} r_i & d_i \\ 0 & 1 \end{bmatrix}$, leads to the dual code given for the left-to-right case. As an example, an iteration with pair $(5, 3)$ can be computed with the addition chain $1{+}1 = 2; 1{+}2 = 3; 2{+}3 = 5$. Using the three registers P, T, and working space P', this can be achieved by the space-aware chain $P \to P'; P' \times P' \to P'; P \times P' \to P; T \times P \to T; P \times P' \to_I P$ where the subscript $_I$ indicates the operation with an initialisation to 1_G. The dual sub-chain is $P \to P'; P \times T \to P; P' \times P \to P'; P' \times P' \to P'; P \times P' \to_I P$. One can readily check that the numbers of each type of operation are the same in this example: one squaring, one multiplication with initialisation, two other multiplications and one copying. Thus there is the effect of having a table which includes M^3 without having to reserve the space for it or spend extra time computing it. Previously it was unclear which digits could be generated this way in a left-to-right algorithm as there was no obvious construction for the required addition sub-chain. Duality solves that problem, as illustrated here with the pair $(5, 3)$.

As the time efficiency is the same for both directions, it is possible to use figures from [12] to see that the algorithm has very similar execution time to the usual algorithms which use similar space (e.g. three registers). When D is fixed

for many exponentiations, the cost of the mixed base recoding can be amortised over the lifetime of the key, and the times from [11] apply. This recoding can be biased to make the best use of any composite operations on G, such as a Frobenius map, which are cheaper than their components. Depending upon where the multiplication by T occurs in the sub-chain, one can also apply one of the double-and-add, triple-and-add or quintuple-and-add formulae for composite elliptic curve operations [6, 10, 9]. Consequently, the new algorithm appears particularly suitable for SSL servers re-using the same key many times. Of course, the time is the same for both directions only using the coarse measurement of counting doubles and adds on the elliptic curve. Use of the composite operations makes modest but different improvements in time to both directions (*see* [1]).

Finally, a few further words on the efficiency of the code. In order to present symmetric versions of the algorithms, there is some extra copying to have a single register for I/O. This is unnecessary in software, but often required in hardware. So, in Fig. 2, M might have been read directly into T instead of P in the left-to-right algorithm, and, dually, the output returned from T rather than P in the right-to-left algorithm. Deletion of the two copyings would still have left dual algorithms computing the same values, although not symmetric, because the matrix for the computation has a single non-zero value. Lastly, if the rules of normalisation are followed when converting the recoding into a space-aware chain then the multiplications by P^{d_i} or T^{d_i} are automatically removed when $d_i = 0$, rendering unnecessary the condition $d_i \neq 0$ that appeared in Fig. 1.

8 Miscellaneous Space Issues for Dual Chains

Returning to general exponentiation algorithms, there may be cost issues in storing the mixed base representation (1). If D is given in binary but one is allowed to choose a base r_i which is not a power of 2, then the recoding must be done from right to left. This can be done on-the-fly for a right-to-left exponentiation method so that minimal additional storage is required for the recoding. However, the left-to-right algorithm requires the complete recoding to be determined and stored in advance. This may not be able to re-use space occupied by D if the key must be kept, but it makes the left-to-right version use more space. On the other hand, as in Figs. 1 and 2, the initial value of M is normally destroyed in the right-to-left direction, but preserved in the left-to-right direction. So extra storage space for input M may be required to retain it in the right-to-left case.

The way in which registers are used also tends to differ between the two directions. Only P is updated in the example left-to-right algorithms, whereas both P and T are updated in the right-to-left cases. This suggests more data movement is required in right-to-left algorithms, especially if the hardware can only write to memory from one register. Thus, although dual exponentiation schemes nominally use the same time and space, there are often relevant secondary space and data movement issues to consider when duality is used in practice.

9 Conclusion

A straight-forward duality mechanism has been provided for addition chains which enables exponentiation algorithms to process digits of the exponent in either direction. In terms of counts of basic operations on the group in which exponentiation takes place, and storage for such elements, this mechanism preserves both the time and space usage of an exponentiation scheme. It thereby improves on current methods which only address time issues. Time and space differences between the two directions are mainly confined to the recoding phase of an exponentiation and data preservation. The duality was illustrated using Brauer's and Yao's algorithms, and applied to derive a new, compact left-to-right algorithm. This algorithm can make use of composite elliptic curve operations to achieve very competitive execution speeds, and is useful in both embedded crypto-systems and SSL servers.

References

1. R. M. Avanzi, *Delaying and Merging Operations in Scalar Multiplication: Applications to Curve-Based Cryptosystems*, SAC 2006, LNCS **4356**, Springer-Verlag, 2007, pp. 203–219.
2. D. J. Bernstein, *Pippenger's Exponentiation Algorithm*, http://cr.yp.to/papers/pippenger.pdf, 2002.
3. A. Brauer, *On Addition Chains*, Bull. Amer. Math. Soc., **45** (10), 1939, 736–739.
4. V. Dimitrov & T. Cooklev, *Two Algorithms for Modular Exponentiation using Non-Standard Arithmetics*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E78-A**(1), Jan 1995, pp. 82–87.
5. V. S. Dimitrov, G. A. Jullien & W. C. Miller, *Theory and Applications for a Double-Base Number System*, Proc. ARITH 13, Monterey, CA, IEEE, 1997, pp. 44–51.
6. V. S. Dimitrov, L. Imbert & P. K. Mishra, *Efficient and Secure Elliptic Curve Point Multiplication using Double-Base Chains*, ASIACRYPT 2005, LNCS **3788**, Springer-Verlag, 2005, pp. 59–78.
7. D. M. Gordon, *A Survey of Fast Exponentiation Algorithms*, Journal of Algorithms, **27**, 1998, pp. 129–146.
8. D. E. Knuth, *The Art of Computer Programming*, vol. **2**, "Seminumerical Algorithms", §4.6.3, 3rd Edition, Addison-Wesley, 1998, pp. 465–485.
9. P. Longa & A. Miri, *New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields*, PKC 2008, LNCS **4939**, Springer-Verlag, 2008, pp. 229–247.
10. P. K. Mishra & V. Dimitrov, *Efficient Quintuple Formulas for Elliptic Curves and Efficient Scalar Multiplication using Multibase Number Representation*, Information Security – ISC 2007, LNCS **4779**, Springer-Verlag, 2007, pp. 390–406.
11. C. D. Walter, *Exponentiation using Division Chains*, Proc. ARITH 13, Monterey, CA, IEEE, 1997, pp. 92–98.
12. C. D. Walter, *MIST: An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis*, CT-RSA 2002, LNCS **2271**, Springer-Verlag, 2002, 53–66.
13. C. D. Walter, *Sliding Windows succumbs to Big Mac Attack*, CHES 2001, LNCS **2162**, Springer-Verlag, 2001, pp. 286–299.
14. A. C.-C. Yao, *On the Evaluation of Powers*, SIAM J. Comput. **5**(1), 1976, 100–103.