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

Multiscalar multiplication on elliptic curves is used in wide cryptographic domains such as digital signatures, zero knowledge proofs, and verifiable computation, where it enables efficient and secure cryptographic protocols. In this work, we present two methods for optimizing multi scalar multiplication over elliptic curves. The first method is used when $\mathbf{3^d - 1}$ points of the target curve can be stored. The second method is suitable for large values of $\boldsymbol{d}$, where storing $\mathbf{3^d - 1}$ points becomes infeasible. Our first method achieves up to **24%** better performance on elliptic curves with efficient tripling, speeding up ECDSA signing by **2.88**× and verification by **1.67**×. The second method achieves an efficiency improvement of **11.34%** compared to the similar existing approach. These improvements are supported by operation counts and practical benchmarks.

Keywords: Elliptic curves, multidimensional scalar multiplication (MSM), scalar multiplication, complexity

1 Introduction

Multidimensional scalar multiplication (MSM) computes a sum of scalar multiplications on elliptic curve points:

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

where $d \geq 2$ is an integer, $(\alpha_i)_{1\leq i\leq d}$ are integer scalars, and each $(P_i)_{1\leq i\leq d}$ are points on an elliptic curve E. MSM is a fundamental operation in elliptic curve cryptography. For example, signature verification in ECDSA [16] and the SIDH protocol [15] both use MSM with $d = 2$. Moreover, MSM can accelerate single scalar multiplications by decomposing them into multiple smaller ones [14]. Recently, MSM algorithms have attracted significant attention as they optimize the computation of many scalar multiplications simultaneously. These techniques are widely applied in digital signatures, zero-knowledge proofs, and pairing-based cryptography. Improvements such as windowing and grouping methods reduce computation time and resource usage, making cryptographic systems more efficient and scalable. Key MSM algorithms are detailed in the literature [2–4, 7, 8, 12–14, 22]. Some works focus on fixed values of d [2, 3, 7, 22], while others present general approaches valid for any d [4, 8, 12–14]. The Bos-Coster method is a recursive technique for multi-scalar multiplication over elliptic curves, minimizing group operations by iteratively reducing the largest scalar coefficients. The d-mul methods [12–14] rely on matrix transformations, while the bucket method [4] uses windowing on the binary representation of scalars for efficient precomputation. According

to Bernstein et al. [4], the bucket algorithm demonstrates superior performance among all methods when dealing with large MSM sizes.
This paper builds on these foundations to further optimize MSM calculations in order to improve cryptographic performance.

Our contribution

In this paper, we present two methods for computing MSM:
The first method applies when precomputation and storage are feasible. It consists of adapting the Strauss–Shamir method to the ternary representation, aiming to:

- Reduce the main operation length, thereby decreasing the number of elliptic curve additions.
- Exploit the efficiency of tripling formulas on certain elliptic curves.
- Increase the capacity for precomputation and storage compared to the binary case.

The second method is designed for scenarios where precomputation and storage are infeasible. It improves upon the bucket method, which is well known for handling MSM with large parameters, by introducing a variant based on the signed ternary representation. This enhanced version provides:

- Minimizing the number of required windows, which reduces the number of group operations.
- Reducing the number of buckets, thereby decreasing both the number of operations and the memory requirements.

Notations

Let $a, b, x \in \mathbb{Z}$. 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.
- $p \geq 5$ is a prime number.
- E is an elliptic curve defined over $\mathbb{F}_{p^k}$, with $k \in \mathbb{N}^*$.
- P_∞ denotes the point at infinity in the elliptic curve group E.
- $\mathbf{A}$ is an addition on E.
- $\mathbf{D}$ is a doubling on E.
- $\mathbf{T}$ is a tripling on E.
- $\mathbf{M_k}$ is the multiplication over $\mathbb{F}_{p^k}$.
- $\mathbf{S_k}$ is the squaring over $\mathbb{F}_{p^k}$.

The paper is organized as follows. Section 2 reviews existing methods for multi scalar multiplication (MSM). Section 3 introduces two new MSM techniques: the first uses precomputation for small parameters d, inspired by Shamir's trick when $d = 2$; the second is a signed ternary version of the bucket method without precomputation. Section 4 compares the complexity and runtime of these methods with existing approaches. Section 6 demonstrates the first method in ECDSA [16] and the second in the BLS signature on $BLS12$-381, improving public key and signature aggregation. These methods are especially effective on curves supporting efficient tripling formulas. The paper concludes with a summary of the main results and contributions.
In the remainder of this paper, we set $\ell = \max_{1 \leq i \leq d}\big(\log_2(\alpha_i)\big)$.

2 State of the Art

In this section, we offer an overview of the existing approaches to MSM algorithms with a focus on complexities.

2.1 Case of d=2

2.1.1 Simultaneous Scalar Multiplication (Shamir's trick)

Shamir's trick allows the computation of $[\alpha_1]P_1 + [\alpha_2]P_2$. This method is given in detail in [23]. The binary representations of α_1 and α_2 are given by:

$$(\alpha_1)_2 = (b_{l-1}^{(1)} b_{l-2}^{(1)} \cdots b_1^{(1)} b_0^{(1)})_2,$$

$$(\alpha_2)_2 = (b_{l-1}^{(2)} b_{l-2}^{(2)} \cdots b_1^{(2)} b_0^{(2)})_2.$$

We define the $2 \times l$ matrix C by:

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

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

$$\begin{aligned} [\alpha_1]P_1 + [\alpha_2]P_2 &= (P_1P_2)C \begin{pmatrix} 2^{l-1} \\ \vdots \\ 2^1 \\ 2^0 \end{pmatrix} \\ &= [2^{l-1}]([b_{l-1}^{(1)}]P_1 + [b_{l-1}^{(2)}]P_2) + \cdots + [2^1]([b_1^{(1)}]P_1 \\ &\quad + [b_1^{(2)}]P_2) + [2^0]([b_0^{(1)}]P_1 + [b_0^{(2)}]P_2). \end{aligned}$$

This computation can be carried out using Algorithm 1 from [23]. The complexity of the main operation is approximately:

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

The precomputation needs only one addition is performed. To further optimize the Shamir calculation, the non-adjacent form (NAF [10]) of α_1 and α_2 can be used in place of their binary representations.

2.1.2 Bernstein's method (DJB)

The concept of the differential addition chain [3] that is an addition chain in which P_1 and P_2 are represented by their difference $P_1 - P_2$. Montgomery observed in [20] 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 [3] 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 [3]. The complexity of the main operation in DJB is given by:

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

Table 1 compares the Bernstein and Shamir methods.

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

Table 1: Comparison of the Bernstein and Shamir methods.

2.2 Case of d > 2

To our knowledge, three main techniques are used in the literature for computing multi scalar multiplication (MSM): state matrices, the bucket method, and interleaving. In this context, we introduce three existing methods: the Straus-Shamir method, d-mul, and bucket method (Pippenger Algorithm).

2.2.1 Straus-Shamir method

Let w be a small integer denoting the window width in the binary representation. According to [19], the Straus-Shamir method computes the multi scalar multiplication $[\alpha_1]P_1 + \cdots + [\alpha_d]P_d$ using the following steps:

1. **Precomputation**: For a small integer w, it precomputes all combinations of points $\{c_1P_1 + \cdots + c_dP_d\}$, where $0 \leq c_i < 2^w$ for $i = 1, \ldots, d$, resulting in 2^{dw} points stored in a lookup table.

2. **Scalar Segmentation**: Represent each scalar α_i in its binary form by partitioning it into windows of size w. Accordingly, we write $\alpha_i = \sum_{j=0}^{h-1} \alpha_{i,j} 2^{jw}$, where $h = \lceil l/w \rceil$ and $0 \leq \alpha_{i,j} < 2^w$.
3. **Initialize Result**: Start with $R = P_\infty$.
4. **Iterative Processing**: For each window position j from $h-1$ to 0:
   - Update the accumulated sum by performing w point doublings: $R \leftarrow 2^w R$.
   - Fetch the precomputed point $\alpha_{1,j} P_1 + \cdots + \alpha_{d,j} P_d$ from the table and accumulate it into R.
5. **Output**: After completing the processing of all windows, the result satisfies $R = [\alpha_1]P_1 + \cdots + [\alpha_d]P_d$.

This approach improves efficiency in multi scalar multiplication by minimizing the number of point operations through shared doublings and the use of precomputed points. The Straus-Shamir method is outlined in Algorithm 1.

Algorithm 1 Straus-Shamir method for multi scalar Multiplication

Require: Scalars $\alpha_1, \alpha_2, \ldots, \alpha_d$, points $P_1, P_2, \ldots, P_d$ on an elliptic curve E, window size w
Ensure: $R = [\alpha_1]P_1 + \cdots + [\alpha_d]P_d$
1: **Precomputation**: Compute $T = \{c_1 P_1 + \cdots + c_d P_d; 0 \leq c_i < 2^w, i = 1, \ldots, d\}$.
2: Compute $h = \lceil l/w \rceil$.
3: Express each α_i as $\alpha_i = \sum_{j=0}^{h-1} \alpha_{i,j} 2^{jw}$, where $0 \leq \alpha_{i,j} < 2^w$.
4: $R \leftarrow P_\infty$.
5: **for** $j = h-1$ down to 0 **do**
6: $\quad R \leftarrow [2^w]R$
7: $\quad R \leftarrow R + T[\alpha_{1,j}, \ldots, \alpha_{d,j}]$
8: **return** Q

The complexity associated with the current approach is as follows:

- **Precomputation:** $d \cdot 2^w[w\mathbf{D} + (\frac{w}{2} + d - 1)\mathbf{A}]$ (the worst case).
- **Main operation:** $l\mathbf{D} + \left\lceil \frac{l}{w} \right\rceil \mathbf{A}$.

2.2.2 *d*-mul methods

We summarize the three d-mul methods chronologically:
The original d-mul [13] transforms scalars into a state matrix A, builds a chain of matrices $(A^{(i)})_{i=1}^{l}$, then computes linear combinations yielding:

$$[\alpha_1]P_1 + \cdots + [\alpha_d]P_d.$$

Randomised d-mul [12] uses random binary strings and permutations to construct matrices from magnitude 1, performing point computations at each step. Optimised d-mul [14] extends this to prefixed scalars via bit permutations, XOR operations, and negative scalar handling (replacing $\alpha_i < 0$ with $-\alpha_i$ and P_i with $-P_i$). **All methods share the same complexity:** $\ell\mathbf{D} + (l+1)d\mathbf{A}$, where ℓ is bit length, $\mathbf{D}$ is doubling, and $\mathbf{A}$ is addition. The authors recommend randomised d-mul for random scalars and optimised d-mul for fixed scalars, abandoning the original version.

2.2.3 Bucket method (Pippenger Algorithm)

This method was proposed in [4], further studied in [9, 18, 24], and is described in detail as Algorithm 1 in [21]. It is based on the Pippenger algorithm, which was originally designed to efficiently compute the multi exponent multiplication [24]. It is hardware optimised in [24] for MSM speed-up. It is parallelized in [18] to provide an efficient GPU implementation of zkSNARK. It is used in [9] with a new coordinate system for twisted Edwards curves to speed up the d-mul for a large d. The binary bucket method proceeds as follows:

- Set $w_b = \log_2(d) - c$, with $c \in [|2, 5|]$. c is chosen such that w_b is optimal.
- Initialize the accumulator R to the identity element P_∞.
- Determine the maximum bit length ℓ among the scalars α_i.
- Compute the number of windows: $N_b = \left\lceil \frac{\ell}{w_b} \right\rceil$.
- Let the number of buckets be $n_b = 2^{w_b}$.
- For each window j from $N_b - 1$ down to 0:
  - Scale the accumulator R by 2^{w_b} using w_b successive doublings: $R \leftarrow [2^{w_b}]R$.
  - Initialize all buckets B_k for $k = 0, \ldots, 2^{w_b} - 1$ to P_∞.

– For each scalar α_i, extract the j-th window bits $\alpha_{i,j}$ of size w_b.
– If $\alpha_{i,j} \neq 0$, add the corresponding point P_i to bucket $B_{\alpha_{i,j}}$: $B_{\alpha_{i,j}} \leftarrow B_{\alpha_{i,j}} + P_i$.
– Initialize an accumulator acc $\leftarrow P_\infty$.
– Aggregate buckets in descending order:

$$\text{For } b = 2^{w_b} - 1 \text{ down to } 1: \quad \text{acc} \leftarrow \text{acc} + B_b, \quad R \leftarrow R + \text{acc}.$$

- Return the final result R.

This method is given in Algorithm 1 in [24]. Its main operation complexity is given by:

$$C_b = (\lceil \frac{l}{w_b} \rceil) w_b \mathbf{D} + \lceil \frac{l}{w_b} \rceil (d + 2^{w_b+1}) \mathbf{A}.$$

For large d and fixed window width w_b, the bucket method yields

$$\mathcal{O}(d + 2^{w_b}).$$

The temporal memory of this method is dominated by 2^{w_b} curve points.

3 Our methods

In this section, we present two efficient MSM methods based on storage constraints. Given storage for at most $3^M - 1$ points, we classify d as small ($d \leq M$) or large ($d > M$). For small d, we adapt the non-windowed Straus–Shamir method to ternary representation, called **Ternary Straus–Shamir (TSS)**, to reduce storage, shorten the main loop, and leverage efficient tripling formulas. For large d, we improve the bucket method using signed ternary representation, termed the **Signed Ternary Bucket Method**.

3.1 Ternary Straus–Shamir (TSS)

As the method relies on precomputation, we assume that the points $(P_i)_{1 \leq i \leq d}$ are fixed. While several MSM methods exist, the Straus–Shamir technique has not been applied in ternary form.This section introduces a ternary-based variant of the method. We will see that it reduces additions during the main operation and improves efficiency on curves with fast tripling formulas, such as Hessian, Huff, Jacobi-intersection, and Edwards curves [17].

3.1.1 Description

We assume d is small, as storing precomputed points is required. We set $n = \max_{1 \leq i \leq d}(\lceil \log_3(\alpha_i) \rceil)$ and write the scalars $\alpha_1, \alpha_2, \ldots, \alpha_d$ in ternary as follows:

$$\alpha_i = (t_{n-1}^{(i)} t_{n-2}^{(i)} \cdots t_1^{(i)} t_0^{(i)})_3,$$

where $i \in [|1, d|]$, and $t_j^{(i)} \in \{0, 1, 2\}$, for $j \in [|0, n-1|]$. We present the scalars $(\alpha_i)_{1 \leq i \leq d}$ in the matrix C as follows:

$$C = \begin{pmatrix} t_{n-1}^{(1)} & \cdots & t_1^{(1)} & t_0^{(1)} \\ t_{n-1}^{(2)} & \cdots & t_1^{(2)} & t_0^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ t_{n-1}^{(d)} & \cdots & t_1^{(d)} & t_0^{(d)} \end{pmatrix}$$

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

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

We place the powers $(3^j)_{0 \leq j \leq n-1}$ in the matrix column S as follows:

$$S = \begin{pmatrix} 3^{n-1} \\ \vdots \\ 3^1 \\ 3^0 \end{pmatrix}$$

Thus, we get:

$$[\alpha_1]P_1 + [\alpha_2]P_2 + \cdots + [\alpha_d]P_d = PCS = P \begin{pmatrix} t_{n-1}^{(1)} & \cdots & t_1^{(1)} & t_0^{(1)} \\ t_{n-1}^{(2)} & \cdots & t_1^{(2)} & t_0^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ t_{n-1}^{(d)} & \cdots & t_1^{(d)} & t_0^{(d)} \end{pmatrix} \begin{pmatrix} 3^{n-1} \\ \vdots \\ 3^1 \\ 3^0 \end{pmatrix}$$

$$\begin{aligned} &= [3^{n-1}]([t_{n-1}^{(d)}]P_d + \cdots + [t_{n-1}^{(2)}]P_2 + [t_{n-1}^{(1)}]P_1) \\ &+ [3^{n-2}]([t_{n-2}^{(d)}]P_d + \cdots + [t_{n-2}^{(2)}]P_2 + [t_{n-2}^{(1)}]P_1) \\ &\vdots \\ &+ [3^1]([t_1^{(d)}]P_d + \cdots + [t_1^{(2)}]P_2 + [t_1^{(1)}]P_1) \\ &+ [3^0]([t_0^{(d)}]P_d + \cdots + [t_0^{(2)}]P_2 + [t_0^{(1)}]P_1) \end{aligned}$$

From this expression, we can precompute points $\sum_{i=1}^{d}[t_j^{(i)}]P_i$ for $j \in [0, n-1]$. We arrange these points in a table T of size n, defined by:

$$T[x_j] = \sum_{i=1}^{d}[t_j^{(i)}]P_i,$$

where

$$x_j = \sum_{i=1}^{d} t_j^{(i)} \cdot 3^{i-1}.$$

Then, we can simplify the computation of the MSM as follows:

$$[\alpha_1]P_1 + [\alpha_2]P_2 + \cdots + [\alpha_d]P_d = [3^{n-1}]T[x_{n-1}] + [3^{n-2}]T[x_{n-2}] + \cdots + [3^1]T[x_1] + [3^0]T[x_0].$$

This computation can be carried out using Algorithm 2.

Algorithm 2 Ternary Straus-Shamir Algorithm (TSS)

Require: Scalars $\alpha_1 = (t_{n-1}^{(1)} \dots t_0^{(1)})_3, \dots, \alpha_d = (t_{n-1}^{(d)} \dots t_0^{(d)})_3$ with $t_j^{(i)} \in \{0,1,2\}$, points $P_1, P_2, \dots, P_d \in E$ (tripling-friendly curve)
Ensure: $R = [\alpha_1]P_1 + \cdots + [\alpha_d]P_d$
1: **Precompute** $V = \{T[0], T[1], \dots, T[3^d - 1]\}$, ▷ T is defined above
2: Initialize array $x[j]$ for $j = 0$ to $n-1$
3: **for** $j = n-1$ **down to** 0 **do**
4: $\quad x[j] \leftarrow \sum_{i=1}^{d} t_j^{(i)} \cdot 3^{i-1}$
5: $R \leftarrow P_\infty$
6: **for** $j = n-1$ **down to** 0 **do**
7: $\quad R \leftarrow [3]R$
8: $\quad R \leftarrow R + V[x[j]]$
9: **return** R

3.1.2 Complexity

This method costs a storage of $3^d - 1$ points and requires $[(2^d - 1)\mathbf{D} + (3^d - 2^d - d)\mathbf{A}$ additions as precomputation. The main operation in Algorithm 2 contains $n = \frac{\log(2)}{\log(3)}\ell$ steps. During each step $j \in [0, n-1]$, we perform one tripling and one addition . Consequently, the total complexity of the main operation in this method is

$$(\frac{\log(2)}{\log(3)}\ell)\mathbf{T} + (\frac{\log(2)}{\log(3)}\ell)\mathbf{A}.$$

The binary variant of this algorithm corresponds to the Straus-Shamir method (Algorithm 1) with a window size of $w = 1$. For this variant, the storage costs $2^d - 1$ points, the precomputation entails $(2^d - d - 1)\mathbf{A}$, and the complexity of the main operation is $l\mathbf{D} + l\mathbf{A}$. In Table 2, we present a comparative summary of the

Steps	Ternary variant	Binary variant
Main operation	$(\frac{\log(2)}{\log(3)}\ell)\mathbf{T} + (\frac{\log(2)}{\log(3)}\ell)\mathbf{A}$	$l\mathbf{D} + l\mathbf{A}$
Precomputation	$[(2^d-1)\mathbf{D} + (3^d - 2^d - d)\mathbf{A}$	$(2^d - d - 1)\mathbf{A}$
Storage	$(2^d - 1)$ points	$(3^d - 1)$ points

Table 2: Complexity of Ternary vs. Binary Straus–Shamir Variants

complexity associated with the ternary and binary versions of the Straus–Shamir method. To highlight the efficiency of the ternary version over the binary one in main operation complexity, we present an example using Huff curves, which allow fast tripling.

Example 1.

Let $d = 3$, $\alpha_1 = 58 = (2011)_3$, $\alpha_2 = 47 = (1202)_3, \alpha_3 = 34 = (1021)_3$, *and* $P_1, P_2, P_3 \in E$, *where* E *is a Huff Curve that is studied in [17] and supports a fast tripling. Let* $n = \log_3(58) = 4$. *As explained at the beginning of this method, table* T *contains all the possibilities of the points* $\sum_{i=1}^{d} [t_j^{(i)}]P_i$, *with* $0 \leq j \leq 3$. *In this example,* T *is given by:*

$$\begin{aligned} T = \{\, & P_\infty,\ P_1,\ [2]P_1,\ P_2,\ P_1+P_2,\ [2]P_1+P_2,\ [2]P_2,\ P_1+[2]P_2,\ [2]P_1+[2]P_2,\ P_3,\ P_1+P_3,\ [2]P_1+P_3, \\ & P_2+P_3,\ P_1+P_2+P_3,\ [2]P_1+P_2+P_3,\ [2]P_2+P_3,\ P_1+[2]P_2+P_3,\ [2]P_1+[2]P_2+P_3, \\ & [2]P_3,\ P_1+[2]P_3,\ [2]P_1+[2]P_3,\ P_2+[2]P_3,\ P_1+P_2+[2]P_3,\ [2]P_1+P_2+[2]P_3, \\ & [2]P_2+[2]P_3,\ P_1+[2]P_2+[2]P_3,\ [2]P_1+[2]P_2+[2]P_3 \,\}. \end{aligned}$$

Thus,

$$[58]P_1+[47]P_2+[34]P_3 = \begin{pmatrix} P_1 P_2 P_3 \end{pmatrix} \begin{pmatrix} 2.3^3 + 0.3^2 + 1.3^1 + 1.3^0 \\ 1.3^3 + 2.3^2 + 0.3^1 + 2.3^0 \\ 1.3^3 + 0.3^2 + 2.3^1 + 1.3^0 \end{pmatrix} = [3^3]T[14]+[3^2]T[6]+[3^1]T[19]+[3^0]T[16].$$

The sum is obtained by applying Algorithm 2 to the scalars $\alpha_1 = 58$, $\alpha_2 = 47$, $\alpha_3 = 34$, *and the corresponding fixed points* P_1, P_2, P_3. *The complexity in this example is interpreted as follows:*

- *The main operation:* $4\mathbf{D} + 4\mathbf{A}$.
- *The precomputation:* $7\mathbf{D} + 16\mathbf{A}$.
- *The storage:* 26 *points.*

Based on Table 2 in [17], and assuming that $\mathbf{S} = 0.8\,\mathbf{M_1}$, *the main operation cost on Huff curves is:*

$$4\mathbf{T} + 4\mathbf{A} = 4(10\mathbf{M_1} + 5\mathbf{S}) + 4(10\mathbf{M_1}) = 96\mathbf{M_1}$$

using Algorithm 2, and:

$$6\mathbf{D} + 6\mathbf{A} = 6(6\mathbf{M_1} + 5\mathbf{S}) + 6(10\mathbf{M_1}) = 120\mathbf{M_1}$$

for the binary variant of the Straus-Shamir method. This example confirms the superiority of the ternary variant over the binary one when an efficient tripling is available.

3.1.3 Security analysis and countermeasures

The ternary Straus-Shamir algorithm (Algorithm 2) is susceptible to side-channel attacks due to non-uniform operations. In each iteration j of the main operation, it performs a tripling $R \leftarrow [3]R$ and an addition $R \leftarrow R + V[x[j]]$, where $x[j] = \sum_{i=1}^{d} t_j^{(i)} \cdot 3^{i-1}$ depends on ternary digits $t_j^{(i)} \in \{0, 1, 2\}$. With probability $\frac{1}{3^d}$, $x[j] = 0$, adding the point at infinity P_∞, while with probability $1 - \frac{1}{3^d}$, a non-trivial addition occurs. This variability enables simple timing attacks (STA) and simple power analysis (SPA) by exposing distinct execution patterns. To counter this, we propose dummy additions, such as a redundant operation $R \leftarrow (R + S) - S$ using a precomputed random point S and its negation $-S$ when $x[j] = 0$, ensuring uniform tripling and addition in every iteration. Constant time arithmetic and secure precomputation of S and $-S$ are critical to prevent cache timing or differential power analysis (DPA) attacks. Algorithm 3 implements this countermeasure.

Algorithm 3 Modified Ternary Straus-Shamir Algorithm

Require: Scalars $\alpha_1 = (t_{n-1}^{(1)} \dots t_0^{(1)})_3, \dots, \alpha_d = (t_{n-1}^{(d)} \dots t_0^{(d)})_3$ with $t_j^{(i)} \in \{0,1,2\}$, points $P_1, \dots, P_d \in E$ (tripling-friendly curve)
Ensure: $R = [\alpha_1]P_1 + \dots + [\alpha_d]P_d$
1: **Precompute** $V = \{T[0], T[1], \dots, T[3^d - 1]\}$, ▷ T is defined above
2: **Precompute** random point $S \in E$ and $-S$ ▷ For dummy additions
3: Initialize array $x[j]$ for $j = 0$ to $n-1$
4: **for** $j = n-1$ **down to** 0 **do**
5: $\quad x[j] \leftarrow \sum_{i=1}^{d} t_j^{(i)} \cdot 3^{i-1}$
6: $R \leftarrow P_\infty$
7: **for** $j = n-1$ **down to** 0 **do**
8: $\quad R \leftarrow [3]R$ ▷ Tripling always performed
9: $\quad$ **if** $x[j] = 0$ **then** ▷ Dummy addition case
10: $\quad\quad R \leftarrow (R + S) - S$ ▷ Redundant addition
11: $\quad$ **else**
12: $\quad\quad R \leftarrow R + V[x[j]]$ ▷ Real addition
13: **return** R

Employing Algorithm 3 instead of Algorithm 2 introduces minimal computational overhead, as the redundant addition $(R+S)-S$ for dummy operations when $x[j] = 0$ occurs with a low probability of $\frac{1}{3^d}$, while true additions dominate with probability $1 - \frac{1}{3^d}$.

3.2 Second method (Signed Ternary Bucket MSM)

In many cryptographic protocols such as zkSNARKs, multi scalar multiplication (MSM) involves a large number of scalars, often $d = 2^{22}$ or more, posing significant memory and performance challenges, especially on resource-limited devices. Bernstein *et al.* [4] established the Bucket method as the most efficient approach for large scale elliptic curve MSM. This motivated us to adapt the Bucket method using a windowed signed ternary representation to enable smaller window sizes and fewer buckets, thereby minimizing the number of group operations and reducing memory requirements.

3.2.1 Description

The signed ternary representation of a scalar α_i is defined as

$$\alpha_i = (t_{n-1}^{(i)} t_{n-2}^{(i)} \cdots t_1^{(i)} t_0^{(i)}),$$

where $n = max_{1 \le i \le d}(\lceil \log_3(\alpha_i) \rceil)$ and $t_j^{(i)} \in \{-1, 0, 1\}$ for all $j \in [0, n-1]$. Let w_t denote a window for the signed ternary representation (see tha section 4 for the optimal choice of w_t). Each scalar α_i is divided into $N_t = \left\lceil \frac{n}{w_t} \right\rceil$ blocks of length w_t. Thus, each scalar can be expressed in the base 3^{w_t} as

$$\alpha_i = (\alpha_{i,N_t-1} \alpha_{i,N_t-2} \cdots \alpha_{i,1} \alpha_{i,0})_{3^{w_t}}, \quad i = 1, 2, \dots, d,$$

where each $\alpha_{i,j} \in \left[-\frac{3^{w_t}-1}{2}, \frac{3^{w_t}-1}{2}\right]$ represents a ternary block of w_t signed trits for $j \in [0, N_t - 1]$.
The multi scalar multiplication (MSM) can be expressed as

$$\sum_{i=1}^{d} \alpha_i P_i = [3^{(N_t-1)w_t}] Q_{N_t-1} + [3^{(N_t-2)w_t}] Q_{N_t-2} + \cdots + [3^{w_t}] Q_1 + Q_0,$$

where the intermediate points $(Q_j)_{j \in [0 \cdots N_t - 1]}$ are computed as

$$\begin{pmatrix} Q_{N_t-1} \\ Q_{N_t-2} \\ \vdots \\ Q_0 \end{pmatrix} = \begin{pmatrix} \alpha_{1,N_t-1} & \cdots & \alpha_{d-1,N_t-1} & \alpha_{d,N_t-1} \\ \alpha_{1,N_t-2} & \cdots & \alpha_{d-1,N_t-2} & \alpha_{d,N_t-2} \\ \vdots & \ddots & \vdots & \vdots \\ \alpha_{1,1} & \cdots & \alpha_{d-1,1} & \alpha_{d,1} \\ \alpha_{1,0} & \cdots & \alpha_{d-1,0} & \alpha_{d,0} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_d \end{pmatrix}.$$

At each window $j \in [0 \cdots N_t - 1]$, the signed ternary bucket method performs the following steps:

- Execute w_t successive triplings using an optimized tripling formula, which is more efficient than iterative doublings and additions.
- Accumulate the points $(P_i)_{i\in[1\cdots d]}$ into buckets according to the digit $\alpha_{i,j}$. Since the digit range $\left[-\frac{3^{w_t}-1}{2}, \frac{3^{w_t}-1}{2}\right]$ is symmetric, only half of the possible buckets are needed.
- Combine the bucket sums to obtain the final result.

This method is described in Algorithm 4.

Algorithm 4 Signed Ternary Bucket Method

Require: Scalars $(\alpha_i)_{i=1}^d$, points $(P_i)_{i=1}^d$, window size w_t, precomputed powers 3^k.
Ensure: $R = \sum_{i=1}^d \alpha_i P_i$.

Convert each scalar α_i to its signed ternary representation.
Let $n = \max_i \text{len}(\alpha_i)$, $N_t = (n + w_t - 1)/w_t$, and $n_b = \frac{3^{w_t}-1}{2} + 1$.
Initialize $R \leftarrow P_\infty$ and all buckets $B[b] \leftarrow P_\infty$ for $b \in [0, n_b - 1]$.
for $j = N_t - 1$ down to 0 **do**
    Perform w_t successive triplings: $R \leftarrow [3^{w_t}]R$.
    Reset all buckets $B[b] \leftarrow P_\infty$.
    for $i = 1$ to d **do**
        Compute window value $\nu = \text{digit}(\alpha_{i,j})$.
        if $\nu > 0$ **then**
            $B[|\nu|] \leftarrow B[|\nu|] + P_i$.
        else if $\nu < 0$ **then**
            $B[|\nu|] \leftarrow B[|\nu|] - P_i$.
    Initialize $A \leftarrow P_\infty$.
    for $b = n_b - 1$ down to 1 **do**
        $A \leftarrow A + B[b]$.
        $R \leftarrow R + A$.
return R.

We present the following example to illustrate the operation of the method and compare its computational complexity with that of the bucket method using a binary window $w_b = 2$.

Example 2. *Let $\alpha_1 = 5324$, $\alpha_2 = 4970$, $\alpha_3 = 2344$, and $P_1, P_2, P_3 \in E$, where E is an elliptic curve group. We want to compute*

$$[\alpha_1]P_1 + [\alpha_2]P_2 + [\alpha_3]P_3.$$

The scalars are first written in their signed ternary representations as follows:

$$\begin{aligned}
\alpha_1 &= (1\,\bar{1}\,1\,1\,0\,\bar{1}\,1\,\bar{1}\,\bar{1})_3,\\
\alpha_2 &= (1\,\bar{1}\,1\,\bar{1}\,1\,1\,0\,1\,\bar{1})_3,\\
\alpha_3 &= (0\,1\,0\,1\,\bar{1}\,0\,\bar{1}\,1\,1)_3,
\end{aligned}$$

where $\bar{x} = -x$. We regroup the digits in windows of size $w_t = 3$ to write the scalars in the base 3^{w_t}:

$$\begin{aligned}
\alpha_1 &= (7\ 8\ 5)_{3^3},\\
\alpha_2 &= (7\ \bar{5}\ 2)_{3^3},\\
\alpha_3 &= (3\ 6\ \bar{5})_{3^3}.
\end{aligned}$$

The MSM takes the following structured form:

$$[\alpha_1]P_1 + [\alpha_2]P_2 + [\alpha_3]P_3 = \sum_{j=0}^{2} \left([\alpha_{1,j}]P_1 + [\alpha_{2,j}]P_2 + [\alpha_{3,j}]P_3\right)[3^{3j}],$$

or it is written as :

$$\begin{pmatrix} P_1 & P_2 & P_3 \end{pmatrix} \begin{pmatrix} 7 & 8 & 5 \\ 7 & \bar{5} & 2 \\ 3 & 6 & \bar{5} \end{pmatrix} \begin{pmatrix} 3^{3\times 2} \\ 3^{3\times 1} \\ 3^{3\times 0} \end{pmatrix}$$

which expands as:

$$[3^6]\Big([7](P_1+P_2)+[3]P_3\Big)+[3^3]\Big([8]P_1-[5]P_2+[6]P_3\Big)+[1]\Big([5](P_1-P_3)+[2]P_2)\Big).$$

This allows the application of the signed ternary bucket method (Algorithm 4).

3.2.2 Complexity Analysis

We analyze the complexity of Algorithm 4 in terms of elliptic curve group operations.

Window shifting

At each window, the algorithm performs w_t successive triplings to align the accumulator with the next ternary block. The total number of triplings is therefore

$$C_T = N_t \cdot w_t.$$

Bucket accumulation

At window $j \in [0, N_t - 1]$, each scalar α_i contributes a signed digit $\alpha_{i,j}$ from the range $\left[-\frac{3^{w_t}-1}{2}, \frac{3^{w_t}-1}{2}\right]$. A point P_i is added to or subtracted from a bucket only when $\alpha_{i,j} \neq 0$. For a uniform ternary distribution, the probability that a window digit is nonzero is $\theta = 1 - \frac{1}{3^{w_t}}$. Hence, the expected number of additions from bucket assignment is

$$C_A^{(\text{bucket})} \approx N_t \cdot \theta \cdot d = N_t \cdot d\left(1 - \frac{1}{3^{w_t}}\right).$$

Bucket summation

After accumulation, the algorithm combines bucket sums in descending order. This phase uses at most two additions per nonempty bucket: one for the running accumulator, one for adding to the result:

$$C_A^{(\text{sum})} \approx N_t \cdot 2 \cdot (n_b - 1) = N_t \cdot 2 \cdot \left(\frac{3^{w_t}-1}{2}\right) \approx N_t \cdot (3^{w_t} - 1).$$

Total group operations

The total cost in group operations is thus

$$C_t = \left(\left\lceil \frac{n}{w_t} \right\rceil \cdot w_t\right)\mathbf{T} + \left[\left\lceil \frac{n}{w_t} \right\rceil \cdot \left[d\left(1 - \frac{1}{3^{w_t}}\right) + (3^{w_t} - 1)\right]\right]\mathbf{A}.$$

The first term, in the number of addition, grows linearly with the number of scalars d, while the second term depends exponentially on w_t. Moreover, the method needs 3^{w_t} curve points as temporal memory. Choosing w_t optimally balances the number of triplings and additions and the memory requirements.

Asymptotic complexity

For large d and fixed window width w_t, the signed ternary bucket method yields

$$\mathcal{O}(d + 3^{w_t})$$

group operations. Because $\theta = 1 - 1/3^{w_t}$ approaches 1 quickly with increasing w_t, most windows provide nonzero contributions, making the method efficient for moderate w_t values.

3.2.3 Parallelization

The signed ternary bucket method (Algorithm 4) supports efficient parallelization through several distinct stages.

Scalar Conversion

If not precomputed, each scalar's conversion to signed ternary form can be independently executed in parallel across multiple threads or cores, as this step is computationally light.

Window-wise Parallelism

Scalars are divided into windows of size w_t. Each window's bucket processing and summation are independent, enabling simultaneous processing of multiple windows. This can be implemented either by pipelining stages on hardware supporting concurrent computations or by distributing windows across threads.

Bucket Accumulation

Within a window, points are allocated to buckets based on their digit values. Parallelism is achieved by partitioning the points among threads, each managing local buckets to minimize synchronization overhead, followed by combining local buckets into a global bucket per window.

Parallel Bucket Summation

Bucket point additions within each window are aggregated using a parallel tree based summation algorithm. Pairs of buckets are summed concurrently at each tree level, resulting in $O(\log n)$ addition depth where n is the number of buckets.

Final Combination with Powers of 3

Partial sums Q_j from each window are multiplied by $3^{w_t \cdot j}$ scalars independently, enabling parallel scalar multiplications. Large powers are efficiently computed using precomputation and fast exponentiation. Subsequently, the weighted sums $\sum_j [3^{w_t \cdot j}]Q_j$ are aggregated via parallel tree based addition, similarly reducing runtime.

Load Balancing and Synchronization

Optimized load balancing among threads and minimizing synchronization overhead are critical for achieving high parallel efficiency, particularly during bucket aggregation.
This structured, multi-level parallelization maximizes hardware utilization and throughput for efficient signed ternary multi scalar multiplication.
In the remainder of this paper, we do not take parallelization into account.

4 Comparison

In this section, we compare our method to state of the art approaches, evaluating their complexity and properties such as uniformity, precomputation, windowing, and parallelization. The comparison is conducted on an efficient tripling-friendly elliptic curve, considering both small and large values of d.
We start by outlining the key properties of the presented methods and compare them with ours in Table 3.

Method	Precomputation	Windowing	Parallelization	Uniformity
Shamir's trick	✓	×	×	✓
Bernstein	×	×	✓	✓
Binary Bucket method	×	✓	✓	✓
Optimised d-mul	✓	×	✓	✓
Binary Straus Shamir	✓	✓	×	×
Ternary Straus Shamir (TSS)	✓	×	×	✓
Ternary Bucket method	×	✓	✓	✓

Table 3: Comparison of the properties of our methods to the existing methods.

4.1 Comparison for small d

TSS is designed for fixed-base settings, allowing one-time precomputation and storage. For $d = 2$, Table 4 shows that its main operation costs $\frac{\ell}{\log_2(3)}\mathbf{T} + \frac{\ell}{\log_2(3)}\mathbf{A} \approx 15.12\ell\mathbf{M_1}$ on a Huff curve ($\mathbf{S_1} = 0.8\mathbf{M_1}$), outperforming Bernstein ($\approx 30\ell\mathbf{M_1}$) and slightly improving on Shamir's trick ($\approx 17.5\ell\mathbf{M_1}$). Although its precomputation cost ($3\mathbf{D} + 3\mathbf{A} \approx 60\mathbf{M_1}$) and storage of 8 points are higher, this reduces the main operation cost, making TSS suitable for long term applications like signature verification. For variable bases or memory constrained contexts, Bernstein's zero precomputation method remains preferable. For $d > 2$, Table 4 compares our method with existing approaches, using $w = 1$ for the binary Straus-Shamir method to simplify analysis. According to Table 4 and for $d > 2$, TSS is optimized for fixed base MSM where points remain fixed, delivers a highly efficient main operation with a cost of $\frac{\ell}{\log_2(3)}\mathbf{T} + \frac{\ell}{\log_2(3)}\mathbf{A}$, translating to approximately $15.12\ell\mathbf{M_1}$ on a Huff curve. This outperforms d-mul ($\approx (10\ell d + 10\ell + 10d)\mathbf{M_1}$),

Method	Main operation	Precomputation	Storage
Shamr's trick ($d = 2$)	$\ell\mathbf{D}+\frac{3}{4}l\mathbf{A}$	$1\mathbf{A}$	3 points
Bernstein ($d = 2$)	$\ell\mathbf{D}+2l\mathbf{A}$	–	2 points
d-mul	$\ell\mathbf{D} + (\ell + 1)d\mathbf{A}$	–	d points
Straus-Shamir	$\ell\mathbf{D} + \ell\mathbf{A}$	$(2^d - d - 1)\mathbf{A}$	$2^d - 1$ points
TSS	$\frac{\ell}{\log_2(3)}\mathbf{T} + \frac{\ell}{\log_2(3)}\mathbf{A}$	$(2^d - 1)\mathbf{D} + (3^d - 2^d - d)\mathbf{A}$	$3^d - 1$ points

Table 4: The comparison of complexity for small $d \geq 2$.

and binary Straus-Shamir ($\approx 20\ell\mathbf{M_1}$) for small d, due to the ternary representation reducing operation frequency. However, its precomputation of $(2^d - 1)\mathbf{D} + (3^d - 2^d - d)\mathbf{A} \approx (10 \cdot 3^d - 10d)\mathbf{M_1}$ and storage of $3^d - 1$ points are significantly higher than the zero precomputation of d-mul or the $(2^d - d - 1)\mathbf{A}$ and $2^d - 1$ points of binary Straus-Shamir, making it less suitable for memory constrained or variable base settings. TSS performs especially well in fixed base applications like signature verification, where the associated costs are spread out over repeated use.
To highlight the efficiency of Straus-Shamir method with ternary representation, Table 5 compares its main operation cost with other methods for $d = 4$ and $\ell = 256$ bits on the Huff curve. Table 5 demonstrates

Method	**Main operation**
d-mul	$12840\mathbf{M_1}$
Straus-Shamir	$5120\mathbf{M_1}$
TSS	$3871\mathbf{M_1}$

Table 5: Main operation cost comparison for $d = 4$, $\ell = 256$ on the Huff curve.

that the Straus-Shamir method with ternary representation achieves superior main operation efficiency, requiring approximately $3871M$ on the Huff curve. This advantage stems from its precomputation for fixed points.

4.2 Comparison for large d

In this section, we focus on comparing the signed ternary bucket method with the binary bucket method, providing both theoretical and practical comparisons.

4.2.1 Theoretical Comparison

We consider multi scalar multiplication over d scalars (or points), each bounded by a maximum bit-length l. The signed ternary and binary bucket methods encode scalars with window sizes w_t and w_b respectively. We analyze their asymptotic costs accounting for the average sparsity of ternary digits and the impact on the number of windows.

Number of windows per scalar

Due to the representation efficiency, the number of windows per scalar is approximated by

$$\text{Ternary: } \frac{0.63 \times l}{w_t}, \quad \text{Binary: } \frac{l}{w_b}$$

where the factor 0.63 accounts for the average density of nonzero digits in signed ternary representation.

Cost expressions

Total cost over all d scalars is approximated as

$$C_t = \mathcal{O}\left(\frac{0.63 \times l}{w_t} \times (d + 3^{w_t})\right)$$

$$C_b = \mathcal{O}\left(\frac{l}{w_b} \times (d + 2^{w_b})\right)$$

Condition for ternary Advantage

To ensure the ternary method is more efficient, we need that $C_t < C_b$, which yields

$$\frac{0.63}{w_t}(d + 3^{w_t}) < \frac{1}{w_b}(d + 2^{w_b})$$

This inequality can be reduced to $\frac{0.63}{w_t}3^{w_t} < \frac{1}{w_b}2^{w_b}$, which gives $3^{w_t} < \frac{w_t}{0.63w_b} \times 2^{w_b}$. Since $\frac{w_t}{0.63w_b}$ is usually close to 1 for typical window sizes, the critical exponent comparison remains

$$w_t \lesssim \frac{w_b}{\log_2 3} \approx 0.63 w_b$$

Interpretation

- The ternary representation reduces the number of windows proportionally by the factor 0.63, due to digit sparsity.
- Bucket size grows exponentially with window size, so minimizing w_t is critical.
- The derived inequality quantifies the maximum w_t allowed to maintain a computational cost advantage over optimized binary window w_b.

Other Considerations

- Both methods support parallel bucket accumulation and summation. Ternary often achieves better load balancing due to sparser bucket fills.
- Binary digit extraction is simpler and hardware friendly. Ternary digit extraction incurs additional software overhead.
- Precomputation costs, while not directly included, are comparable but generally higher for the ternary method due to digit decoding complexity.

The balance between bucket size and window count determines optimal parameters for MSM. By accounting for bit-length l, scalar count d, and digit sparsity (0.63 factor), we confirm that $w_t \lesssim 0.63 \times w_b$ ensures the signed ternary bucket method achieves better or comparable computational complexity than the binary bucket method for large-scale multi scalar multiplication.

4.2.2 Practical comparison

We implemented both the binary bucket and signed ternary bucket methods with `C` using the **RELIC** cryptographic library [1]. We analyze the group operation counts and running times on the $BLS12$ curve with 256-bit scalars. Here, we will assume that $\mathbf{T} = \mathbf{D} = \mathbf{A}$ due to the relatively small number of triplings and doublings compared to additions.

Number of group operations

Table 6 presents the number of group operations (additions) and the percentage gain achieved by the signed ternary method compared to the binary method. The binary and ternary window sizes, w_b and w_t, are chosen to be optimal. The ternary bucket method reduces the number of required additions by

d	wb	wt	Binary Adds	Signed Adds	Gain %
2^{10}	7	5	47124	41943	10.99
2^{11}	8	6	81842	75114	8.23
2^{12}	8	6	147111	130410	11.34
2^{13}	9	6	265848	241002	9.34
2^{14}	10	7	478461	445848	6.82
2^{15}	11	8	875483	826056	5.65
2^{16}	12	8	1612722	1514184	6.12
2^{17}	13	9	2948439	2713734	7.97
2^{18}	14	9	5567148	5073030	8.88
2^{19}	15	9	10026967	9791622	2.35
2^{20}	16	11	18874306	18386040	2.59
2^{21}	17	11	35389439	34114635	3.61
2^{22}	17	11	66846445	65572365	1.90

Table 6: Total number of group additions required by the standard binary method and the proposed signed ternary bucket method.

1.9 % − −11.3 % relative to the conventional binary method, achieving the largest savings (11.3 %) for MSM instances of size $d \leq 2^{13}$.

Running time comparison

Table 7 reports the measured running times (in seconds) for the binary bucket method and the signed ternary bucket method on MSM instances of size $d = 2^{10}$ to $d = 2^{22}$. For each method we provide two timings: one excluding scalar recoding ("w/o recoding") and the full execution time including recoding and all other overheads ("full"). The rightmost columns show the absolute differences (signed ternary bucket minus binary bucket) for both measurements. The signed ternary bucket method is generally faster or

d	Binary Bucket		Signed Ternary Bucket		Differences (Signed - Binary)	
	w/o bin rep	full	w/o tern rep	full	w/o rep	full
2^{10}	0.069	0.069	0.066	0.081	-0.003	+0.012
2^{11}	0.123	0.123	0.115	0.146	-0.008	+0.023
2^{12}	0.234	0.234	0.218	0.280	-0.016	+0.046
2^{13}	0.417	0.417	0.413	0.533	-0.004	+0.116
2^{14}	0.775	0.775	0.759	0.998	-0.016	+0.223
2^{15}	1.458	1.459	1.392	1.921	-0.066	+0.462
2^{16}	2.658	2.660	2.655	3.606	-0.003	+0.946
2^{17}	4.906	4.910	4.644	6.570	-0.262	+1.660
2^{18}	9.828	9.834	9.182	13.081	-0.646	+3.247
2^{19}	18.805	18.819	18.363	27.008	-0.442	+8.189
2^{20}	35.143	35.165	34.267	49.694	-0.876	+14.529
2^{21}	68.809	68.851	66.230	96.746	-2.579	+27.895
2^{22}	141.199	141.280	136.607	199.232	-4.592	+57.952

Table 7: Running times and differences (seconds).

comparable to the binary bucket when disregarding representation computation costs (w/o rep), especially for larger d (negative differences). However, when including representation computation (full), the signed ternary method runs slower, with increasingly larger overhead differences for bigger d.
The total number of group operations shows a consistent operation count gain of approximately 2–11% for the signed ternary method. This gain is partially offset by the representation computation overhead in running time. The current implementation relies on the RELIC library [1], which is primarily optimized for binary scalar representations. Consequently, on the fly conversion to signed ternary digits introduces non negligible overhead. This makes the signed ternary bucket method particularly advantageous in scenarios where scalars are known in advance, thereby allowing precomputation and storage of their signed ternary representation to fully exploit the reduced number of additions.

5 Application

In this section, we present two practical scenarios where multi scalar multiplication (MSM) can be effectively applied. For a small number of points d, we combine the TSS method with scalar decomposition techniques to enhance the efficiency of the Elliptic Curve Digital Signature Algorithm (ECDSA) numerical signaling. For large d, we apply the signed ternary bucket method to optimize multi scalar multiplication in the context of BLS signatures.

5.1 Application to the ECDSA Scheme

5.1.1 Presenting ECDSA [11, 16]

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely adopted digital signature scheme based on elliptic curve cryptography.

Key Generation

A private key d is randomly selected from the interval $[|1, n-1|]$, where n is the order of the base point G on the elliptic curve. The corresponding public key Q is computed by scalar multiplication $Q = [d]G$.

Signature

To sign a message m, the signer computes the signature pair (r, s) as follows:

1. Select a random nonce $k \in [|1, n-1|]$.

2. Compute the curve point $(x_1, y_1) = kG$.
3. Compute $r = x_1 \bmod n$, and if $r = 0$, choose a different k.
4. Compute the hash of the message $e = H(m)$ using a secure hash function.
5. Compute the signature scalar $s = k^{-1}\big(e + rd\big) \bmod n$, where k^{-1} is the modular inverse of k modulo n. If $s = 0$, select a different k.

The signature is the pair (r, s).

Signature verification

Using the signature (r, s), the message m, and the public key Q, the signature verification proceeds as follows:

1. Check that $r, s \in [1, n-1]$, if not, reject the signature.
2. Compute the message hash $e = H(m)$.
3. Compute $w = s^{-1} \bmod n$, $u_1 = ew \bmod n$, and $u_2 = rw \bmod n$.
4. Compute the curve point $(x_1, y_1) = u_1 G + u_2 Q$.
5. The signature is valid if $r \equiv x_1 \bmod n$, and invalid otherwise.

5.1.2 Applying TSS to ECDSA

Signature

We exploit the fact that the point G is fixed. We precompute and temporarily store, relative to a basis b, points of the form

$$P_0 = G, \quad P_1 = [b]G, \quad P_2 = [b^2]G, \quad \ldots, \quad P_m = [b^m]G,$$

where m is a positive integer. When generating the scalar k, we decompose it in the basis b as follows:

$$k = k_0 + k_1 b + \ldots + k_{d-1} b^{d-1},$$

with $d \leq m$. Therefore, computing $[k]G$ is equivalent to calculating the multi scalar multiplication

$$[k]G = [k_0]P_0 + [k_1]P_1 + \ldots + [k_{d-1}]P_{d-1}.$$

At this stage, we apply the Ternary Strauss Shamir (TSS) method to efficiently compute this MSM. We implemented the signature generation step in C using the RELIC cryptographic library [1]. To evaluate performance, we measured the running time of the original signature process and the signature process using TSS over 1000 random messages. The comparative results are presented as two curves in Figure 1. The implementation was carried out on Intel Core i7-1165G7.

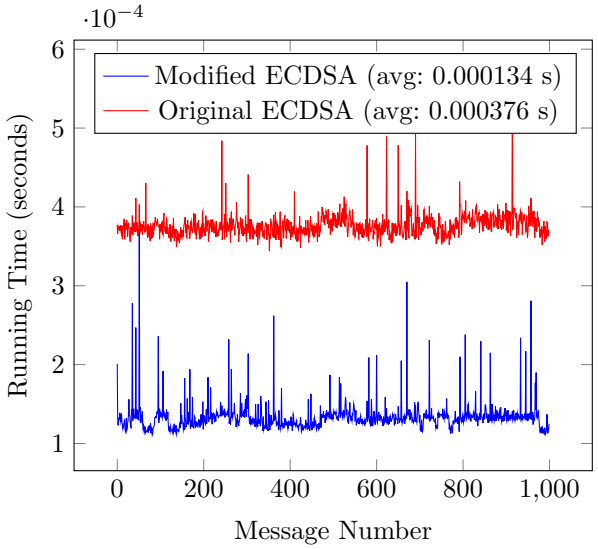

Figure 1: Running Times for ECDSA Signatures on NIST-P256

From Figure 1, it is evident that the TSS-based ECDSA variant achieves an average signature time of **0.000134 s**, which is approximately **2.88× faster** than the conventional implementation with an average of **0.000376 s**.

Signature verification

For the signature verification, we need to compute the MSM $[u_1]G + [u_2]Q$. Therefore, we directly apply TSS with parameter $d = 2$. We implemented the verification step for both the original ECDSA scheme and our modified version, and we measured the running time over 1000 randomly generated messages. The comparative results are presented in Figure 2.

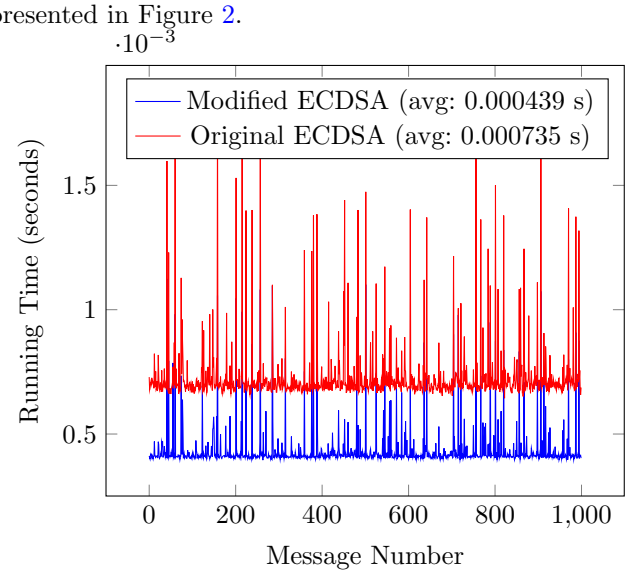

Figure 2: Running Times for ECDSA veifications on NIST-P256

From Figure 2, it is evident that the TSS-based ECDSA verification achieves an average time of **0.000439 s**, which is approximately **1.67× faster** than the conventional implementation with an average of **0.000735 s**.

Integrating TSS into standard ECDSA significantly improves both signing and verification efficiency. This gain arises from distributed precomputation of fixed-base scalar multiplications, a technique that is particularly effective when the base point G remains constant across operations.

5.2 Application to BLS signatures

This section demonstrates the MSM in BLS multi signature aggregation. Multi signatures allow d signers to produce a single short signature σ on message m. BLS signatures support public key aggregation, replacing d individual public keys with one compact aggregated key. This greatly improves blockchain scalability by reducing bandwidth for keys and signatures [**?**], which is why BLS has seen increasing adoption in blockchains.

5.2.1 Presenting BLS signatures

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ 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. We assume that we have d signers. The ith signer's private key is s_i, randomly chosen from $\mathbb{Z}_q$. The public key of this signer is $P_i = [s_i]G_2 \in \mathbb{G}_2$. The signer generates the BLS signature on message m by computing $\Sigma_i = [s_i]H_1(m) \in \mathbb{G}_1$. To verify signature Σ_i under public key P_i on message m, check that

$$e(\Sigma_i, G_2) = e\big(H_1(m), P_i\big).$$

The BLS signature scheme can aggregate multi signatures, as mentioned in [5**?** , 6] and the references therein. Considering this, we introduce the multi signatures aggregation approach that was suggested in [5]. Let $H_2 : \{0,1\}^* \to \mathbb{Z}_q$ be a new hash function. The signer sends the signature Σ_i to a designated 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 $\alpha_i = H_2(P_{k_i}||PKs)$, and $PKs = \{P_{k_1}, P_{k_2}, \cdots, P_{k_d}\}$ is the set of signers public keys. The aggregated form of the public keys is

$$APK = [\alpha_1]P_{k_1} + [\alpha_2]P_{k_2} + \cdots + [\alpha_d]P_{k_d}. \tag{2}$$

To verify signature Σ on message m, check that

$$e(\Sigma, G_2) = e\big(H_1(m), APK\big).$$

5.2.2 Our improvements

We introduce modifications aimed at improving the calculation of signatures and public keys aggregations in equations (1) and (2). In several cases, particularly those based on blockchain technology, the number of signers d is dynamic. Thus, each time the designated combiner must compute $2d$ scalar multiplications in (1) and (2). Consequently, the cost of the calculation increases as d does. To ease the designated combiner's workload, the authors of [6] suggest having each signer calculate $\alpha_i = H_2(P_{k_i}||PKs)$ and $[\alpha_i]\Sigma_i$, with $i \in [|1, d|]$. In many cases, signers lack resources, which could weaken their efforts. Since the α_i are public, signers forward their α_i, Σ_i to the combiner, which computes (1) and (2). We contribute to the designated combiner's work by enhancing computation in (1) and (2). Since the scalars α_i remain fixed until a new signer joins the group, we can leverage precomputed ternary representations of these α_i to efficiently apply the signed ternary bucket method.
We implemented BLS signature aggregation and public-key aggregation in C using the RELIC toolkit on the BLS12-381 curve, supporting both the classical binary bucket method and the signed ternary bucket method, with pairings computed via the optimal ate pairing. Benchmarks were performed on an Intel Core i7-1165G7 with `-O3 -march=native` and near-optimal window sizes. Results for public-key aggregation in $\mathbb{G}_2$ and signature aggregation in $\mathbb{G}_1$ (2^{10} to 2^{15} signers) are summarized in Table 8. The signed ternary

d	$\mathbb{G}_2$ (public keys)		$\mathbb{G}_1$ (signatures)	
	Time (s)	Operations	Time (s)	Operations
2^{10}	0.183 / **0.167**	47 057 / **41 943**	0.070 / 0.074	47 064 / **41 943**
2^{11}	0.318 / **0.288**	81 850 / **75 114**	0.120 / **0.119**	81 824 / **75 114**
2^{12}	0.589 / **0.542**	147 103 / **130 410**	0.230 / **0.217**	147 121 / **130 410**
2^{13}	1.065 / **1.052**	265 426 / **241 002**	0.419 / **0.406**	265 460 / **241 002**
2^{14}	1.925 / **1.901**	478 228 / **445 848**	0.740 / 0.749	478 207 / **445 848**
2^{15}	3.607 / **3.522**	873 140 / **826 056**	1.573 / **1.472**	873 249 / **826 056**

Table 8: BLS12-381 aggregation performance: binary bucket vs. signed ternary bucket (lower is better). Bold = better result.

bucket method reduces the number of group operations by 5.4–11.3 % in both groups. In the costly $\mathbb{G}_2$ group it provides clear runtime improvements of up to 9.4 %. In the much faster $\mathbb{G}_1$ group the runtime benefit is smaller (0–6 %) and occasionally disappears at 2^{15} signers due to implementation overhead, yet it never incurs a significant penalty.

6 Conclusion

In this paper, we explored ternary representations to enhance multi scalar multiplication on elliptic curves, considering both small and large numbers of points. For small sets, we introduced a ternary variant of the Straus–Shamir method, which shortens the main operation and leverages efficient tripling formulas with precomputed points. Applied to Huff curves for $d = 4$, $l = 256$, our approach achieves up to 69% fewer $\mathbb{F}_p$ multiplications compared to existing methods. When integrated into ECDSA, signing is 2.88× faster and verification 1.66× faster. For large sets, we proposed a signed ternary bucket method. Compared to the classical bucket method, it reduces group operations by up to 11.34%. Although initially slower in practice, the ternary bucket method becomes faster than the binary method when both representations are precomputed, making it especially suitable for BLS signatures and public-key aggregation. In these contexts, the ternary bucket method outperforms the binary method in both operation count and running time.

References

[1] Aranha DF, et al (2010) Relic is an efficient library for cryptography. https://github.com/relic-toolkit/relic, version 0.7.0, accessed 2025

[2] Azarderakhsh R, Karabina K (2016) Efficient algorithms and architectures for double point multiplication on elliptic curves. In: Proceedings of the 3rd Workshop on Cryptography and Security in Computing Systems. ACM, Prague, Czech Republic, pp 25–30, doi: 10.1145/2858930.2858932

[3] Bernstein DJ (2006) Differential addition chains. Self-published, URL http://cr.yp.to/ecdh/diffchain-20060219.pdf

[4] Bernstein DJ, Doumen J, Lange T, et al (2012) Faster batch forgery identification. In: Progress in Cryptology – INDOCRYPT 2012, Lecture Notes in Computer Science, vol 7668. Springer, Kolkata, India, pp 454–473, doi: 10.1007/978-3-642-34931-7_26

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

[6] Boneh D, Drijvers M, Neven G (2018) Compact multi-signatures for smaller blockchains. In: Advances in Cryptology – ASIACRYPT 2018, Lecture Notes in Computer Science, vol 11273. Springer, Brisbane, Australia, pp 435–464, doi: 10.1007/978-3-030-03329-3_15

[7] Bos JW, Costello C, Hisil H, et al (2013) High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In: Cryptographic Hardware and Embedded Systems – CHES 2013, Lecture Notes in Computer Science, vol 8086. Springer, Santa Barbara, CA, USA, pp 331–348, doi: 10.1007/978-3-642-40349-1_19

[8] Brown DRL (2006) Multi-dimensional Montgomery ladders for elliptic curves. Cryptology ePrint Archive, Paper 2006/220, URL https://eprint.iacr.org/2006/220

[9] El Housni Y, Botrel G (2022) EdMSM: Multi-scalar-multiplication for SNARKs and faster Montgomery multiplication. Cryptology ePrint Archive, Paper 2022/377, URL https://eprint.iacr.org/2022/377

[10] Hankerson D, Menezes A (2021) Elliptic curve cryptography. In: Encyclopedia of Cryptography, Security and Privacy. Springer, p 1–2, doi: 10.1007/978-3-642-27739-9_53-2

[11] Hankerson D, Menezes A, Vanstone S (2004) Guide to Elliptic Curve Cryptography. Springer

[12] Hisil H, Hutchinson A, Karabina K (2018) d-MUL: Optimizing and implementing a multidimensional scalar multiplication algorithm over elliptic curves. In: Security, Privacy, and Applied Cryptography Engineering – SPACE 2018, Lecture Notes in Computer Science, vol 11346. Springer, Bengaluru, India, pp 198–217, doi: 10.1007/978-3-030-05072-6_12

[13] Hutchinson A, Karabina K (2019) Constructing multidimensional differential addition chains and their applications. Journal of Cryptographic Engineering 9(1):1–19. doi: 10.1007/s13389-017-0179-y

[14] Hutchinson A, Karabina K (2020) A new encoding algorithm for a multidimensional version of the Montgomery ladder. In: Cryptology in Africa – AFRICACRYPT 2020, Lecture Notes in Computer Science, vol 12176. Springer, Dakar, Senegal, pp 403–422, doi: 10.1007/978-3-030-51938-4_20

[15] Jao D, De Feo L (2011) Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Lecture Notes in Computer Science, vol 7071. Springer, Taipei, Taiwan, pp 19–34, doi: 10.1007/978-3-642-25405-5_2

[16] Johnson D, Menezes A, Vanstone S (2001) The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security 1(1):36–63. doi: 10.1007/s102070100002

[17] Li W, Yu W, Wang K (2015) Improved tripling on elliptic curves. In: International Conference on Information Security and Cryptology, Springer, pp 193–205

[18] Lu T, Wei C, Yu R, et al (2023) Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. IACR Transactions on Cryptographic Hardware and Embedded Systems 2023(3):194–220

[19] Luo G, Fu S, Gong G (2023) Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp 358–380

[20] Montgomery PL (1987) Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177):243–264. doi: 10.2307/2007888

[21] Pottier X, de Ruijter T, Bertels J, et al (2025) Optimsm: Fpga hardware accelerator for zero-knowledge msm. IACR Transactions on Cryptographic Hardware and Embedded Systems 2025(2):489–510

[22] Rao S, Rao S (2016) Three dimensional Montgomery ladder, differential point tripling on Montgomery curves and point quintupling on Weierstrass and Edwards curves. In: Progress in Cryptology – AFRICACRYPT 2016, Lecture Notes in Computer Science, vol 9646. Springer, Fes, Morocco, pp 84–106, doi: 10.1007/978-3-319-31517-1_5

[23] Sakai Y, Sakurai K (2002) Algorithms for efficient simultaneous elliptic scalar multiplication with reduced joint Hamming weight representation of scalars. In: Information Security – ISC 2002, Lecture Notes in Computer Science, vol 2433. Springer, Sao Paulo, Brazil, pp 484–500, doi: 10.1007/3-540-45811-5_37

[24] Xavier CF (2022) PipeMSM: Hardware acceleration for multi-scalar multiplication. Cryptology ePrint Archive, Paper 2022/346

Author Contributions

All authors contributed to the development of this paper. The first draft of the manuscript was written by W. Haddaji. L. Ghammam, N. El Mrabet, and L. B. Abdelghani revised and provided feedback on previous versions of the manuscript. All authors read and approved the final version of the manuscript.

Funding

The authors have not disclosed any funding.

Data Availability

No datasets were generated or analysed during the current study.

Declarations

Conflict of interest

The authors have no conflict of interest to declare that are relevant to the content of this article.