A New Encoding Algorithm for a Multidimensional Version of the Montgomery Ladder

Aaron Hutchinson[1] and Koray Karabina[2,3]

[1] University of Waterloo
a5hutchinson@uwaterloo.ca
[2] Florida Atlantic University
kkarabina@fau.edu
[3] National Research Council Canada
koray.karabina@nrc-cnrc.gc.ca

Abstract. We propose a new encoding algorithm for the simultaneous differential multidimensional scalar point multiplication algorithm d-MUL. Previous encoding algorithms are known to have major drawbacks in their efficient and secure implementation. Some of these drawbacks have been avoided in a recent paper in 2018 at a cost of losing the general functionality of the point multiplication algorithm. In this paper, we address these issues. Our new encoding algorithm takes the binary representations of scalars as input, and constructs a compact binary sequence and a permutation, which explicitly determines a regular sequence of group operations to be performed in d-MUL. Our algorithm simply slides windows of size two over the scalars and it is very efficient. As a result, while preserving the full generality of d-MUL, we successfully eliminate the recursive integer matrix computations in the originally proposed encoding algorithms. We also expect that our new encoding algorithm will make it easier to implement d-MUL in constant time. Our results can be seen as the efficient and full generalization of the one dimensional Montgomery ladder to arbitrary dimension.

Keywords: d-MUL, scalar multiplication algorithm, scalar encoding, Montgomery ladder
MSC: 94A60, 11Y16

1 Introduction

Efficient and secure scalar multiplication algorithms are essential in modern cryptography. A (single dimensional) *scalar multiplication algorithm* for a group $\mathbb{G}$ is one which takes an integer α and group element $P \in \mathbb{G}$ as input and produces the element αP as output. Such an algorithm is required in numerous protocols such as Diffie-Hellman key exchange, and digital signature generation and verification. In such group based cryptographic schemes, scalar multiplication dominate the run time of the system, and therefore it is crucial to minimize its cost. Some cryptographic applications can further make use of *multidimensional* scalar multiplication algorithms, which take vectors $(\alpha_1, \ldots, \alpha_d)$ of integers and $(P_1, \ldots, P_d)$ of group elements as input and produces the element $\alpha_1 P_1 + \cdots + \alpha_d P_d$ as output. For example, verifying a signature in the Elliptic Curve Digital Signature Algorithm (ECDSA) requires computing a point $uP + vQ$, where P and Q are public parameters and u and v are derived from the given signature. Multidimensional scalar multiplication can also speed up single scalar multiplication with a fixed base P. For $\lambda = \lfloor |\mathbb{G}|^{1/d} \rfloor$ and $\lambda_i = \lambda^{i-1}$, one can write $\alpha = \sum_{i=1}^{d} \alpha_i \lambda_i$ for $0 \le \alpha_i < \lambda$, precompute $P_i = \lambda_i P$, and compute

$$\alpha P = (\sum_{i=1}^{d} \alpha_i \lambda_i) P = \sum_{i=1}^{d} \alpha_i P_i$$

through multiscalar multiplication with input $\alpha_i, P_i,\ i = 1, ..., d$. If the group $\mathbb{G}$ is equipped with efficiently computable endomorphisms, one can use similar techniques to speed up single scalar multiplication with variable base P because the cost of precomputating P_i becomes negligible compared to the overall cost; see [4,3].

Scalar multiplication algorithms have been studied heavily in the past. One very interesting single dimensional algorithm is the Montgomery ladder [7]. A key difference between the Montgomery ladder and the double-and-add algorithm is that the Montgomery ladder is *regular* in the sense that every iteration of the main loop performs the same operations. It is known that irregularity of algorithms can be exploited through side-channel analysis and underlying scalars may be recovered by attackers; see [9]. Therefore, regularity is essential for security when the scalar α must be kept secret, such as in Diffie-Hellman public key derivation. Another interesting key feature of the Montgomery ladder is that it allows the use of differential point addition $(P, Q, P - Q \mapsto P + Q)$, where the knowledge of the difference of the points helps to write more efficient formulas [8]. As an example, $73P$ can be computed in seven steps by setting $[T, B] = [0, P]$, tracing the bits b_i of 73 from left to right, updating

$$[T, B] \leftarrow [2T, T + B] \text{ if } b_i = 0,$$
$$[T, B] \leftarrow [T + B, 2B] \text{ if } b_i = 1,$$

and so performing one addition and one doubling at each step; see Table 1. Note that the difference of the points to be added is always known (0 or P).

Bernstein [1] proposed a regular two dimensional differential addition chain (the DJB algorithm). The DJB algorithm computes $\alpha_1 P_1 + \alpha_2 P_2$ for ℓ-bit scalars in ℓ steps, performing two additions and one doubling at each step. In particular, the DJB algorithm initiates $T[1] \leftarrow 0$, $T[2] \leftarrow P_1$, $T[3] \leftarrow P_2$, and at each step, $[T[1], T[2], T[3]]$ is updated by doubling one $T[i]$ and adding two distinct pairs of points. Given the bit sequence of α_1 and α_2, a recursive formula was presented in [1] to encode a sequence for the update rules. Table 2 shows an example for computing $73P + 59Q$ in seven steps, performing 1 doubling and 2 additions per

i		1	2	3	4	5	6	7
b_i		1	0	0	1	0	0	1
T	0	P	$2P$	$4P$	$9P$	$18P$	$36P$	$73P$
B	P	$2P$	$3P$	$5P$	$10P$	$19P$	$37P$	$74P$

Table 1: Montgomery ladder for $\alpha = 73$

step. Note that the difference of the points to be added is always known (0, P, Q, or $P \pm Q$).

i		1	2	3	4	5	6	7
$T[1]$	0	$P+Q$	$3P+Q$	$5P+3Q$	$9P+7Q$	$19P+15Q$	$37P+29Q$	$73P+59Q$
$T[2]$	P	$2P$	$2P+2Q$	$4P+4Q$	$10P+8Q$	$18P+14Q$	$36P+30Q$	$74P+58Q$
$T[3]$	Q	$2P+Q$	$3P+2Q$	$5P+4Q$	$9P+8Q$	$18P+15Q$	$37P+30Q$	$74P+59Q$

Table 2: The DJB algorithm for computing $73P + 59Q$

In 2017, a generalization of the Montgomery ladder to d dimensions was made in [6] by means of an algorithm called d-MUL, originally based on an algorithm of Brown from 2006 in [2]. d-MUL uses a sequence of *state matrices* (defined in Section 2.1) to derive an encoding of the scalar vector $(\alpha_1, \ldots, \alpha_d)$, which is used to perform the scalar multiplication. For ℓ-bit scalars α_i, the encoding algorithm in [6] requires dealing with $(d+1) \times d$ integer matrices with ℓ-bit integers. Even though the underlying matrix arithmetic is simple, it introduces non-trivial overhead cost, and makes it harder to resist against side-channel attacks. For example, a constant time implementation of d-MUL at the 128-bit security level in [5] reported about $10,000$ cycle counts for the encoding phase. After encoding, d-MUL loops through ℓ steps, where one doubling and d (differential) addition are performed per step in a regular fashion.

A second paper [5] further explored d-MUL. The motivation in [5] is to bypass the encoding step, and immediately start scalar multiplication by a carefully chosen sequence of group operations: d additions and 1 doubling per step, for a total number of ℓ steps. In particular, a bijection was established between $2^{\ell d} d!$ different choices of (r, σ), where r is a length-ℓd bitstring and σ is a permutation on $\{1, 2, ..., d\}$, and the set of all state matrices containing (at most) ℓ-bit odd scalars $[\alpha_1, ..., \alpha_d]$. In short, by sampling r and σ at random, one can compute a point $\alpha_1 P_1 + \cdots + \alpha_d P_d$, for some α_i sampled at random among ℓ-bit odd integers without explicitly constructing α_i, or their binary representation.

When $d = 1$ and $d = 2$, the algorithms in [5], which we call *randomized d-MUL*, greatly simplify. When $d = 1$, there is only one choice of $\sigma = [1]$, and given r, the scalar multiplication algorithm starts with

$$T[1] \leftarrow 0, \ T[2] \leftarrow P;$$

bits r_i of r are traced from left to right, and $T[1]$ and $T[2]$ are updated as follows

$$[T[1], T[2]] \leftarrow [2T[r_i + 1], T[1] + T[2]].$$

Table 3 gives an example with $r = [1\ 1\ 0\ 1\ 1\ 0\ 1]$, which in the end computes $73P$. Note that the relation between the scalar and the r-sequence is not obvious. This may be compared to the Montgomery ladder computation in Table 1.

i		1	2	3	4	5	6	7
r_i		1	1	0	1	1	0	1
$T[1]$	0	$2P$	$2P$	$4P$	$10P$	$18P$	$36P$	$74P$
$T[2]$	P	P	$3P$	$5P$	$9P$	$19P$	$37P$	$73P$

Table 3: Randomized d-MUL with $r = [1\ 1\ 0\ 1\ 1\ 0\ 1]$

When $d = 2$, there are two choices of $\sigma \in \{[1,2],[2,1]\}$, and given r, the scalar multiplication algorithm starts with

$$T[1] \leftarrow 0,\ T[2] \leftarrow P,\ T[3] \leftarrow P+Q,\ \text{if } \sigma = [1,2],$$
$$T[1] \leftarrow 0,\ T[2] \leftarrow Q,\ T[3] \leftarrow P+Q,\ \text{if } \sigma = [2,1];$$

bits r_i of r are traced from left to right, and $T[1]$ and $T[2]$ are updated such that

$$[T[1],T[2],T[3]] \leftarrow [2T[r_{2i-1}+r_{2i}+1], T[r_{2i}+1]+T[r_{2i}+2], T[1]+T[3]].$$

Table 4 gives an example with $\sigma = [1,2]$ and $r = [01\ 11\ 00\ 10\ 11\ 01\ 01]$, which in the end computes $73P + 59Q$. As in the case of $d = 1$, the relation between the scalars and the r-sequence is not obvious. One may compare this computation to the DJB algorithm example in Table 2.

i		1	2	3	4	5	6	7
$r_{2i-1}r_{2i}$		01	11	00	10	11	01	01
$T[1]$	0	$2P$	$2P+2Q$	$4P+4Q$	$10P+8Q$	$18P+14Q$	$36P+30Q$	$74P+60Q$
$T[2]$	P	$2P+Q$	$3P+2Q$	$5P+4Q$	$9P+8Q$	$18P+15Q$	$37P+30Q$	$74P+59Q$
$T[3]$	$P+Q$	$P+Q$	$3P+3Q$	$5P+3Q$	$9P+7Q$	$19P+15Q$	$37P+29Q$	$73P+59Q$

Table 4: Randomized d-MUL with $\sigma = [1,2]$ and $r = [01\ 11\ 00\ 10\ 11\ 01\ 01]$

The randomized d-MUL method [5] may be useful for some applications where one is interested in computing $\sum \alpha_i P_i$ for some random scalars α_i, but not for some specific (priori-fixed) values α_i. Therefore, applications of this method are limited despite it being very efficient. Deriving α_i from a given (r, σ) was made explicit but the connection between (r, σ) and the corresponding α_i in the other direction was not entirely clear in [5]. In particular, it is not known how to derive (r, σ) from given α_i other than running the original d-MUL encoding as mentioned before, which has its own efficiency and potential security drawbacks.

2 Preliminaries and Our Contributions

In this paper, we derive many theoretical results which explore the connection between (r, σ) and the scalars $(\alpha_1, \ldots, \alpha_d)$ appearing in the output of the d-MUL algorithm from [5]. We use these theoretical results to derive an efficient and compact encoding of an integer vector $(\alpha_1, \ldots, \alpha_d)$ as a bitstring, which we use to build a regular scalar multiplication algorithm similar to that of [5]. In particular, our new encoding algorithm takes the bitstring representations of α_i's and constructs a pair (r, σ) by simply sliding windows of size two from right to left. As a result, while preserving the full generality of d-MUL, we successfully eliminate the recursive integer matrix computations in the original encoding algorithm as proposed in [6]. Therefore, we expect significant time and memory savings in the encoding phase of d-MUL. We also expect that our new encoding algorithm will make it easier to implement d-MUL in constant time.

When α_i are ℓ-bit odd positive integers for $i = 1, ..., d$, our encoding algorithm simplifies to Algorithm 1. Note that Algorithm 1 processes two bits at a time and uses small tables, large integer matrices are not required, and there is no if/else branch in the algorithm. These are some desired features for an efficient and secure implementation of an algorithm. As an example, running Algorithm 1 with $\alpha = 73$ yields the r-sequence as in Table 3, and running it with $[\alpha_1, \alpha_2] = [73, 59]$ yields the r-sequence as in Table 4 and the permutation $\sigma = [1, 2]$. We should emphasize again that previous encoding algorithms do not offer such an efficient algorithm to construct the r-sequence from a given scalar sequence for general $d \geq 1$. Given the r-sequence and σ, point multiplication can be performed using the same rules as described above, or more generally, as described in [5]. Our algorithm in its full generalization to ℓ-bit scalars, including the point multiplication part, is presented later in this paper in Algorithm 4.

Below we give some preliminaries before formally stating the contributions and organization of this paper in Subsection 2.2.

2.1 Preliminaries

In this subsection we summarize some key definitions and results from [6] and [5] as points of reference. Details can be found in the respective papers. We point out that d-dimensional scalar multiplication algorithms in a group $\mathbb{G}$ correspond to those in $\mathbb{Z}^d$ by identifying combinations $\alpha_1 P_1 + \cdots + \alpha_d P_d$ with the vector $(\alpha_1, \ldots, \alpha_d)$; this identification is a group isomorphism modulo the order of P_i in component i, and so we restrict to studying algorithms in $\mathbb{Z}^d$.

Notation. Throughout this paper, we will write $(b_1 b_2 \cdots b_n)_2$ for the binary representation of an integer, where b_1 is the most significant digit and b_n is the parity digit. For binary strings r_1 and r_2 we use $r_1 || r_2$ to denote their concatenation. As usual for a matrix A, we write A_i for the i^{th} row of A, and $A_{i,j}$ for the entry in the i^{th} row and j^{th} column. Matrix indices always begin at 1. We use e_j to denote the unit basis row vector with a 1 in the j^{th} column and 0s elsewhere.

The primary structure that the d-MUL algorithm is built on is a state matrix.

Definition 1. *A $(d+1) \times d$ **state matrix** A is integer-valued and satisfies:*

Algorithm 1: New Encoding for d-MUL

```
Input: Odd integers $\alpha_1, \dots, \alpha_d \in [0, 2^\ell)$, points $P_1, \dots, P_d \in \mathbb{G}$, $\mathbb{G}$ abelian
Output: A binary sequence $r$ of length $\ell d$ bits and a permutation $\sigma$ on $\{1, ..., d\}$
Let $B[i]$ be the binary representation of $\alpha_i$, with extra leading 0.
$\sigma \leftarrow [d - i : i = 0, ..., (d-1)]$
$r \leftarrow [\ ]$
for $k = \ell$ down to 1 do
    $t \leftarrow [\ ], r_t \leftarrow [\ ]$
    for $i = 1$ to $d$ do
        $t[i] \leftarrow (B[i][k] + B[i][k+1]) \mod 2$
    end

    $h \leftarrow 0$
    for $i = 1$ to $d$ do
        $r_t[i] \leftarrow t[\sigma[i]]$
        $h \leftarrow h + r_t[i]$
    end

    $r \leftarrow r_t || r$
    $L \leftarrow [\ ],\ c_0 \leftarrow 0,\ c_1 \leftarrow 0$
    for $i = 1$ to $d$ do
        $w_0 \leftarrow (1 - r_t[i]),\ c_0 \leftarrow c_0 + w_0$
        $w_1 \leftarrow r_t[i],\ c_1 \leftarrow c_1 + w_1$
        $sgn \leftarrow (1 - 2r_t[i])$
        $L[h + sgn \cdot (w_0 \cdot c_0 + w_1 \cdot (c_1 - 1))] \leftarrow \sigma[i]$
    end
    $\sigma \leftarrow L$
end
return $r$, $\sigma$
```

1. *each row A_i has $i-1$ odd entries.*
2. *for $1 \le i \le d$, we have $A_{i+1} - A_i \in \{e_j, -e_j\}$ for some $1 \le j \le d$.*

The ***difference vector*** *for A is $c^A := A_{d+1} - A_1$. We define a bijection $\sigma_A : \{2, \dots, d+1\} \to \{1, \dots, d\}$, called the* ***column sequence*** *of A, by letting $\sigma_A(i)$ be the position in which $A_i - A_{i-1}$ is nonzero. The* ***magnitude*** *of A is defined as $|A| = \max_{i,j}\{|A_{ij}|\}$.*

By "matrix" we will always mean a state matrix unless otherwise stated. All state matrices considered in this paper will have a common size of $(d+1) \times d$ for some dimension d; we will never consider matrices of different sizes simultaneously. We mostly consider matrices with non-negative values. Our interest will lie in pairs of state matrices having special properties, which we introduce shortly in Definition 3. We first state a few necessary results which were proved in [5].

Lemma 1. *For a state matrix A, the row sum $A_m + A_n$ has $|m-n|$ odd entries.*

Corollary 1. *Let A and B be state matrices such that every row in A is the sum of two rows from B. Then for every k there is some m such that $A_k = B_m + B_{m+k-1}$. In particular, $A_1 = 2B_{h+1}$, where h is the number of odd entries in the integer row vector $\frac{1}{2}A_1$.*

Theorem 1. *For a state matrix A, there is a unique state matrix B such that every row in A is the sum of two rows from B.*

Definition 2. *Let A and B be state matrices such that every row in A is the sum of two rows from B. The **addition sequence** $\{a_k\}_{k=1}^{d+1}$ for A corresponding to B is defined to be $a_k = (x_k, y_k)$, where x_k and y_k are the unique row indices for which $A_k = B_{x_k} + B_{y_k}$*

As it turns out, there are exactly 2^d many addition sequences corresponding to a $(d+1) \times d$ matrix B which each yield a different matrix A. The following definition gives a bijection between binary strings and additions sequences, which we use to encode the sequence as a binary string.

Definition 3. *Let B be a $(d+1) \times d$ state matrix and r a binary string of length d. Let h be the number of 1's in r. Define a recursive sequence $a_k = (x_k, y_k)$ of ordered pairs by $x_1 = y_1 = h + 1$ and*

$$a_k = \begin{cases} (x_{k-1}, y_{k-1} + 1) & \text{if } r_{k-1} = 0 \\ (x_{k-1} - 1, y_{k-1}) & \text{if } r_{k-1} = 1 \end{cases}$$

*for $2 \leq k \leq d+1$. The **extension matrix** of B corresponding to r is the $(d+1) \times d$ state matrix A having addition sequence a_k with respect to the matrix B.*

$$B \qquad\qquad\qquad\qquad A$$

$$\begin{bmatrix} 2 & 4 & 2 & 2 \\ 2 & 4 & 2 & 3 \\ 3 & 4 & 2 & 3 \\ 3 & 3 & 2 & 3 \\ 3 & 3 & 3 & 3 \end{bmatrix} \qquad \begin{bmatrix} 6 & 8 & 4 & 6 \\ 5 & 8 & 4 & 6 \\ 5 & 7 & 4 & 6 \\ 5 & 7 & 5 & 6 \\ 5 & 7 & 5 & 5 \end{bmatrix}$$

$$\sigma_B : (4123) \qquad\qquad \sigma_A : (1234)$$

$$c^B = (1, -1, 1, 1) \qquad\qquad c^A = (-1, -1, 1, -1)$$

Fig. 1: Two state matrices A and B of dimension $d = 4$, along with their column sequences and difference vectors. A is the extension matrix of B corresponding to the bitstring $r = 1001$.

Figure 1 gives an example of an extension matrix. Iterating the construction in Definition 3 allows us to built a sequence of matrices given a long binary string.

Definition 4. *Let B be a $(d+1) \times d$ state matrix. Let $r_1, \ldots, r_\ell$ be binary strings of length d, and $r = r_1 || \cdots || r_\ell$. The* ***extension sequence*** *with base B corresponding to r is a sequence $\{A^{(i)}\}_{i=1}^{\ell+1}$ of $(d+1) \times d$ state matrices defined recursively by $A^{(1)} = B$, and $A^{(i+1)}$ is the extension matrix of $A^{(i)}$ corresponding to r_i.*

This definition gives us a way of encoding an entire sequence of matrices $\{A^{(i)}\}_{i=1}^{\ell+1}$ as a simple pair (B, r). Note also that by Theorem 1 the entire sequence is uniquely determined by the final matrix A^{ℓ}. The idea of the randomized d-MUL algorithm in [5] is to randomly choose a $\{0,1\}$-valued state matrix B and binary string of length ℓd, and output the last row of the last matrix of the corresponding extension sequence. The group version of the algorithm can these operations without constructing the matrix sequence explicitly by using the encoding given in Definition 4.

2.2 Contributions and Organization

The main contributions of this paper are:

1. We derive many theoretical results on state matrices and extension sequences. In particular, we determine the exact relationship between the pair (B, r) and the last row of the last matrix of the corresponding extension sequence $\{A^{(i)}\}$. This relationship is stated precisely in Theorem 4, which details how the sequence of matrices built in the algorithm of [6] can be modeled and encoded using the efficient framework of [5].
2. Using the results of Theorem 4 we detail a new version of d-MUL, a d-dimensional scalar multiplication algorithm which is a full generalization of the Montgomery ladder to d dimensions. This version of d-MUL recodes the ℓ-bit input scalars $(\alpha_1, \ldots, \alpha_d)$ very efficiently into a ℓd-length bitstring r, a process only involving permuting the XOR of consecutive bits of the α_i. After recoding the scalars, we use the algorithm of [5] to perform the scalar multiplication with the careful choice of the bitstring r. In particular, this version retains the pattern of 1 point doubling $\mathbf{D}$ and d point additions $\mathbf{A}$ for each bit of the input scalars, giving an operation cost of $\ell\mathbf{D} + \ell d\mathbf{A}$ for the point addition stage. Furthermore, every addition can be performed as a differential addition. Our algorithm does not require storage of any precomputed points, unless differential additions are employed.

In Section 3 we state and prove many theoretical results on extension sequences of state matrices with the aim of optimizing the d-MUL algorithm. In Section 4 we apply the results of Section 3 to construct a new version of the d-MUL algorithm.

3 Theoretical Results

In this section we solve the following two problems:

1. Let $\{A^{(k)}\}_{k=1}^{\ell}$ be an extension sequence with $|A^{(1)}| = 1$. Given only the binary representation of the entries in the row vector $A_1^{(\ell)} + A_{d+1}^{(\ell)}$, find a simple expression giving the binary representations of the entries in $A_1^{(k)}$ for all $k = 1, \ldots, \ell$.
2. Let A be an extension matrix of B corresponding to the bitstring r, and let σ_A and σ_B be the column sequences for A and B, respectively. Find a simple method for determining (σ_B, r) given only (A_1, σ_A).

We make use of the solution to these two problems in the following manner. For a vector $(\alpha_1, \ldots, \alpha_d)$ of positive odd ℓ bit integers, choose a matrix $A^{(\ell)}$ such that $A_1^{(\ell)} + A_{d+1}^{(\ell)} = \begin{bmatrix}\alpha_1 \cdots \alpha_d\end{bmatrix}$ and let $\{A^{(k)}\}_{k=1}^{\ell}$ be the derived extension sequence. Then using the solution to (1) we can determine $A_1^{(k)}$ for every k, and by iterating the solution to (2) we can determine all column sequences σ_k for each matrix $A^{(k)}$ as well as the bitstring r for the entire sequence $\{A^{(k)}\}_{k=1}^{\ell}$. This allows us to determine (r, σ_1) without ever having to construct any matrices. Furthermore $A^{(1)}$ is completely determined by σ_1 since $|A^{(1)}| = 1$. This entire process can then be turned into a method for constructing an efficient addition chain algorithm which uses only the bits of the α_i and the initial choice of column sequence σ_ℓ, and which has very small storage costs and encoding phase.

This section will solve problems (1) and (2) above, whose solutions yield Theorem 4 giving an equivalence of two extension sequence constructions. Section 4 will use the solutions to these problems to detail an efficient scalar multiplication algorithm similar to the original d-MUL algorithm of [6].

3.1 Determining the Bits of an Extension Sequence

The output of the addition chain constructed in Theorem 4 of [5] is always determined by the last row of the final matrix, and so it makes sense to analyze how these final rows change throughout the sequence of state matrices. Our first result of this section finds the connection between the last rows of successive matrices.

Theorem 2. *Let A be an extension matrix of B. Let $B_{d+1,i} = B_{1,i} + c_i$ and $B_{1,i} + B_{d+1,i} = (b_1 b_2 \cdots b_{n-1} 1)_2$. If $A_1 = 2B_{h+1}$, then*

$$A_{1,i} + A_{d+1,i} = \begin{cases} (b_1 b_2 \cdots b_{n-1} 11)_2 & \text{if } (B_{h+1,i} \text{ is even and } c_i = -1) \\ & \text{or } (B_{h+1,i} \text{ is odd and } c_i = 1) \\ (b_1 b_2 \cdots b_{n-1} 01)_2 & \text{if } (B_{h+1,i} \text{ is even and } c_i = 1) \\ & \text{or } (B_{h+1,i} \text{ is odd and } c_i = -1) \end{cases}$$

Proof. We consider two cases.

1. Suppose $B_{h+1,i}$ is even. Then

$$\begin{aligned}
A_{1,i} + A_{d+1,i} &= 2B_{h+1,i} + (B_{1,i} + B_{d+1,i}) \\
&= 2B_{1,i} + (B_{1,i} + B_{d+1,i}) && \text{since } B_{h+1,i} \text{ is even} \\
&= B_{1,i} + B_{d+1,i} - c_i + (B_{1,i} + B_{d+1,i}) \\
&= 2 \cdot (b_1 b_2 \cdots b_{n-1} 1)_2 - c_i \\
&= (b_1 b_2 \cdots b_{n-1} 10)_2 - c_i
\end{aligned}$$

2. Suppose $B_{h+1,i}$ is odd. Then

$$\begin{aligned}
A_{1,i} + A_{d+1,i} &= 2B_{h+1,i} + (B_{1,i} + B_{d+1,i}) \\
&= 2B_{d+1,i} + (B_{1,i} + B_{d+1,i}) && \text{since } B_{h+1,i} \text{ is odd} \\
&= B_{1,i} + B_{d+1,i} + c_i + (B_{1,i} + B_{d+1,i}) \\
&= 2 \cdot (b_1 b_2 \cdots b_{n-1} 1)_2 + c_i \\
&= (b_1 b_2 \cdots b_{n-1} 10)_2 + c_i
\end{aligned}$$

The result follows when considering $c_i = 1$ and $c_i = -1$ in both cases.

With this theorem we can relate the top and bottom rows in a sequence of matrices with the bits of the final matrix, as described in the following corollary.

Corollary 2. *Let $\{A^{(i)}\}_{i=1}^{\ell}$ be an extension sequence such that $|A^{(1)}| = 1$. Let $A^{(\ell)}_{1,i} + A^{(\ell)}_{d+1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1} 1)_2$. Then for $1 \le k \le \ell$,*

(1) $A^{(k)}_{1,i} + A^{(k)}_{d+1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{k-1} 1)_2$,
(2) $A^{(k)}_{1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{k-1})_2 + b^{(i)}_{k-1}$,
(3) $A^{(k)}_{d+1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{k-1})_2 + 1 - b^{(i)}_{k-1}$.

with $b^{(i)}_0 := 0$.

Proof. Note that (2) and (3) follow immediately from (1) since any odd integer a with binary representation $(b_1 b_2 \cdots b_{k-1} 1)_2$ can be written as $a = t + (t + 1)$ for some unique integer t, with the even integer in $\{t, t+1\}$ expressible as $(b_1 b_2 \cdots b_{k-1})_2 + b_{k-1}$ and the odd integer expressible as $(b_1 b_2 \cdots b_{k-1})_2 + 1 - b_{k-1}$.

To prove (1), we use backwards induction on k. The base case $k = \ell$ is given by assumption. Assume that $A^{(k+1)}_{1,i} + A^{(k+1)}_{d+1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_k 1)_2$ for some k. By Theorem 2 the binary expansion of $A^{(k+1)}_{1,i} + A^{(k+1)}_{d+1,i}$ is exactly that of $A^{(k)}_{1,i} + A^{(k)}_{d+1,i}$ with a single bit inserted between the final two bits, and so $A^{(k)}_{1,i} + A^{(k)}_{d+1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{k-1} 1)_2$.

The above corollary solves problem (1) posed at the beginning of this section.

3.2 Determining the Column Sequence and Bitstring from an Extension Matrix

In this subsection we solve problem (2) detailed at the introduction to this section. The following theorem provides an alternative method for describing the addition sequence for a given extension matrix, which will be needed in the results to come.

Theorem 3. *Let A be an extension matrix of B. Let $A_1 = \begin{bmatrix} 2\alpha_1 \; 2\alpha_2 \cdots 2\alpha_d \end{bmatrix}$, let σ_A be the column sequence for A, and let $a_k = (x_k, y_k)$ be the addition sequence for A corresponding to B. Then for $k \geq 1$ we have*

$$a_{k+1} = \begin{cases} (x_k - 1, y_k) & \text{if } \alpha_{\sigma_A(k+1)} \text{ is odd} \\ (x_k, y_k + 1) & \text{if } \alpha_{\sigma_A(k+1)} \text{ is even} \end{cases}$$

Proof. Fix $k \geq 1$. Then

$$2\alpha_{\sigma_A(k+1)} = B_{x_k,\sigma_A(k+1)} + B_{y_k,\sigma_A(k+1)} = A_{k,\sigma_A(k+1)} \equiv 0 \bmod 2$$

and

$$B_{x_{k+1},\sigma_A(k+1)} + B_{y_{k+1},\sigma_A(k+1)} = A_{k+1,\sigma_A(k+1)} \equiv 1 \bmod 2$$

and so we have

$$\begin{aligned} & a_{k+1} = (x_k - 1, y_k) \\ \iff & B_{x_k,\sigma_A(k+1)} \equiv 1 \bmod 2 \quad \text{and} \quad B_{x_{k+1},\sigma_A(k+1)} \equiv 0 \bmod 2 \\ & \qquad (\text{since } x_{k+1} < x_k) \\ \iff & \alpha_{\sigma_A(k+1)} \text{ is odd} \end{aligned}$$

and similarly

$$\begin{aligned} & a_{k+1} = (x_k, y_k + 1) \\ \iff & B_{y_k,\sigma_A(k+1)} \equiv 0 \bmod 2 \quad \text{and} \quad B_{y_{k+1},\sigma_A(k+1)} \equiv 1 \bmod 2 \\ & \qquad (\text{since } y_{k+1} > y_k) \\ \iff & \alpha_{\sigma_A(k+1)} \text{ is even.} \end{aligned}$$

We can now derive an expression for the binary string giving the addition sequence for two state matrices A and B using only the column sequence for A and the row which was doubled from B.

Corollary 3. *Let A be an extension matrix of B. Let $A_1 = \begin{bmatrix} 2\alpha_1 \; 2\alpha_2 \cdots 2\alpha_d \end{bmatrix}$ and let σ_A be the column sequence for A. Then*

$$r = (\alpha_{\sigma_A(2)} \; mod \; 2) || \cdots || (\alpha_{\sigma_A(d+1)} \; mod \; 2)$$

is the binary string giving the addition sequence for A corresponding to B, where $||$ denotes concatenation of bits.

Proof. Let $a_k = (x_k, y_k)$ be the addition sequence for A corresponding to B, and let $\hat{a}_k = (\hat{x}_k, \hat{y}_k)$ be the recursive sequence obtained from r using Definition 3. We show that $a_k = \hat{a}_k$ for every k by induction on k. For $k = 1$, we have $A_1 = \begin{bmatrix} 2\alpha_1 \; 2\alpha_2 \; \cdots \; 2\alpha_d \end{bmatrix} = 2B_{h+1}$ by Corollary 1, where h is the number of odds in $\begin{bmatrix} \alpha_1 \; \alpha_2 \; \cdots \; \alpha_d \end{bmatrix}$, and so $a_1 = (h+1, h+1)$. By the definition of an extension matrix, we have $\hat{x}_1 = \hat{y}_1 = 1 + \sum_{i=1}^{d} (\alpha_{\sigma_A(i+1)} \bmod 2) = 1 + \sum_{i=1}^{d} (\alpha_i \bmod 2) = 1 + h$ since σ is a bijection. Therefore $a_1 = \hat{a}_1$.

Let r_i be the ith bit in r. If $k \geq 1$, we have

$$\begin{aligned}
\hat{a}_{k+1} &= \begin{cases} (\hat{x}_k - 1, \hat{y}_k) \text{ if } r_k = 1 \\ (\hat{x}_k, \hat{y}_k + 1) \text{ if } r_k = 0 \end{cases} \\
&= \begin{cases} (\hat{x}_k - 1, \hat{y}_k) \text{ if } \alpha_{\sigma_A(k+1)} \text{ is odd} \\ (\hat{x}_k, \hat{y}_k + 1) \text{ if } \alpha_{\sigma_A(k+1)} \text{ is even} \end{cases} && \text{by definition of } r \\
&= \begin{cases} (x_k - 1, y_k) \text{ if } \alpha_{\sigma_A(k+1)} \text{ is odd} \\ (x_k, y_k + 1) \text{ if } \alpha_{\sigma_A(k+1)} \text{ is even} \end{cases} && \text{by inductive hypothesis} \\
&= a_{k+1} && \text{by Theorem 3.}
\end{aligned}$$

We can now relate the column sequences of the two state matrices A and B through the following definition. Lemma 2 to follow shows this relationship explicitly.

Definition 5. *Let $\sigma : \{2, 3, \ldots, d+1\} \to \{1, 2, \ldots, d\}$ be a bijection and let $b_1, \ldots, b_d$ be bits. Define the bijection $\tau : \{2, 3, \ldots, d+1\} \to \{1, 2, \ldots, d\}$ as follows:*

1. *Initialize two empty lists L_0 and L_1.*
2. *For $i = 1$ to d, append $\sigma(i+1)$ to the end of L_{b_i}.*
3. *Let $L = \text{REVERSE}(L_1)||L_0$, where $||$ denotes concatenation.*
4. *Define $\tau(i+1) = L(i)$ for $1 \leq i \leq d$.*

Define Ψ as the function giving τ from σ and $b_1, \ldots, b_d$; that is,

$$\Psi(\sigma, (b_1, \ldots, b_d)) = \tau.$$

When given a list as input, the function REVERSE returns the list in reverse order. Note that τ is a bijection since L contains each of the values $\sigma(2), \sigma(3), \ldots, \sigma(d+1)$ exactly once.

Lemma 2. *Let A be an extension matrix of B. Let σ_A and σ_B be the column sequences for A and B, respectively, and let $A_1 = \begin{bmatrix} 2\alpha_1 \; \cdots \; 2\alpha_d \end{bmatrix}$. Then*

$$\sigma_B = \Psi\left(\sigma_A, (\alpha_{\sigma_A(2)} \bmod 2, \ldots, \alpha_{\sigma_A(d+1)} \bmod 2)\right).$$

Proof. Let $\tau = \Psi\left(\sigma_A, (\alpha_{\sigma_A(2)} \bmod 2, \ldots, \alpha_{\sigma_A(d+1)} \bmod 2)\right)$. We begin by noting that at step 3 in defining τ we have that the size of L_1 is $|\{i : \alpha_i = 1 \bmod 2\}| = h$. Let $1 \leq k \leq d$. We examine two cases.

Suppose $\alpha_{\sigma_A(k+1)}$ is odd. Then

$$A_{k+1} = A_k + c^A_{\sigma_A(k+1)} e_{\sigma_A(k+1)} = B_{x_k} + B_{y_k} + c^A_{\sigma_A(k+1)} e_{\sigma_A(k+1)}$$

and by Theorem 3 we have $a_{k+1} = (x_{k+1}, y_{k+1}) = (x_k - 1, y_k)$ and

$$A_{k+1} = B_{x_{k+1}} + B_{y_{k+1}} = B_{x_k - 1} + B_{y_k} = B_{x_k} - c^B_{\sigma_B(x_k)} e_{\sigma_B(x_k)} + B_{y_k}$$

Equating these two expressions for A_{k+1} gives $\sigma_B(x_k) = \sigma_A(k+1)$. We point out that $|\{\alpha_{\sigma_A(i)} : 2 \leq i \leq k+1, \alpha_{\sigma_A(i)} \text{ odd}\}| = h + 1 - x_{k+1}$ since $x_1 = h + 1$ and x_i decreases exactly when an odd α_j is found. In defining τ, step 2 would put $\sigma_A(k+1)$ into $L_{\alpha_{\sigma_A(k+1)} \bmod 2} = L_1$ and we would have $L_1(h+1-x_{k+1}) = \sigma_A(k+1)$. Since the order of L_1 is reversed to form L, we have $\tau(x_k) = L(x_k - 1) = L(x_{k+1}) = L_1(h+1-x_{k+1}) = \sigma_A(k+1) = \sigma_B(x_k)$.

Suppose now $\alpha_{\sigma_A(k+1)}$ is even. Then $A_{k+1} = B_{x_k} + B_{y_k} + c^A_{\sigma_A(k+1)} e_{\sigma_A(k+1)}$ as before, and by Theorem 3 we have $a_{k+1} = (x_{k+1}, y_{k+1}) = (x_k, y_k + 1)$ and so

$$A_{k+1} = B_{x_{k+1}} + B_{y_{k+1}} = B_{x_k} + B_{y_k + 1} = B_{x_k} + B_{y_k} + c^B_{\sigma_B(y_k+1)} e_{\sigma_B(y_k+1)}$$

Equating these two expressions for A_{k+1} gives $\sigma_B(y_k + 1) = \sigma_A(k+1)$. Similarly to the first case we have $|\{\alpha_{\sigma_A(i)} : 2 \leq i \leq k+1, \alpha_{\sigma_A(i)} \text{ even}\}| = y_{k+1} - (h+1)$ since $y_1 = h + 1$ and y_i increases exactly when an even α_j is found. Step 2 in τ's definition would put $\sigma_A(k+1)$ into $L_{\alpha_{\sigma_A(k+1)} \bmod 2} = L_0$ and we would have $L_0(y_{k+1} - (h+1)) = \sigma_A(k+1)$. Since L_0 is concatenated to the end of L_1 when forming L, we have $\tau(y_k + 1) = L(y_k) = L_0(y_k - h) = L_0(y_{k+1} - (h+1)) = \sigma_A(k+1) = \sigma_B(y_k + 1)$.

Since the sequence $\{x_i\}_{i=1}^{d+1}$ takes on every value in $\{1, 2, \ldots, h+1\}$ and $\{y_i\}_{i=1}^{d+1}$ takes on every value in $\{h+1, h+2, \ldots, d+1\}$, we have that $\sigma_B = \tau$. This concludes the proof.

With Corollary 3 and Lemma 2, we have solved problem (2).

3.3 Alternative Construction of an Extension Sequence

We now arrive at our primary result of this section, which uses the results from the previous subsections to directly construct the binary string for an extension sequence yielding a given d-tuple.

Theorem 4. *Suppose the following are given:*

- $(\alpha_1, \alpha_2, \ldots, \alpha_d)$*, where each* α_i *is an odd positive integer with* ℓ *bits or less*
- $\sigma_\ell : \{2, 3, \ldots, d+1\} \to \{1, 2, \ldots, d\}$ *a bijection.*

From this information, let $\alpha_i = (b_1^{(i)} b_2^{(i)} \cdots b_{\ell-1}^{(i)} 1)_2$ *and:*

1. *Let $A^{(\ell)}$ be the state matrix having*
   i) $A^{(\ell)}_{1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1})_2 + b^{(i)}_{\ell-1}$*,*
   ii) $A^{(\ell)}_{d+1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1})_2 + 1 - b^{(i)}_{\ell-1}$*,*
   iii) column sequence σ_ℓ*.*
   Let $\{A^{(i)}\}_{i=1}^{\ell}$ *be the unique (Theorem 1) sequence of state matrices such that every row from* $A^{(i)}$ *is the sum of two rows from* $A^{(i-1)}$ *for* $1 < i \leq \ell$*, and let* σ_i *be the column sequence for* $A^{(i)}$*.*
2. *Define a recursive sequence by* $\hat{\sigma}_\ell = \sigma_\ell$ *and*
$$\hat{\sigma}_k = \Psi\left(\hat{\sigma}_{k+1}, ((b^{(\hat{\sigma}_{k+1}(2))}_{k-1} \oplus b^{(\hat{\sigma}_{k+1}(2))}_{k}), \dots, (b^{(\hat{\sigma}_{k+1}(d+1))}_{k-1} \oplus b^{(\hat{\sigma}_{k+1}(d+1))}_{k}))\right)$$
   for $1 \leq k < \ell$*, where* $b^{(i)}_0 := 0$ *and "*$\oplus$*" denotes XOR of bits. Let*
$$r^{(k)} = (b^{(\hat{\sigma}_{k+1}(2))}_{k-1} \oplus b^{(\hat{\sigma}_{k+1}(2))}_{k})||\cdots||(b^{(\hat{\sigma}_{k+1}(d+1))}_{k-1} \oplus b^{(\hat{\sigma}_{k+1}(d+1))}_{k})$$
   for $1 \leq k < \ell$*, where* $||$ *denotes concatenation.*

Then $\sigma_k = \hat{\sigma}_k$ *for* $1 \leq k \leq \ell$ *and* $\{A^{(i)}\}_{i=1}^{\ell}$ *is the extension sequence corresponding to* $r = r^{(1)}||r^{(2)}||\cdots||r^{(\ell-1)}$ *and having a base given by a matrix having magnitude 1 and column sequence* $\hat{\sigma}_1$*.*

Proof. We first note that for any $0 \leq k < \ell$ and $1 \leq i \leq d$, Corollary 2 gives $A^{(k+1)}_{1,i} = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{k-1} b^{(i)}_k)_2 + b^{(i)}_k$. In both cases that $b^{(i)}_k = 0$ or $b^{(i)}_k = 1$, we see that $\frac{1}{2}A^{(k+1)}_{1,i} \bmod 2 = b^{(i)}_{k-1} \oplus b^{(i)}_k$, where $b^{(i)}_j := 0$ for $j < 1$.

We show $\sigma_k = \hat{\sigma}_k$ for all k by backwards induction on k. When $k = \ell$ we have $\sigma_\ell = \hat{\sigma}_\ell$ by definition. Suppose $\sigma_{k+1} = \hat{\sigma}_{k+1}$ for some k. Taking $A = A^{(k+1)}$ and $B = A^{(k)}$ in the supposition of Lemma 2, we conclude that
$$\begin{aligned}\sigma_k &= \Psi\left(\sigma_{k+1}, (\tfrac{1}{2}A^{(k+1)}_{1,\sigma_{k+1}(2)} \bmod 2, \dots, \tfrac{1}{2}A^{(k+1)}_{1,\sigma_{k+1}(d+1)} \bmod 2)\right)\\ &= \Psi\left(\sigma_{k+1}, (b^{(\sigma_{k+1}(2))}_{k-1} \oplus b^{(\sigma_{k+1}(2))}_{k}, \dots, b^{(\sigma_{k+1}(d+1))}_{k-1} \oplus b^{(\sigma_{k+1}(d+1))}_{k})\right)\\ &= \hat{\sigma}_k\end{aligned}$$
since $\sigma_{k+1} = \hat{\sigma}_{k+1}$.

Now we show that $A^{(k+1)}$ is the extension matrix of $A^{(k)}$ corresponding to $r^{(k)}$ for a fixed k. Taking $A = A^{(k+1)}$ and $B = A^{(k)}$ in the supposition of Corollary 3, we have that the binary string giving the addition sequence for $A^{(k+1)}$ corresponding to $A^{(k)}$ is
$$\begin{aligned}&(\tfrac{1}{2}A^{(k+1)}_{1,\sigma_{k+1}(2)} \bmod 2)||\cdots||(\tfrac{1}{2}A^{(k+1)}_{1,\sigma_{k+1}(d+1)} \bmod 2)\\ &= (b^{(\sigma_{k+1}(2))}_{k-1} \oplus b^{(\sigma_{k+1}(2))}_{k})||\cdots||(b^{(\sigma_{k+1}(d+1))}_{k-1} \oplus b^{(\sigma_{k+1}(d+1))}_{k})\\ &= r^{(k)}\end{aligned}$$
since we've already shown $\sigma_k = \hat{\sigma}_k$ for all k.

By definition we now have that $\{A^{(i)}\}_{i=1}^{\ell}$ is the extension sequence with base $A^{(1)}$ corresponding to r. By Theorem 4.4 of [6], $A^{(1)}$ has magnitude 1 and by definition has column sequence $\sigma_1 = \hat{\sigma}_1$. This concludes the proof of the theorem.

In the context of Theorem 4, note that

$$\begin{aligned}A^{(\ell)}_{1,i} + A^{(\ell)}_{d+1,i} &= \left[(b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1})_2 + b^{(i)}_{\ell-1}\right] + \left[(b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1})_2 + 1 - b^{(i)}_{\ell-1}\right] \\ &= 2 \cdot (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1})_2 + 1 = (b^{(i)}_1 b^{(i)}_2 \cdots b^{(i)}_{\ell-1} 1)_2 = \alpha_i.\end{aligned}$$

The significance of Theorem 4 is the following. The d-MUL algorithm, Algorithm 3 in [6], is performed using the method of item (1) in Theorem 4; that is, it computes the sequence $\{A^{(i)}\}^{\ell}_{i=1}$ explicitly and stores the addition sequence information for each matrix. This is a very costly operation in terms of clock cycles and storage. Theorem 4 shows that the algorithm can be performed instead using item (2) by only computing the sequence $\{\sigma_i\}^{\ell}_{i=1}$ (given by Ψ) and the bit string r, therefore bypassing any matrix or integer arithmetic and allowing us to begin computing points immediately after r has been constructed. An algorithm similar to that of Algorithm 2 of [5] can then be used to compute the same output as running the original d-MUL with the input $(a_1, \ldots, a_d)$ and a choice for σ_ℓ.

4 Optimized d-MUL

In this section we present Algorithm 4, which is essentially Algorithm 3.2 of [5] in which the bitstring r is constructed through the method of item (2) in Theorem 4 to give a desired set of output scalars. This is in contrast to choosing r uniformly at random as in [5].

In addition to using the alternative method of computation given by Theorem 4, we address a potential security issue when formulating Algorithm 4. The algorithm in [5] and many of the results in this paper have produced an integer vector with odd entries, and with the intention of subtracting off a binary vector v to yield an output vector with entries of arbitrary parity. How exactly the point corresponding to this vector v is subtracted off has not yet been discussed.

Let P_i be the points of a desired linear combination. If all 3^d elements of the set $\{c_1P_1 + \cdots + c_dP_d : c_i \in \{0, 1\}\}$ are stored, such as when using differential additions, then the point corresponding to the binary vector v is one such point; this point may then be looked up and a single addition can be performed to complete the scalar multiplication.

If these 3^d points are not stored, then more care should be taken. If each P_i satisfying $v_i = 1$ is to be subtracted off from the output in succession, then this may leak information about the scalars of the desired linear combination (or at the very least the number of even scalars). One solution is to simply not perform the subtraction by v at all and settle for an output in which all scalars are odd. This would cut down the size of the output space by a factor of 2^d. This may or may not be acceptable for a given application of the algorithm.

We give an alternative solution to this problem now, which essentially just adds another iteration in the state matrix sequence. That is, we make the sacrifice of an additional d additions and 1 doubling for added security and a uniform output. Suppose we wish to compute the point $\alpha_1P_1 + \cdots + \alpha_dP_d$ for arbitrary

ℓ-bit α_i (not necessarily odd or positive). If any α_i is negative, we may negate α_i and P_i and treat $\alpha_i P_i$ as $(-\alpha_i)(-P_i)$. With negligible preprocessing we may therefore assume every α_i is positive. Let $(b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2$ be the binary representation of α_i, and define $\hat{\alpha}_i$ as $(b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2 + b_\ell^{(i)} - 1$. Then $\alpha_i - \hat{\alpha}_i \in \{0, 1\}$, and $2\hat{\alpha}_i + 1$ has $\ell + 1$ bits. We then apply Theorem 4 to the odd integers $2\hat{\alpha}_i + 1$ for $1 \leq i \leq d$ and some column sequence σ. By item (1) of the same theorem, we get a state matrix $A^{(\ell+1)}$ satisfying:

1. $A^{(\ell+1)}_{1,i} = (b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2 + b_\ell^{(i)}$,
2. $A^{(\ell+1)}_{d+1,i} = (b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2 + 1 - b_\ell^{(i)}$,
3. $A^{(\ell+1)}$ has column sequence σ.

The matrix $A^{(\ell+1)}$ therefore contains all of the original values $\alpha_1, \ldots, \alpha_d$. If σ is chosen carefully, then this matrix will contain the row $\begin{bmatrix}\alpha_1 \ \alpha_2 \ \cdots \ \alpha_d\end{bmatrix}$. Specifically, we may choose σ as any bijection in which the indices for all odd α_i come before those which are even. The index corresponding to this row will be exactly $h := 1 + \sum(\alpha_i \mod 2)$.

We note that Theorem 4 doesn't use the last parity bits of the α_i, but in this context we are applying the theorem to the integers $2\hat{\alpha}_i + 1$. Therefore the final "1" bit of $2\hat{\alpha}_i + 1$ will be ignored, but the rest will be used to construct a bitstring r of length ℓd. That is, we use exactly the bits of $\hat{\alpha}_i$ with an extra leading "0" bit.

Details of Algorithm 4: Here we give some details regarding Algorithm 4. The notation **Ai**(j) refers to line j of Algorithm i.

1. To simplify the presentation we deal with negative integer inputs by calling Algorithm 3, Sanitize, using the method described at the beginning of this section. This is, if α_i is negative we replace α_i by $-\alpha_i$ and P_i by $-P_i$. If working in a setting such as a Montgomery curve using XZ-coordinates, this step isn't necessary since P_i is identified with $-P_i$.
2. Similarly, we separate the process of choosing an initial column sequence σ into a different algorithm, Algorithm 2: ChooseSeq. We choose any permutation for which the indices of the odd α_i are placed before the indices for the even α_i. The RandomPermutation function seen in Algorithm 2 returns a permutation of the input set chosen uniformly at random, represented in list form. The lists σ_E and σ_O are concatenated to form a single permutation.
3. The binary representation in line **A4**(4) is computed with the most significant bit of $\hat{\alpha}_i$ being $b_2^{(i)}$ and the parity bit being $b_{\ell+1}^{(i)}$.
4. The loop **A4**(6) follows Definition 5 while also constructing the bitstring r simultaneously.
5. The loop **A4**(14) is essentially the same as that seen in the Randomized d-MUL algorithm of [5]. The conditional seen in [5] has been replaced in favor of a much simpler, compact, and equivalent assignment for both x and y.

```
Algorithm 2: ChooseSeq
Input: Integers $\alpha_1, \ldots, \alpha_d$
Output: Permutation on $\{1, 2, \ldots, d\}$
1 Evens ← $\{i : \alpha_i \equiv 0 \mod 2\}$
2 Odds ← $\{i : \alpha_i \equiv 1 \mod 2\}$
3 $\sigma_E$ ← RandomPermutation(Evens)
4 $\sigma_O$ ← RandomPermutation(Odds)
5 return $\sigma_O || \sigma_E$
```

```
Algorithm 3: Sanitize
Input: Integers $\alpha_1, \ldots, \alpha_d$, points $P_1, \ldots, P_d \in \mathbb{G}$, $\mathbb{G}$ abelian
Output: Positive integers $\alpha_1, \ldots, \alpha_d$, points $P_1, \ldots, P_d \in \mathbb{G}$, $\mathbb{G}$ abelian
1 for $i = 1$ to $d$ do
2     if $\alpha_i < 0$ then
3         $\alpha_i \leftarrow -\alpha_i$
4         $P_i \leftarrow -P_i$
5     end
6 end
7 return $\alpha, P$
```

A special case is when all scalars α_i are positive and odd. In this case, the Sanitize step has no effect, and ChooseSeq amounts to choosing any permutation on d elements. Furthermore, the $\hat{\alpha}_i$ calculated in Algorithm 4 are equal to the input α_i. This special case leads to an encoding given by the implementation-oriented Algorithm 1, where we skip sanitization and always make the same choice of initial σ. In addition, the construction of the array L is done without an if/else branch for side-channel resistance.

A basic Magma implementation of Algorithm 4 can be found here:

`https://github.com/AaronHutchinson/d-MUL-Optimized-2020-`

4.1 Differential Additions

This subsection aims to outline an alternate version of Algorithm 4 which utilizes differential additions. Our only sacrifice to gain knowledge of point differences is storing each column sequence σ generated in the loop on line 6 of Algorithm 4. We can compute point differences using the following theorem.

Theorem 5. *Let A be an extension matrix of B with addition sequence $\{a_k\}_{k=1}^{d+1}$. If σ is the column sequence for B and c is the difference vector for B, then $B_{y_1} - B_{x_1}$ is the zero row matrix and for $2 \le k \le d+1$ we have*

$$B_{y_k} - B_{x_k} = \sum_{i=x_k+1}^{y_k} c_{\sigma(i)} e_{\sigma(i)}.$$

Proof. We use induction on k. When $k = 1$ we have $x_1 = y_1$ by definition of an addition sequence, and so $B_{y_1} - B_{x_1}$ is zero. Assume that $B_{y_k} - B_{x_k} = \sum_{i=x_k+1}^{y_k} c_{\sigma(i)} e_{\sigma(i)}$ for some k with $1 \leq k \leq d$. We have either that $a_{k+1} = (x_k - 1, y_k)$ or $a_{k+1} = (x_k, y_k + 1)$.

Suppose that $a_{k+1} = (x_k - 1, y_k)$ so that $y_{k+1} = y_k$ and $x_{k+1} = x_k - 1$. Then

$$\begin{aligned} B_{y_{k+1}} - B_{x_{k+1}} &= B_{y_k} - B_{x_k - 1} = B_{y_k} - (B_{x_k} - c_{\sigma(x_k)} e_{\sigma(x_k)}) \\ &= (B_{y_k} - B_{x_k}) + c_{\sigma(x_k)} e_{\sigma(x_k)} = \sum_{i=x_k+1}^{y_k} c_{\sigma(i)} e_{\sigma(i)} + c_{\sigma(x_k)} e_{\sigma(x_k)} \\ &= \sum_{i=x_k}^{y_k} c_{\sigma(i)} e_{\sigma(i)} = \sum_{i=x_{k+1}+1}^{y_{k+1}} c_{\sigma(i)} e_{\sigma(i)}. \end{aligned}$$

If $a_{k+1} = (x_k, y_k + 1)$ then $y_{k+1} = y_k + 1$ and $x_{k+1} = x_k$, and so

$$\begin{aligned} B_{y_{k+1}} - B_{x_{k+1}} &= B_{y_k+1} - B_{x_k} = (B_{y_k} + c_{\sigma(y_k+1)} e_{\sigma(y_k+1)}) - B_{x_k} \\ &= (B_{y_k} - B_{x_k}) + c_{\sigma(y_k+1)} e_{\sigma(y_k+1)} \\ &= \left(\sum_{i=x_k+1}^{y_k} c_{\sigma(i)} e_{\sigma(i)} \right) + c_{\sigma(y_k+1)} e_{\sigma(y_k+1)} \\ &= \sum_{i=x_k+1}^{y_k+1} c_{\sigma(i)} e_{\sigma(i)} = \sum_{i=x_{k+1}+1}^{y_{k+1}} c_{\sigma(i)} e_{\sigma(i)}. \end{aligned}$$

This concludes the proof.

Suppose that all rows in the set $S = \{[t_1, \ldots, t_d] : t_i \in \{0, 1, -1\}\}$ are stored. Then the above theorem tells us exactly how to find the proper element of S for the difference which corresponds to a sum $B_i + B_j$. The only knowledge required to compute this row is the column sequence σ and the difference vector c. We will now show that only a slight modification of Algorithm 4 will allow us to perform differential additions.

Let α_i and $\hat{\alpha}_i$ for $i = 1, \ldots, d$ be as in Section 4, and let σ be any column sequence. Again by Theorem 4 we may derive a sequence $\{A^{(k)}\}_{k=1}^{\ell+1}$ of state matrices where each row in $A^{(k+1)}$ is the sum of two rows from $A^{(k)}$ and the final matrix $A^{(\ell+1)}$ satisfies:

1. $A^{(\ell+1)}_{1,i} = (b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2 + b_\ell^{(i)}$,
2. $A^{(\ell+1)}_{d+1,i} = (b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2 + 1 - b_\ell^{(i)}$,
3. $A^{(\ell+1)}$ has column sequence σ

where $(b_1^{(i)} b_2^{(i)} \cdots b_\ell^{(i)})_2$ is the binary representation of α_i. We recall that the difference vector c for any state matrix A is defined to be $A_{d+1} - A_1$. Applying Corollary 2 to our current scenario, we find that the i^{th} entry of the difference vector for $A^{(k)}$ is exactly

$$\begin{aligned} A^{(k)}_{d+1,i} - A^{(k)}_{1,i} &= \left((b_1^{(i)} b_2^{(i)} \cdots b_{k-1}^{(i)})_2 + 1 - b_{k-1}^{(i)} \right) - \left((b_1^{(i)} b_2^{(i)} \cdots b_{k-1}^{(i)})_2 + b_{k-1}^{(i)} \right) \\ &= 1 - 2b_{k-1}^{(i)} \end{aligned}$$

```
Algorithm 4: Optimized $d$-MUL
Input: Integers $\alpha_1, \dots, \alpha_d \in (-2^\ell, 2^\ell)$, points $P_1, \dots, P_d \in \mathbb{G}$, $\mathbb{G}$ abelian
Output: Group element $\alpha_1 P_1 + \cdots + \alpha_d P_d$
1  $\alpha, P \leftarrow \textsc{Sanitize}(\alpha, P)$.
2  $\sigma \leftarrow \textsc{ChooseSeq}(\alpha)$.
3  $\hat{\alpha} \leftarrow (\alpha_1 + (\alpha_1 \mod 2) - 1, \dots, \alpha_d + (\alpha_d \mod 2) - 1)$
4  Let $(0\, b_2^{(i)} b_3^{(i)} \cdots b_\ell^{(i)} b_{\ell+1}^{(i)})_2$ be the binary form of $\hat{\alpha}_i$, with extra leading 0.
5  Initialize an empty binary array $r$ of length $\ell d$.
6  for $k = \ell$ down to 1 do
7      For $i = 1$ to $d$, assign $r_{(k-1)d+i} \leftarrow b_k^{(\sigma(i))} \oplus b_{k+1}^{(\sigma(i))}$.
8      Initialize empty lists $L_0$ and $L_1$ of length $d$.
9      For $i = 1$ to $d$, append $\sigma(i)$ to the end of $L_{b_k^{(\sigma(i))} \oplus b_{k+1}^{(\sigma(i))}}$.
10     Overwrite $\sigma \leftarrow \textsc{Reverse}(L_1) || L_0$, where $||$ denotes concatenation.
11 end
12 Initialize group elements $Q_1, \dots, Q_{d+1}, R_1, \dots R_{d+1}$ as $id(\mathbb{G})$.
13 For $i = 1$ to $d$, assign $Q_{i+1} \leftarrow Q_i + P_{\sigma(i)}$.
14 for $k = 1$ to $\ell$ do
15     $h, x, y \leftarrow r_{(k-1)d+1} + \cdots + r_{kd} + 1$
16     $R_1 \leftarrow 2Q_h$
17     for $i = 1$ to $d$ do
18         $x \leftarrow x - r_{(k-1)d+i}$, $y \leftarrow y + 1 - r_{(k-1)d+i}$
19         $R_{i+1} \leftarrow Q_x + Q_y$
20     end
21     $Q \leftarrow R$
22 end
23 $h \leftarrow (\alpha_1 \mod 2) + \cdots + (\alpha_d \mod 2) + 1$
24 return $Q_h$
```

Therefore the entries of this difference vector are given “for free”, as they only depend on the bits in position $k-1$ of the α_i.

With this discussion in mind, Algorithm 4 may be altered so that each σ derived in the loop beginning on line 6 is saved in a table so that the column sequence for matrix $A^{(i)}$ is stored as σ_i. One may then use Theorem 5 to find the difference corresponding to each sum; it is exactly

$$A_{y_k}^{(i)} - A_{x_k}^{(i)} = \sum_{i=x_k+1}^{y_k} (1 - 2b_{i-1}^{(\sigma_i(k))}) e_{\sigma_i(k)}.$$

5 Conclusions

There are now three versions of the d-MUL algorithm: Original d-MUL (Algorithm 3 of [6]), Randomized d-MUL (Algorithm 2 of [5]), and Optimized d-MUL (Algorithm 4 in this paper). Optimized d-MUL seems to be a direct improvement over Original d-MUL, since the storage of two $(d+1) \times d$ matrices with large entries, ℓ many arrays D, and large integer arithmetic is exchanged for the storage of a single ℓd length bitstring and the computation of ℓ many simple permutations. We therefore see no reason to use Original d-MUL over Optimized d-MUL.

We believe that Randomized d-MUL may still be preferable over Optimized d-MUL in certain special situations. If a given application only calls for a random linear combination, then it would be more efficient to employ Randomized d-MUL over Optimized d-MUL since in the former case we need only generate a random bit string rather than derive it from random scalars as in the latter case. The efficiency gain is slightly more dramatic when the scalars of the combination need not be known, since the derivation of the scalars in Randomized d-MUL is split off into an independent algorithm. On the other hand, if the setting calls for a specific linear combination to be computed from given points, we see no way to use Randomized d-MUL in such a setting and so Optimized d-MUL seems to be the best option out of these three algorithms.

Acknowledgment

This research has been partially supported by the U.S. Army Research Office (ARO) under the award number W911NF-17-1-0311. The content is solely the responsibility of the authors and does not necessarily represent the official views of the ARO. The authors thank reviewers for their comments.

References

1. D. Bernstein. Differential addition chains. Technical report, 2006. Available at http://cr.yp.to/ecdh/diffchain-20060219.pdf.
2. D. Brown. Multi-Dimensional Montgomery Ladders for Elliptic Curves. ePrint Archive: Report 2006/220. Available at http://eprint.iacr.org/2006/220.
3. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. *Journal of Cryptology*, 24:446–469, 2011.
4. R. Gallant, R. Lambert, and S. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. *Advances in Cryptology - CRYPTO 2011, Lecture Notes in Computer Science*, 2139:190–200, 2001.
5. H. Hisil, A. Hutchinson, and K. Karabina. d-MUL: Optimizing and Implementing a Multidimensional Scalar Multiplication Algorithm over Elliptic Curves. *8th International Conference on Security, Privacy, and Applied Cryptography Engineering - SPACE 2018, Lecture Notes in Computer Science*, 11348:198–217, 2018.
6. A. Hutchinson and K. Karabina. Constructing Multidimensional Differential Addition Chains and Their Applications. *Journal of Cryptographic Engineering*, 9(1):1–19, 2019.
7. P.L. Montgomery. Evaluating Recurrences of the Form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas Chains. Available at https://cr.yp.to/bib/1992/montgomery-lucas.ps, 1983.
8. P.L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48:243–264, 1987.
9. J. Jaffe P. Kocher and B. Jun. Differential Power Analysis. *Advances in Cryptology — CRYPTO '99, Lecture Notes in Computer Science*, 1666:388–397, 1999.