d-MUL: Optimizing and Implementing a Multidimensional Scalar Multiplication Algorithm over Elliptic Curves

Huseyin Hisil[1], Aaron Hutchinson[2], and Koray Karabina[2(✉)]

[1] Yasar University, İzmir, Turkey
[2] Florida Atlantic University, Boca Raton, USA
kkarabina@fau.edu

Abstract. This paper aims to answer whether *d*-MUL, the multidimensional scalar point multiplication algorithm, can be implemented efficiently. *d*-MUL is known to access costly matrix operations and requires memory access frequently. In the first part of the paper, we derive several theoretical results on the structure and the construction of the addition chains in *d*-MUL. These results are interesting on their own right. In the second part of the paper, we exploit our theoretical results, and propose an optimized variant of *d*-MUL. Our implementation results show that *d*-MUL can be very practical for small d, and it remains as an interesting algorithm to further explore for parallel implementation and cryptographic applications.

Keywords: *d*-MUL · Elliptic curve scalar multiplication
Differential addition chain · Isochronous implementation

1 Introduction

Let G be an abelian group of order $|G| = N$. A *single point multiplication algorithm* in G takes as input $a \in [1, N)$ and $P \in G$ and outputs the point aP. More generally, a *d-dimensional point multiplication algorithm* in G takes $a_1, \ldots, a_d \in [1, N)$ and points $P_1, \ldots, P_d \in G$ and produces the point $a_1P_1 + \cdots + a_dP_d$. Secure and efficient point multiplication algorithms are critical in cryptography and have received much attention over the years. Some of these algorithms are specialized for certain tasks or only work in specific settings, such as when the P_i are fixed versus variable, or when the scalars a_i are public versus secret. See [1,3–5,7,8,10] for examples of such algorithms. In some cases, a linear combination $a_1P_1 + \cdots + a_dP_d$ must be computed for a_i chosen uniformly at random.

In particular, the *d*-MUL algorithm in [4,7] is a multidimensional point multiplication algorithm which offers uniform operations in its execution, differential additions with each point addition, and has potential for an isochronous implementation. The *d*-MUL paper [7] notes that fixing the dimension parameter d

A. Chattopadhyay et al. (Eds.): SPACE 2018, LNCS 11348, pp. 198–217, 2018.
https://doi.org/10.1007/978-3-030-05072-6_12

as 1 yields the Montgomery chain [9], while taking $d = 2$ yields the chain given by Bernstein in [2]. d-MUL takes advantage of state matrices in its underlying structure.

Definition 1. *A $(d+1) \times d$ **state matrix** A has non-negative entries and satisfies:*

1. *each row A_i has $(i-1)$ odd entries.*
2. *for $1 \leq i \leq d$, we have $A_{i+1} - A_i \in \{e_j, -e_j\}$ for some $1 \leq j \leq d$, where e_j is the row matrix having 1 in the jth column and 0's elsewhere.*

*We define the **magnitude** of A to be $|A| = \max\{|A_{ij}| : 1 \leq i \leq d+1, 1 \leq j \leq d\}$.*

At a high level, on input $a_1, \ldots, a_d \in \mathbb{Z}$ and $P_1, \ldots, P_d \in G$ the d-MUL algorithm, as described in [7], consists of three stages:

(1) construct a $(d+1) \times d$ state matrix A having a row consisting of the scalars $a_1, \ldots, a_d$,
(2) construct a sequence $\{A^{(i)}\}_{i=1}^{\ell}$ of state matrices such that $A^{(\ell)} = A$, the entries of $A^{(1)}$ are in the set $\{0, 1, -1\}$, and every row of $A^{(i+1)}$ is the sum of exactly two (possibly not distinct) rows from $A^{(i)}$,
(3) Compute the linear combinations Q_i of the $P_1, \ldots, P_d$ corresponding to the rows of $A^{(1)}$, and then use the row relations among the consecutive $A^{(j)}$ to add pairs of Q_i's together until reaching the final matrix $A^{(\ell)}$.

Suppose that one wishes to compute a random linear combination of $P_1, \ldots, P_d$ such that the coefficients of the combination are scalars having ℓ bits or less. One approach is to choose d many scalars a_i from the interval $[0, 2^\ell)$ and run the d-MUL algorithm with input $a_1, \ldots, a_d$ and $P_1, \ldots, P_d$. This method has some concerning drawbacks:

- A large amount of integer matrix computation is necessary in item (2) above before any point additions can be performed.
- A large amount of storage space is necessary to store the matrices $A^{(i)}$, which each consist of $d(d+1)$ many i bit integers.

In an effort to avoid these drawbacks, one might instead consider a variant of d-MUL which starts with a state matrix $A^{(1)}$ having entries in $\{0, 1, -1\}$ and builds a random sequence $\{A^{(i)}\}_{i=1}^{\ell}$ as in (2) above. In this setting, point additions can begin immediately alongside the generation of the matrices, which also reduces the storage space on the matrix sequence to that of just a single $(d+1) \times d$ state matrix. This new procedure still comes with some concerns, such as how to build such a random sequence of state matrices, and how to ensure that the final output point isn't biased in any way. This idea is the primary focus of this paper.

Contributions: Our contributions can be summarized as follows:

- We present a variant of d-MUL, the multidimensional differential addition chain algorithm described in [7]. The algorithm takes as input points $P_1, \ldots, P_d$, a $(d \cdot \ell)$-bit string r, a permutation $\sigma \in S_d$, and a binary vector v of length d, and it produces a point P from the set $S = \{a_1P_1 + \cdots + a_dP_d : 0 \leq a_i < 2^\ell\}$.
- The algorithm is performed as d-MUL, except that the scalar coefficients are not chosen in advance, and no matrix arithmetic is required prior to point addition; instead, the row additions are chosen through r. Moreover, the scalars a_i for the output point $P = a_1P_1 + \cdots + a_dP_d$ can be determined through matrix arithmetic either before or after the point P is computed. In particular, our algorithm maintains the uniform pattern of

$$1 \text{ doubling}, d \text{ additions}, 1 \text{ doubling}, d \text{ additions}, \ldots$$

  that d-MUL features.
- We prove that there is a uniform correspondence between the input parameters (r, σ, v), and the scalars $a_1, \ldots, a_d$ determining P. More precisely, if r, σ, and v are chosen uniformly at random, then the point P will be uniformly distributed in S.
- We make some observations to modify and speed up the original algorithm, resulting in a constant-time friendly description. In particular, our algorithm can still benefit from parallelization, and we reduce the storage requirements for the scalar computation algorithm from $O(d^2)$ (the cost of storing a state matrix) to $O(1)$ (a single scalar).
- We report on results from implementations of our algorithm. Initial constant-time implementations of the algorithm gave run times nearing 500 000 cycle counts; with the modifications mentioned in the previous bullet this was reduced to 97 300 for $d = 2$, 109 800 for $d = 3$, and 123 300 for $d = 4$ with constant-time operations and differential additions used in each case (combined cost of computing P and $(a_1, \ldots, a_d)$ in each case).

The rest of this paper is organized as follows. Section 2 provides new theoretical results and builds the foundation of the scalar multiplication algorithm. Section 3 details the scalar multiplication algorithm which produces a random linear combination of points. Section 4 shows implementation oriented optimization alternatives on the proposed algorithms. Section 5 reports on the results of these optimizations in an implementation. We derive our conclusions in Sect. 6.

2 Theoretical Results

This section is devoted to developing theoretical results that culminate in our scalar multiplication algorithm. The outline of this section is as follows. Let A and B be state matrices such that every row in A is the sum of two rows from B.

1. We prove Theorem 1: for a fixed A, the matrix B is unique.

2. We prove Lemma 6: for a fixed B, there are exactly 2^d distinct matrices A.
3. In proving Lemma 6 we gain insight on how to construct all such matrices A from a given B: the matrices A are in one-to-one correspondence with binary strings r of length d. We formalize the construction of A from a given B and binary string r.
4. In Theorem 4 we show that iterating the construction of A from B with a bitstring r chosen randomly at each iteration will produce a uniformly random integer row vector $\begin{bmatrix} a_1 \cdots a_d \end{bmatrix}$ with $0 \leq a_i < 2^\ell$ for $0 \leq i \leq d$. This results in a version of d-MUL which produces a random output with negligible precomputation.

Throughout this section, row and column indices of all matrices start at 1.

2.1 Uniqueness

In this section, we aim to show that the output of Algorithm 2 in [7] is unique in the following sense: if A and B are $(d+1) \times d$ state matrices (defined below) such that every row in A is the sum of two rows from B, then B is the output of Algorithm 2 in [7] when ran with input A. The proof will be by induction on d, but we first prove several lemmas and corollaries which are required in the main proof. Proofs of some of these lemmas and corollaries, when they are rather mechanical, are omitted due to space restrictions. The results of this section will be used in Sect. 3 to attain an algorithm for generating a uniformly random linear combination of group elements.

Throughout this paper, we will be working with the notion of a state matrix as in Definition 1. This is the same definition used in [7], but we restrict to state matrices with non-negative entries. A simple consequence of Definition 1 is that each index j in property (2) above is obtained from a unique i.

Lemma 1. *Let A be a state matrix. If $A_{m+1} - A_m = c_i e_i$, and $A_{n+1} - A_n = c_j e_j$ with $m \neq n$, then $i \neq j$. As a consequence, each column of A has the form*

$$\begin{bmatrix} 2x \cdots 2x\ 2x + (-1)^k \cdots 2x + (-1)^k \end{bmatrix}^T$$

for some k and some x, where the index at which $2x$ changes to $2x + (-1)^k$ is different for each column.

Remark 1. Since every column of a state matrix A has the form stated in Lemma 1, $|A|$ can be computed by only looking at the rows A_1 and A_{d+1}.

Definition 2. *Let A be a $(d+1) \times d$ state matrix. The **column sequence** for A is the function $\sigma_A : \{2, \ldots, d+1\} \to \{1, \ldots, d\}$, where $\sigma_A(i)$ is the position in which the row matrix $A_i - A_{i-1}$ is nonzero. When A is clear from the context, we will sometimes write σ instead of σ_A. By Lemma 1, σ_A is a bijection.*

Definition 3. *Let A be a $(d+1) \times d$ state matrix. The **difference vector** for A is the vector $c^A := A_{d+1} - A_1$. When A is clear from the context, we will sometimes write c instead of c^A.*

With these definitions, we have $A_k - A_{k-1} = c^A_{\sigma_A(k)} e_{\sigma_A(k)}$ for $2 \leq k \leq d+1$. Next, we formulate results on the number of odd entries in the sum and difference of two rows from a state matrix.

Lemma 2. *If B is a $(d+1) \times d$ state matrix, then $B_m + B_n$ has $|m-n|$ odd entries.*

The following simple corollary will be used extensively throughout the rest of the paper.

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 each k, there is some m such that $A_k = B_m + B_{m+k-1}$.*

Proof. Write $A_k = B_m + B_n$, with $m \leq n$. Property (2) of state matrices says that A_k has $k-1$ odds. By Lemma 2, $B_m + B_n$ has $n-m$ odds. So $k-1 = n-m$ and $n = m+k-1$.

Corollary 2. *Let A and B be state matrices such that every row in A is the sum of two rows from B. Let h be the number of odds in the integer row matrix $\frac{1}{2}A_1$. Then $2B_{h+1} = A_1$.*

Proof. By Corollary 1 we have $A_1 = 2B_m$ for some index m. By assumption, B_m has h odd entries. By the definition of a state matrix, B_m has $m-1$ odd entries. So $m = h+1$ and $A_1 = 2B_{h+1}$.

Lemma 3. *If B is a $(d+1) \times d$ state matrix, then $B_m - B_n$ has*

(1) $|m-n|$ odd entries, all of which are either 1 or -1,
(2) $d - |m-n|$ even entries, all of which are 0.

We now show that we can write c^B as a function of c^A when every row in A is the sum of two rows from B.

Lemma 4. *Let A and B be state matrices such that every row in A is the sum of two rows from B. Write $A_1 = \begin{bmatrix} 2\alpha_1 & \cdots & 2\alpha_d \end{bmatrix}$. Then*

$$c^B_j = \begin{cases} c^A_j & \text{if } \alpha_j \text{ is even} \\ -c^A_j & \text{if } \alpha_j \text{ is odd} \end{cases}$$

We can also relate σ_A and σ_B. An explicit formula for σ_A in terms of σ_B can be found, but for our purposes only knowing $\sigma_A(2)$ suffices. The following lemma will be one of the keys to proving Theorem 1 to follow.

Lemma 5. *Let A and B be state matrices such that every row in A is the sum of two rows from B. Write $A_1 = \begin{bmatrix} 2\alpha_1 & \cdots & 2\alpha_d \end{bmatrix}$ and let h be the number of α_i which are odd. Then*

$$\sigma_A(2) = \begin{cases} \sigma_B(h+1) & \text{if } \alpha_{\sigma_A(2)} \text{ is odd} \\ \sigma_B(h+2) & \text{if } \alpha_{\sigma_A(2)} \text{ is even} \end{cases}$$

We now have all the tools required to prove the main result of this subsection.

Theorem 1. *Let A be any state matrix of size $(d+1) \times d$. Then there is a unique state matrix B such that every row in A is the sum of two rows from B. In particular, Algorithm 2 in [7] gives a construction for B.*

Proof. We use induction on d. Let $d = 1$ and suppose B is such a matrix. Then B has only two rows, one of which is determined uniquely by Corollary 2. By Corollary 1, we have $A_2 = B_1 + B_2$; two of the three row matrices in this equation are determined already, and so the third is determined as well.

Assume the theorem holds for matrices of size $d \times (d-1)$. Let A be a $(d+1) \times d$ state matrix and suppose B and C satisfy the condition stated in the theorem. Write $A_1 = \begin{bmatrix} 2\alpha_1 & \cdots & 2\alpha_d \end{bmatrix}$ and let h be the number of α_i which are odd. Throughout the rest of the proof, for any matrix X we will let ${}^{i}[X]^{j}$ denote the matrix obtained by deleting the ith row and jth column of X. Let $A' = {}^{1}[A]^{\sigma_A(2)}$. That A' is a state matrix follows from A being a state matrix and that the only odd in A_2 occurs in column $\sigma_A(2)$.

1. Suppose $\alpha_{\sigma_A(2)}$ is odd. By Lemma 5, we have $\sigma_B(h+1) = \sigma_C(h+1) = \sigma_A(2)$. Let $B' = {}^{h+1}[B]^{\sigma_B(h+1)}$ and $C' = {}^{h+1}[C]^{\sigma_C(h+1)}$.
   We'll now show that B' is a state matrix. For $2 \le i \le h$, we have

   $$B'_i - B'_{i-1} = [B]_i^{\sigma_B(h+1)} - [B]_{i-1}^{\sigma_B(h+1)} = [B_i - B_{i-1}]^{\sigma_B(h+1)} = [c^B_{\sigma_B(i)} e_{\sigma_B(i)}]^{\sigma_B(h+1)}$$

   which is still a unit basis row matrix since $\sigma_B(h+1) \neq \sigma_B(i)$. Similarly, for $h+2 \le i \le d+1$ we have

   $$B'_i - B'_{i-1} = [B]_{i+1}^{\sigma_B(h+1)} - [B]_i^{\sigma_B(h+1)} = [B_{i+1} - B_i]^{\sigma_B(h+1)} = [c^B_{\sigma_B(i+1)} e_{\sigma_B(i+1)}]^{\sigma_B(h+1)}$$

   (the row index increases by one to account for the deleted row $h+1$) which is still a unit basis row matrix since $\sigma_B(h+1) \neq \sigma_B(i+1)$. Looking at $i = h+1$, we have

   $$\begin{aligned} B'_{h+1} - B'_h &= [B]_{h+2}^{\sigma_B(h+1)} - [B]_h^{\sigma_B(h+1)} \\ &= [B_{h+2}]^{\sigma_B(h+1)} - [B_h]^{\sigma_B(h+1)} \\ &= [B_h + c^B_{\sigma_B(h+2)} e_{\sigma_B(h+2)} + c^B_{\sigma_B(h+1)} e_{\sigma_B(h+1)}]^{\sigma_B(h+1)} - [B_h]^{\sigma_B(h+1)} \\ &= [c^B_{\sigma_B(h+2)} e_{\sigma_B(h+2)}]^{\sigma_B(h+1)} + [c^B_{\sigma_B(h+1)} e_{\sigma_B(h+1)}]^{\sigma_B(h+1)} \\ &= [c^B_{\sigma_B(h+2)} e_{\sigma_B(h+2)}]^{\sigma_B(h+1)} + 0. \end{aligned}$$

   So B' satisfies the second requirement of being a state matrix. For the first requirement involving parities, we note that $B_{i,\sigma_B(h+1)} = \alpha_{\sigma_B(h+1)} + c^A_{\sigma_B(h+1)}$ (which is even) for $1 \le i \le h+1$ and $B_{i,\sigma_B(h+1)} = \alpha_{\sigma_B(h+1)}$ (which is odd) for $h+2 \le i \le d+1$. So for $1 \le i \le h+1$, $[B]_i^{\sigma_B(h+1)}$ is obtained from B_i by deleting an even entry, and so the number of odds isn't affected. Similarly, for $h+1 \le i \le d$, $[B]_i^{\sigma_B(h+1)}$ is obtained from B_{i+1} by deleting an odd entry, and so has $i-1$ odds. This shows B' is a $d \times (d-1)$ state matrix.

We now show that every row in A' is the sum of two rows from B'. We have

$$A'_i = [A]_{i+1}^{\sigma_A(2)} = [A_{i+1}]^{\sigma_A(2)} = [B_j + B_{j+i}]^{\sigma_B(h+1)} = [B]_j^{\sigma_B(h+1)} + [B]_{j+i}^{\sigma_B(h+1)}$$

for some index j. If neither j or $j+i$ are $h+1$, then both the above row matrices correspond to rows of B'. If one is $h+1$, we just see that

$$[B]_{h+1}^{\sigma_B(h+1)} = [B_h + c^B_{\sigma_B(h+1)} e_{\sigma_B(h+1)}]^{\sigma_B(h+1)} = [B_h]^{\sigma_B(h+1)} = B'_h.$$

Thus B' is a $d \times (d-1)$ state matrix such that every row in A' is the sum of two rows from B'.

An entirely identical argument shows C' is a $d \times (d-1)$ state matrix such that every row from A' is the sum of two rows from C'. Our inductive hypothesis gives that $B' = C'$. We already have $B_{h+1} = \frac{1}{2}A_1 = C_{h+1}$ from Corollary 2. Since $B_{h+1,\sigma_B(h+1)} = C_{h+1,\sigma_C(h+1)}$ and $\sigma_B(h+1) = \sigma_C(h+1)$, we have that column $\sigma_B(h+1)$ is identical in both matrices by Lemma 1. Thus $B = C$.

2. Suppose $\alpha_{\sigma_A(2)}$ is even. The proof is mostly identical to case 1. We get $\sigma_B(h+2) = \sigma_C(h+2) = \sigma_A(2)$ by Lemma 5 and take $B' = {}^{h+2}[B]^{\sigma_B(h+2)}$ and $C' = {}^{h+2}[C]^{\sigma_C(h+2)}$.

To wrap up this section, we prove one additional corollary which will be needed later on.

Corollary 3. *Let A and B be state matrices such that every row in A is the sum of two rows from B. If $A_k = B_m + B_{m+k-1}$ and also $A_k = B_n + B_{n+k-1}$, then $m = n$.*

2.2 Generating Randomness

The task of generating random group elements has many applications in cryptography, most notably in the first round of the Diffie-Hellman key agreement protocol. We will now make use of the results in Subsect. 2.1 to tackle the problem of choosing and computing an element from the set $\{a_1P_1 + \cdots + a_dP_d : 0 \le a_i < 2^\ell\}$ uniformly at random for a fixed set of points P_i in an abelian group G and for a fixed parameter ℓ. We of course would like to be as efficient and uniform with our intermediate computations as possible.

Many solutions to this problem exist already. One such solution is to choose $a_i \in [0, 2^\ell)$ uniformly at random for $1 \le i \le d$, and then run the d-MUL algorithm of [7] with the input $(a_1, \ldots, a_d)$. This method has the advantage of being uniform with all group operations (see Remark 5.1 in [7]), but comes with the overhead of first computing a sequence $\{A^{(i)}\}_{i=1}^{\ell}$ of $(d+1) \times d$ matrices before any group addition is done. Once the sequence has been computed the remaining computations within G are relatively efficient, requiring ℓ point doublings and $\ell \cdot d$ point additions.

We propose an alternative solution to the problem by considering a "reversed" version of d-MUL which bypasses the computation of this sequence of

matrices. Instead of choosing d many $a_i \in [0, 2^\ell)$, we choose a single $r \in [0, 2^{\ell d})$ uniformly at random, which will be used to construct a unique sequence $\{A^{(i)}\}_{i=1}^{\ell}$ which we will utilize in the same way as above to perform the group computations. Taking all such sequences $\{A^{(i)}\}_{i=1}^{\ell}$ corresponding to every $r \in [0, 2^{\ell d})$, the distribution of all integer d-tuples corresponding to all rows in the final matrices $A^{(\ell)}$ are *not* uniform; however by only considering the final rows $A_{d+1}^{(\ell)}$ we find an output which is uniform over all odd integer d-tuples, which we state as a main result in Theorem 3. By subtracting a binary vector from the output, we find a uniformly random d tuple.

In this subsection, we define the tools used to explore the problem and prove many results, culminating in Theorem 4 which gives a method for producing uniformly random output. These results will be used in Sect. 3 to give an algorithm for computing a uniformly random linear combination of group elements.

We now find interest in sequences of state matrices having special properties, as described in the following definition.

Definition 4. *A **state matrix chain** is a sequence $\left(A^{(i)}\right)_{i=1}^{\ell}$ of state matrices $A^{(i)}$ with d columns such that*

1. *each row of $A^{(i+1)}$ is the sum of two rows from $A^{(i)}$ for $1 \leq i < \ell$,*
2. *$\{|A^{(i)}|\}_{i=1}^{\ell}$ is a strictly increasing sequence,*
3. *$|A^{(1)}| = 1$.*

We say ℓ is the length of the chain $\left(A^{(i)}\right)_{i=1}^{\ell}$.

The sequence of matrices produced by Algorithm 3 in [7] is a state matrix chain. Note that a sequence $\left(A^{(i)}\right)_{i=1}^{\ell}$ satisfying (1) and (3) may be "trivially extended" to have an arbitrarily greater number of matrices by defining

$$B^{(i)} = \begin{cases} A^{(1)} & i \leq n \\ A^{(i-n)} & i > n \end{cases}$$

which is a sequence containing $\left(A^{(i)}\right)_{i=1}^{\ell}$ and still satisfying (1) and (3) of the above definition. We therefore attain some degree of uniqueness in excluding such trivial extensions from the current discussion by requiring (2). Note that by Theorem 1, a state matrix chain $\left(A^{(i)}\right)_{i=1}^{\ell}$ is uniquely determined by $A^{(\ell)}$.

Definition 5. *An **augmented** state matrix chain is a pair $\left(\left(A^{(i)}\right)_{i=1}^{\ell}, h\right)$, where $\left(A^{(i)}\right)_{i=1}^{\ell}$ is a state matrix chain with matrices having d columns and $1 \leq h \leq d+1$. h is called the output row for the augmented chain. Let SMC_d denote the set of all augmented state matrix chains (of varying length) with matrices having d columns. We define a function*

$$\begin{aligned} \text{OUTPUT} : SMC_d &\longrightarrow \mathbb{Z}^{1\times d} \\ \left(\left(A^{(i)}\right)_{i=1}^{\ell}, h\right) &\longmapsto A_h^{(\ell)}. \end{aligned}$$

The function OUTPUT (as with any function) naturally gives equivalence classes on its domain defined by the preimage of elements in the codomain; specifically, say augmented chains (C, h) and (C', h') are equivalent if and only if $\text{OUTPUT}(C, h) = \text{OUTPUT}(C', h')$. Since $\text{OUTPUT}(C, h) = A_h$ for some state matrix A, we have that $h-1$ is the number of odd entries in the row matrix A_h; likewise, $h'-1$ is the number of odd entries in the row matrix $A'_{h'}$, and since $A_h = A'_{h'}$ we have $h = h'$. That is, the output row is constant over all augmented state matrix chains in the equivalence class $[(C, h)]$. The length of the chains in $[(C, h)]$ is, in general, not constant.

Theorem 2. *For $s \in \mathbb{Z}^{1\times d}$ having h odd entries, we have*

$$|\text{OUTPUT}^{-1}(s)| = 2^d(d-h)!h!$$

That is, there are $2^d(d-h)!h!$ many state matrix chains which give s as an output.

Proof. By Theorem 1 the number of chains giving s as an output is equal to the number of state matrices containing s as a row. We count all such matrices.

Row $h+1$ must be s. For rows 1 through h, an odd entry must be selected to change to an even entry by either adding or subtracting 1, giving a total of $\prod_{i=1}^{h} 2i$ possibilities. Similarly, in choosing rows $h+2$ through $d+1$ an even entry must be changed to an odd entry by either adding or subtracting 1, giving a total of $\prod_{i=1}^{d-h} 2i$ possibilities. All together, we have

$$\left(\prod_{i=1}^{h} 2i\right) \cdot \left(\prod_{i=1}^{d-h} 2i\right) = 2^d(d-h)!h!$$

many possible matrices.

Note that for a fixed s the number of chains producing s as an output is independent of the bit size of the entries of s.

Lemma 6. *Let B be a $(d+1)\times d$ state matrix. Then there are exactly 2^d pairwise distinct state matrices A such that every row in A is the sum of two rows from B.*

Proof. Fix $0 \le h \le d$ and consider all matrices A such that $A_1 = 2B_{h+1}$ (every A has such a unique h by Corollary 2). By Corollary 3, for every k there are unique x_k and y_k such that $A_k = B_{x_k} + B_{y_k}$ with $x_k \le y_k$. This defines a sequence of pairs $a_k = (x_k, y_k)$ such that $a_1 = (h+1, h+1)$ and $a_{d+1} = (1, d+1)$. By Corollary 3 and Algorithm 2 in [7], we have either $a_{k+1} = (x_k - 1, y_k)$ or $a_{k+1} = (x_k, y_k + 1)$ for each k, and either choice for each k defines a valid and unique state matrix satisfying the conditions stated in the lemma. Since the x_k's must decrease to 1, we have h possible indices k to choose where to place the -1's in the first coordinates of a_{k+1}, and so $\binom{d}{h}$ sequences are possible. Summing over all h, we have $\sum_{h=0}^{d} \binom{d}{h} = 2^d$ total matrices.

The above proof gives insight into the method used in the algorithms to come. There is a one-to-one correspondence between the integers in the interval $[0, 2^d - 1]$ and the possible matrices A stated in the theorem. The number of 1's in the binary expansion of a chosen integer will determine h, and the placement of the 1's determines the positions to place the -1's in the sequence a_k defined in the proof. In particular, choosing an integer in the interval $[0, 2^d - 1]$ uniformly at random corresponds to choosing a matrix A uniformly at random out of all matrices satisfying the conditions in Lemma 6, and defines how to construct the chosen matrix A. We make this formal below.

Definition 6. *Let A and B be $(d+1) \times d$ 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 by $a_k = (x_k, y_k)$, where x_k and y_k are the unique row indices such that $A_k = B_{x_k} + B_{y_k}$.*

Remark 2. Uniqueness follows from Corollary 3.

Definition 7. *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 $2 \leq k \leq d+1$. We define the **extension matrix** of B corresponding to r as the $(d+1) \times d$ state matrix A having the addition sequence a_k with respect to the matrix B.*

By choosing ℓ many binary strings of length d, we may iterate Definition 7 to produce a sequence of matrices.

Definition 8. *Let B be a $(d+1) \times d$ state matrix and r a binary string of length $\ell \cdot d$. Let $r_1, \ldots, r_\ell$ be the partition of r into ℓ blocks of length d, with r_i being the sequence whose terms are bits $(i-1) \cdot d + 1$ through $i \cdot d$ of r. We define the extension sequence with base B corresponding to r as the sequence of $\ell + 1$ many $(d+1) \times d$ state matrices $\left(A^{(i)}\right)_{i=1}^{\ell+1}$ defined recursively as $A^{(1)} = B$ and $A^{(i+1)}$ is the extension matrix of $A^{(i)}$ corresponding to r_i.*

By the definition of an extension matrix, every row in $A^{(i)}$ is the sum of two rows from $A^{(i-1)}$ for each i. Note however that not all extension sequences are state matrix chains since $|B| = 1$ is not required, and even if this condition were satisfied many sequences would have B repeated many times (such as when r is the zero string) and so $|A^{(i)}|$ is not strictly increasing in such cases.

Corollary 4. *Fix a $(d+1) \times d$ state matrix B. Every sequence of $(d+1) \times d$ state matrices $\left(A^{(i)}\right)_{i=1}^{\ell+1}$ satisfying*

1. $A^{(1)} = B$,
2. *For* $1 < i \leq \ell + 1$, *every row in* $A^{(i)}$ *is the sum of two rows from* $A^{(i-1)}$

is an extension sequence with base B *corresponding to some binary sequence* r.

Proof. For each i, there is an addition sequence for $A^{(i)}$ corresponding to $A^{(i-1)}$. Concatenating these sequences yields the sequence r.

Since there are $2^{\ell d}$ binary strings of length ℓd, there are $2^{\ell d}$ extension sequences of a fixed matrix B. We now arrive at the primary result of this section.

Theorem 3. *Let* $\mathcal{B} = \{B \in \mathbb{Z}^{(d+1)\times d} : |B| = 1, B \text{ a state matrix}\}$. *Let*

$$\mathcal{S} = \left\{ \left(A^{(i)}\right)_{i=1}^{\ell+1} : \left(A^{(i)}\right)_{i=1}^{\ell+1} \text{ is an extension of some } B \in \mathcal{B} \right\}$$

Then the map OUTPUT $\left(\left(A^{(i)}\right)_{i=1}^{\ell+1}\right) = A_{d+1}^{(\ell+1)}$ *defines a* $2^d d!$*-to-*1 *correspondence from* $\mathcal{S}$ *to the set of row matrices of length* d *consisting of positive odd* ℓ*-bit or less integers.*

Proof. Let s be such a row matrix. By Theorem 2, there are $2^d d!$ distinct state matrix chains which give s as an output. Let $\left(A^{(i)}\right)_{i=1}^{n}$ be one such a state matrix chain. Then Theorem 4.3 in [7] and Theorem 1 above give $n \leq \ell + 1$, and since $|A^{(1)}| = 1$ the chain may be uniquely extended into a sequence in $\mathcal{S}$ while still having output s by defining

$$B^{(i)} = \begin{cases} A^{(i-\ell-1+n)} & \text{if } \ell + 1 - n < i \leq \ell + 1 \\ A^{(1)} & \text{if } 1 \leq i \leq \ell + 1 - n \end{cases}$$

This essentially says to repeat matrix $A^{(1)}$ sufficiently many times to get a sequence of length $\ell + 1$. This produces a valid extension since it corresponds to choosing the addition sequence on $A^{(1)}$ corresponding to the zero bitstring (in other words, doubling the first row of all 0's produces the same matrix since $|A^{(1)}| = 1$). This is possible for each chain $\left(A^{(i)}\right)_{i=1}^{n}$, and so each fixed s yields $2^d d!$ many distinct extension sequences in $\mathcal{S}$. There are $2^{(\ell-1)d}$ such row matrices s, and each gives $2^d d!$ extensions in $\mathcal{S}$ (different choices of s give disjoint sets of extensions since the last row in the last matrix of each extension is s) for a total of $2^{\ell d} d!$ extensions. Since there are $d!$ state matrices B satisfying $|B| = 1$ and $2^{\ell d}$ extensions for each B, this is exactly the size of S.

The implication of the above theorem is that choosing both a matrix B satisfying $|B| = 1$ and an integer in $[0, 2^{\ell d} - 1]$ uniformly at random defines a unique extension sequence of B, which in turn yields a row matrix s chosen uniformly at random from the set of row matrices consisting of ℓ-bit or less odd entries, which is given by the last row in the last matrix of the extension sequence. An arbitrary ℓ-bit row matrix may be obtained by choosing a binary vector uniformly at random and subtracting it from the output row s.

Theorem 4. *Choose the following parameters uniformly at random from their respective sets:*

1. *r a binary string of length ℓd,*
2. *B a $(d+1) \times d$ state matrix satisfying $|B| = 1$,*
3. *v a binary row matrix of length d.*

Let $\left(A^{(i)}\right)_{i=1}^{\ell+1}$ be the extension sequence with base B corresponding to r, and define $s = A_{d+1}^{(\ell+1)}$. Then $s - v$ is an element chosen uniformly at random from the set of row matrices of length d having entries in $[0, 2^\ell - 1]$.

Proof. By Theorem 3, s is chosen uniformly at random from the set of row matrices of length d consisting of odd ℓ-bit or less integers. For each s, define $T_s = \{s - v : v \text{ is a binary vector}\}$. Then $\{T_s : s \text{ has odd } \ell\text{-bit or less integers}\}$ is a partition of the set of row matrices of length d having entries in $[0, 2^\ell - 1]$. Choosing r and B together specify a T_s, and v further specifies an arbitrary vector.

It's worth noting that Theorem 3 (and so also Theorem 4) will not hold true when instead choosing $h = i$ for $i \neq d+1$ (i.e., $s = A_i^{(\ell+1)}$ for $i \neq d+1$ in the above). If s contains an entry which is 0 or 2^ℓ, many of the state matrices containing s as a row counted in Theorem 2 will necessarily contain the entry -1 (which we don't allow) or $2^\ell + 1$, and so the conditions of Theorem 4.3 in [7] will not be satisfied. In turn, producing such s containing $2^\ell + 1$ would take one additional iteration of Algorithm 3 in [7] and so yields an extension sequence of length $\ell + 2$. This results in sequences of varying lengths, which we wish to avoid.

3 Algorithms

Given Theorem 4, our scalar multiplication algorithm is now simple. Let $P = \begin{bmatrix} P_1 \dots P_d \end{bmatrix}$ be a row matrix of points and choose r, B, and v as in Theorem 4. B can be constructed by applying a random permutation to the columns of a lower triangular matrix whose lower triangle consists of all 1's. We then construct the extension sequence with base B corresponding to the bitstring r. The scalars in the linear combination of the output are given by subtracting v from the last row of the last matrix. These rules are reflected in Algorithm 1.

We perform the scalar multiplication by using the same addition and doubling rules specified by the binary string r on the point column matrix $B \cdot P^T$. Upon arriving at the final point column matrix, we subtract $v \cdot P^T$ from the last entry and return this point as an output. We remark that this process only yields a point; the coefficients of the resulting linear combination are computed through Theorem 4 using Algorithm 1. These rules for point multiplication are reflected in Algorithm 2.

The scalars in Algorithm 1 and the points in Algorithm 2 can be merged in order to scan r only a single time. Alternatively, the two algorithms can

Algorithm 1. d-MUL scalars

Input: bitsize parameter ℓ; bitstring r of length ℓd; τ a bijection on $\{0, 1, \ldots, d-1\}$; v a bitstring of length d
Output: a row matrix $\begin{bmatrix} a_1 & \cdots & a_d \end{bmatrix}$

$B[0] \leftarrow \begin{bmatrix} 0 & \cdots & 0 \end{bmatrix}$
for $i = 0$ **to** $d - 1$ **do**
  $B[i+1] \leftarrow B[i] + e[\tau[i]]$ `// Initial state matrix`
end
for $i = 0$ **to** $\ell - 1$ **do**
  $h, x, y \leftarrow r[i \cdot d] + \cdots + r[(i+1)d - 1]$
  $A[0] \leftarrow 2B[h]$
  for $j = 0$ **to** $d - 1$ **do**
    if $r[i \cdot d + j] = 1$ **then**
      $x \leftarrow x - 1$
    else
      $y \leftarrow y + 1$
    end
    $A[j+1] \leftarrow B[x] + B[y]$
  end
  $B \leftarrow A$
end
$a \leftarrow B[d] - v$
return a `// Scalars`

be computed independent of each other. We prefer the latter case because the column vectors of B in Algorithm 1 constitutes a redundant representation. We eliminate this redundancy in Sect. 4.

We take a moment to point out some special cases. As in the original d-MUL algorithm of [7], when taking $d = 1$ here we get a scalar multiplication algorithm reminiscent of the Montgomery chain [9]. To compute aP the Montgomery chain tracks two variables Q_1 and Q_2; each iteration adds these variables together and doubles one of them to get the next pair $\begin{bmatrix} Q_1, Q_2 \end{bmatrix}$. If the point to double is chosen uniformly at random in the Montgomery chain, one gets an algorithm identical to Algorithm 2.

Similarly, when $d = 2$ Algorithm 2 resembles a variant of the "new binary chain" of [2]. To compute $a_1P_1 + a_2P_2$ this chain tracks a triple of points (Q_1, Q_2, Q_3) which are linear combinations of P_1 and P_2 for which the scalars of each combination correspond to pairs from the set $S = \{(s,t), (s+1,t), (s,t+1), (s+1,t+1)\}$ for some (s,t). The missing pair from the set always contains exactly one odd entry. Suppose that Q_1 corresponds to the (even,even) tuple, Q_2 corresponds to the (odd,odd) tuple, and Q_3 is the mixed parity tuple. Then the triple (Q'_1, Q'_2, Q'_3) for the next iteration satisfies $Q'_1 = 2Q_i$ for some i, $Q'_2 = Q_1 + Q_2$, and either $Q'_3 = Q_1 + Q_3$ or $Q'_3 = Q_2 + Q_3$ such that the resulting scalars from each linear combination are still of the form given by S for a new pair (s', y'). This leaves $4 = 2^2$ options for (Q'_1, Q'_2, Q'_3) from a fixed triple

(Q_1, Q_2, Q_3), and so choosing an option at random is equivalent to Algorithm 2 when $d = 3$. See [2] for more details on the new binary chain.

Algorithm 2. Simplified d-MUL

Input: bitsize parameter ℓ; $P = \begin{bmatrix} P_1 & \cdots & P_d \end{bmatrix}$ points on curve E; bitstring r of length ℓd; τ a bijection on $\{0, 1, \ldots, d-1\}$; v a bitstring of length d

Output: Q satisfying $Q = a_1 P[1] + \cdots + a_d P[d]$ for $0 \leq a_i < 2^\ell$ chosen uniformly at random; the row matrix $\begin{bmatrix} a_1 & \cdots & a_d \end{bmatrix}$

```
1  Q[0] ← id(E)
2  for i = 0 to d − 1 do
3  |  Q[i + 1] ← Q[i] + P[τ[i]]                    // Initial state matrix
4  end
5  for i = 0 to ℓ − 1 do
6  |  h, x, y ← r[i · d] + ··· + r[(i + 1)d − 1]
7  |  R[0] ← 2Q[h]
8  |  for j = 0 to d do
9  |  |  if r[i · d + j] = 1 then
10 |  |  |  x ← x − 1
11 |  |  else
12 |  |  |  y ← y + 1
13 |  |  end
14 |  |  R[j + 1] ← Q[x] + Q[y]
15 |  end
16 |  Q ← R
17 end
18 T ← Q[d] − v[0] · P[0] − ··· − v[d − 1] · P[d − 1]
19 a ← d-MUL-Scalars(ℓ,r, τ, v)
20 return a, T                                    // Scalars and output point
```

The point addition $R[j+1] \leftarrow Q[x] + Q[y]$ at line 14 of Algorithm 2 can be implemented using a differential addition $Q[x] \oplus Q[y]$ if $Q[x] \ominus Q[y]$ is known in advance. Algorithm 3 computes a difference vector Δ which satisfies:

$$Q[x] \ominus Q[y] = \Delta[0] \cdot P[0] \oplus \ldots \oplus \Delta[d-1] \cdot P[d-1].$$

Using Δ it's possible to arrange a look-up function for difference points TBL from which the difference point is extracted $\texttt{TBL}(\Delta)$. We provide explicit construction of TBL for $d = 2, 3, 4$ in Sect. 4.

Algorithms 1, 2, and 3 use an implementation oriented notation where arrays are used in the place of vectors and the index of an array always starts from zero.

Algorithm 3. Simplified d-MUL with differential additions

Input: bitsize parameter ℓ; $P = \begin{bmatrix} P_1 & \cdots & P_d \end{bmatrix}$ points on curve E; bitstring r of length ℓd; τ a bijection on $\{0, 1, \ldots, d-1\}$; v a bitstring of length d; TBL a look-up function for difference points

Output: Q satisfying $Q = a_1 P[1] + \cdots + a_d P[d]$ for $0 \leq a_i < 2^\ell$ chosen uniformly at random; the row matrix $\begin{bmatrix} a_1 & \cdots & a_d \end{bmatrix}$

$Q[0] \leftarrow \mathrm{id}(E)$
for $i = 0$ **to** $d - 1$ **do**
  $Q[i+1] \leftarrow Q[i] + P[\tau[i]]$ `// Initial state matrix`
end
$\kappa^{\mathrm{col}} \leftarrow \tau$
$\kappa^{\mathrm{row}} \leftarrow [1 \colon i \in [0, \ldots, d-1]]$
for $i = 0$ **to** $\ell - 1$ **do**
  $h, x, y \leftarrow r[i \cdot d] + \cdots + r[(i+1)d - 1]$
  $\kappa^{\mathrm{row}'}, \kappa^{\mathrm{col}'}, \Delta \leftarrow [0 \colon i \in [0, \ldots, d-1]]$
  $R[0] \leftarrow 2Q[h]$
  for $j = 0$ **to** d **do**
    if $r[i \cdot d + j] = 1$ **then**
      $x \leftarrow x - 1$
      $\kappa^{\mathrm{row}'}[\kappa^{\mathrm{col}}[x]] \leftarrow -\kappa^{\mathrm{row}}[\kappa^{\mathrm{col}}[x]]$
      $\kappa^{\mathrm{col}'}[j] \leftarrow \kappa^{\mathrm{col}}[x]$
    else
      $\kappa^{\mathrm{row}'}[\kappa^{\mathrm{col}}[y]] \leftarrow \kappa^{\mathrm{row}}[\kappa^{\mathrm{col}}[y]]$
      $\kappa^{\mathrm{col}'}[j] \leftarrow \kappa^{\mathrm{col}}[y]$
      $y \leftarrow y + 1$
    end
    $\Delta[\kappa^{\mathrm{col}'}[j]] \leftarrow \kappa^{\mathrm{row}}[\kappa^{\mathrm{col}'}[j]]$
    $R[j+1] \leftarrow Q[x] \oplus Q[y]$ `//` $Q[x] \ominus Q[y] = \mathrm{TBL}(\Delta)$
  end
  $\kappa^{\mathrm{row}} \leftarrow \kappa^{\mathrm{row}'}$
  $\kappa^{\mathrm{col}} \leftarrow \kappa^{\mathrm{col}'}$
  $Q \leftarrow R$
end
$T \leftarrow Q[d] - v[0] \cdot P[0] - \cdots - v[d-1] \cdot P[d-1]$
$a \leftarrow d$-MUL-Scalars(ℓ,r, τ, v)
return a, T `// Scalars and output point`

4 Optimizations

Let A be the extension matrix of B corresponding to the bitstring r. Our first optimization involves simplifying the computation of A. We notice that the ith column of A is a function of only the ith column of B and the bitstring r and is independent of the other columns of B. This means that when computing an extension sequence $\left(A^{(i)}\right)_{i=1}^{\ell+1}$, the columns of $A^{(\ell+1)}$ can be computed one at a time, reducing storage costs to only a single column of a state matrix.

Furthermore, the columns of state matrices have a very strict form. Specifically, a column of a state matrix A looks like

$$\left[2x \cdots 2x\ 2x + (-1)^k \cdots 2x + (-1)^k\right]^T,$$

for some integer k. The representation of this column can take the much simpler form $\{2x, (-1)^k, i\}$, where i is the highest index for which the entry of the column is $2x$. This simple representation reduces storage costs further to only storing one large integer $2x$, one bit sign information $(-1)^k$, and one small integer i. In this direction, Algorithm 4 provides an optimized version of Algorithm 1.

We point out that by taking $k = d+1$ in Corollary 1 we always have $A_{d+1}^{(\ell+1)} = A_1^{(\ell)} + A_{d+1}^{(\ell)}$, and by the uniqueness in Corollary 3 this is always the case for any extension sequence. One might consider skipping the computation of $A^{(\ell+1)}$ and simply outputting $A_1^{(\ell)} + A_{d+1}^{(\ell)}$ instead of $A_{d+1}^{(\ell+1)}$ (and likewise with the point additions in Algorithm 2). In our implementation with differential additions we found it difficult to retrieve the point corresponding to the difference $A_1^{(\ell)} + A_{d+1}^{(\ell)}$ in a secure fashion. This approach is viable in an implementation which doesn't take advantage of differential additions. Furthermore, this means the final d bits of the bitstring r are unused. These bits may be used in place of the binary vector v if desired.

Algorithm 4. d-`MUL` scalars (Optimized)

Input: bitstring r of length ℓd, τ a bijection on $\{0, 1, \ldots, d-1\}$
Output: Array of scalars k corresponding to r and τ

for $i = 0$ **to** $d - 1$ **do**
  $k[i] \leftarrow 0, \quad \delta \leftarrow 1, \quad \texttt{index} \leftarrow i$
  for $j = 0$ **to** $d(\ell - 2)$ **by** d **do**
    $h \leftarrow r[j] + \ldots + r[j + d - 1]$
    $z \leftarrow \texttt{BoolToInt}(h > \texttt{index})$
    $k[i] \leftarrow 2(k[i] + \delta)$
    $\delta \leftarrow (1 - 2z) \cdot \delta$
    $q \leftarrow \texttt{index} + 1 - h, \quad a \leftarrow 0, \quad \texttt{index} \leftarrow -1,$
    $q \leftarrow \texttt{Select}(q, -q, \texttt{BoolToInt}(q > 0)) + z$
    for $t = 0$ **to** $d - 1$ **do**
      $a \leftarrow a + \texttt{Xnor}(z, r[j + t])$
      $\texttt{index} \leftarrow \texttt{Select}(t, \texttt{index}, \texttt{BoolToInt}((a == q) \wedge (\texttt{index} == -1)))$
    end
  end
  $k[\tau[i]] \leftarrow 2k[i] + \delta - r[(\ell - 1) \cdot d + \tau[i] - 1]$
end
return k `// Array of scalars`

Algorithm 4 uses two auxiliary functions: `BoolToInt` sends `true` to 1 and `false` to 0. The `Select` function sends the inputs to the first input or to the second input if the third input is false or true, respectively.

Algorithm 2 inputs an array of points which is denoted by P for simplicity. Algorithm 2 also inputs the same ordering v and the random bitstring r as Algorithm 4, and outputs k as the d scalars corresponding to v and r. Algorithm 2 also outputs the point $T = k[0] \cdot P[0] + \cdots + k[d] \cdot P[d]$. In Algorithm 2, the `if` statement (also given below on the left) can be eliminated as given below on the right. This latter notation sacrifices readability but it helps in simplifying the implementation.

if $r[i \cdot d + j] = 1$ **then**
  $x \leftarrow x - 1$
else
  $y \leftarrow y + 1$

$x \leftarrow x - r[i \cdot d + t]$
$y \leftarrow y - r[i \cdot d + t] + 1$

4.1 Constant Time

Algorithm 2 can be implemented to give a constant-time implementation regardless of whether regular additions or differential additions are used. For this purpose, both d and ℓ are fixed. The following additional modifications are also mandatory.

- A table look-up on P that depends on the secret information $\tau[i-1]$ is performed in Line 3 of Algorithm 2. This look-up is implemented with constant-time scanning method. We note that this scanning does not cause a performance bottleneck for small values of d since it is not a part of the main loop (lines 5–13).
- Additional secret dependent table look-ups are performed at Line 7 and Line 10 of Algorithm 2. These look-ups are also implemented with constant-time scanning method. However, this time, the scanning constitute a performance bottleneck. To minimize the performance penalty, the actual implementation in Sect. 5, further optimizes Algorithm 2 by removing the assignment $Q \leftarrow R$ in line 12 and letting the intermediate point array to oscillate between the arrays Q and R. The indexes that can occur for Q and R are given by the sequences $[0 \cdots (d-i)]$ and $[i \cdots d]$, respectively. These indexes are perfectly suitable for constant-time scanning method since they are linearly sequential.
- The table look-ups $Q[0]$ at Line 1, $Q[i], Q[i-1]$ at Line 3, and $Q[0]$, $Q[d]$, $P[0]$, $\ldots$, $P[d-1]$, $r[d(\ell-1)]$, $\ldots$, $r[d(\ell-1)+d-1]$ at Line 14 of Algorithm 2 do not depend on the secret information. Therefore, these lines can be implemented in the way they are written. On the other hand, each of $r[d(\ell-1)] \cdot P[i]$ is equal to either the identity element or $P[i]$. This intermediate values must be selected in constant item with `Select`. The `Select` function processes secret data in Algorithm 1 and therefore `Select` is implemented to run in constant-time.
- Line 15 of Algorithm 1 also requires a secret dependent assignment. The left hand side $k[\tau[i]]$ also requires constant-time scanning.

4.2 Differential Addition

The number of distinct values of Δ in Algorithm 3 increases exponentially with d. Nevertheless, `TBL` is manageable for small d. We investigate the fixed cases $d = 2, 3, 4$, separately, and the explicit table entries for $d = 2, 3$ (the case $d = 4$ is omitted due to space restrictions) are as follows:

- *Case* $d = 2$*:* The table size is 4. The first iteration selects out of the 2 points $[P_0, P_1]$; and the second iteration selects out of the 2 points $[P_0 - P_1, P_0 + P_1]$
- *Case* $d = 3$*:* The table size is 13. The first iteration selects out of the 3 points $[P_0, P_1, P_2]$; the second iteration selects out of the 6 points $[P_1 - P_2, P_1 + P_2, P_0 - P_1, P_0 - P_2, P_0 + P_2, P_0 + P_1]$; and the third iteration selects out of the 4 points $[P_0 - P_1 - P_2, P_0 - P_1 + P_2, P_0 + P_1 - P_2, P_0 + P_1 + P_2]$.

Computing Δ with the help of variables κ^{col}, κ^{row}, $\kappa^{\text{row}'}$, $\kappa^{\text{col}'}$ is considerably inefficient. In order to emulate Δ, we derived dedicated boolean functions for each of the cases $d = 2, 3, 4$. We refer to the implementation for these expressions. Our experience is that simplification of computing Δ is open to further investigation.

Since the iterations selects through sequential indexes, the look-ups can be implemented with constant-time scanning method in a subsequence of `TBL`. The overhead of constant-time scanning is not a bottleneck for $d = 2, 3$ but starts becoming one for $d > 3$.

5 Implementation Results

Sections 2 and 3 provided simplifications on d-MUL by eliminating all of the redundancies and Sect. 4 put d-MUL into an optimized format with low-level expressions. Our main aim in this section is to show that optimized d-MUL can be implemented to give fast implementations. We implemented the optimized d-MUL algorithm for $d = 2, 3, 4$ with point addition method being (i) differential; (ii) regular (i.e. non-differential). In all experiments, we used $\mathbb{F}_p^2 = \mathbb{F}(i)$ where $p = 2^{127} - 1$ and $i^2 = -1$. We did not exploit any endomorphism. We used Montgomery differential addition formulas [9] for (i) and twisted Edwards ($a = -1$) unified addition formulas in extended coordinates [6] for (ii). Since d-MUL is a generic multidimensional scalar point multiplication algorithm, one can switch to other possibly faster fields.

We used a single core of an i7-6500U Skylake processor, where the other cores are turned-off and turbo-boost disabled. GNU-gcc version 5.4.0 with flags `-m64 -O2 -fomit-frame-pointer` was used to compile the code. The code can be executed on any `amd64` or `x64` processor since we use the generic 64×64 bit integer multiplier. In all of 12 implementations, we used the constant-time method for the scalar generation since the elimination of the branches lead to a slightly faster implementation. Table 1 provides cycle counts for our non-constant time implementation.

As dimension increases, so does the memory access due to look-ups from Q and `TBL`. On the other hand, the number of additions decreases as d increases.

Table 1. Non-constant time implementation of optimized d-Dmul.

Implementation	Scalars	Point	Total
Regular, $d = 2$	9 100	135 900	145 000
Regular, $d = 3$	12 900	127 600	140 500
Regular, $d = 4$	10 700	125 200	135 900
Differential, $d = 4$	10 700	88 200	98 900
Differential, $d = 3$	12 900	84 600	97 500
Differential, $d = 2$	9 100	86 600	95 700

Therefore, there is a trade-off between the number of memory accesses and the number of point additions, depending on d. In case (i), the fastest dimension turns out to be $d = 2$ for overall computation. Profiling the code we see that the number of memory accesses is dominated by the selection of difference points from `TBL` for higher dimensions. In case (ii), the fastest dimension turns out to be $d = 4$ since no look-up occurs from `TBL`.

The variation between the cycle counts in the scalars column of Table 1 is easy to explain. In (i), each scalar is represented by 2 limbs when $d = 2$; 1 limb when $d = 4$. In both cases, almost all available bits are used and the adder circuit is utilized. The case $d = 2$ is slightly faster than $d = 4$ since less effort is spent on scanning r. The $d = 3$ is slower because 84-bit scalars are represented by 2 limbs and more effort is spent on scanning r.

Table 2 provides the cycle counts when all input-dependent computations and input-dependent table look-ups are emulated with arithmetic operations. These implementations run in constant time for all inputs.

Table 2. Constant time implementation of optimized d-Dmul.

Dimension	Scalars	Point	Total
Regular, $d = 2$	9 100	143 500	152 600
Regular, $d = 3$	12 900	135 300	148 200
Regular, $d = 4$	10 700	131 200	141 900
Differential, $d = 4$	10 700	112 600	123 300
Differential, $d = 3$	12 900	96 900	109 800
Differential, $d = 2$	9 100	88 200	97 300

We immediately see that the ranking does not change and switching to the constant-time setting does not constitute a big speed penalty.

6 Concluding Remarks

We presented several theoretical results on the structure and the construction of the addition chains in d-MUL. Using our theoretical results, which are interesting on their own right, we proposed an optimized version of d-MUL. Our implementation results show that the optimized d-MUL gains some significant speed ups. In particular, we were able to reduce the cycle counts of our initial isochronous implementation of the original d-MUL algorithm from nearly 500 000 to under 125 000 cycles.

Acknowledgements. The authors would like to thank reviewers for their comments and corrections. Research reported in this paper was supported by the Army Research Office 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 Army Research Office.

References

1. Azarderakhsh, R., Karabina, K.: Efficient algorithms and architectures for double point multiplication on elliptic curves. In: Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2 2016 (2016)
2. Bernstein, D.: Differential addition chains. Technical report (2006). http://cr.yp.to/ecdh/diffchain-20060219.pdf
3. Bos, J.W., Costello, C., Hisil, H., Lauter, K.: High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 331–348. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40349-1_19
4. Brown, D.: Multi-dimensional montgomery ladders for elliptic curves. ePrint Archive: Report 2006/220. http://eprint.iacr.org/2006/220
5. Costello, C., Longa, P.: Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the mersenne prime. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 214–235. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_10
6. Hisil, H., Wong, K.K.-H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 326–343. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89255-7_20
7. Hutchinson, A., Karabina, K.: Constructing multidimensional differential addition chains and their applications. J. Cryptogr. Eng. 1–19 (2017). https://doi.org/10.1007/s13389-017-0177-2
8. Joye, M., Tunstall, M.: Exponent recoding and regular exponentiation algorithms. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 334–349. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02384-2_21
9. Montgomery, P.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**, 243–264 (1987)
10. Subramanya Rao, S.R.: Three dimensional montgomery ladder, differential point tripling on montgomery curves and point quintupling on Weierstrass' and Edwards curves. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 84–106. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31517-1_5