On Computing the Multidimensional Scalar Multiplication on Elliptic Curves

Walid Haddaji[1,3*], Loubna Ghammam[2], Nadia El Mrabet[3], Leila Ben Abdelghani[4]

[1,3*]Science and technology for defense lab LR19DN01, Center for military research, military academy, Tunis, Tunisia.
[2]ITK Engineering GmbH, Im Speyerer Tal 6 Rülzheim76761Germany.
[3]Laboratory of Secure System and Architecture (SSA), Ecole des Mines de Saint Etienne, 880 Rte de Mimet, Campus Georges Charpak Provence, 13120, Gardanne, France.
[4]Laboratory of Analysis, Probability and Fractals, Faculty of Sciences, Environment Avenue, Omrane, 5000, Monastir, Tunisia.

*Corresponding author(s). E-mail(s): haddajiwalid95@gmail.com;
Contributing authors: loubna.ghammam@itk-engineering.de;
nadia.elmrabet@emse.fr; leila.benabdelghani@fsm.rnu.tn;

Abstract

A multidimensional scalar multiplication ($\boldsymbol{d}$-mul) consists of computing $[\boldsymbol{a_1}]\boldsymbol{P_1} + \cdots + [\boldsymbol{a_d}]\boldsymbol{P_d}$, where $\boldsymbol{d}$ is an integer ($\boldsymbol{d \geq 2}$), $\boldsymbol{\alpha_1, \cdots, \alpha_d}$ are scalars of size $\boldsymbol{l} \in \mathbb{N}^*$ bits, $\boldsymbol{P_1, P_2, \cdots, P_d}$ are points on an elliptic curve $\boldsymbol{E}$. This operation ($\boldsymbol{d}$-mul) is widely used in cryptography, especially in elliptic curve cryptographic algorithms. Several methods in the literature allow to compute the $\boldsymbol{d}$-mul efficiently (e.g., the bucket method [1], the Karabina et al. method [2–4]). This paper aims to present and compare the most recent and efficient methods in the literature for computing the $\boldsymbol{d}$-mul operation in terms of with, complexity, memory consumption, and proprieties. We will also present our work on the progress of the optimisation of $\boldsymbol{d}$-mul in two methods. The first method is useful if $\boldsymbol{2^d - 1}$ points of $\boldsymbol{E}$ can be stored. It is based on a simple precomputation function. The second method works efficiently when $\boldsymbol{d}$ is large and $\boldsymbol{2^d - 1}$ points of $\boldsymbol{E}$ can not be stored. It performs the calculation on the fly without any precomputation. We show that the main operation of our first method is $\boldsymbol{100(1 - \frac{1}{d})\%}$ more efficient than that of previous works, while our second exhibits a $\boldsymbol{50\%}$ improvement in efficiency.

These improvements will be substantiated by assessing the number of operations and practical implementation.

Keywords: Elliptic curves, multidimensional scalar multiplication (***d***-mul), scalar multiplication, complexity

1 Introduction

Elliptic curve scalar multiplication is the main operation of the elliptic curve based algorithms such as EdDSA [5] for the signature scheme, and Elliptic Curve Diffie Hellman protocol ECDH [6]. This operation enables the repetition of adding a point P from an elliptic curve E to itself k times to result in a new point on E. This point is denoted as $[k]P$, where $k \in \mathbb{Z}$. Several works optimising this operation have been proposed in this context. Among these methods, one can cite the double and add method [7], the addition subtraction method [7] and the window method (w-method [8]). Cryptographic applications can benefit from multidimensional scalar multiplication algorithms (d-mul). For example, the signature verification of ECDSA [9] and the SIDH protocol [10] require a d-mul for $d = 2$. Multidimensional scalar multiplication can also speed up simple scalar multiplication [4]. In recent years, d-mul algorithms have attracted attention in this context. A d-mul takes d scalars $\alpha_1, \alpha_2, \ldots, \alpha_d$ and d points from the elliptic curve E, denoted as $P_1, P_2, \ldots, P_d$, and produces the result $[\alpha_1]P_1 + [\alpha_2]P_2 + \ldots + [\alpha_d]P_d$, where $l \in \mathbb{N}^*$ and $log_2(\alpha_i) = l$ for $1 \leq i \leq d$. The best known d-mul algorithms are discussed in papers such as [2–4, 11–15]. While [11, 13–15] focus on special cases of d, [2–4, 12] generalise d-mul for elliptic curves.

Our contribution:

Initially, we were motivated by the usefulness of a d-mul with a small d in certain cryptographic schemes. For this reason, we exploited the possibility of precomputation and storage to design a first method with an efficient main operation. While preparing this first method, we discovered newer schemes based on zkSNARK [16, 17] that require a d-mul with a large number of scalars. In this case, our first method is no longer applicable. Furthermore, according to [16, 17] the computation of the d-mul is the bottleneck in zkSNARK based schemes. This motivated us to develop a second method that doesn't require precomputation and storage to efficiently compute a d-mul with a large d. Our methods represent a revitalisation of the design, in contrast, to [17–19] which focus on optimising the bucket method through software and hardware enhancements. These methods include some countermeasures against side channel attacks. In addition, the second method, designed for large d, is algorithmically simple and does not require many memory accesses.

Notations:

Let $a, b, x \in \mathbb{Z}$, and C be a matrix. In the rest of this paper, we use the following notations:

- $[|a, b|]$ is the set of integers between a and b,
- $HW(x)$ is the Hamming weight of x,
- C^t is the transposition of C,
- p is a prime number,
- $\mathbb{K}$ is a finite field of characteristic p,
- E is an elliptic curve defined over $\mathbb{K}$,
- P_∞ represents the identity of the group $(E(\mathbb{K}), +)$,
- **A** is an addition on E,
- **D** is a doubling on E.

This paper is structured as follows: First, we present in Section 2 a State of the Art of the known methods for multidimensional scalar multiplication (d-mul). Section 3 introduces two new methods for efficient computation of d-mul. The first deals with the case of a small parameter d using precomputation. It takes inspiration from Shamir's trick, namely when $d = 2$. The second method does not rely on precomputation. Instead, it performs d-mul calculations on the fly. It processes simultaneously the bits of scalars α_i, where $i \in [|1, d|]$. In Section 5, we make a detailed comparison of the complexity of our work with that of other existing methods. We compare some key properties of the presented methods, with a special focus on comparing our methods with the optimized version of Karabina et al.'s algorithms in terms of main operation complexity, as these represent the latest advances in d-mul. This comparison concerns the level of curve arithmetic (number of additions and doublings) and coordinate systems (affine, projective, and Jacobian). For clarity, we refer to the methods of Karabina et al. [2–4] as d-mul methods. We then apply the optimized version of d-mul methods and our first method on the $secp256k1$ curve [20] used in ECDSA [9] and the $Montgomery$ curve [21] used in SQIsign [22] to compare running times for small d. We repeat this comparison by applying the same version of d-mul methods and our second method to the $BLS12-381$ curve [23] used in zkSNARK protocols for large d. Note that the improvements proposed in this paper are applicable to any elliptic curve. Our paper concludes with a summary of the results and contributions.

2 State of the Art

The d-mul problem consists of computing $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$, where d, l, and $\alpha_1, \cdots, \alpha_d$ are integers such that $d \geq 2$, $l \geq 1$, with $1 \leq i \leq d$, $\alpha_i \in [0, 2^l - 1]$ and $P_1, P_2, \cdots, P_d \in E$.

2.1 Case of d=2

2.1.1 Simultaneous Scalar Multiplication (Shamir's trick [24, 25])

Shamir's trick allows the computation of $[\alpha_1]P_1 + [\alpha_2]P_2$. This method is given in detail in [26]. Let $l \in \mathbb{N}^*$, α_1, α_2 be two integers such that $l = log_2(\alpha_i)$, with $i \in \{1, 2\}$.

The binary representations of α_1 and α_2 are given by:

$$\begin{aligned}(\alpha_1)_2 &= (b^{(1)}_{l-1}b^{(1)}_{l-2}\cdots b^{(1)}_1 b^{(1)}_0)_2,\\ (\alpha_2)_2 &= (b^{(2)}_{l-1}b^{(2)}_{l-2}\cdots b^{(2)}_1 b^{(2)}_0)_2.\end{aligned}$$

Note that $\forall j \in [0, l-1], \forall i \in \{1,2\}, b^{(i)}_j \in \{0,1\}$, $b^{(i)}_{l-1}$ is denoted by the most significant bit, and $b^{(i)}_0$ is the least significant bit. We define the $2 \times l$ matrix C by:

$$C = \begin{pmatrix} b^{(1)}_{l-1} & \cdots & b^{(1)}_1 & b^{(1)}_0 \\ b^{(2)}_{l-1} & \cdots & b^{(2)}_1 & b^{(2)}_0 \end{pmatrix}$$

where, for each integer k, we use $[k]$ to express the scalar multiplication of a given point on E by k. We remark that

$$\begin{aligned}[\alpha_1]P_1 + [\alpha_2]P_2 &= (P_1, P_2)C\begin{pmatrix} 2^{l-1} \\ \vdots \\ 2^1 \\ 2^0 \end{pmatrix} \\ &= [2^{l-1}]([b^{(1)}_{l-1}]P_1 + [b^{(2)}_{l-1}]P_2) + \cdots + [2^1]([b^{(1)}_1]P_1 \\ &\quad + [b^{(2)}_1]P_2) + [2^0]([b^{(1)}_0]P_1 + [b^{(2)}_0]P_2).\end{aligned}$$

This computation leads to the Algorithm 1, which takes as inputs:

- α_1, $\alpha_2 \in \mathbb{N}$,
- l: the size (in bits) of the longest scalar,
- $P_1, P_2 \in E$.

The number of additions depends on the so-called Joint Hamming Weight (JHW) of α_1 and α_2 defined in [26] as the number of non zero columns in the matrix C. It is possible to extend the JHW definition to any finite number of scalars. Since α_1 and α_2 are randomly generated and have a size of l bits, then $JHW(\alpha_1, \alpha_2) \approx \frac{3}{4}l$. Thus, the main operation of this method involves l steps. At each step $j \in [0, l-1]$, one doubling is performed, and if $(b^{(1)}_j, b^{(2)}_j) \neq (0,0)$, one addition is also performed. So the complexity of the main operation is approximately

$$\frac{3}{4}l\mathbf{A} + (l-1)\mathbf{D}.$$

For the precomputation, one addition is performed. The non-adjacent form (NAF [8]) of α_1 and α_2 can be used instead of their binary representations to optimize the complexity of Shamir's trick. In fact, NAF is one of the signed binary representations

Algorithm 1 Simultaneous Scalar Multiplication (Shamir's trick [24, 25])

Input: $l \geq 1$, $\alpha_1 = (b_{l-1}^{(1)} \cdots b_1^{(1)} b_0^{(1)})_2$, $\alpha_2 = (b_{l-1}^{(2)} \cdots b_1^{(2)} b_0^{(2)})_2$, $P_1, P_2 \in E$
Output: $[\alpha_1]P_1 + [\alpha_2]P_2$

1. **precomputation** $G \leftarrow P_1 + P_2$.
2. $R \leftarrow P_\infty$.
3. **for** $j = l-1$ **down to** 0 **do**:
 3. 1 $R \leftarrow [2]R$.
 3. 2 **if** $((b_j^{(1)}, b_j^{(2)}) = (1,0))$ **then** $R \leftarrow R + P_1$.
 3. 3 **else if** $((b_j^{(1)}, b_j^{(2)}) = (0,1))$ **then** $R \leftarrow R + P_2$.
 3. 4 **else if** $((b_j^{(1)}, b_j^{(2)}) = (1,1))$ **then** $R \leftarrow R + G$.
4. **return** R.

admitting the smallest number of non zero digits. We illustrate it in the following example:

$$\alpha = (11100011111011)_2 = (100\bar{1}00100000\bar{1}0\bar{1})_{NAF},$$

where $\bar{1} = -1$. Compared to the binary representation, the use of this representation allows us to reduce the number of additions. In fact, $HW(\alpha_1) \approx HW(\alpha_2) \approx \frac{l}{3}$, which makes $JHW(\alpha_1, \alpha_2) \approx \frac{5}{9}l$. Consequently, if we use the NAF representation, Shamir's trick requires the precomputation of two points, $G = P_1 + P_2$ and $H = P_1 - P_2$, and has the following main operation complexity:

$$\frac{5}{9}l\mathbf{A} + l\mathbf{D}.$$

2.1.2 Bernstein's method (DJB)

Let $l \in \mathbb{N}^*$, and $\alpha_1, \alpha_2 \in [0, 2^l - 1]$. The DJB method allows to compute $[\alpha_1]P_1 + [\alpha_2]P_2$ based on the concept of the differential addition chain [11] that is an addition chain in which P_1 and P_2 are represented by their difference $P_1 - P_2$. Montgomery observed in [27] that for P_1 and P_2 two points of the curve $y^2 = x^3 + ax^2 + x$ (Montgomery's curve), we can efficiently compute the $x-$coordinate of $P_1 + P_2$ from the $x-$coordinates of P_1, P_2, and $P_2 - P_1$. Based on this observation and the concept of a differential addition chain, Bernstein designed in [11] a method to compute $[\alpha_1]P_1 + [\alpha_2]P_2$ on the Montgomery curve for l-bit scalars α_1 and α_2. At each step, one doubling and two additions are performed, which ensures the uniformity of the operations. More precisely, three temporal variables T_1, T_2 and T_3 are initialized, respectively, to P_∞, P_1 and P_2. Then, they are updated at each step by doubling one of the T_i and adding two different pairs of points. The rules of this updating are based on a recursive formula applied to the bits of α_1 and α_2 given in [11]. As an example, we show in Figure 1 how this formula works to construct a differential addition chain of $(13, 17)$. The complexity of the main operation in DJB is given by:

$$2l\mathbf{A} + l\mathbf{D}.$$

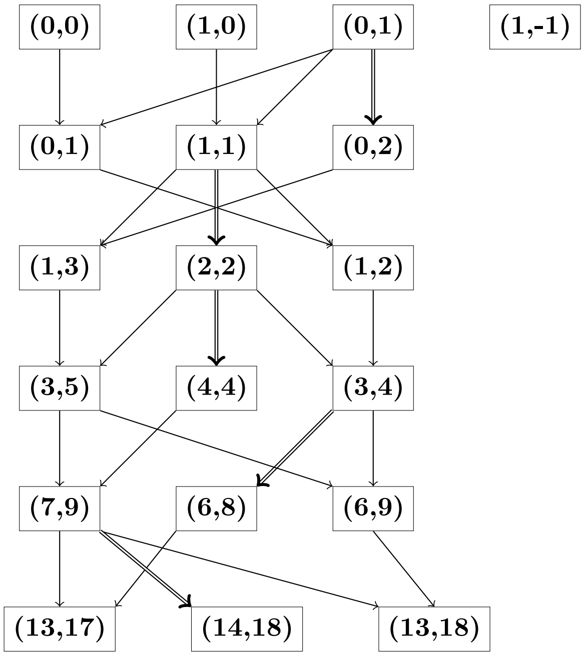

Figure 1: The differential addition chain of $(13, 17)$.

Table 1 compares the Bernstein and Shamir methods.

Method	Main operation	Precomputation	Storage	Uniformity
Shamir's trick	$l\mathbf{D}+\frac{3}{4}l\mathbf{A}$	$1\mathbf{A}$	3 points	$\times$
Bernstein	$l\mathbf{D}+2l\mathbf{A}$	$-$	2 points	✓

Table 1: Comparison of the Bernstein and Shamir methods.

2.2 Case of d > 2

Recall that our goal is to compute efficiently $[\alpha_1]P_1+[\alpha_2]P_2+\cdots+[\alpha_d]P_d$, where $d \geq 2$, $\alpha_1, \cdots, \alpha_d \in \mathbb{N}$ with $log_2(\alpha_i) = l$, for some $l \in \mathbb{N}^*$, $1 \leq i \leq d$ and $P_1, \cdots, P_d \in E$.

2.2.1 Interleaving with NAFs

Interleaving is a technique, as described in [8], used to prepare precomputed points for each $[\alpha_i]P_i$ using distinct methods. During each step of the main operation, we perform a doubling operation and addition by employing the precomputed points corresponding to each $[\alpha_i]P_i$. In this context, we introduce an interleaving approach based on window non adjacent forms (wNAFs). Specifically, for $1 \leq i \leq d$, we assign a window width

$w_i \in \mathbb{N}^*$ to α_i using Algorithm 3 in [28]. We calculate $w_i\mathrm{NAF}(\alpha_i) = \sum_{j=0}^{l_i-1}(k_j^i 2^i)$, where $k_j^i \in \{1, 3, \cdots, 2^{w_i}-1\}$ and l_i is the number of digits k_j^i for each i. Subsequently, for $1 \le i \le d$, we precompute the points $[j]P_i$ for odd values of $j < 2^{w_i-1}$. During the main operation, the digits of scalars α_i are jointly processed from left to right to perform a single doubling. This method is outlined in Algorithm 3.51 in [8]. It requires $\sum_{i=1}^{d}(2^{w_i-2})$ as storage points. Its precomputation is given by:

$$\#\{i; w_i > 2\}\mathbf{D} + \sum_{i=1}^{d}(2^{w_i-2} - 1)\mathbf{A}$$

and its main operation complexity is

$$l\mathbf{D} + l\sum_{i=1}^{d}(\frac{1}{w_i+1})\mathbf{A}$$

2.2.2 Bucket method (Pippenger Algorithm)

This method is proposed in [1] and described in detail in [17–19]. It is based on the Pippenger algorithm, which was originally designed to efficiently compute the multi exponent multiplication [17]. It is hardware optimised in [17] for d-mul speed-up. It is parallelized in [19] to provide an efficient GPU implementation of zkSNARK. It is used in [18] with a new coordinate system for twisted Edwards curves to speed up the d-mul for a large d. The bucket method works as follows:

- We select an integer $w \ge 2$ such that $w \approx log_2(d)$, w is referred to as the 'window width',
- For each $i \le d$, we write $\alpha_i = (\alpha_{i,\lceil \frac{l}{w} \rceil}, \cdots, \alpha_{i,1})_{2^w}$. This implies that we partition each α_i into $\lceil \frac{l}{w} \rceil$ words, each word is of size w bits,
- For each $0 \le j \le \lceil \frac{l}{w} \rceil - 1$:
  - Let $P^{[j]} = [\alpha_{1,j}]P_1 + [\alpha_{2,j}]P_2 + \cdots + [\alpha_{d,j}]P_d$,
  - We range the points $P_1, \cdots, P_d$ into 2^w buckets $(B_0^{[j]}, B_1^{[j]}, \cdots, B_{2^w-1}^{[j]})$ according to $\alpha_{i,j}$ and we eliminate the bucket 0, with $1 \le i \le d$,
  - We add the points in each bucket to obtain the sums $S_1^{[j]}, S_2^{[j]}, \cdots, S_{2^w-1}^{[j]}$,
  - We compute $P^{[j]} = \sum_{k=1}^{2^w-1}[k]S_k^{[j]}$ using an efficient method proposed in [1].
- After computing of all $P^{[j]}$, we calculate the final result, $\sum_{j=0}^{\lceil \frac{l}{w} \rceil - 1}[2^{jw}]P^{[j]}$, using an inverse recursive method described in [19].

This method is given in Algorithm 1 in [17]. Its main operation complexity is given by:

$$l\mathbf{D} + \lceil \frac{l}{w} \rceil (d + 2^{w+1})\mathbf{A}$$

2.2.3 *d*-mul methods

We present the three versions of d-mul methods in chronological order as follows:

***d*-mul**

The d-mul is described in Algorithm 1 in [2]. It requires, as a first step, transforming the scalars into a particular matrix A known as the state matrix [3]. This transformation is called the initialisation step. It is illustrated in Algorithm 1 in [2]. Then it constructs a matrices'chain $(A^{(i)})_{i=1}^{l}$ from A such that $A^{(1)} = A$ using Algorithm 2 in [2]. This chain is called a chain of state matrices. Finally, it carries out successive linear combinations on the points P_i, moving in the opposite direction to that of the construction, until obtaining the sum $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.

Randomised d-mul

Recall that the magnitude of a state matrix is defined in [3] as being the largest value among the absolute values of the matrix entries. Let $d, l \in \mathbb{N}^*$, $P_1, P_2, \cdots, P_d \in E$, r a binary string of size ld. Let v be a binary vector of size d. r and v are generated uniformly at random. Let σ be a permutation on $\{2, \cdots, d+1\}$. Randomised d-mul (Algorithm 2 in [3]) produces a random point of the form $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$, where $\alpha_1, \alpha_2, \cdots, \alpha_d \in [0, 2^l - 1]$ are randomly generated from r. Unlike d-mul, randomised d-mul avoids constructing a chain of state matrices from a matrix of a given magnitude. In fact, using a permutation on $\{0, 1, \cdots, d-1\}$, it constructs a chain of state matrices from a matrix of magnitude 1. The magnitudes of the matrices in this chain form a strictly increasing sequence. The magnitude of the final matrix is determined by the binary chain r. Furthermore, randomised d-mul performs the needed computation on the points P_i at each construction step. It ends with a random point as a linear combination of these points. We can use Algorithm 1 in [3] to recover the scalars.

Optimised d-mul

Randomised d-mul does not allow the scalars to be chosen apriory. Therefore, Hutchinson and Karabina proposed the optimised d-mul [4] to ensure this correctness. It optimises computations compared to d-mul by using bit permutations and XOR operation. The use of negative scalars is supported by the optimised d-mul. In fact, if a scalar is negative, it must be replaced by its symmetric, which is positive, and the point associated with it must also be replaced by its symmetric. Therefore, the Algorithm 4 in [4] presents a version of optimised d-mul with positive scalars Furthermore, optimised d-mul requires the generation of a random permutation σ on $\{1, \cdots, d\}$ using the parity of the scalars α_i (Algorithm 1 in [4]).
We compare d-mul methods in Table 2.

Method	Main operation complexity	Precomputation	Static table	Temporal storage
d-mul	$l\mathbf{D}+ld\mathbf{A}$.	$d\mathbf{A}$. $2ld$ conditional tests. $2ld$ sums of 2 rows of a $(d+1)\times d-$matrix.	d points.	$l(d+1)d$ scalars.
Randomised d-mul	$l\mathbf{D}+ld\mathbf{A}$. ld bits additions . ld conditional tests.	$d\mathbf{A}$.	d points.	ld bits.
Optimised d-mul	$l\mathbf{D}+ld\mathbf{A}$.	ld 2−bit XORs. $d\mathbf{A}$.	d points.	ld bits.

Table 2: Comparison of d-mul methods

The authors of all versions of d-mul methods have recommended to:

- Forget about using the first version of d-mul methods [2],
- Use the randomised d-mul method [2] when the cryptographic application requires the use of random scalars,
- Employ optimised d-mul [4] when those scalars are prefixed.

Since we are working in the context of elliptic curve optimisation, we will use the latest version of d-mul methods (optimised d-mul) for our next comparisons.

3 Our methods

This paper aims to compute efficiently the following d-mul:

$$[\alpha_1]P_1 + [\alpha_2]P_2 + \cdots + [\alpha_d]P_d,$$

where $d \geq 2$, $l \in \mathbb{N}^*$, $\alpha_1, \cdots, \alpha_d$ are scalars with $log_2(\alpha_i) = l$, and $P_1, \cdots, P_d \in E$. In this work, we distinguish between two cases. The first case is when d is small, and the second one is when d is large.
In the following, we explain when d is small or large:
we suppose that our available storage space allows us to store at most $2^M - 1$ points, where $M \in \mathbb{N}^*$, then

- If $d \leq M$, d is small,
- If $d > M$, d is large.

In this section, we present two methods of an efficient computation of d-mul. The first method when d is small and we have precomputation steps. It is referred to as **Multidimensional Shamir's trick**. The second method consists in the calculation of the d-mul without precomputation and when d is large. This method is called **Multiple double and add**.

3.1 First method (Multidimensional Shamir's trick)

While various methods for multidimensional scalar multiplication exist in the literature, it's worth mentioning that Shamir's trick (Algorithm 1) hasn't been adapted for any d parameter. In response, rather than offering a direct extension, we present, in this section, a generalisation based on a specific precomputation technique.

3.1.1 Description

Recall that this method is considered when d is small (see above for the definition of small). As we need to store some precomputed points. This case applies to several cryptographic algorithms, including digital signature algorithms EdDSA [5], ECDSA [9], and post quantum signature SQISign [22].

This method aims to efficiently compute the following d-mul:

$$[\alpha_1]P_1 + [\alpha_2]P_2 + \cdots + [\alpha_d]P_d,$$

where $P_1, \cdots, P_d \in E$ and $\alpha_1, \cdots, \alpha_d \in \mathbb{N}$, with $log_2(\alpha_i) = l$, $1 \leq i \leq d$. Note that if there are $i \in [|1, d|]$ such that $log_2(\alpha_i) = m < l$, we add $l - m$ zeros to the left of the most significant bits of α_i. This is because, at each step $j \in [0, l-1]$, we need to precompute elliptic curve points in the format $[c_1]P_1 + [c_2]P_2 + \cdots + [c_d]P_d$, with $c_1, c_2, \cdots, c_d$ represent the d bits that are used at each j^{th} step in the main operation. Let us write the scalars $\alpha_1, \alpha_2, \cdots, \alpha_d$ in their binary representations as follows:

$$\begin{aligned}
\alpha_1 &= (b_{l-1}^{(1)} b_{l-2}^{(1)} \cdots b_1^{(1)} b_0^{(1)})_2, \\
\alpha_2 &= (b_{l-1}^{(2)} b_{l-2}^{(2)} \cdots b_1^{(2)} b_0^{(2)})_2, \\
&\vdots \\
\alpha_{d-1} &= (b_{l-1}^{(d-1)} b_{l-2}^{(d-1)} \cdots b_1^{(d-1)} b_0^{(d-1)})_2, \\
\alpha_d &= (b_{l-1}^{(d)} b_{l-2}^{(d)} \cdots b_1^{(d)} b_0^{(d)})_2.
\end{aligned}$$

Note that, $\forall j \in [0, l-1], \forall i \in [|1, d|]$, $b_j^{(i)} \in \{0, 1\}$, $b_{l-1}^{(i)}$ is denoted by the most significant bit and $b_0^{(i)}$ is the least significant bit. We present the scalars $(\alpha_i)_{1 \leq i \leq d}$ in the matrix C as follows:

$$C = \begin{pmatrix} b_{l-1}^{(1)} & \cdots & b_1^{(1)} & b_0^{(1)} \\ b_{l-1}^{(2)} & \cdots & b_1^{(2)} & b_0^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{l-1}^{(d)} & \cdots & b_1^{(d)} & b_0^{(d)} \end{pmatrix}$$

We arrange the points $(P_i)_{1 \leq i \leq d}$ within the matrix row P in the following manner:

$$P = \begin{pmatrix} P_1 & P_2 & \cdots & P_d \end{pmatrix}$$

We place the powers $(2^j)_{0\leq j\leq l-1}$in the matrix column S as follows:

$$S = \begin{pmatrix} 2^{l-1} \\ \vdots \\ 2^1 \\ 2^0 \end{pmatrix}$$

Thus, we get:

$$\begin{aligned}
[\alpha_1]P_1 + [\alpha_2]P_2 + \cdots + [\alpha_d]P_d &= PCS \\
&= P \begin{pmatrix} b_{l-1}^{(1)} & \cdots & b_1^{(1)} & b_0^{(1)} \\ b_{l-1}^{(2)} & \cdots & b_1^{(2)} & b_0^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{l-1}^{(d)} & \cdots & b_1^{(d)} & b_0^{(d)} \end{pmatrix} \begin{pmatrix} 2^{l-1} \\ \vdots \\ 2^1 \\ 2^0 \end{pmatrix} \\
&= [2^{l-1}]([b_{l-1}^{(d)}]P_d + \cdots + [b_{l-1}^{(2)}]P_2 + [b_{l-1}^{(1)}]P_1) \\
&+ [2^{l-2}]([b_{l-2}^{(d)}]P_d + \cdots + [b_{l-2}^{(2)}]P_2 + [b_{l-2}^{(1)}]P_1) \\
&\vdots \\
&+ [2^1]([b_1^{(d)}]P_d + \cdots + [b_1^{(2)}]P_2 + [b_1^{(1)}]P_1) \\
&+ [2^0]([b_0^{(d)}]P_d + \cdots + [b_0^{(2)}]P_2 + [b_0^{(1)}]P_1)
\end{aligned}$$

From our computation, we remark that we can precompute and store the points of the form $\sum_{\substack{i=1 \\ b_j^{(i)} \neq 0}}^{d} P_i$ in a table T. It is possible to use a storage T since we assumed that d is small. Then, we can simplify the computation of the d-mul and we get:

$$[\alpha_1]P_1 + [\alpha_2]P_2 + \cdots + [\alpha_d]P_d = [2^{l-1}]T[x_{l-1}] + [2^{l-2}]T[x_{l-2}] + \cdots + [2^1]T[x_1] + [2^0]T[x_0].$$

with $\forall j \in [0, l-1], x_j = \bigoplus_{i=1}^{d}(b_j^{(i)} << (i-1))$ and $T[x_j] = \sum_{\substack{i=1 \\ b_j^{(i)} \neq 0}}^{d} P_i$, where:

$$\begin{cases} \bullet \bigoplus \quad \textit{is the XOR operation}, \\ \bullet << \quad \textit{is the left shift operation}. \end{cases}$$

The $T[x_i]$ are precomputed, stored, and considered later in the main operation. Note that the table T is constructed in the following way:

$$
\begin{aligned}
P_\infty &\longrightarrow T[0],\\
P_1 &\longrightarrow T[1],\\
P_2 &\longrightarrow T[2],\\
P_1 + P_2 &\longrightarrow T[3],\\
P_3 &\longrightarrow T[4],\\
P_1 + P_3 &\longrightarrow T[5],\\
P_2 + P_3 &\longrightarrow T[6],\\
&\vdots\\
P_1 + P_2 + \cdots + P_d &\longrightarrow T[2^d - 1].
\end{aligned}
$$

This method is detailed in Algorithm 2, maintaining its similarity to Shamir's trick when $d = 2$. This algorithm takes as inputs the scalars $\alpha_1, \alpha_2, \cdots, \alpha_d$. It outputs the point $Q = [\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.
For each $j \in [0, l-1]$, in the j^{th} step of the main operation, we use an integer $x_j = \bigoplus_{i=1}^{d}(b_j^{(i)} << (i-1))$ to locate the stored points to be processed. The parameter x_j is the positive integer whose binary representation is $b_j^{(d)} b_j^{(d-1)} \cdots b_j^{(2)} b_j^{(1)}$.

Algorithm 2 Multidimensional Shamir's trick

Input: $\alpha_1 = (b_{l-1}^{(1)} \cdots b_0^{(1)})_2, \cdots, \alpha_d = (b_{l-1}^{(d)} \cdots b_0^{(d)})_2$,
$P_1, P_2, \cdots, P_d \in E$.
Output: $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.

1. **Precompute** $V = \{T[0], T[1], \cdots, T[2^d - 1]\}$.
2. $Q = P_\infty$.
3. **for** $j = l-1$ **down to** 0 **do**
   3. 1 $Q \leftarrow [2]Q$.
   3. 2 $x_j \leftarrow \bigoplus_{i=1}^{d}(b_j^{(i)} << (i-1))$.
   3. 3 **if** $x_j \neq 0$ **then** $Q \leftarrow Q + V[x_j]$.
4. **return** Q.

3.1.2 Complexity

This method costs a storage of $2^d - 1$ points and requires $2^d - d - 1$ additions as precomputation. When uniformity is not a concern, we use the Algorithm 2, where the main operation contains l steps. During each step $j \in [0, l-1]$, we perform one doubling and one addition if at least one of the bits $b_j^{(i)}$ is non null, with $j \in [0, l-1]$ and $1 \leq i \leq d$. Consequently, the total complexity of the main operation in this method consists of l doublings and $JHW(\alpha_1, \alpha_2, \cdots, \alpha_d) \approx (1 - \frac{1}{2^d})$ additions. However, if uniformity is a priority, we opt for Algorithm 4, where the main operation precisely involves l additions and l doublings. The complexity of this method is shown in Table 3.

	Without uniformity (Algorithm 2)	With uniformity (Algorithm 4)
Main operation	$l\mathbf{D}$+$(1-\frac{1}{2^d})l\mathbf{A}$	$l\mathbf{D}$+$l\mathbf{A}$
Precomputation	$(2^d - d - 1)\mathbf{A}$	
Storage	$2^d - 1$ points	

Table 3: The complexity of the first method.

To illustrate how our method works, we present the following example:
Example 1.
Let $d = 3$*,* $\alpha_1 = 13 = (01101)_2$*,* $\alpha_2 = 17 = (10001)_2, \alpha_3 = 21 = (10101)_2$*, and* $P_1, P_2, P_3 \in E$*. Here* $l = log_2(21) = 5$*.*
As explained at the beginning of this method, table T *contains all the possibilities of the points* $\sum_{\substack{i=1 \\ b_j^{(i)} \neq 0}}^{d} P_i$*, with* $0 \leq j \leq l-1$*. In this example,* T *is given by:*

$$T = \{P_\infty, P_1, P_2, P_1 + P_2, P_3, P_1 + P_3, P_2 + P_3, P_1 + P_2 + P_3\}.$$

Thus,

$$\begin{aligned}
[13]P_1 + [17]P_2 + [21]P_3 &= PCS \\
&= \begin{pmatrix} P_1, P_2, P_3 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2^4 \\ 2^3 \\ 2^2 \\ 2^1 \\ 2^0 \end{pmatrix} \\
&= \begin{pmatrix} P_1, P_2, P_3 \end{pmatrix} \begin{pmatrix} 2^3 + 2^2 + 2^0 \\ 2^4 + 2^0 \\ 2^4 + 2^2 + 2^0 \end{pmatrix} \\
&= [2^4]T[6] + [2^3]T[1] + [2^2]T[5] + [2^1]T[0] + [2^0]T[7]
\end{aligned}$$

We compute this sum using the Algorithm 2. The complexity in this example is interpreted as follows:

- *The main operation's complexity:* $5\boldsymbol{D}+4\boldsymbol{A}$,
- *The precomputation's complexity:* $4\boldsymbol{A}$,
- *The storage: 7 points.*

If we use the Algorithm 4, the main operation's complexity is $5\boldsymbol{D}+5\boldsymbol{A}$

3.2 Second method (Multidimensional Double and add)

3.2.1 Description

In some cryptographic algorithms, such as the zkSNARK protocol, the d-mul is used for a large d (e.g. $d = 2^{22}$). Simply put, these schemes use a large number of scalars to compute the d-mul. This poses a challenge for storage on resource constrained devices. Thus, we can't store efficiently precomputed points as explained in Section 3.1. Therefore, in this section, we present an efficient method for computing d-mul with a large d. This method is called **Multidimensional double and add**. It does not rely on the precomputation. Similarly to the first method, for $1 \leq i \leq d$, the binary representation of α_i is as follows:

$$\alpha_i = (b_{l-1}^{(i)} b_{l-2}^{(i)} \cdots b_1^{(i)} b_0^{(i)})_2,$$

where l is the size of the longest scalar.
For $j \in [0, l-1]$, this method acts simultaneously on the bits $b_j^{(1)}, b_j^{(2)}, \cdots, b_j^{(d-1)}, b_j^{(d)}$. Let $x_j = \bigoplus_{i=1}^{d} (b_j^{(i)} << (i-1))$. If $x_j \neq 0$, we perform one doubling, we compute on the fly the points $T[x_j] = \sum_{\substack{i=1 \\ b_j^{(i)} \neq 0}}^{d} P_i$, and we add the result of the doubling to $T[x_j]$. We assume that we always have $x_j \neq 0$ since the probability of $x_j = 0$ is negligible when d is large. This method is detailed in Algorithm 3. In fact, the condition $x_j = 0$ is equivalent to $b_j^{(i)} = 0, \forall i \in [|1, d|]$. Since the scalars are generated uniformly at random and d is large, $x_j = 0$ has a probability of $1/2^d$, which is very small. For example, in a practical case, d could be 2^{22}. Therefore, the probability of $x_j = 0$ is $\frac{1}{2^{4194304}}$.

3.2.2 Complexity

Let $j \in [0, l-1]$, $x_j = \bigoplus_{i=1}^{d} (b_j^{(i)} << (i-1))$. At each j^{th} step, this method performs one doubling, $HW(x_j) - 1$ additions to compute the point $T[x_j] = \sum_{\substack{i=1 \\ b_j^{(i)} \neq 0}}^{d} P_i$, and one addition to add the result of the doubling to $T[x_j]$. In total, we perform one doubling and $HW(x_j)$ additions at each step. Thus, this method requires exactly $\sum_{j=0}^{l-1} HW(x_j)$ additions and l doublings. It is worth noting that x_j represents the binary sequence $b_j^{(d)} b_j^{(d-1)} \cdots b_j^{(1)} b_j^{(1)}$, where $j \in [0, l-1]$. For $i \in [|1, d|]$, each bit $b_j^{(i)}$ is derived from the scalar α_i. As α_i is generated uniformly at random, $HW(x_j)$ is approximately $\frac{d}{2}$.

Algorithm 3 Multidimensional double and add

Input: $\alpha_1 = (b_{l-1}^{(1)} \cdots b_0^{(1)})_2, \cdots, \alpha_d = (b_{l-1}^{(d)} \cdots b_0^{(d)})_2$,
$P_1, P_2, \cdots, P_d \in E$.
Output: $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.

1. $Q \leftarrow P_\infty$
2. **for** $j = l-1$ **down to** 0 **do**
 2. 1 $Q \leftarrow [2]Q$.
 2. 2 $T = P_\infty$.
 2. 3 **for** $i = 1$ **to** d **do**
  if $b_j^{(i)} \neq 0$ **then** $T \leftarrow T + P_i$.
 2. 4 $Q = Q + T$.
3. **return** Q.

The complexity of this method is approximately given by:

$$\frac{d}{2} l\mathbf{A} + l\mathbf{D}$$

To illustrate this method, we give the following example:
Example 2.
Let $d = 4$, $\alpha_1 = 17 = (10001)_2, \alpha_2 = 25 = (11001)_2, \alpha_3 = 28 = (11100)_2, \alpha_4 = 12 = (01100)_2$ *and* $P_1, P_2, P_3, P_4 \in E$. *In this example* $l = 5$ *and,*

$$\begin{aligned}
[\alpha_1]P_1 + [\alpha_2]P_2 + [\alpha_3]P_3 + [\alpha_4]P_4 &= [17]P_1 + [25]P_2 + [28]P_3 + [12]P_4 \\
&= [16](P_1 + P_2 + P_3) + [8](P_2 + P_3 + P_4) \\
&+ [4](P_3 + P_4) + [2]P_\infty + (P_1 + P_2) \\
&= [2^4]([1]P_1 + [1]P_2 + [1]P_3 + [0]P_4) \\
&+ [2^3]([0]P_1 + [1]P_2 + [1]P_3 + [1]P_4) \\
&+ [2^2]([0]P_1 + [0]P_2 + [1]P_3 + [1]P_4) \\
&+ [2^1]([0]P_1 + [0]P_2 + [0]P_3 + [0]P_4) \\
&+ [2^0]([1]P_1 + [1]P_2 + [0]P_3 + [0]P_4).
\end{aligned}$$

According to the second method for this example $x_0 = 3$, $x_1 = 0$, $x_2 = 12$, $x_3 = 14$, *and* $x_4 = 7$. *As* $HW(7) = 3$, $HW(14) = 3$, $HW(12) = 2$, $HW(0) = 0$ *and* $HW(3) = 2$, *In this example, we interpret the complexity as follows:*

- *the main operation's complexity:* $5\boldsymbol{D}+10\boldsymbol{A}$,
- *the storage:* 4 *points.*

4 Security analysis and countermeasures

In this section, we examine security concerns related to our methods and provide the corresponding countermeasures.

4.1 First method

Let $j \in [0, l-1]$ and $x_j = \bigoplus_{i=1}^{d}(b_j^{(i)} << (i-1))$. Although the probability of getting a non zero x_j is negligible, we consider this situation to be a security problem. Indeed, in the Algorithm 2, an adversary could perform a power attack to deduce that the bits of the scalars α_i used in the j^{th} step are all equal to zero. To solve this problem, we provide Algorithm 4 to consider the two possible cases, $x_j \neq 0$ and $x_j = 0$. Then we inject a false addition when $x_j = 0$ to ensure uniformity of the elliptic curve operations. Thus the complexity of the main operation reaches exactly l additions and l doublings with a slight increase compared to Algorithm 2. In other words, we do not go to great lengths to ensure this uniformity.

Algorithm 4 Uniform Multidimensional Shamir's trick

Input: $\alpha_1 = (b_{l-1}^{(1)} \cdots b_0^{(1)})_2, \cdots, \alpha_d = (b_{l-1}^{(d)} \cdots b_0^{(d)})_2$,
$P_1, P_2, \cdots, P_d \in E$.
Output: $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.

1. **Precompute** $V = \{T[0], T[1], \cdots, T[2^d - 1]\}$.
2. $Q = P_\infty$, $F \leftarrow V[d-1]$.
3. **for** $j = l-1$ **down to** 0 **do**
   3.1 $Q \leftarrow [2]Q$.
   3.2 $x_j \leftarrow \bigoplus_{i=1}^{d}(b_j^{(i)} << (i-1))$.
   3.3 **if** $x_j \neq 0$ **then** $Q \leftarrow Q + V[x_j]$.
   3.4 **else** $F \leftarrow F + Q$. //A fake addition.
4. **return** Q.

4.2 Second method

Let $j \in [0, l-1]$ and x_j be the same parameter used in the first method. Viewing the main operation within Algorithm 3 as a comprehensive entity, we consistently uphold uniformity across all steps with a high probability. In fact, the only distinct case arises when $x_j = 0$, indicating that $b_j^{(i)} = 0$ for $1 \leq i \leq d$. However, this event occurs with an extremely low probability $(\frac{1}{2^d})$ given a large d. In this rare scenario, we have the opportunity to enhance the security of our algorithm and achieve complete uniformity by introducing a fake addition operation. Although this practice is not expensive in terms of curve operations, its cost escalates significantly as d increases. This increased cost arises from the calculation of x_j and the associated conditional statements. Consequently, in Algorithm 3, we have intentionally omitted the conditional check on x_j. This decision ensures that the same block of operations

is approximately practised at every step, regardless of the value of x_j. This strategy enhances efficiency, as d grows larger.
Let us examine the Algorithm 3 more closely, concentrating on loop 2.3. It becomes apparent that an adversary has the ability to determine the values of the bits $b_j^{(i)}$ at every j^{th} step, where $1 \leq i \leq d$. This can be achieved by manipulating a power attack during the running of the addition operations to decide whether the bit $b_j^{(i)}$ should be interpreted as 0 or 1. To solve this problem, and to thwart the adversary's ability to deduce the bits of the scalars α_i, a simple solution is to apply an efficient random permutation, such as the Fisher-Yates shuffle algorithm stated in [29], to the bits $b_j^{(i)}$ during each j^{th} step. This is referred to as **full use** of permutations. Because the running time of such a permutation depends on a large d, using it for all the l steps could affect the efficiency of the method. To address this concern, we propose a solution based on the following two scenarios: one scenario where the scalars are predetermined and another where they are generated for single use.

– First scenario:
  - the scalars are predetermined,
  - we predefine a set of indices called $Perm$ from the set $\{0, 1, \cdots, l-1\}$,
  - at every j^{th} step, we generate an efficient permutation σ over the set $\{1, \cdots, d\}$ and apply it to $\{b_j^{(1)}, \cdots, b_j^{(d)}\}$ and $\{P_1, \cdots, P_d\}$ if $j \in Perm$,
  - if $j \notin Perm$, σ acts as the identity map on the set $\{1, \cdots, d\}$,
  - This operation is called **partial use** of permutations.

– Second scenario:
  - the scalars are generated for single use,
  - we choose the set $Perm$ at random from the set $\{0, \cdots, l-1\}$ each time the algorithm runs,
  - The same approach is used as in the first scenario.

In both scenarios, we ensure that the bits $b_j^{(i)}$ are hidden during certain steps, which significantly improves security. Specifically, if we assume that the cardinality of $Perm$ is denoted as $m \in [|1, l|]$, an adversary would have to perform an exhaustive attack with a complexity of 2^{dm} to reveal the hidden bits. This complexity is exceptionally high, mainly due to the large integer d in this method. We can strengthen security by extending the set $Perm$, taking into account the available computational resources. Furthermore, in the second scenario, we can weaken the adversary's effort by modifying the steps affected by the random permutations in each algorithm run. This does not affect the security of the scalars as they are used only once. At the end of each j^{th} step, we can free the memory from the generated permutation to ameliorate the efficiency of our algorithm. All of the aforementioned adjustments are represented by algorithm 5.

5 Comparison

In this section, we compare our method to those outlined in the state of the Art, assessing their complexity and various properties such as uniformity, precomputation,

Algorithm 5 Multidimensional double and add (modified)

Input: $\alpha_1 = (b_{l-1}^{(1)} \cdots b_0^{(1)})_2, \cdots, \alpha_d = (b_{l-1}^{(d)} \cdots b_0^{(d)})_2$,
$P_1, P_2, \cdots, P_d \in E$.
Output: $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.

1. $Q \leftarrow P_\infty$
2. $Perm \xleftarrow{r} \{0, 1, \cdots, l-1\}$. //Pick uniformly at random a subset $Perm$ from the set $\{0, 1, \cdots, l-1\}$.
3. **for** $j = l-1$ **down to** 0 **do**
   3.1 $Q \leftarrow [2]Q$.
   3.2 $T = P_\infty$.
   3.3 **if** $j \in Perm$ **then** generate a random permutation σ on $\{1, \cdots, d\}$.
   3.4 **else** $\sigma \leftarrow id_{\{1,\cdots,d\}}$. // σ operates as it is the identity map.
   3.5 **for** $i = 1$ **to** d **do**
   if $b_j^{(\sigma(i))} \neq 0$ **then** $T \leftarrow T + P_{\sigma(i)}$.
   3.6 $Q = Q + T$.
4. **return** Q.

windowing, and parallelisation. Our primary focus is comparing the complexity of the main operation and the running time of our method to that of the optimised d-mul, which represents the latest advancement in optimising d-mul. We initiate our comparative analysis by examining the existing methods, especially in the specific case where $d = 2$. We provide a comprehensive comparison, as shown in Table 5, for this particular case. Subsequently, we broaden our evaluation to encompass the general case, considering two scenarios: one scenario involving a small d and the other with a larger d. To simplify the comparison, we employ a uniform window size w for all scalars α_i in the Interleaving method.
Note that all our practical experiments are carried out on an Intel i7 Personal Workstation with 8GB RAM and SSD storage, using SageMath.
Before proceeding with the categorisation of our comparison based on the parameter d, it is important to note the presence of certain properties in the presented methods and then compare them to our approaches in Table 4.

Method	Precomputation	Windowing	Parallelisation	Uniformity
Shamir's trick	✓	×	×	✓
Bernstein	×	×	✓	✓
Bucket method	×	✓	✓	✓
Optimised d-mul	✓	×	✓	✓
Our first method	✓	×	×	✓
Our second method	×	×	✓	✓

Table 4: Comparison of the proprieties of our method to the existing methods.

– **The case of $d = 2$:**
In this case, if we do not need uniformity (as in algorithm 2), our method is

Method	Main operation	Precomputation	Storage
Naive	$(\alpha_1 + \alpha_2 - 1)\mathbf{A}$	–	2 points
Shamr's trick	$l\mathbf{D}+\frac{3}{4}l\mathbf{A}$	$1\mathbf{A}$	3 points
Bernstein	$l\mathbf{D}+2l\mathbf{A}$	–	2 points
Optimised 2-mul	$l\mathbf{D}+2l\mathbf{A}$	$2\mathbf{A}$	2 points
Our method	$l\mathbf{D}$+l$\mathbf{A}$	$1\mathbf{A}$	3 points

Table 5: The comparison of complexity for $d = 2$.

equivalent to Shamir's trick in terms of the complexity of the main operation, precomputation and storage, where storage refers to the number of points that need to be stored. However, this equivalence only holds for precomputation and storage if we introduce uniformity into our method (algorithm 4). In this case, Shamir's trick outperforms our method in terms of the efficiency of the main operation. Moreover, this work surpasses Bernstein's method in terms of the main operation. It conserves l point additions.

– **The case of small** $d > 2$**:**
We proceed to compare our first method, in Table 6, with existing methods when d is small. Table 6 showcases a notable achievement: our method surpasses optimised

Method	Main operation	Precomputation	Storage
Naive	$(\sum_{i=1}^{d} \alpha_i - 1)\mathbf{A}$	–	d points
Interleaving	$l\mathbf{D}+\frac{ld}{w+1}\mathbf{A}$	$d\mathbf{D}+(2^{w-2} - 1)\mathbf{A}$	$d2^{w-2}$ points
Optimised d-mul	$l\mathbf{D}+ld\mathbf{A}$	$d\mathbf{A}$	d points
Our method	$l\mathbf{D}+l\mathbf{A}$	$(2^d - d - 1)\mathbf{A}$	$2^d - 1$ points

Table 6: The comparison of complexity for small $d > 2$.

d-mul in terms of the complexity of the main operation. Furthermore, it outperforms the Interleaving method when d exceeds $w+1$. However, our method requires more precomputation and storage compared to optimised d-mul. Additionally, the Interleaving method necessitates the calculation of wNAFs (Non Adjacent Forms) of the scalars α_i at each step and does not guarantee the uniformity of operations. When assessing the overall complexity, which includes both the main operation and the precomputation, while keeping d and w constant, we offer the following comparisons:

- Our method overestimates the efficiency of optimised d-mul if $l \geq \frac{2^d-2d-1}{d-1}$,
- Our method also outperforms the Interleaving method when $l \geq \frac{(w+1)(2^d-2^{w-2}-d)}{d}$.

To simplify the general comparison (Table 6) and to present it only as a function of the parameter l, which is considerably larger than d and w, we offer the updated comparison in Table 7 with $d = 6$ and $w = 4$.

Method	Main operation	Precomputation	Storage
Naive	$(\sum_{i=1}^{6} \alpha_i - 1)\mathbf{A}$	–	6 points
Interleaving	$l\mathbf{D}+\frac{6}{5}l\mathbf{A}$	$6\mathbf{D}+18\mathbf{A}$	24 points
Optimised d-mul	$l\mathbf{D}+6l\mathbf{A}$	$6\mathbf{A}$	6 points
Our method	$l\mathbf{D}+l\mathbf{A}$	$57\mathbf{A}$	63 points

Table 7: The comparison of complexity, for $d = 6$ and $w = 4$.

Our method's main operation complexity surpasses that of optimised d-mul. This latter offers advantages like uniformity and parallelisation. Thus, our primary focus is to compare our method with d-mul.

Table 8 shows that our method maintains a constant main operation complexity with respect to d, while optimised d-mul's complexity varies. Additionally, for each d, optimised d-mul consumes $d - 1$ more additions than our method.

d	Optimised d-mul	**Our method**
3	$l\mathbf{D}+3l\mathbf{D}$	$l\mathbf{D}+l\mathbf{A}$
4	$l\mathbf{D}+4l\mathbf{A}$	$l\mathbf{D}+l\mathbf{A}$
5	$l\mathbf{D}+5l\mathbf{A}$	$l\mathbf{D}+l\mathbf{A}$
6	$l\mathbf{D}+6l\mathbf{A}$	$l\mathbf{D}+l\mathbf{A}$
7	$l\mathbf{D}+7l\mathbf{A}$	$l\mathbf{D}+l\mathbf{A}$
8	$l\mathbf{D}+8l\mathbf{A}$	$l\mathbf{D}+l\mathbf{A}$

Table 8: The comparison of the main operation complexity.

Table 9 presents a detailed analysis considering the arithmetic field and different coordinate systems. We assume $\mathbf{S} = \frac{3}{4}\mathbf{M}$, where $\mathbf{M}$, $\mathbf{S}$ and $\mathbf{I}$ represent multiplication, squaring and inversion in $\mathbb{K}$. The data in Table 9 clearly show that our method outperforms optimised d-mul regarding main operation complexity for small $d \geq 2$ across different coordinate systems. In particular, it shows remarkable efficiency when using Jacobian coordinates. To confirm this comparison, we applied our method ('Mu-S-secp256k1' and 'Mu-S-Mont') and optimised d-mul ('d-mul-secp256k1' and 'd-mul-Mont') on both the $secp256k1$ curve [20] used in ECDSA [9] and the $Montgomery$ curve [21] used in SQIsign [22]. For a closer look at the running time comparison, we set d to 4 and vary the parameter l within the set $\{32, 64, 128, 192, 256, 320\}$, generating four approximate graphs in Figure 2 that plot the running time as a function of l. These graphs show that our method is significantly faster than the optimised d-mul overall scalar sizes l and whichever curve we use. Furthermore, for each curve, we observe an increasing divergence between the graph of our method and that of d-mul as l increases. This divergence arises because optimised d-mul consistently processes 4 scalars of l bits, while our method appears to process only one scalar of l bits due to precomputational advantages. Consequently, as the parameter l escalates, the processing time naturally increases. In addition, we evaluate the running time by varying the number of scalars d within the set $\{2, 3, 4, 5, 6, 7, 8\}$ while keeping the scalar size at $l = 256$. This evaluation yields four plots in Figure 3, which show that the running time of optimised d-mul grows linearly with d, while that of our method remains nearly constant as a function of d. This confirms that the complexity of the main operation in our method is independent of the number of scalars.

Coordinates	Optimised d-mul	**Our method**
Affine	$l(d+1)\mathbf{I} + l(\frac{7}{2} + \frac{11}{4}d)\mathbf{M}$	$l(2\mathbf{I} + \frac{25}{2}\mathbf{M})$
Projective	$l(\frac{43}{4} + \frac{27}{2}d)\mathbf{M}$	$\frac{97}{4}l\mathbf{M}$
Jacobian	$l(\frac{17}{2} + 15d)\mathbf{M}$	$\frac{47}{2}l\mathbf{M}$

Table 9: Comparison of main operation complexity for different coordinate systems.

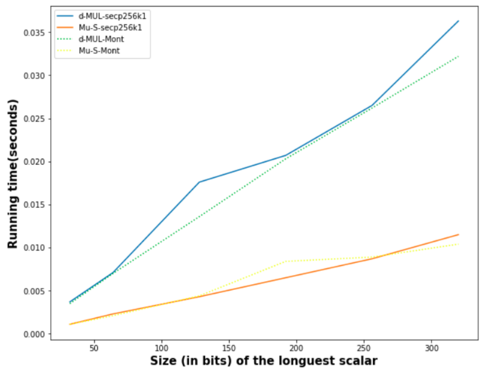

Figure 2: Running time of the main operation as a function of l for $d = 4$.

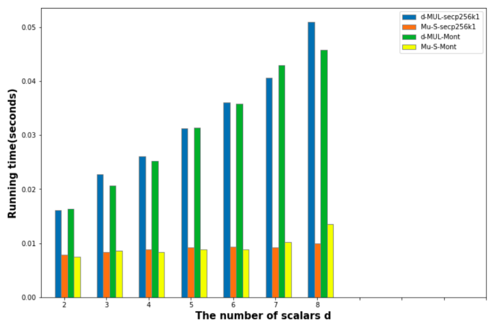

Figure 3: Comparison of main operation running time as a function of d for $l = 256$.

– **The case of a large d:**
In this case, we focus on comparing our method to some efficient methods for large d. We use Table 10 to compare the complexity of our second method to optimised d-mul and the bucket method in terms of main operation complexity, precomputation and storage. From Table 10 we can see that all three methods have the same memory requirements. However, optimised d-mul stands out by requiring d additions for precomputation, which is significant for large d. They all use the same number of doublings in their main operations but differ in the number of additions. Our method requires about $l\frac{d}{2}$ additions in the main operation, outperforming d-mul. However, for large d with $w = \log(d)$, the bucket method outperforms our method. The challenge with the bucket method is that it requires many buckets, each accumulating curve points. For example, with $d = 2^{22}$, it needs to initialize about $\frac{2^{22}}{22}l$ buckets, resulting in more memory access. In contrast, our method efficiently tracks and uses the points P_i in the main operation with simple, low cost operations, with $1 \leq i \leq d$.
We used the $BLS12-381$ curve to compare the running time of our method to optimised d-mul over different d values from the set $\{2^s; s \in [9, 19]\}$, keeping l fixed at 256. This comparison examines our method in three different scenarios:

Method	Main operation	Precomputation	Storage
Bucket method	$l\mathbf{D} + \lceil \frac{l}{w} \rceil (d + 2^{w+1})\mathbf{A}$	–	d
Optimised d-mul	$l\mathbf{D}+ld\mathbf{A}$	$d\mathbf{A}$	d
Our method	$l\mathbf{D}+l\frac{d}{2}\mathbf{A}$	–	d

Table 10: The comparison of complexity for the some window $w \approx log_2(d)$.

- In the first scenario, no permutations are applied. This is suitable when the security of the scalars α_i is not a priority, and the primary focus is on achieving high computational efficiency,
- The second scenario involves applying random permutations to steps selected by the set $Perm$ (which contains 50 steps in this comparison). It balances security and efficiency based on the cardinality of the set $Perm$,
- In the third scenario, random permutations are applied to all computation steps with $j \in [0, l-1]$. This approach provides robust scalar security with minimal loss of efficiency.

Figure 4 illustrates the comparison using a multiple bar graph. It shows the running time of both our method ('Mu-D1', 'Mu-D2' and 'Mu-D3' representing the three scenarios) and the optimised d-mul ('d-mul'). In Figure 4, the main operation of our method runs faster than the optimised d-mul, even with partial or full use of permutations. As d increases, the performance gap widens. This is because optimised d-mul involves costly computations for scalar bits, and these costs escalate with higher d values. Furthermore, when we use permutations, the time difference between our method and optimised d-mul is always more significant than without

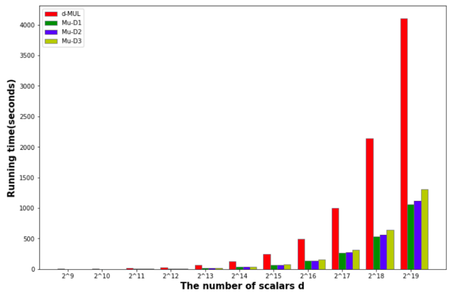

Figure 4: Running time of the main operation as a function of a large d for $l = 256$.

permutations. This is due to the permutation complexity, which is $O(d)$ for each permutation on the set $\{1, \cdots, d\}$.

6 Application

We illustrate the application of our d-mul in BLS multi signatures aggregation in this section. Permitting d signers to produce a brief signature σ on the same message m is known as multi signatures. Public key aggregation is supported by certain multi signatures methods. This means that a brief aggregated public key is needed in place of an explicit list including all d public keys. This characteristic is crucial to the scalability of blockchain systems because it permits the compression of public keys and signatures, which reduces bandwidth [30]. Since BLS signature [31] permits public key aggregation, it has been progressively deployed on the blockchain, which is of special relevance to us.

6.1 BLS signature

We begin by reviewing the conventional BLS signature scheme [31], adapting the notations for additive group operations. Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be groups of prime order q, G_1 and G_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, $H_1 : \{0,1\}^* \to \mathbb{G}_1$ be a hash function and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable pairing. A signer's private key is s_k, randomly chosen from $\mathbb{Z}_q$. The public key of this signer is $P_k = [s_k]G_2 \in \mathbb{G}_2$. The BLS signature works in the following way:

- **BLS.Setup (λ):** Given a security parameter λ, this function outputs the settings:

$$S = (q, \mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T, e, G_1, G_2),$$

- **BLS.KeyGen (S):** Given the settings S, this function outputs the private and public keys $(s_k, P_k = [s_k]G_2)$,
- **BLS.Sign(S, s_k, m):** Given the settings S, the private key s_k, and the message m, this function outputs the signature $\Sigma = [s_k]H_1(m) \in \mathbb{G}_1$,

- **BLS.Verify** (S, P_k, Σ, m)**:** Given the settings S, the public key P_k, the signature Σ, and the massage m, this function outputs 1 if $e(\Sigma, G_2) = e(H_1(m), P_k)$ and 0 otherwise.

6.2 BLS multi signatures aggregation

The BLS signature scheme can aggregate multi signatures, as mentioned in [30, 32, 33] and the references therein. Considering this, we introduce the multi signatures aggregation approach that was suggested in [32]. As mentioned in reference [30], this technique is denoted by **MS**. $H_2 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ is a new hash function that is introduced with the same parameters as before. Similar to [30], we provide **MS** as follows:

- **MS.Setup** (λ)**:** Given a security parameter λ, the output is **BLS.Setup** (λ),
- **MS.KeyGen** (S)**:** Given the settings S, the output is **BLS.KeyGen**(S),
- **MS.Sign** (S, PKs, s_{k_i}, m)**:** Given the settings S, the set of public keys of d signers, the i^{th} signer's private key s_{k_i}, and the message m, this function computes the signature $\Sigma_i = [s_{k_i}]H_1(m)$. Σ_i is sent to a designed combiner that can be one of the signers or an external entity. This combiner aggregates all the signatures by computing:
$$\Sigma = [\alpha_1]\Sigma_1 + [\alpha_2]\Sigma_2 + \cdots + [\alpha_d]\Sigma_d, \tag{1}$$
where d is the signers' number, $\alpha_i = H_2(P_{k_i}||PKs)$, and $PKs = \{P_{k_2}, P_{k_2}, \cdots, P_{k_d}\}$ is the set of signers public keys,
- **MS.KeyAggr**(S, PKs)**:** Given the settings S and the set of public keys, this function outputs the aggregated form of the public keys:
$$APK = [\alpha_1]P_{k_1} + [\alpha_2]P_{k_2} + \cdots + [\alpha_d]P_{k_d}, \tag{2}$$
- **MS.Verify**(S, APK, Σ, m)**:** Given the settings S, the aggregated form of public keys APK, the signature Σ, and the message m, this function outputs **BLS.Verify**(S, APK, Σ, m).

6.3 Our improvements

Multi signatures aggregation is generally useful in a wide range of fields, including finance, healthcare, cryptocurrency, and smart contracts. In certain situations, the number of signers d is small, whereas it is large in other situations. In several cases, particularly those based on blockchain technology, the number of signers d is dynamic. Thus, each time the designed combiner must compute $2d$ scalar multiplications in (1) and (2). Consequently, the cost of the calculation increases as d does. To ease the designed combiner's workload, the authors of [33] suggest having each signer calculate their $\alpha_i = H_2(P_{k_i}||PKs)$ and their signatures $\Sigma_i = [\alpha_i s_{k_i}]H_1(m)$, with $i \in [|1, d|]$. In many cases, signers lack resources, which could weaken their efforts. Since α_i is public and revealing it doesn't compromise security, we contribute to the designed combiner's work by enhancing computation in (1) and (2) mentioned above. In this way, we eliminate the need for signers to perform this calculation. To demonstrate our improvements, we proceed as follows:

- **Case of a small d:**
  - **The multi signatures aggregation (1) :**
    We use our second method (Algorithm 5) since the signatures $\Sigma_1, \Sigma_2, \cdots, \Sigma_d$ utilized in (1) change as a function of the message m. In this instance, the computational complexity is at most the same as when we employ optimized d-mul,
  - **The aggregation of the public keys (2):**
    Given that the points $P_{k_1}, P_{k_2}, \cdots, P_{k_d}$ are fixed and the scalars $\alpha_1, \alpha_2, \cdots, \alpha_d$ are variables, we use our first method (Algorithm 4). Compared to using optimized d-mul, this method's precomputation improves the computation by a factor of $100(1-\frac{1}{d})\%$ on the number of additions,
- **Case of a large d:**
  We are unable to utilize precomputation in this situation. Consequently, we compute (1) and (2) using our second method. Here we gain approximately 50% on the number of additions compared to using optimised d-mul.

Example 3.
In this example, we take the signers' number $d = 4$. Let E be the curve $BLS12-381$. Let $\mathbb{G}_1 = E(\mathbb{F}_p)[r]$, $\mathbb{G}_2 = E(\mathbb{F}_{p^{12}})[r]$ and $\mathbb{G}_T = \{\mu \in \mathbb{F}_{p^{12}}; \mu^r = 1\}$. For a message $m \in \mathbb{G}_1$, the signers produce the signatures $\Sigma_1 = [s_{k_1}]H_1(m)$, $\Sigma_2 = [s_{k_2}]H_1(m)$, $\Sigma_3 = [s_{k_3}]H_1(m)$ and $\Sigma_4 = [s_{k_4}]H_1(m)$. The designed combiner applies the equation (1) to compute the following aggregated signature:

$$\Sigma = [\alpha_1]\Sigma_1 + [\alpha_2]\Sigma_2 + [\alpha_3]\Sigma_3 + [\alpha_4]\Sigma_4.$$

Using our second method, this aggregation requires the following complexity:

$$254\boldsymbol{D} + 510\boldsymbol{A}.$$

To compute the aggregated form of the public keys, the designed combiner applies (2) to get:

$$APK = [\alpha_1]P_{k_1} + [\alpha_2]P_{k_2} + [\alpha_3]P_{k_3} + [\alpha_4]P_{k_4}.$$

As the points $P_{k_1}, P_{k_2}, P_{k_3}, P_{k_4}$ are fixed, it uses our first method to perform this computation. Thus, the needed complexity is $11\boldsymbol{A}$ as a precomputation and $254\boldsymbol{D} + 254\boldsymbol{A}$ for the main operation. The multi signatures verification requires two pairings over the curve $BLS12-381$. The full complexity of the multi signatures scheme is shown in Table 11.

Steps	Using our methods		Without our methods
Σ	$254\mathbf{D} + 510\mathbf{A}$		$254\mathbf{D} + 1020\mathbf{A}$
APK	Main operation	Precomputaion	$254\mathbf{D} + 1020\mathbf{A}$
	$254\mathbf{D} + 254\mathbf{A}$	$11\mathbf{A}$	

Table 11: The comparison of the multi signatures complexity with and without our methods.

7 Conclusion

In this paper, we have proposed a novel approach for computing multidimensional scalar multiplication on elliptic curves. Our work has been presented in two distinct methods to treat two different cases:
The first method is designed for the case where the number of scalars d is small. This method uses precomputation to efficiently perform the main operation, outperforming existing methods in this context. Our method is practical for schemes with a small number of scalars, such as ECDSA with $d = 2$, and speeds up the classic scalar multiplication. it is also a strong candidate for improving the efficiency of isogeny based postquantum cryptosystems such as SQISign. We confirmed the effectiveness of this method by evaluating the running time by applying it to the $secp256k1$ [20] and $Montgomery$ [21] curves used in ECDSA and SQIsign respectively. Note that this method is preferable when the scalars are prefixed.
The second method is tailored for the case where the number of scalars d is large. In such cases, there is no precomputation, and all calculations are performed on the fly. This method is similar to the double and add method. It simultaneously processes the bits of the scalars arranged in the same column from left to right at each step. Its main operation is more efficient than that of the optimised d-mul method and requires fewer memory accesses compared to the bucket method. Our method is highly practical for proving and verification algorithms such as the Succinct Non interactive Argument of Knowledge (zkSNARK) protocol that uses a large number of scalars. We tested its efficiency by evaluating its running time on the $BLS12 - 381$ curve [23]. Note that this method incorporates security measures against power attacks.

References

[1] Bernstein, D.J., Doumen, J., Lange, T., Oosterwijk, J.-J.: Faster batch forgery identification, 454–473 (2012). Springer

[2] Hutchinson, A., Karabina, K.: Constructing multidimensional differential addition chains and their applications. Journal of Cryptographic Engineering **9**(1), 1–19 (2017)

[3] Hisil, H., Hutchinson, A., Karabina, K.: d-mul: optimizing and implementing a multidimensional scalar multiplication algorithm over elliptic curves, 198–217 (2018). Springer

[4] Hutchinson, A., Karabina, K.: A new encoding algorithm for a multidimensional version of the montgomery ladder, 403–422 (2020). Springer

[5] Josefsson, S., Liusvaara, I.: Edwards-curve digital signature algorithm (EdDSA)(No. rfc8032). Technical report (2017. Online https://www.rfc-editor.org/rfc/rfc8032.html)

[6] Haakegaard, R., Lang, J.: The elliptic curve diffie-hellman (ecdh). Online at https://koclab.cs.ucsb.edu/teaching/ecc/ (2015)

[7] Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems, 292–302 (1999). Springer

[8] Hankerson, D., Menezes, A.: Elliptic curve cryptography. In: Encyclopedia of Cryptography, Security and Privacy, pp. 1–2. Springer, ??? (2021)

[9] Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security **1**, 36–63 (2001)

[10] Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies, 19–34 (2011). Springer

[11] Bernstein, D.J.: Differential addition chains (2006. http://cr.yp.to/ecdh/diffchain-20060219.pdf)

[12] Brown, D..: Multi-dimensional montgomery ladders for elliptic curves. (2006)

[13] Bos, J.W., Costello, C., Hisil, H., Lauter, K.: High-performance scalar multiplication using 8-dimensional glv/gls decomposition, 331–348 (2013). Springer

[14] Azarderakhsh, R., Karabina, K.: Efficient algorithms and architectures for double point multiplication on elliptic curves, 25–30 (2016)

[15] Rao, S., Rao, S.: Three dimensional montgomery ladder, differential point tripling on montgomery curves and point quintupling on weierstrass' and edwards curves, 84–106 (2016). Springer

[16] Nitulescu, A.: zk-SNARKs: a gentle introduction. Technical report (2020)

[17] Xavier, C.F.: Pipemsm: Hardware acceleration for multi-scalar multiplication. Cryptology ePrint Archive (2022)

[18] El Housni, Y., Botrel, G.: Edmsm: Multi-scalar-multiplication for snarks and faster montgomery multiplication. Cryptology ePrint Archive (2022)

[19] Lu, T., Wei, C., Yu, R., Chen, C., Fang, W., Wang, L., Wang, Z., Chen, W.: Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. Cryptology ePrint Archive (2022)

[20] Bi, W., Jia, X., Zheng, M.: A secure multiple elliptic curves digital signature algorithm for blockchain. arXiv (2018)

[21] Costello, C., Smith, B.: Montgomery curves and their arithmetic: The case of large characteristic fields. Journal of Cryptographic Engineering **8**(3), 227–240 (2018)

[22] De Feo, L., Kohel, D., Leroux, A., Petit, C., Wesolowski, B.: Sqisign: compact post-quantum signatures from quaternions and isogenies, 64–93 (2020). Springer

[23] Wahby, R.S., Boneh, D.: Fast and simple constant-time hashing to the bls12-381 elliptic curve. Cryptology ePrint Archive (2019)

[24] Straus, E.G.: Addition chains of vectors (problem 5125). American Mathematical Monthly **70**(806-808), 16 (1964)

[25] ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985)

[26] Sakai, Y., Sakurai, K.: Algorithms for efficient simultaneous elliptic scalar multiplication with reduced joint hamming weight representation of scalars, 484–500 (2002). Springer

[27] Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Mathematics of computation **48**(177), 243–264 (1987)

[28] King, B.: wnaf*, an efficient left-to-right signed digit recoding algorithm, 429–445 (2008). Springer

[29] Hazra, T.K., Ghosh, R., Kumar, S., Dutta, S., Chakraborty, A.K.: File encryption using fisher-yates shuffle, 1–7 (2015). IEEE

[30] Baldimtsi, F., Chalkias, K.K., Garillot, F., Lindstrom, J., Riva, B., Roy, A., Sonnino, A., Waiwitlikhit, P., Wang, J.: Subset-optimized bls multi-signature with key aggregation. Cryptology ePrint Archive (2023)

[31] Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 514–532 (2001). Springer

[32] Boneh, D., Drijvers, M., Neven, G.: Bls multi-signatures with public-key aggregation. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html (2018)

[33] Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 435–464 (2018). Springer