Constructing Multidimensional Differential Addition Chains and their Applications

Aaron Hutchinson and Koray Karabina

April 8, 2017

Abstract

We propose new algorithms for constructing multidimensional differential addition chains and for performing multidimensional scalar point multiplication based on these chains. Our algorithms work in any dimension and offer some key efficiency and security features. In particular, our scalar point multiplication algorithm is uniform, it has high potential for constant time implementation, and it can be parallelized. It also allows trading speed for precomputation cost and storage requirements. These key features and our theoretical estimates indicate that this new algorithm may offer significant performance advantages over the existing point multiplication algorithms in practice. We also report some experimental results and verify some of our theoretical findings.

keywords: differential addition chains; side channel resistance; elliptic curves; scalar multiplication; cryptographic algorithms
subjclass: 94A60; 14G50

1 Introduction

Let $\mathbb{G}$ be an additive abelian group of order $|\mathbb{G}| = N$. In a typical cryptographic application, $\mathbb{G}$ would be chosen as a prime order group generated by an element $P \in \mathbb{G}$. For example, elliptic curve digital signature algorithm (ECDSA) and elliptic curve Diffie-Hellman key agreement protocol (ECDH) require a suitably chosen prime order subgroup $\mathbb{G}$ of an elliptic curve defined over a finite field. Efficient implementation of such cryptographic schemes are only possible through an efficient implementation of point multiplication algorithms. A *single point multiplication* algorithm in $\mathbb{G}$ takes as input a scalar $a \in [1, N)$ and a point $P \in \mathbb{G}$, and outputs aP. More generally, a *d dimensional point multiplication* algorithm in $\mathbb{G}$ (shortly, a *d-point multiplication algorithm*) takes as input a sequence of scalars $a_1, a_2, \ldots, a_d$, and a sequence of points $P_1, P_2, \ldots, P_d \in \mathbb{G}$, and outputs $\sum_{i=1}^{d} a_i P_i$. From security and efficiency point of views, there are two important cases to consider in a d-point multiplication algorithm: the use of public vs. secret scalars a_i, and the use of variable vs. fixed points P_i.

Public vs. secret scalars: Point multiplication algorithm in some cryptographic applications is performed using secret scalars. For example, in the signature generation algorithm in ECDSA, signer has to compute aP for some randomly chosen secret integer a and a domain parameter P. A part of the public key in Okamato's identification scheme [24] is generated by computing $a_1P_1 + a_2P_2$ for a secret pair of integers (a_1, a_2), and randomly chosen points P_1 and P_2. In ECDH, aP is computed for a secret scalar a. Moreover, it is known that 1-point algorithm $(a, P \to aP)$ may be performed more efficiently using a d-point multiplication algorithm by rewriting $aP = \sum_{i=1}^{d} a_iP_i$, where a_i $(|a| \approx N, |a_i| \approx N^{1/d})$ and P_i are computed as a function of a and P, respectively (see [12, 11, 14] for $d = 2$, [29] for $d = 3$, [19, 13] for $d = 4$, and [6] for $d = 8$.).

In some other cryptographic applications, point multiplication algorithm is performed using publicly known scalars. For example, in the signature verification algorithm on ECDSA, verifier computes $a_1P_1 + a_2P_2$, where a_i are derived as a function of the message and the signature, P_1 is a domain parameter, and P_2 is the public key of a signer. Signature verification can significantly be improved when the signature verification equation is replaced by checking whether $\sum_{i=1}^{d} a_iP_i$ (for some $d \in \{4, 6\}$) is the identity element for some suitably constructed publicly known scalars a_i and points P_i; see [1].

Performing point multiplication algorithm with respect to a secret sequence of scalars is more challenging from a security point of view because power analysis attacks may recover some information about the scalars when the execution of the underlying multiplication algorithm can be distinguished based on the algebraic structure of the scalars (e.g. the size of a_i, or the number of zeros in the binary representation of a_i). Therefore, if the scalars a_i are secret, then underlying point multiplication algorithm should perform a uniform sequence of operations and output $\sum a_iP_i$ in constant time. This provides some level of resistance against simple power analysis attacks.

Variable vs. fixed points: In certain cases in the computation of $\sum a_iP_i$, the base points P_i are fixed (i.e. known in advance), whereas in some other cases P_i are variable (received from other parties or generated on the fly). Considering some of our earlier examples, signer in ECDSA computes aP for a fixed domain parameter P. Communicating parties in ECDH have to compute aP for a fixed point P, but they also have to compute aQ for a variable point Q sent by other parties. The base points P_i introduced in the efforts for accelerating ECDSA verification can possibly be considered as fixed because they can be included in the signer's certificate.

Performing point multiplication algorithm with respect to a variable basis is more challenging from an efficiency point of view because, when P_i are fixed, one may precompute and store a certain number of extra points to speed up the algorithm. For a concrete example, consider computing aP for a scalar $a \in [1, N)$ and a point $P \in \mathbb{G}$. Let $m = \lfloor N^{1/2} \rfloor$. Using the Euclidean algorithm one can write $a = a_1 + a_2m$ for $a_i \approx m$, and so aP can be computed as $a_1P_1 + a_2P_2$, where $P_1 = P$ and $P_2 = mP$. If P is a fixed point, then P_2 can be precomputed and, using Straus-Shamir's simultaneous double point multiplication algorithm (see Algorithm 14.88 in [20]), one can save half of the point doubling $(Q \to 2Q)$

operations and save one fourth of the point addition $(P, Q \rightarrow P + Q)$ operations in a naive comparison with a traditional double-and-add type algorithm. The efficiency gain would not be significant if $P_2 = mP$ had to be computed each time because computing P_2 takes about half the time of computing aP. On the other hand, we should note that some of the groups $\mathbb{G}$, such as (hyper)elliptic curve groups with efficiently computable endomorphisms, allow efficient decomposition of a scalar a into a_i, and efficient computation of P_i from a point P so that multidimensional point multiplication techniques still yield significant speed ups when computing $aP = \sum a_i P_i$ for a variable point P; see [12, 11, 14, 29, 19, 13, 6].

As a result, it has been very motivating to design efficient and secure (constant-time) multidimensional point multiplication algorithms, and to evaluate their performance over cryptographically interesting groups (i.e. binary/prime (hyper)elliptic curves) in software and hardware; see [26, 3, 8, 9, 6, 2, 13] for some recent work in this field.

Contributions: We investigate multidimensional differential addition chains and multidimensional point multiplication algorithms derived from these chains. Our contributions are summarized as follows:

- We describe a d-point multiplication algorithm d-MUL (Algorithm 3) that builds upon a new construction of a d-dimensional differential addition chain. d-MUL takes as input a sequence of integers $a_1, a_2, \ldots, a_d$ and points $P_1, P_2, ..., P_d$, $d \geq 1$; constructs a differential addition chain for its input $a_1, ..., a_d$ as a subroutine; and computes $\sum_{i=1}^{d} a_i P_i$. d-MUL has the following key properties:

  1. The algorithm d-MUL has a uniform structure (see Section 5) and high potential for constant-time implementation with respect to the number of bits of the input. For an input of ℓ-bit scalars a_i and points P_i, $\sum_{i=1}^{d} a_i P_i$ can be computed in exactly ℓ steps, where each step requires d point additions and a single point doubling. All the addition and doubling operations in d-MUL can be performed in parallel at each step. See Remark 5.1 and Section 5.2 for more details.
  2. The algorithm d-MUL allows trading speed for precomputation cost and storage requirements. If differential addition formulas are utilized in d-MUL and if the cost of negating a point is negligible in the underlying group, then d-MUL requires a precomputation of $(3^d - 2d - 1)/2$ point additions and a storage of $(3^d - 1)/2$ points in a lookup table. If traditional addition formulas are utilized in d-MUL, then it requires a precomputation of only $(d - 1)$ point additions. A lookup table is not required in this case. Note that the second case may make d-MUL more attractive over other existing d-point multiplication algorithms as they all seem to require a lookup table of exponential size (exponential in d); see for example [6]. See Table 1 and Section 5.2 for more details.

- We present a concrete cost analysis of d-MUL for some cryptographically interesting scenarios including (hyper)elliptic curve point multiplication using d-dimensional GLV/GLS decomposition technique for $d = 2, 3, 4, 8$. Our analysis shows that d-MUL

offers significant performance advantages over some recent point multiplication algorithms and their implementations [17, 8, 6, 26]. For $d = 2$, d-MUL is equivalent to the 2-dimensional differential addition chain construction by Bernstein [4]. For $d = 3$, we show that d-MUL saves at least one point addition at each step in the main loop of the algorithm in comparison with the 3-dimensional Montgomery ladder algorithm in [26]. This improves the efficiency of the 3 dimensional point multiplication algorithm in [26] up to 21% when the group arithmetic is performed over a Montgomery curve defined over large characteristic finite fields; see Table 2. Another advantage of 3-MUL over the one in [26] is that 3-MUL is uniform whereas the one in [26] is not. For $d = 4$, we estimate that if d-MUL is fully parallelized then it offers up to 18% speed-up over the fully parallelized 4-point multiplication algorithm by Joye and Tunstall [17] at the 128-bit security level; see Table 4. For $d = 8$, we show that d-MUL can reduce the precomputation and storage requirements in the implementation of the 8-dimensional point multiplication algorithm in [6] from computing and storing 32 points down to 7 points. This may be particularly advantageous for implementing curve arithmetic in resource restricted devices.

- Finally, we confirm our theoretical findings on the increasing efficiency of our algorithm d-MUL for larger values of d through our experimental results. In our prototype implementation, we work with a $n = 252$-bit prime order subgroup $\mathbb{G}$ of a Montgomery curve $y^2 = x^3 + 486662x^2 + x$ defined over the prime field of size $2^{255} - 19$, and compute $\sum_{i=1}^{d} a_i P_i$ for $d = 1, 2, 3, 4$ and scalars a_i each of which is $\ell = \lfloor n/d \rfloor$-bit. See Table 5 and Section 5.2 for more details.

The rest of this paper is organized as follows. In Section 2, we provide a survey of some known d-point multiplication algorithms. In Section 3, we describe two algorithms towards constructing d-dimensional differential addition chains. We describe our d-point multiplication algorithm d-MUL in Section 4, and provide a complexity analysis and comparisons in Section 5. We provide a toy example in Appendix A and conclude in Section 6.

2 Related Work

In the following, we provide a survey of some known d-point multiplication algorithms focusing more on the ones with uniform structure. First, we set some notation and state some of our assumptions.

Notation and assumptions: The underlying group $\mathbb{G}$ is always assumed to be an additive group of prime order N represented by n-bits. For simplicity, we assume that all scalars a_i involved in the computation of $\sum a_i P_i$ are ℓ-bit positive scalars unless otherwise stated[1]. In

[1]Accurate analysis of algorithms should consider varying length inputs a_i because certain properties (correctness, constant-time, etc.) should ideally be independent of the input size. We do consider this general case in the paper when necessary.

typical cryptographic applications, it is common to have $\ell \approx n$ (e.g. signature verification), or $\ell \approx n/d$ (e.g. speeding up aP by rewriting $aP = \sum_{i=1}^{d} a_i P_i$). We assume that one can always rewrite $aP = \sum_{i=1}^{d} a_i P_i$, where a_i and P_i are efficiently derived from a and P[2]. Throughout this paper, we call this technique the *d-decomposition* technique.

A survey of algorithms: A d-point multiplication $\sum_{i=1}^{d} a_i P_i$ can be achieved after performing d of 1-point multiplications $a_i P_i$, and summing them over. If parallelization is not an option, then this is not a very interesting method because the cost of computing aP with an n-bit scalar would be comparable to the cost of computing $\sum_{i=1}^{d} a_i P_i$ after deploying the d-decomposition technique, and performing d of 1-point multiplications in a row with n/d bit scalars a_i. A potentially more efficient method is to compute $\sum_{i=1}^{d} a_i P_i$ simultaneously. Straus-Shamir's trick (see Algorithm 14.88 in [20]), interleaving [21], and comb [18] methods are three such popular methods. Unfortunately, these techniques do not yield constant-time point multiplication algorithms mainly because the number of point additions strictly depends on the number of non-zero elements in the binary (more generally, q-ary or width-w NAF) representation of the scalars. These techniques have been modified with a *protected recoding* of the scalars that allows constant-time implementations [25, 15, 10, 17]. The signed digit recoding algorithm in [17] yields a uniform d-point multiplication algorithm, that we call the JT algorithm. The JT algorithm, with the choice of a parameter $m = 2^k$, requires a precomputation (and storage) of dm points at a cost of $2d(m/2 - 1) + (d - 1)$ additions and $2d$ doublings[3]. After that, $\sum_{i=1}^{d} a_i P_i$, for ℓ-bit scalars a_i, can be computed in $\lceil \ell/k \rceil$ steps performing k doublings and d additions at each step in a uniform pattern.

Simultaneous d-point multiplication algorithms generally require non-trivial precomputation and storage of group elements (exponential in dimension d or width w). More recently, several new protected recoding algorithms have been introduced in [9] which seem to offer better performance over the previously known algorithms [25, 15, 10] in terms of precomputation cost and storage requirements. For example, deploying the so-called GLV-SAC recoding algorithm [9] in a d-point multiplication algorithm requires a precomputation (and storage) of 2^{d-1} points at a cost of $(2^{d-1} - 1)$ group additions. After that, $\sum_{i=1}^{d} a_i P_i$ can be computed with ℓ doublings and ℓ additions in a uniform sequence of add, double, add, double, ..., add operations; see [9] for more details. This point multiplication algorithm is called the d-GLV-SAC algorithm in this paper.

Constructing a d-dimensional differential addition chain for a vector of scalars $[a_1, a_2, ..., a_d]$ naturally yields a d-point multiplication algorithm where the chain sequence describes explicitly how to compute $\sum_{i=1}^{d} a_i P_i$; see [4] for an overview and examples of multidimensional differential addition chains and their applications. Differential addition chains are of particular interest in certain groups $\mathbb{G}$ (such as elliptic curve groups) where addition can be performed more efficiently using differential addition and doubling formulas when the dif-

[2]As we pointed out before, this is a fairly reasonable assumption when d is small ($d \leq 8$) and P is fixed, or when $\mathbb{G}$ is an (hyper)elliptic curve with efficiently computable endomorphisms.

[3]Precomputation cost and storage requirements can be reduced by half at an expense of computing the inverse of points.

ferences of the input points are known. In the elliptic curves setting, differential addition and doubling formulas are generalization of Montgomery's formulas, which use only the x-coordinates of points [22]. In general, addition becomes much faster when differences of the points are fixed and precomputed in advance. For example, in the elliptic curves setting, one may represent the difference of points in affine coordinates rather than projective coordinates for improved performance.

One dimensional and two dimensional differential addition chains have extensively been studied [23, 22, 28, 4, 2]. One of the challenges in this area is to construct a d-dimensional differential addition chain that yields a constant time d-point multiplication algorithm performing a uniform sequence of operations. The Montgomery ladder constructed from a one-dimensional differential addition chain yields a uniform 1-point multiplication algorithm, which we call the ML algorithm (reads as the Montgomery ladder algorithm). The ML algorithm computes aP, for an ℓ-bit scalar a, in $(\ell - 1)$ steps performing one addition and doubling at each step. Bernstein [4] proposed two dimensional differential addition chain constructions one of which yields a uniform 2-point multiplication algorithm, also known as the DJB algorithm. The DJB algorithm computes $a_1P_1 + a_2P_2$, for ℓ-bit scalars a_i, in ℓ steps performing two additions and one doubling at each step. Azarderakhsh and Karabina [2] proposed a two dimensional differential addition chain construction that yields a uniform 2-point multiplication algorithm, also known as the AK algorithm. The AK algorithm computes $a_1P_1 + a_2P_2$, for ℓ-bit scalars a_i, in about 1.4ℓ steps[4] performing one addition and one doubling at each step. Brown, in an unpublished paper [7] in 2006, proposed a d-dimensional differential addition chain construction for general d. We believe that the idea of using d-dimensional addition chains in d-point multiplication algorithms have been overlooked in the literature since 2006. For example, we are not aware of any efficiency analysis or implementation results for general d. Similarly, we are not aware of any performance comparisons between a differential addition chain based d-point multiplication algorithm and others like JT [17] and d-GLV-SAC [9]. In fact, the following is noted in [7]: "*Careful analysis is needed in each case to ascertain whether the algorithm presented outperforms existing efficient alternatives.*". In this paper, we are trying to close this gap by presenting explicit construction of multidimensional differential addition chains and simultaneous scalar point multiplication algorithms. We also provide rigorous correctness and efficiency analysis of our algorithms, and discuss some cryptographically interesting features such as *uniformity*, *constant time*, and *parallelization*. We should note that our starting point for this work was Brown's paper [7], and that our construction has some similarities with [7]. Therefore, we inherit some notation from [7] as appropriate for the ease of presentation and for convenient comparisons with [7].

[4]This is based on a heuristic estimate with small variance in practice; see [2].

3 A construction of multidimensional differential addition chains

Throughout this section, $A, A^{(k)}, B$, etc. will denote matrices of size $(d+1) \times d$ for some positive integer d, unless otherwise specified. For any matrix A, we will denote by A_i the ith row of A, and by $A_{i,j}$ the entry of A in the ith row and jth column. All row and column indices start at 1. We will let e_j be the row matrix consisting of a 1 in the jth column and 0s elsewhere.

Definition 3.1. A $(d+1) \times d$ *state matrix* A satisfies:

i) each row A_i has $(i-1)$ odd entries.

ii) for $1 \leq i \leq d$, we have $A_{i+1} - A_i \in \{e_j, -e_j\}$ for some $1 \leq j \leq d$.

An easy consequence of this definition is the following. Row A_1 consists of all even entries, while row A_{d+1} consists of all odds. Since there are d many columns and the difference between successive rows is e_j, the number of rows forces each column's entries to change exactly once. In other words, each column in A has the form $\begin{bmatrix} x & \cdots & x & x+(-1)^k & \cdots & x+(-1)^k \end{bmatrix}^T$ for some integer k, where the index at which x changes to $x+(-1)^k$ must be distinct for each column.

In the following, we describe two algorithms: initialization and chain sequence generation. These two algorithms yield a d-dimensional differential addition chain algorithm that takes as input a sequence of integers $a_1, a_2, \ldots, a_d$, $d \geq 1$, and constructs a differential addition chain for its input. For a convenient comparison of our algorithms to that of Brown's [7], we choose our notation similar to the notation in [7].

3.1 Initialization step

The initialization step takes as input a sequence of integers $a_1, a_2, \ldots, a_d$, $d \geq 1$ and generates a state matrix.

Algorithm 1. Initialization

input : Integers $a_1, \ldots, a_d$
output: A $(d+1) \times d$ state matrix A

1 Let $h \in \{0, 1, \ldots, d\}$ be the number of a_i which are odd.
2 Define $A_{h+1} = \begin{bmatrix} a_1 & a_2 & \cdots & a_d \end{bmatrix}$.
3 Define the remaining rows A_i recursively in the following way:
4 If $0 \leq i < h+1$, define $A_i = A_{i+1} \pm e_j$, where j is chosen such that $A_{i+1,j}$ is odd. The choice of adding or subtracting e_j is left to the user.
5 If $h+1 < i \leq d+1$, define $A_i = A_{i-1} \pm e_j$, where j is chosen such that $A_{i-1,j}$ is even. The choice of adding or subtracting e_j is left to the user.
6 **return** A

The choice in adding or subtracting e_j does not make much of a difference for our purposes, however in the rare event that increasing the magnitude of an entry would push the integer into the next bit level, this could add one extra iteration in Algorithm 3 to come.

Correctness of the algorithm: We must prove that the output of the algorithm, the matrix A, is indeed a state matrix. For the remainder, the notation $\mathbf{A1}(i)$ will be used to refer to line i in Algorithm 1. Step $\mathbf{A1}(2)$ ensures row $h+1$ of A is defined to have h many odd entries, satisfying (i) of the definition of a state matrix for this row. Each row above A_{h+1} defined in $\mathbf{A1}(3)$ will have one less odd entry than the row below it, preserving (i) as we move up the rows from A_{h+1}. Similarly, the rows after A_{h+1} are defined so that they contain exactly one additional odd entry than the row above them, making (i) preserved throughout the entire matrix. As for property (ii), the successive rows are defined by $A_i = A_{i\pm1} \pm e_j$, and so $A_i - A_{i\pm1} = \pm e_j$.

We'll justify that $\mathbf{A1}(3)$ is always valid within the bounds $1 \leq i \leq d+1$. By the choice of h in (1), the initial row A_{h+1} contains h many odd entries. There are h many rows to define above A_{h+1}, and each new row turns exactly one odd entry into an even one. The number of odd entries will decrease to zero exactly when the top row, A_1, is constructed. Similarly, there are $(d+1)-(h+1) = d-h$ many rows below A_{h+1} to be constructed, and each new row turns an even into an odd. Since there are $d-h$ many evens in A_{h+1}, they will be switched to odds one by one until row A_{d+1} is reached, which will have zero even entries.

3.2 Chain sequence generation

Algorithm 2 below takes a state matrix A as input and constructs a new state matrix B having the property that each row in A is the sum of two (not necessarily distinct) rows of B. We use an array D to store the information pertaining to which of the rows of B sum to the rows in A, and we also keep track of the differences of the rows involved in those sums (also in D using a row matrix R).

The variables above deserve explanation. Perhaps the most convoluted variable used here is σ. By definition of a state matrix, we have that $A_i - A_{i-1} = \pm e_j$ for some index j. σ is defined so that $\sigma(i) = j$, meaning that $\sigma(i)$ is column number which has changed when moving from A_{i-1} to A_i. Since each column changes exactly once, and only one for each row, σ is a bijection. The c_j represents whether the jth column has increased or decreased.

The variables x and y represent the upper and low bounds, respectively, on the rows of the matrix B which have been defined at each step of the algorithm. They both are initialized to $h+1$ since that is the only row for B which is initially defined. As rows are appended to the top of B, x will decrease; as rows are appended to the bottom of B, y will increase.

A typical column of D will look like $\begin{bmatrix} k & x & y & R \end{bmatrix}^T$, which is interpreted as

$$B_x + B_y = A_k$$
$$B_x - B_y = R$$

Algorithm 2. Chain sequence generation

input : A $(d+1) \times d$ state matrix A
output: A $(d+1) \times d$ state matrix B,
An array D having 4 rows and $d+1$ columns.

Let $\begin{bmatrix} 2\alpha_1 & 2\alpha_2 & \cdots & 2\alpha_d \end{bmatrix}$ denote A_1.
Define $\sigma : \{2, \ldots, d+1\} \to \{1, \ldots, d\}$ such that $\sigma(i)$ is the position in which the row matrix $A_i - A_{i-1}$ is nonzero.
For $1 \leq j \leq d$, set $c_j \leftarrow A_{d+1,j} - A_{1,j}$. `// Each` c_j `is either` 1 `or` -1
Define h as the number of α_i which are odd. Set $x \leftarrow h+1$ and $y \leftarrow h+1$.
Set $B_{h+1} \leftarrow \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_d \end{bmatrix}$. Let R be a $1 \times d$ zero matrix.
Set $D = \begin{bmatrix} 1 & h+1 & h+1 & R \end{bmatrix}^T$.
for $k = 2$ **to** $d+1$ **do**
  if $\alpha_{\sigma(k)}$ *is odd* **then**
    $B_{x-1} \leftarrow B_x + c_{\sigma(k)} e_{\sigma(k)}$, $R \leftarrow R + c_{\sigma(k)} e_{\sigma(k)}$.
    $x \leftarrow x - 1$.
  if $\alpha_{\sigma(k)}$ *is even* **then**
    $B_{y+1} \leftarrow B_y + c_{\sigma(k)} e_{\sigma(k)}$, $R \leftarrow R - c_{\sigma(k)} e_{\sigma(k)}$.
    $y \leftarrow y + 1$.
  end
  Append the column $\begin{bmatrix} k & x & y & R \end{bmatrix}^T$ to the end of D.
end
return B *and* D.

where R is a row matrix consisting of only the integers $0, 1, -1$ and having $k-1$ nonzero entries. When the algorithm completes, we will have $x = 1$, $y = d+1$, and $R = B_1 - B_{d+1}$.

Correctness of the algorithm: We must prove that the output matrix B is both a state matrix and has the property that each row of A is the sum of two rows from B. For the remainder, the notation **A2**(i) will be used to refer to line i in Algorithm 2.

First, note that σ defined in **A2**(2) is a bijection. This follows from the definition of a state matrix. σ denotes the order in which the ± 1 is introduced as we move down the rows of the state matrix.

We show that B is a state matrix of the correct size. The rows of B are built to satisfy (ii) of the definition of a state matrix when we define them in **A2**(7). As for (i), row B_{h+1} meets the condition by construction in **A2**(5). Rows above are constructed exactly when we find an odd α_i, in which we define the new row using the previous one and changing the α_i's parity. Thus the number of odds in each row will decrease by one as we move up the rows, beginning at B_{h+1} with h odds and ending at B_1 with 0 odds. Similarly, the number of odds

increases as we move down the rows from B_{h+1}.

Now we show that each row in A is the sum of two rows in B by proving the following.

Theorem 3.1. In the k^{th} iteration of the loop in **A2**(7), let x_k denote the least index of the rows of B which have been defined by the end of the iteration, and let y_k denote the greatest index of the rows of B which have been defined by the end of the iteration; then $A_k = B_{x_k} + B_{y_k}$.

Proof. Naturally, we prove this by induction on k. The base case $k = 1$ actually happens outside of the loop during **A2**(5): we have $A_1 = 2B_{h+1} = B_{x_1} + B_{y_1}$, using the values $x_1 = y_1 = h + 1$. These definition of x_1 and y_1 are consistent with the definitions stated in the theorem, since by step (5) only B_{h+1} has been defined.

Assume now that $A_k = B_{x_k} + B_{y_k}$ holds during the k^{th} iteration of the loop (or the base case of $k = 1$). During the $(k+1)^{\text{th}}$ iteration, we construct one new row of B depending on the parity of $\alpha_{\sigma(k+1)}$. We look at each case in turn.

Suppose $\alpha_{\sigma(k+1)}$ is odd. Then the first **if** statement would be executed in **A2**(7), and so we get

$$\begin{aligned} B_{x_k-1} &= B_{x_k} + c_{\sigma(k+1)} e_{\sigma(k+1)} \\ B_{x_k-1} &= (A_k - B_{y_k}) + c_{\sigma(k+1)} e_{\sigma(k+1)} \\ B_{x_k-1} + B_{y_k} &= A_k + c_{\sigma(k+1)} e_{\sigma(k+1)} \\ B_{x_{k+1}} + B_{y_{k+1}} &= A_{k+1} \end{aligned}$$

where the last equality follows since

- $x_{k+1} = x_k - 1$ (the least index decreased by one during this iteration)
- $y_{k+1} = y_k$ (the greatest index hasn't increased during this iteration)
- by definition, $\sigma(k+1)$ satisfies $A_{k+1} - A_k = \pm e_{\sigma(k+1)}$, where the sign is determined by $c_{\sigma(k+1)}$.

This proves the statement in the odd case.

Suppose instead that $\alpha_{\sigma(k+1)}$ is even. Similarly, executing the second **if** statement, we find

$$\begin{aligned} B_{y_k+1} &= B_{y_k} + c_{\sigma(k+1)} e_{\sigma(k+1)} \\ B_{y_k+1} &= (A_k - B_{x_k}) + c_{\sigma(k+1)} e_{\sigma(k+1)} \\ B_{y_k+1} + B_{x_k} &= A_k + c_{\sigma(k+1)} e_{\sigma(k+1)} \\ B_{y_{k+1}} + B_{x_{k+1}} &= A_{k+1}, \end{aligned}$$

where the substitutions on the last line follow similarly to the odd case. ∎

We also claim the correctness of the differences, stored in row 4 of D.

Theorem 3.2. Let x_k and y_k be as in Theorem 3.1, and let R_k be the current state of the row matrix R upon completion of the kth iteration of the loop **A2**(7). Then $R_k = B_{x_k} - B_{y_k}$.

Theorem 3.2 can be proved by induction on k similar to Theorem 3.1, with $R_1 = 0 = B_{h+1} - B_{h+1}$ being the initialization of R as the base case.

At this time we should mention the array D. The only purpose for D's existence is to keep track of which rows in B sum to the rows in A, as well as their differences. That the algorithm chooses the correct rows is reflected by Theorems 3.1 and 3.2.

4 d-MUL: A d-point multiplication algorithm

We are now ready to describe our d-point multiplication algorithm d-MUL. First, we make the following definition for the ease of presentation.

Definition 4.1. For a matrix A, we define the *magnitude* of A to be $|A| = \max\{|A_{i,j}|\}$.

Algorithm 3. d**-MUL:** d**-point multiplication**

---

input : Positive integers $a_1, \ldots, a_d$ and points $P_1, \ldots, P_d$.
output: $a_1P_1 + \cdots + a_dP_d$.

1 Run Algorithm 1 with the input $a_1, \ldots, a_d$. Let $A^{(1)}$ be the output.
2 Let h be the number of a_i which are odd. Assign $k \leftarrow 1$.
3 **while** $\left|A^{(k)}\right| > 1$ **do**
4 | Run Algorithm 2 with the input $A^{(k)}$. Let $A^{(k+1)}$ and $D^{(k)}$ be the respective outputs.
5 | $k \leftarrow k + 1$.
6 **end**
7 For $1 \leq i \leq d+1$, assign $Q_i^{(k)} \leftarrow A_i^{(k)} \cdot \begin{bmatrix} P_1 & \cdots & P_d \end{bmatrix}^T$.
8 $k \leftarrow k - 1$.
9 **while** $k > 0$ **do**
10 | For $n = 1$ to $d+1$, assign $Q^{(k)}_{D^{(k)}_{1,n}} \leftarrow Q^{(k+1)}_{D^{(k)}_{2,n}} + Q^{(k+1)}_{D^{(k)}_{3,n}}$.
| `/* The above addition can be performed using differential addition formulas, where the difference is given by` $D^{(k)}_{4,n} \cdot \begin{bmatrix} P_1 & \cdots & P_d \end{bmatrix}^T$ `*/`
11 |
12 | $k \leftarrow k - 1$.
13 **end**
14 **return** $Q^{(1)}_{h+1}$.

The idea here is to construct an initial state matrix $A^{(1)}$ out of the integers $a_1, \ldots, a_d$ using Algorithm 1, and then repeatedly apply Algorithm 2 to create a sequence of state matrices

$A^{(k)}$ until we arrive at one having only 0 and 1 entries. Then we add the corresponding linear combinations of the P_i together using $D^{(k)}$ while working our way back through the sequence until we arrive at the initial state matrix, upon which we have computed $Q_{h+1}^{(1)} = a_1P_1 + \cdots + a_dP_d$.

Correctness of the algorithm It suffices to prove that the first while loop in Algorithm 3 eventually terminates because $Q_{h+1}^{(1)} = a_1P_1 + \cdots + a_dP_d$ follows immediately from the description of Algorithm 2. In Theorem 4.1, we show that the sequence $|A^{(k)}|$ as generated in Algorithm 3 is a strictly decreasing sequence, as required.

Theorem 4.1. Let A be the input to Algorithm 2 and B be the output. If $|A| > 1$, then $|B| < |A|$.

Proof. Let $|B| = \beta$. It follows from the description of Algorithm 2 that there exists a pair of indices (j, k) such that $\beta + \beta_j = \alpha_k$, where β_j and β are in the same column of B (that is, $\beta - \beta_j = 0$ or $\beta - \beta_j$=1), and α_k is an entry of A. There are two cases to consider: α_k is even or α_k is odd. If α_k is even, then we must have $\beta + \beta_j = \alpha_k$ and $\beta - \beta_j = 0$. This yields $\beta = \alpha_k/2$. Now, if $\alpha_k = 0$, then $\beta = 0 < |A|$, by the assumption of the theorem. If $\alpha_k > 0$, then $\beta = \alpha_k/2 < \alpha_k \leq |A|$. If α_k is odd, then we must have $\beta + \beta_j = \alpha_k$ and $\beta - \beta_j = 1$. This yields $\beta = (\alpha_k + 1)/2$. Now, if $\alpha_k = 1$, then $\beta = 1 < |A|$, by the assumption of the theorem. If $\alpha_k > 1$, then $\beta = (\alpha_k + 1)/2 < \alpha_k \leq |A|$, as required. ■

Next, we use Theorem 4.1 and derive stronger results on the complexity of Algorithm 3.

Lemma 4.1. Let $\{A^{(k)}\}_{k=1}^{\ell+1}$ be a sequence of matrices generated as in Algorithm 3 with $|A^{(1)}| > 1$, $|A^{(\ell+1)}| \leq 1$, and $\ell > 0$. Then $|A^{(\ell+1)}| = 1$ and $|A^{(\ell)}| = 2$.

Proof. If $|A^{(\ell+1)}| \leq 1$ then all entries in $A^{(\ell+1)}$ must be either 0 or 1 because $A^{(\ell+1)}$ is a state matrix as shown before. In particular, the row $A_1^{(\ell+1)}$ consists of all zeros, the row $A_{d+1}^{(\ell+1)}$ consists of all ones, and $|A^{(\ell+1)}| = 1$. By the description of Algorithm 2, any row in $A^{(\ell)}$ is either twice a row of $A^{(\ell+1)}$, or is a sum of two distinct rows of $A^{(\ell+1)}$. In both cases, we obtain $|A^{(\ell)}| \leq 2$ because $|A^{(\ell+1)}| = 1$. We must also have $|A^{(\ell)}| \geq 2$ because otherwise, $|A^{(\ell)}| < 2$, which implies $|A^{(\ell)}| = 1 = |A^{(\ell+1)}|$. This is a contradiction because if $|A^{(\ell)}| = 1$, then Algorithm 3 would not generate the next state matrix $A^{(\ell+1)}$. ■

Lemma 4.2. Let $\{A^{(k)}\}_{k=1}^{\ell+1}$ be a sequence of matrices generated as in Algorithm 3. Then $|A^{(i)}| \leq 2|A^{(i+1)}|$ for any $i = 1, .., \ell$.

Proof. Choose an arbitrary $i \in \{1, ..., \ell\}$ and set $A = A^{(i)}$, $B = A^{(i+1)}$. Note that B is an output of Algorithm 2 given A as input. Let $\alpha = |A|$ and $\beta = |B|$. As observed previously in the proof of Theorem 4.1, we have $\alpha = \beta_1 + \beta_2$, where $\beta_1 \leq \beta_2$ are two entries in the same column of B such that $\beta_1 - \beta_2 \in \{0, 1\}$. There are two possible cases. In the first case, α is even, $\beta_1 = \beta_2$, and $\alpha = 2\beta_2$. This implies $|A| = \alpha = 2\beta_2 \leq 2|B|$. In the second case, α is odd, $\beta_2 = \beta_1 + 1$, and $\alpha = 2\beta_2 - 1$. This implies $|A| = 2\beta_2 - 1 \leq 2|B| - 1$. In both cases, $|A| \leq 2|B|$, as required. ■

Lemma 4.3. Let $\{A^{(k)}\}_{k=1}^{\ell+1}$ be a sequence of matrices generated as in Algorithm 3. Then $|A^{(i)}| \geq 2|A^{(i+1)}| - 1$ for any $i = 1, .., \ell$.

Proof. Choose an arbitrary $i \in \{1, ..., \ell\}$ and set $A = A^{(i)}$, $B = A^{(i+1)}$. Note that B is an output of Algorithm 2 given A as input. Let $\beta = |B|$. It follows from the proof of Theorem 4.1 that there exists an element α in A such that $\alpha \in \{2\beta - 1, 2\beta, 2\beta + 1\}$. Hence, $|A^{(i)}| \geq \alpha \geq 2\beta - 1 \geq 2|A^{(i+1)}| - 1$. ■

Theorem 4.2. Let $\{A^{(k)}\}_{k=1}^{\ell+1}$ be a sequence of matrices generated as in Algorithm 3. If $|A^{(1)}| > 2^{\ell-1}$, then Algorithm 3 makes at least ℓ calls to Algorithm 2.

Proof. Suppose that Algorithm 3 makes exactly m calls to Algorithm 2, resulting in the sequence $\{A^{(k)}\}_{k=1}^{m+1}$. By Lemma 4.1 and Lemma 4.2, we can write $2^{\ell-1} < |A^{(1)}| \leq 2^{m-1}|A^{(m)}| = 2^m$, whence $m \geq \ell$. ■

Theorem 4.3. Let $\{A^{(k)}\}_{k=1}^{\ell+1}$ be a sequence of matrices generated as in Algorithm 3. If $|A^{(1)}| \leq 2^{\ell}$, then Algorithm 3 makes at most ℓ calls to Algorithm 2.

Proof. Suppose that Algorithm 3 makes exactly $m \geq \ell + 1$ calls to Algorithm 2, resulting in the sequence $\{A^{(k)}\}_{k=1}^{m+1}$. By Lemma 4.1 and Lemma 4.3, we can write $|A^{(1)}| \geq 2^{m-1}|A^{(m)}| - (2^{m-1} - 1) = 2^m - 2^{m-1} + 1 = 2^{m-1} + 1 \geq 2^{\ell} + 1$, contradiction Hence, $m \leq \ell$. ■

Theorem 4.4. Let $a_1, \ldots, a_d$ be ℓ-bit input coefficients to Algorithm 3 such that not all a_i are equal to $2^{\ell-1}$. Then Algorithm 3 makes exactly ℓ calls to Algorithm 2.

Proof. By Theorem 4.2 and Theorem 4.3, we have that if $2^{\ell-1} < |A^{(1)}| \leq 2^{\ell}$, then Algorithm 2 is called exactly ℓ times. This occurs provided that the largest a_i satisfies $2^{\ell-1} < a_i \leq 2^{\ell} - 1$ because then $|A^{(1)}| \geq a_i > 2^{\ell-1}$ and $|A^{(1)}| \leq a_i + 1 \leq 2^{\ell}$. Here, the second last inequality follows because $A^{(1)}$ is a state matrix. Finally, since a_i are ℓ-bit integers and not all of them are $2^{\ell-1}$ by assumption, at least one a_i satisfies $2^{\ell-1} < a_i \leq 2^{\ell} - 1$, as desired. ■

5 Analysis and comparisons

5.1 Preliminaries

As before, we consider d-point multiplication algorithm in a group $\mathbb{G}$, where the cost of addition and doubling operations are denoted by $\mathbf{A}$ and $\mathbf{D}$, respectively. The most cryptographically interesting groups $\mathbb{G}$ in this context are (sub)groups of (hyper)elliptic curves defined over a finite field. In this case, the cost of multiplication and squaring operations in a finite field are denoted by $\mathbf{M}$ and $\mathbf{S}$, respectively. Motivated by these cryptographically interesting (hyper)elliptic curve groups, we assume that the cost of point negation ($P \to -P$) is negligible. Moreover, the cost of differential addition and doubling operations are denoted by $\mathbf{A}_\Delta$ and $\mathbf{D}_\Delta$, respectively. For concreteness, we provide below two examples of groups $\mathbb{G}$ and the cost of underlying group operations to which we refer throughout this section. In these two examples, $\mathbb{G}$ occurs as a subgroup of an elliptic curve group defined over a finite field of characteristic greater than two.

Montgomery curves: A Montgomery curve is given by the equation $by^2 = x^3 + ax^2 + x$. Differential addition and doubling costs are $\mathbf{A}_\Delta = 4\mathbf{M} + 2\mathbf{S}$ and $\mathbf{D}_\Delta = 2\mathbf{M} + 2\mathbf{S}$; see [5, 22]. A straightforward modification of the affine coordinate addition formula in [5, 22] yields a projective addition formula with a cost of $\mathbf{A} = 15\mathbf{M} + 2\mathbf{S}$. Our motivation for choosing Montgomery curves is that they seem to offer the most efficient differential addition and doubling operations compared to other curve models over non-binary fields.

Twisted Edwards curves: A Twisted Edwards curve is given by the equation $ax^2 + y^2 = 1 + dx^2y^2$. Addition and doubling costs (with extended coordinates) are $\mathbf{A} = 8\mathbf{M}$ and $\mathbf{D} = 3\mathbf{M} + 4\mathbf{S}$; see [5, 16]. Our motivation for choosing Twisted Edwards curves is that they seem to offer the most efficient regular addition and doubling operations compared to other curve models over non-binary fields.

Assumptions: When we consider a d-point multiplication algorithm for computing $\sum_{i=1}^{d} a_i P_i$, we assume that there exists a pair a_i, a_j such that $a_i \neq a_j$. This is not a restrictive assumption because if all a_i are equal, say $a_i = a$, then one can write $\sum_{i=1}^{d} a_i P_i = a(\sum_{i=1}^{d} P_i)$, which is simply a 1-point multiplication algorithm after precomputing $\sum_{i=1}^{d} P_i$. We also make the simplifying assumption that all a_i satisfy $1 \leq a_i \leq 2^\ell - 1$ (a_i are positive and at most ℓ-bit integers) because the dimension d drops by 1 for each zero scalar a_i and that scalars a_i are bounded above by a function of the group order and the dimension d. In most of the applications $\ell = n$ or $\ell = \lfloor n/d \rfloor$, where $|\mathbb{G}|$ is an n-bit number; see also Section 2.

5.2 An analysis of d-MUL

We investigate the complexity of our d-point multiplication algorithm d-MUL (Algorithm 3). We estimate the complexity of d-MUL in terms of the number of point additions and doublings performed during the execution of the algorithm. We point out the uniform pattern of operations in d-MUL as well as remark on the exact number of iterations in the main loop of d-MUL.

Complexity and uniformity: The only steps where point addition and doubling are performed in Algorithm 3 are **A3**(7) and **A3**(10). Each iteration of **A3**(9) performs 1 doubling and d additions in a uniform pattern of 1 doubling, d additions, 1 doubling, d additions, ..., 1 doubling, d additions. As commented in Algorithm 3, all of these additions can be performed using differential addition formulas since the difference of every sum is either the identity element of the group, or it is determined by the last row in D. Therefore, we consider two variants of d-MUL: d-MUL-DIFF and d-MUL-REG, which utilize differential addition formulas and regular addition formulas, respectively. In d-MUL-DIFF, $\texttt{span}_{\{0,1,-1\}}(P_1, \ldots, P_d)$ should be precomputed and stored. There are $3^d - 1$ such points in total (discarding the identity element), and it requires $3^d - 2d - 1$ point additions. Moreover, when the cost of point negation is negligible, the precomputation cost can be deduced to that of computing $(3^d - 1)/2$ points, which requires $(3^d - 2d - 1)/2$ point additions. Note that the cost of **A3**(7)

has already been taken care of because these points belong to the set $\texttt{span}_{\{0,1,-1\}}(P_1, \ldots, P_d)$. In d-MUL-REG, the differences of the points are not needed and so the exponentially many number of additions can be avoided in the precomputation step but one would still need to precompute the points in **A3**(7). This requires $(d-1)$ point additions because entries of the last state matrix of the sequence $\{A^{(k)}\}_k$ are 0 or 1.

It follows from Theorem 4.3 that the main while loop of d-MUL has at most ℓ iterations under our assumptions as stated in Section 5.1. In fact, if one further insists that all a_i have exactly ℓ-bits in their binary representation, then d-MUL has exactly ℓ iterations; see Theorem 4.4. Moreover, the number of such tuples a_i, $i = 1, ..., d$, that lead to exactly ℓ iterations in d-MUL is at least $2^{(\ell-1)d} - 2^{(\ell-1)}$ among at most $2^{\ell d}$ of its inputs.

Our discussion can be summarized as in Remark 5.1. Remark 5.1 justifies that d-MUL has some level of built-in protection against side-channel attacks due to its uniform pattern and the fixed number of iterations for a large portion of its input.

Remark 5.1. Let a_i, $i = 1, ..., d$, be input to d-MUL (Algorithm 3) such that $1 \leq a_i \leq 2^\ell - 1$.

1. d-MUL (after precomputation) performs at most ℓ iterations of loop **A3**(3).
2. If all a_i have exactly ℓ-bits in their binary representation, and $a_i \neq a_j$ for some $i \neq j$, then d-MUL (after precomputation) performs exactly ℓ iterations.
3. d-MUL (after precomputation) performs exactly ℓ iterations for about $1/2^d$ of its inputs.
4. d-MUL performs (per iteration) 1 doubling and d addition operations in a uniform pattern of 1 doubling, d additions, ..., 1 doubling, d additions.

Our analysis yields Table 1 in comparison with two other algorithms JT and d-GLV-SAC.

Table 1: A comparison of d-point multiplication algorithms: d-MUL-DIFF, d-MUL-REG, JT [17], and d-GLV-SAC [9]. Refer to Section 2 for more details on the JT and d-GLV-SAC algorithms. $\mathbf{A}_\Delta$ and $\mathbf{D}_\Delta$ denote differential addition and doubling costs. $\mathbf{A}$ and $\mathbf{D}$ denote regular addition and doubling costs. The second row of the table gives the number of points in the lookup table.

	d-MUL-DIFF	d-MUL-REG	JT [17]	d-GLV-SAC [9]
Precomp	$\left(\frac{3^d - 2d - 1}{2}\right)\mathbf{A}$	$(d-1)\mathbf{A}$	$\frac{d((m-2)\mathbf{A} + 2\mathbf{D})}{2}$	$(2^{d-1} - 1)\mathbf{A}$
Lookup table	$\frac{3^d - 1}{2}$	0	$\frac{d \cdot m}{2}$	2^{d-1}
Main loop	$\ell(d\mathbf{A}_\Delta + \mathbf{D}_\Delta)$	$\ell(d\mathbf{A} + \mathbf{D})$	$\left\lceil \frac{\ell}{k} \right\rceil (d\mathbf{A} + k\mathbf{D})$	$\ell(\mathbf{A} + \mathbf{D})$

Based on our estimates in Table 1, we compare our algorithm d-MUL ($d = 3$) with a 3-point multiplication algorithm 3-ML as recently proposed in [26]. Table 2 shows that 3-MUL-DIFF saves at least one $\mathbf{A}_\Delta$ per iteration in the main loop. When implemented over a Montgomery curve, where $\mathbf{A}_\Delta = 4\mathbf{M} + 2\mathbf{S}$ and $\mathbf{D}_\Delta = 2\mathbf{M} + 2\mathbf{S}$, 3-MUL-DIFF is expected to yield about 21% speed-up over 3-ML (with respect to the total number of field multiplications and squarings, assuming $\mathbf{M} = \mathbf{S}$ or $0.8\mathbf{M} = \mathbf{S}$). Another advantage of 3-MUL over 3-ML is that 3-MUL is uniform (see Remark 5.1) whereas 3-ML is not because the cost per iteration varies between $(4\mathbf{A}_\Delta + \mathbf{D}_\Delta)$ and $5\mathbf{A}_\Delta$; see [26].

Table 2: A comparison of two 3-point multiplication algorithms. Each of the ℓ steps in 3-ML requires $(4\mathbf{A}_\Delta + \mathbf{D}_\Delta)$ or $5\mathbf{A}_\Delta$

	3-MUL-DIFF	3-MUL-REG	3-ML [26]
Precomp	$10\mathbf{A}$	$2\mathbf{A}$	$8\mathbf{A}$
Lookup table	13	0	8
Main loop	$\ell(3\mathbf{A}_\Delta + \mathbf{D}_\Delta)$	$\ell(3\mathbf{A} + \mathbf{D})$	$\geq \ell(4\mathbf{A}_\Delta + \mathbf{D}_\Delta)$
Uniform	✓	✓	✗
Expected speed up over 3-ML		0.21	

Parallelization: All point addition and doubling operations performed at each iteration in the main loop of d-MUL can be fully parallelized. The JT algorithm also allows parallelization of the addition operations (but not of the doubling operations) during its execution. On the other hand, it does not seem to be possible to have the d-GLV-SAC algorithm perform the addition and doubling operations in parallel.

In Table 3, we provide a comparison of the d-MUL and JT algorithms assuming that addition and doubling operations can be performed in parallel using c computing units.

Based on our estimates in Table 3, we provide a more concrete comparison between d-MUL-DIFF and JT in a cryptographically interesting setting in Table 4. We aim for 128-bit security level, and consider $\mathbb{G}$ to be a $n = 256$-bit subgroup of an elliptic curve group. We choose $d = 4$ and consider a_i to be $n/d = 256/4 = 64$-bit integers in the 4-point multiplication algorithm for computing $\sum_{i=1}^{d} a_i P_i$ (see the notation and assumptions in Section 2). In this setting, we assume that 4-MUL-DIFF and JT adapts Montgomery and Twisted Edwards curves, respectively (see Section 5.1 for a justification of the curve selections). We conclude that 4-MUL-DIFF is expected to outperform JT when $c = 5$, but not for $c \leq 4$.

Finally, we compare the relative complexity of d-MUL-DIFF for the values of $d = 1, 2, 3, 4$. Based on our estimates in Table 1, d-MUL-DIFF is expected to introduce about $(1 - 1/d)\mathbf{D}_\Delta/(\mathbf{A}_\Delta + \mathbf{D}_\Delta)\%$ speed up (discarding the precomputation) over 1-MUL-DIFF (which is equivalent to deploying the one dimensional Montgomery ladder). In Table 5, we consider $\mathbb{G}$ as a $n \approx 256$-bit subgroup of an elliptic curve group, and set a_i to be $\ell = \lfloor n/d \rfloor$-bit integers in the d-point multiplication algorithm for computing $\sum_{i=1}^{d} a_i P_i$ (see the notation

Table 3: A comparison of parallelized d-point multiplication algorithms. $\mathbf{A}_\Delta$ and $\mathbf{D}_\Delta$ denote differential addition and doubling costs. $\mathbf{A}$ and $\mathbf{D}$ denote regular addition and doubling costs. It is assumed that addition and doubling operations can be performed in parallel using c computing units; $e_1 = \lfloor d/c \rfloor$, $f_1 = d \pmod c$; $H(x) = 0$ if $x = 0$ and $H(x) = 1$ otherwise. The variable $m = 2^k$ is a parameter in the JT algorithm (see Section 2)

	d-MUL-DIFF	d-MUL-REG	JT [17]
Precomp	$\left\lceil \frac{3^d - 2d - 1}{2c} \right\rceil \mathbf{A}$	$\left\lceil \frac{d-1}{c} \right\rceil \mathbf{A}$	$\left\lceil \frac{d}{c} \right\rceil \left(\frac{(m-2)\mathbf{A} + 2\mathbf{D}}{2} \right)$
Lookup table	$\frac{3^d - 1}{2}$	0	$\frac{d \cdot m}{2}$
Main loop	$\ell((e_1 + H(f_1))\mathbf{A}_\Delta + (1 - H(f_1))\mathbf{D}_\Delta)$	$\ell((e_1 + H(f_1))\mathbf{A} + (1 - H(f_1))\mathbf{D})$	$\left\lceil \frac{\ell}{k} \right\rceil ((e_1 + H(f_1))\mathbf{A} + k\mathbf{D})$

and assumptions in Section 2). We further assume that d-MUL-DIFF adapts Montgomery curves which seem to provide the most efficient differential point addition formulas. In particular, $\mathbf{A}_\Delta = 4\mathbf{M} + 2\mathbf{S}$ and $\mathbf{D}_\Delta = 2\mathbf{M} + 2\mathbf{S}$. In this setting, d-MUL-DIFF is expected to improve 1-DIFF-MUL by up to 30%. In our prototype implementation, we work with a $n = 252$-bit prime order subgroup $\mathbb{G}$ of a Montgomery curve $y^2 = x^3 + 486662x^2 + x$ defined over the prime field of size $2^{255} - 19$, and compute $\sum_{i=1}^{d} a_i P_i$ for $d = 1, 2, 3, 4$ and scalars a_i each of which is $\ell = \lfloor n/d \rfloor$-bit. Our implementation uses the NTL-library [27], runs on a personal computer, and verifies the theoretical expectations. The gap between our theoretical estimates and experimental results may be due to several reasons including the overheads of using lookup tables and the fact that theoretical estimates ignore the cost of field additions.

6 Concluding remarks

We described new algorithms for constructing d-dimensional differential addition chains and for performing d-dimensional scalar point multiplication based on differential addition chains. Our d-point multiplication algorithm d-MUL has some attractive efficiency and security features. In particular, d-MUL is uniform, it has high potential for constant time implementation, and it can be parallelized. d-MUL also allows trading speed for precomputation cost and storage requirements. We find this as an interesting feature especially for competing with some existing d-point multiplication algorithms that require exponentially large lookup tables during their execution. Overall, our theoretical estimates indicate that d-MUL may offer significant performance advantages over the existing point multiplication algorithms in practice. We also reported some experimental results based on our prototype implementation and verified some of our theoretical findings. It would be interesting to have an optimized implementation of d-MUL and compare its performance to some state-of-the-art algorithms in software and hardware.

Table 4: A comparison of parallel d-MUL-DIFF and JT for $d = 4$, $\ell = 64$. It is assumed that the parameter $m = 2^k$ in JT is set to be 16 (i.e. k=4). Refer to Section 2 for more details on the JT algorithm and its parameters. It is assumed that addition and doubling operations are performed in parallel using $1 \leq c \leq 5$ computation units.

	4-MUL-DIFF				JT [17]			
c	Precomp	Lookup table	Main loop		Precomp	Lookup table	Main loop	
1	$36\mathbf{A}$	40	$256\mathbf{A}_\Delta + 64\mathbf{D}_\Delta$		$28\mathbf{A}$	32	$64\mathbf{A} + 64\mathbf{D}$	
2	$18\mathbf{A}$	40	$128\mathbf{A}_\Delta + 64\mathbf{D}_\Delta$		$14\mathbf{A} + 2\mathbf{D}$	32	$32\mathbf{A} + 64\mathbf{D}$	
3	$12\mathbf{A}$	40	$128\mathbf{A}_\Delta$		$14\mathbf{A} + 2\mathbf{D}$	32	$32\mathbf{A} + 64\mathbf{D}$	
4	$9\mathbf{A}$	40	$64\mathbf{A}_\Delta + 64\mathbf{D}_\Delta$		$7\mathbf{A} + 1\mathbf{D}$	32	$32\mathbf{A} + 64\mathbf{D}$	
5	$8\mathbf{A}$	40	$64\mathbf{A}_\Delta$		$7\mathbf{A} + 1\mathbf{D}$	32	$16\mathbf{A} + 64\mathbf{D}$	
	$\mathbf{A} = 15\mathbf{M} + 2\mathbf{S}, \mathbf{A}_\Delta = 4\mathbf{M} + 2\mathbf{S}, \mathbf{D}_\Delta = 2\mathbf{M} + 2\mathbf{S}$				$\mathbf{A} = 8\mathbf{M}, \mathbf{D} = 3\mathbf{M} + 4\mathbf{S}$			
c	Precomp	Main loop	Total		Precomp	Main loop	Total	
4	$135\mathbf{M} + 18\mathbf{S}$	$384\mathbf{M} + 256\mathbf{S}$	$519\mathbf{M} + 274\mathbf{S}$		$59\mathbf{M} + 4\mathbf{S}$	$320\mathbf{M} + 256\mathbf{S}$	$379\mathbf{M} + 260\mathbf{S}$	
4			$\mathbf{S} = \mathbf{M}$	$\mathbf{S} = 0.8\mathbf{M}$			$\mathbf{S} = \mathbf{M}$	$\mathbf{S} = 0.8\mathbf{M}$
4			$793\mathbf{M}$	$738\mathbf{M}$			$639\mathbf{M}$	$587\mathbf{M}$
5	$120\mathbf{M} + 16\mathbf{S}$	$256\mathbf{M} + 128\mathbf{S}$	$376\mathbf{M} + 144\mathbf{S}$		$59\mathbf{M} + 4\mathbf{S}$	$320\mathbf{M} + 256\mathbf{S}$	$379\mathbf{M} + 260\mathbf{S}$	
5			$\mathbf{S} = \mathbf{M}$	$\mathbf{S} = 0.8\mathbf{M}$			$\mathbf{S} = \mathbf{M}$	$\mathbf{S} = 0.8\mathbf{M}$
5			$520\mathbf{M}$	$491\mathbf{M}$			$639\mathbf{M}$	$587\mathbf{M}$
5	Expected speed up over JT		0.18	0.16				

Table 5: Relative complexity of d-MUL-DIFF for the values of $d = 1, 2, 3, 4$, and some experimental results. d-MUL-DIFF adapts Montgomery curves with $\mathbf{A}_\Delta = 4\mathbf{M} + 2\mathbf{S}$ and $\mathbf{D}_\Delta = 2\mathbf{M} + 2\mathbf{S}$.

	$d = 1$		$d = 2$		$d = 3$		$d = 4$	
Main loop	$1536\mathbf{M} + 1024\mathbf{S}$		$1280\mathbf{M} + 768\mathbf{S}$		$1194\mathbf{M} + 682\mathbf{S}$		$1152\mathbf{M} + 640\mathbf{S}$	
	$\mathbf{M} = \mathbf{S}$	$\mathbf{M} = 0.8\mathbf{S}$	$\mathbf{M} = \mathbf{S}$	$\mathbf{M} = 0.8\mathbf{S}$	$\mathbf{M} = \mathbf{S}$	$\mathbf{M} = 0.8\mathbf{S}$	$\mathbf{M} = \mathbf{S}$	$\mathbf{M} = 0.8\mathbf{S}$
	$2560\mathbf{M}$	$2355\mathbf{M}$	$2048\mathbf{M}$	$1894\mathbf{M}$	$1876\mathbf{M}$	$1739\mathbf{M}$	$1792\mathbf{M}$	$1664\mathbf{M}$
Expected speed up over $d = 1$			0.20	0.19	0.26	0.26	0.30	0.29
Experimental speed up over $d = 1$			0.14		0.21		0.23	

A A toy example

Suppose we wish to compute the point $10P_1 + 14P_2 + 9P_3 + 11P_4$ using Algorithm 3.

We are first to run Algorithm 1. There are two odd coefficients, and so $A_3 = \begin{bmatrix}10 & 14 & 9 & 11\end{bmatrix}$. Now we add or subtract 1 to the coefficients one at a time, ensuring that we end up with all evens in A_1 and all odds in A_5. Making random choices for whether to add or subtract, suppose we arrive at the matrix

$$A^{(1)} = \begin{bmatrix}10 & 14 & 10 & 12\\ 10 & 14 & 10 & 11\\ 10 & 14 & 9 & 11\\ 10 & 15 & 9 & 11\\ 11 & 15 & 9 & 11\end{bmatrix}$$

as the output of **A1**.

For **A3**(2), we have $k = 1$ and $h = 2$.

Now we arrive at the loop, **A3**(3). We run **A2** with the matrix $A^{(1)}$ above as input. To compute σ, we look at consecutive rows of $A^{(1)}$. The change between rows $A_1^{(1)}$ and $A_2^{(1)}$ occurs at column 4, and so $\sigma(2) = 4$. The change from $A_2^{(1)}$ to $A_3^{(1)}$ occurs at column 3, and so $\sigma(3) = 3$. Continuing in this fashion, we have $\sigma(4) = 2$ and $\sigma(5) = 1$. To define the c_i's, just subtract the first row of $A^{(1)}$ from the final row to get $\begin{bmatrix}1 & 1 & -1 & -1\end{bmatrix}$, which tells us $c_1 = 1, c_2 = 1, c_3 = -1, c_4 = -1$.

To begin building B, halve the top row of $A^{(1)}$ to get the numbers $5, 7, 5, 6$. There are three odds here, and so $B_4 = \begin{bmatrix}5 & 7 & 5 & 6\end{bmatrix}$. In **A2**(6), we initialize D as

$$D = \begin{array}{|c|}\hline 1\\ \hline 4\\ \hline 4\\ \hline \left[0, 0, 0, 0\right]\\ \hline \end{array}$$

We construct the remaining rows of B iteratively in **A2**(7) by inspecting the successive rows of $A^{(1)}$.

The change in row $A_2^{(1)}$ occurs in column 4 (already noted by $\sigma(2) = 4$) in which the 12 in row 1 decreased to 11. In order to make 11, we need a 5 to go with the 6 already placed in B_4. Since 6 is even we define the new row of B to be placed below the current rows,

consisting of the same numbers as B_4 but changing the 6 to a 5:

$$B = \begin{bmatrix} & & & \\ & & & \\ & & & \\ 5 & 7 & 5 & 6 \\ 5 & 7 & 5 & 5 \end{bmatrix} \qquad A^{(1)} = \begin{bmatrix} 10 & 14 & 10 & 12 \\ 10 & 14 & 10 & 11 \\ 10 & 14 & 9 & 11 \\ 10 & 15 & 9 & 11 \\ 11 & 15 & 9 & 11 \end{bmatrix}$$

These are our results after executing **A2**(11) in the first iteration ($k = 2$). This definition of the newly constructed row of B allows us to satisfy $A_2^{(1)} = B_4 + B_5$. The difference $B_4 - B_5 = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ becomes the new R, and we append the column on to D with this information in **A2**(15):

$D =$

1	2
4	4
4	5
$[0, 0, 0, 0]$	$[0, 0, 0, 1]$

We look at row 3 of $A^{(1)}$. The change occurs at column 3 when 10 decreased to 9 (given by $\sigma(3) = 3$ and $c_3 = -1$). In order to make 9, we need a 4 to go with the 5 in column 3 of B. Since 5 is odd we define the new row of B to be placed above the current rows, consisting of the same numbers as B_4 but changing the 5 to a 4:

$$B = \begin{bmatrix} & & & \\ & & & \\ 5 & 7 & 4 & 6 \\ 5 & 7 & 5 & 6 \\ 5 & 7 & 5 & 5 \end{bmatrix} \qquad A^{(1)} = \begin{bmatrix} 10 & 14 & 10 & 12 \\ 10 & 14 & 10 & 11 \\ 10 & 14 & 9 & 11 \\ 10 & 15 & 9 & 11 \\ 11 & 15 & 9 & 11 \end{bmatrix}$$

The above is matrix B after the second iteration of **A2**(7) ($k = 3$). Again the definition of this new row ensures that $B_3 + B_5 = A_3^{(1)}$. The row numbers and the difference $B_3 - B_5 = \begin{bmatrix} 0 & 0 & -1 & 1 \end{bmatrix}$ are stored into D:

$D =$

1	2	3
4	4	3
4	5	5
$[0, 0, 0, 0]$	$[0, 0, 0, 1]$	$[0, 0, -1, 1]$

We continue in this way, going down the rows of $A^{(1)}$. The change in $A_4^{(1)}$ occurs in column 2, where we need an 8 to go with our 7 in B. 7 is odd, so we append a row at the top of B (and a column onto D, shown further below):

$$B = \begin{bmatrix} & & & \\ 5 & 8 & 4 & 6 \\ 5 & 7 & 4 & 6 \\ 5 & 7 & 5 & 6 \\ 5 & 7 & 5 & 5 \end{bmatrix} \qquad A^{(1)} = \begin{bmatrix} 10 & 14 & 10 & 12 \\ 10 & 14 & 10 & 11 \\ 10 & 14 & 9 & 11 \\ 10 & 15 & 9 & 11 \\ 11 & 15 & 9 & 11 \end{bmatrix}$$

Finally, the last change occurs in column 1. We need a 6 with our current 5 in B in order to make 11, and since 5 is odd, we append at the top once more. This will finish the loop, and we will rename the matrix B to be $A^{(2)}$:

$$A^{(2)} = B = \begin{bmatrix} 6 & 8 & 4 & 6 \\ 5 & 8 & 4 & 6 \\ 5 & 7 & 4 & 6 \\ 5 & 7 & 5 & 6 \\ 5 & 7 & 5 & 5 \end{bmatrix} \qquad A^{(1)} = \begin{bmatrix} 10 & 14 & 10 & 12 \\ 10 & 14 & 10 & 11 \\ 10 & 14 & 9 & 11 \\ 10 & 15 & 9 & 11 \\ 11 & 15 & 9 & 11 \end{bmatrix}$$

$D^{(1)} =$

1	2	3	4	5
4	4	3	2	1
4	5	5	5	5
$[0,0,0,0]$	$[0,0,0,1]$	$[0,0,-1,1]$	$[0,1,-1,1]$	$[1,1,-1,1]$

Assigning $k = 2$, this finishes the first iteration of the loop in **A3**(3). We continue iterating through this loop, constructing a sequence of matrices and arrays just as we did above. We construct the final matrix when $k = 4$ (however, the loop increments once more before exiting to leave us with $k = 5$). In total, we now have:

$A^{(1)}$

$$\begin{bmatrix} 10 & 14 & 10 & 12 \\ 10 & 14 & 10 & 11 \\ 10 & 14 & 9 & 11 \\ 10 & 15 & 9 & 11 \\ 11 & 15 & 9 & 11 \end{bmatrix}$$

$D^{(1)}$

1	2	3	4	5
4	4	3	2	1
4	5	5	5	5
$[0,0,0,0]$	$[0,0,0,1]$	$[0,0,-1,1]$	$[0,1,-1,1]$	$[1,1,-1,1]$

$A^{(2)}$

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

$D^{(2)}$

1	2	3	4	5
3	2	2	2	1
3	3	4	5	5
$[0,0,0,0]$	$[-1,0,0,0]$	$[-1,1,0,0]$	$[-1,1,-1,0]$	$[-1,1,-1,-1]$

$A^{(3)}$

$$\begin{bmatrix} 2 & 4 & 2 & 2 \\ 2 & 4 & 2 & 3 \\ 3 & 4 & 2 & 3 \\ 3 & 3 & 2 & 3 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

$D^{(3)}$

1	2	3	4	5
4	3	2	2	1
4	4	4	5	5
$[0,0,0,0]$	$[0,0,0,1]$	$[1,0,0,1]$	$[1,1,0,1]$	$[1,1,1,1]$

$A^{(4)}$

$$\begin{bmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$D^{(4)}$

1	2	3	4	5
5	4	3	2	1
5	5	5	5	5
$[0,0,0,0]$	$[0,0,-1,0]$	$[-1,0,-1,0]$	$[-1,0,-1,-1]$	$[-1,-1,-1,-1]$

$A^{(5)}$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The arrays $D^{(i)}$ give us a road map for how to compute our desired $A_3^{(1)}$ in terms of the

rows of $A^{(5)}$, which are particularly simple. We compute $Q_i^{(5)} := A_i^{(5)} \cdot \begin{bmatrix} P_1 & P_2 & P_3 & P_4 \end{bmatrix}^T$ for $1 \leq i \leq 5$, as stated in **A3**(7):

$$\begin{aligned}
Q_1^{(5)} &= 0 \\
Q_2^{(5)} &= P_2 \\
Q_3^{(5)} &= P_2 + P_4 \\
Q_4^{(5)} &= P_1 + P_2 + P_4 \\
Q_5^{(5)} &= P_1 + P_2 + P_3 + P_4
\end{aligned}$$

Proceeding into the loop in **A3**(9), we then compute the rows of $A^{(4)}$ listed in the first row of $D^{(4)}$ in terms of the above 5 points:

$$\begin{aligned}
Q_1^{(4)} &= 2Q_5^{(5)} &&= 2P_1 + 2P_2 + 2P_3 + 2P_4 \\
Q_2^{(4)} &= Q_4^{(5)} + Q_5^{(5)} &&= 2P_1 + 2P_2 + P_3 + 2P_4 \\
Q_3^{(4)} &= Q_3^{(5)} + Q_5^{(5)} &&= P_1 + 2P_2 + P_3 + 2P_4 \\
Q_4^{(4)} &= Q_2^{(5)} + Q_5^{(5)} &&= P_1 + 2P_2 + P_3 + P_4 \\
Q_5^{(4)} &= Q_1^{(5)} + Q_5^{(5)} &&= P_1 + P_1 + P_3 + P_4
\end{aligned}$$

Next, compute the rows of $A^{(3)}$ listed in the first row of $D^{(3)}$ using the above points:

$$\begin{aligned}
Q_1^{(3)} &= 2Q_4^{(4)} &&= 2P_1 + 4P_2 + 2P_3 + 2P_4 \\
Q_2^{(3)} &= Q_3^{(4)} + Q_4^{(4)} &&= 2P_1 + 4P_2 + 2P_3 + 3P_4 \\
Q_3^{(3)} &= Q_2^{(4)} + Q_4^{(4)} &&= 3P_1 + 4P_2 + 2P_3 + 3P_4 \\
Q_4^{(3)} &= Q_2^{(4)} + Q_5^{(4)} &&= 3P_1 + 3P_2 + 2P_3 + 3P_4 \\
Q_5^{(3)} &= Q_1^{(4)} + Q_5^{(4)} &&= 3P_1 + 3P_2 + 3P_3 + 3P_4
\end{aligned}$$

Compute the rows of $A^{(2)}$ using $D^{(2)}$ and the above points:

$$\begin{aligned}
Q_1^{(2)} &= 2Q_3^{(3)} &&= 6P_1 + 8P_2 + 4P_3 + 6P_4 \\
Q_2^{(2)} &= Q_2^{(3)} + Q_3^{(3)} &&= 5P_1 + 8P_2 + 4P_3 + 6P_4 \\
Q_3^{(2)} &= Q_2^{(3)} + Q_4^{(3)} &&= 5P_1 + 7P_2 + 4P_3 + 6P_4 \\
Q_4^{(2)} &= Q_2^{(3)} + Q_5^{(3)} &&= 5P_1 + 7P_2 + 5P_3 + 6P_4 \\
Q_5^{(2)} &= Q_1^{(3)} + Q_5^{(3)} &&= 5P_1 + 7P_2 + 5P_3 + 5P_4
\end{aligned}$$

And finally, compute the rows of $A^{(1)}$ using $D^{(1)}$ and the above points:

$$Q_1^{(1)} = 2Q_4^{(2)} \qquad = 10P_1 + 14P_2 + 10P_3 + 12P_4$$

$$Q_2^{(1)} = Q_4^{(2)} + Q_5^{(2)} = 10P_1 + 14P_2 + 10P_3 + 11P_4$$
$$Q_3^{(1)} = Q_3^{(2)} + Q_5^{(2)} = 10P_1 + 14P_2 + 9P_3 + 11P_4$$
$$Q_4^{(1)} = Q_2^{(2)} + Q_5^{(2)} = 10P_1 + 15P_2 + 9P_3 + 11P_4$$
$$Q_5^{(1)} = Q_1^{(2)} + Q_5^{(2)} = 11P_1 + 15P_2 + 9P_3 + 11P_4$$

As we wanted to compute row 3 of $A^{(1)}$, we output $Q_3^{(1)}$.

References

[1] A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik, and S. Vanstone. Accelerated verification of ECDSA signatures. *Selected Areas in Cryptography, SAC 2005, Lecture Notes in Computer Science*, 3897:307–318, 2005.

[2] R. Azarderakhsh and K. Karabina. A New double point multiplication algorithm and its application to binary elliptic curves with endomorphisms. *IEEE Transactions on Computers*, 63:2614–2619, 2014.

[3] R. Azarderakhsh and K. Karabina. Efficient algorithms and architectures for double point multiplication on elliptic curves. Proceedings of the Third Workshop on Cryptography and Security in Computing Systems - CS2 2016, 2016.

[4] D. Bernstein. Differential addition chains. Technical report, 2006. Available at `http://cr.yp.to/ecdh/diffchain-20060219.pdf`.

[5] S. Bernstein and T. Lange. Explicit-formulas database. `http://hyperelliptic.org/EFD/`.

[6] J. Bos, C. Costello, H. Hisil, and K. Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. *Cryptographic Hardware and Embedded Systems - CHES 2013, Lecture Notes in Computer Science*, 8086:2013, 331-348.

[7] D. Brown. Multi-Dimensional Montgomery Ladders for Elliptic Curves. ePrint Archive: Report 2006/220. Available at `http://eprint.iacr.org/2006/220`.

[8] C. Costello and P. Longa. FourQ: Four-dimensional decompositions on a Q-curve over the Mersenne prime. *Advances in Cryptology ASIACRYPT 2015, Lecture Notes in Computer Science*, 9452:214–235, 2015.

[9] A. Faz-Hernandez, P. Longa, and A. Sanchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. *Topics in Cryptology CT-RSA 2014, Lecture Notes in Computer Science*, 8366:1–27, 2014.

[10] M. Feng, B. Zhu, C. Zhao, and S. Li. Signed MSB-set comb method for elliptic curve point multiplication. *Information Security Practice and Experience - ISPEC 2006, Lecture Notes in Computer Science*, 3903:13–24, 2006.

[11] 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.

[12] R. Gallant, R. Lambert, and S. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. *Advances in Cryptology - CRYPTO 2011, LNCS*, 2139:190–200, 2001.

[13] A. Guillevic and S. Ionica. Four-dimensional GLV via the Weil restriction. *Advances in Cryptology, ASIACRYPT 2013, Lecture Notes in Computer Science*, 8269:79–96, 2013.

[14] D. Hankerson, K. Karabina, and A. Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. *IEEE Transactions on Computers*, 58:1411–1420, 2009.

[15] M. Hedabou, P. Pinel, and L. Beneteau. Countermeasures for preventing comb method against SCA attacks. *Information Security Practice and Experience - ISPEC 2005*, 3439:85–96, 2005.

[16] H. Hisil, K. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. *Advances in Cryptology - ASIACRYPT 2008, Lecture Notes in Computer Science*, 5350:326–343, 2008.

[17] M. Joye and M. Tunstall. Exponent recoding and regular exponentiation algorithms. *Lecture Notes in Computer Science, AFRICACRYPT 2009*, 5580:334–349, 2009.

[18] C. Lim and P. Lee. More flexible exponentiation with precomputation . *Advances in Cryptology CRYPTO 94, Lecture Notes in Computer Science*, 839:95–107, 1994.

[19] P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. *Advances in Cryptology, ASIACRYPT 2012, Lecture Notes in Computer Science*, 7658:718–739, 2012.

[20] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. New York, 1996.

[21] B. Möller. Algorithms for multi-exponentiation. *Selected Areas in Computer Science SAC 2001, LNCS*, 2259:165–180, 2001.

[22] P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[23] P. Montgomery. Evaluating recurrences of form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas chains. `www.cwi.nl/ftp/pmontgom/Lucas.ps.gz`, December 13, 1983; Revised March, 1991 and January, 1992.

[24] T. Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. *Advances in Cryptology CRYPTO 92, Lecture Notes in Computer Science*, 740:31–53, 1993.

[25] K. Okeya and T. Takagi. The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. *Topics in Cryptology - CT-RSA 2003, Lecture Notes in Computer Science*, 2612:328–343, 2003.

[26] Srinivasa Rao Subramanya Rao. Three dimensional Montgomery ladder, differential point tripling on Montgomery curves and point quintupling on Weierstrass and Edwards curves. *Progress in Cryptology AFRICACRYPT 2016, Lecture Notes in Computer Science*, 9646:84–106, 2016.

[27] V. Shoup. Ntl: A Library for doing number theory. http://www.shoup.net/ntl/.

[28] M. Stam. *Speeding up Subgroup Cryptosystems*. PhD thesis, Technische Universiteit Eindhoven, 2003.

[29] Z. Zhou, Z. Hu, M. Xu, and W. Song. Efficient 3-dimensional GLV method for faster point multiplication on some GLS elliptic curves. *Information Processing Letters*, 110:1003–1006, 2010.