Extending the Signed Non-zero Bit and Sign-Aligned Columns Methods to General Bases for Use in Cryptography

Abhraneel Dutta[1], Aaron Hutchinson[2], Koray Karabina[3]

[1] Florida Atlantic University, USA
[2] University of Waterloo, Canada
[3] National Research Council/Conseil national de recherches, Canada

Abstract. An efficient scalar multiplication algorithm is a crucial component of elliptic curve cryptosystems. We propose a scalar multiplication algorithm based on scalar recodings that is regular in nature. Our scalar multiplication algorithm is made from two scalar recoding algorithms called Recode and Align. Recode is the generalization of the signed non-zero bit recoding algorithm given by Hedabou, Pinel and Bénéteau in 2005. It recodes the k-ary representation of the given scalar into a signed non-zero form by means of a small lookup table. On the other hand, Align is the generalized k-ary version of the sign-aligned columns recoding algorithm given by Faz-Hernández, Longa and Sánchez in 2014. It recodes the k-ary representation of a scalar in such a way that the sign of each of its digits agrees with a given $\{1, -1\}$-valued sequence. When analyzing the choice of $k \in \{2, 3\}$, we find some theoretical evidence that $k = 3$ may offer better performance in certain scenarios.

Keywords: Elliptic curves, scalar multiplication, scalar recoding

1 Introduction

Elliptic curves have been commonly used in Diffie-Hellman (DH) type cryptographic *key exchange* protocols [7]. The runtime of Elliptic curve Diffie Hellman key exchange protocol (ECDH) and other elliptic curve cryptosystems are dominated by the scalar multiplication operations that can either be a single or a d-dimensional (e.g. Strauss's Algorithm [26]) in nature. We focus our work on a *variable scalar and fixed base* (VS-FB) setting of such algorithms that computes $\sum_{i=0}^{d-1} a_i P_i$, where the input scalars a_i are unknown while the points P_i are fixed in advance. A few examples of this setting would be split and comb algorithm [19, 24, 28], d-MUL [13]. Algorithms in this setting have a wide variety of applications, including the first round of ECDH, signature generation in the Elliptic Curve Digital Signature Algorithm (ECDSA) [16], and computing the kernel generator in the first round of Supersingular Isogeny-based Diffie-Hellman (SIDH) key exchange [14]. The double-and-add algorithm is the most basic scalar multiplication algorithm available. This algorithm is not often used

in cryptography due to its severe security flaws (discussed below), but it serves as a prototype for many modern algorithms used in practice. The algorithm is extremely versatile since no preprocessing is required. Generalizations and improvements on this algorithm can be seen in the window methods of [5], signed digit recodings of [15, 22, 25], and regular recodings of [17, 21, 23]. Furthermore, the scalar(s) may be written in k-ary as opposed to binary, in which the doubling operation is replaced with a k-multiplication.

The setting of VS-FB particularly benefits from optimization techniques since multiples of the base points P_i may be computed in advance for use during the runtime of the algorithm. One immediate consequence of this fact is that any single dimensional scalar multiplication aP with a having n bits may be computed using a d dimensional scalar multiplication algorithm which computes $\sum_{i=0}^{d-1} a_i P_i$, where $P_i = 2^{i \cdot n/d} P$ and the bits of a_i are bits $i \cdot n/d$ through $(i+1)n/d - 1$ of a for $0 \leq i < d$. The points P_i may be precomputed in advance since the point P is fixed. The flexibility of choosing d offers a time and memory tradeoff, where higher values of d generally produce faster algorithms but require storing more points. Algorithms that use this process include regular recoding algorithms like split and comb method [19, 20, 28], d-MUL [13] and regular recoding in [9, 12, 27] to switch from single to multi dimensional VS-FB scalar multiplication algorithm since they help split large scalars into smaller ones and recodes them to achieve side channel protection to some extent.

When selecting a scalar multiplication algorithm for use in cryptography, care must often be taken to ensure that the algorithm does not weaken the security of the cryptosystem. In ECDH for instance, an attacker deploying side-channel analysis (such as simple power analysis) will gain information about the bits of the scalar through any trivial operations performed in the main loop. We therefore often require scalar multiplication algorithms used in cryptography to be *regular*, so that each iteration of the main loop performs the same operations, independent of the input. See [3, 18] for surveys of scalar multiplication algorithms, attack types, and protection mechanisms. One method that such algorithms (Double and Add or Strauss's algorithm) can be made regular is by *recoding* the scalars into a representation which contains no 0 column. For Strauss's Algorithm only one such scalar, say a_0, needs such a recoding since then a nontrivial addition will always be performed. Such recoding methods often make use of negative digits in the representation of the scalars, referred to as *signed digit recodings*, and have been studied as early as the 1950s in [2, 4]. For elliptic curves, having signed digits in the scalars is not an issue since the cost of point inversion $P \mapsto -P$ is negligible. In 2001 Möller in [21] gave a nonzero base k recoding by iteratively changing the value of any zero digit $a_{i,j}$ to $-k$ and exchanging the digits $(a_{i,\ell}, \ldots, a_{i,j+1})$ for those of $(a_{i,\ell}, \ldots, a_{i,j+1})_2 + 1$. Okeya and Takagi in 2003 gave another regular recoding based on window method which converts the width-w NAF to an addition chain. Joye and Tunstall in [17] gave two similar nonzero recoding methods for scalars represented in base $k = 2^m$, one using signed digits (referred to as *JT algorithm*) and one unsigned digit sets, resulting in a regular (single dimensional) scalar multiplication algorithm. In the

unsigned case the scalar is recoded to have digits in the set $\{1, 2, \ldots, 2^m - 1\}$, which is valid for all integer scalars; in the signed case the recoded scalar has digits in $\{\pm 1, \pm 3, \ldots, \pm(2^m - 1)\}$, but the recoding is limited only to odd scalars.

Another recoding algorithm, also known as signed non-zero bit recoding, was presented by Hedabou, Pinel and Bénéteau in [12]. Here, a scalar a is recoded by replacing any string in its binary representation of the form $(1, 0, \ldots, 0, 1)$ of length at least 3 with the equal length string $(1, 1, -1, -1, \ldots, -1)$. Then a standard double-and-add algorithm can be used, which is now regular since there are no 0 digits. On an elliptic curve, no additional storage of points is required since the cost of point inversion is negligible.

In the multidimensional setting, a_0 may be recoded to a nonzero representation to give side channel protection as above. Since this recoding of a_0 already makes the resulting algorithm regular, the remaining scalars $a_1, \ldots, a_{d-1}$ may then use a recoding which increases efficiency by means of reducing the storage space required by the algorithm. This was achieved for the binary version of Straus's Algorithm by Faz-Hernandez, Longa, and Sanchez in [8], in which the authors perform a *sign alignment* on the scalars. This sign alignment recodes each scalar $a_i, 1 \leq i < d$, into a form b_i so that either $b_{i,j} = 0$ or $\mathsf{Sign}(b_{i,j}) = \mathsf{Sign}(b_{0,j})$ for each digit $b_{i,j}$ of b_i. The point added on iteration j of the main loop of the scalar multiplication algorithm then becomes $\sum_{i=0}^{d-1} b_{i,j} P_i = \mathsf{Sign}(b_0) \sum_{i=0}^{d-1} |b_{i,j}| P_i$. Since point inversion is negligible, this modification allows one to store only the points $\sum_{i=0}^{d-1} c_i P_i$ with $0 \leq c_i < 2$ and $c_0 = 1$. Compared to only performing a nonzero recoding on a_0, this sign alignment reduces the storage requirements from 2^d points down to 2^{d-1} points, all of which may be precomputed. This additional recoding on scalars $a_1, \ldots, a_{d-1}$ is referred to as Sign-Aligned Columns (SAC) recoding, in reference to the matrix visualization of Straus's Algorithm. The work of [8] also reports on an efficient, side-channel protected implementation of the scalar multiplication algorithm using a nonzero recoding of a_0 and SAC recoding for $a_1, \ldots, a_{d-1}$ in the case $d = 4$ on a GLV-GLS [10,11] twisted Edwards curve over $\mathbb{F}_{p^2}$.

While the savings gained from this technique is extraordinary given the minimal amount of preprocessing required for recoding scalars $a_1, \ldots, a_{d-1}$, the authors of [8] develop this method specifically for the case of a binary representation of the scalars and extending the technique to higher bases remains untouched. While generalizing the recoding and sign alignment of scalars to base k is an interesting mathematical problem on its own, such a generalization would allow for a scalar multiplication algorithm which has fewer iterations in the main loop at the cost of increased storage. Given a recoding algorithm valid for a general base k, the base could then be chosen as needed for a target application in a time/memory tradeoff.

1.1 Contributions

1. We generalize the signed and nonzero scalar recoding of [12] from base 2 to a general base $k \geq 2$. This recoding takes the k-ary representation $(a_{0,\ell-1}, \ldots, a_{0,0})$ of a positive ℓ-digit scalar a_0 not divisible by k as input, and outputs a nonzero

representation $(b_{0,\ell}, \ldots, b_{0,0})$ of a_0 for which $b_{0,j} \in \{\pm 1, \pm 2, \ldots, \pm(k-1)\}$ for $0 \leq j < \ell$, $b_{0,\ell} = 1$, and $(b_{0,\ell}, \ldots, b_{0,0})_k = a_0$. We present the recoding as Algorithm 1, called Recode, and provide mathematical foundation for its correctness. The digit recoding of Recode relies on lookups in a small table and it is regular in nature. Our algorithm is more versatile than the JT algorithm, where the input scalar must be an odd integer and the base k is of the form $k = 2^m$. In general our recoding is different from JT when running each algorithm with $k = 2^m$.
2. We generalize the sign-alignment algorithm for ℓ-digit scalars $a_1, \ldots, a_{d-1}$ given in [8] from base 2 to a general base $k \geq 2$. After a_0 is recoded to b_0 through the Recode algorithm, our sign-alignment algorithm recodes the k-ary representation $(a_{i,\ell-1}, \ldots, a_{i,0})$ of each scalar a_i for $1 \leq i < d$ into a form $(b_{i,\ell}, \ldots, b_{i,0})$. This recoded form has the properties that $b_{i,j} \in \{0, \pm 1, \pm 2, \ldots, \pm(k-1)\}$ for $0 \leq j < \ell$, $b_{i,\ell} \in \{0, 1\}$, either $b_{i,j} = 0$ or $\mathsf{Sign}(b_{i,j}) = \mathsf{Sign}(b_{0,j})$ for $0 \leq j \leq \ell$, and $(b_{i,\ell}, \ldots, b_{i,0})_k = (a_{i,\ell-1}, \ldots, a_{i,0})_k$. The sign-alignment algorithm is given in a naive form in Algorithm 2, called Align, for which we provide a correctness proof. To our knowledge, sign-alignment was only handled for $k = 2$ and the correctness proof was not provided. We derive another algorithm, called OptimizedAlign, as an alternate form of Align. OptimizedAlign has more resistance against simple power analysis attacks due to its regular nature and is presented in Algorithm 3.
3. We use Recode and OptimizedAlign to build a multidimensional scalar multiplication algorithm, Algorithm 4, which computes aP for a point P of order m in an abelian group $\mathbb{G}$ and any integer $a \in [1, m)$. Our algorithm uses a digit base k, which should be relatively prime to m, and a dimension d as parameters. After performing precomputation and recoding stages in a regular fashion, the scalar multiplication then proceeds as a regular sequence of w steps, with each step performing a point multiplication $\mathbf{K}$ by k and a nontrivial point addition $\mathbf{A}$ in $\mathbb{G}$, for a total computational cost of $w(\mathbf{K} + \mathbf{A})$. The algorithm requires a table of points in $\mathbb{G}$ of size $(k-1)k^{d-1}$ to be precomputed and stored, which may be done offline based on the values of P, k, and d.
4. We report on the performance of our scalar multiplication algorithm under this setup, and also when applying the split and comb method. We compare the efficiency of using our scalar multiplication algorithm over elliptic curves in the VS-FB setting for $k = 2$ and $k = 3$. Finally, we analyze under what conditions using $k = 3$ will outperform $k = 2$, when Twisted Edwards curves with projective coordinates are used.

In Section 2 we introduce the algorithm Recode and prove many facts resulting in its correctness. Section 3 details the sign alignment algorithm Align and its correctness proof, as well as an optimized version OptimizedAlign. Finally, in Section 4 we give our scalar multiplication algorithm and analyze its performance in the VS-FB scenario.

2 New Nonzero Digit Representations

The goal of this section is to detail the Recode algorithm, which is given in Section 2.2 along with a correctness proof. Before that, we prove a few theoret-

ical results on strings of integers in Section 2.1 which will assist in proving the correctness of Recode.

2.1 Theoretical Results

The majority of this section deals with strings of integers having certain forms, which we formally define below.

Definition 1. *For ℓ a non-negative integer, an ℓ-String is defined as a sequence of integers of length ℓ. The set of all strings is denoted by $\mathcal{S} = \{(a_{\ell-1}, \ldots, a_1, a_0) | \ \ell \in \mathbb{N}, a_i \in \mathbb{Z}$ for $0 \leq i \leq \ell - 1\}$. The unique 0-String is called the empty string.*

We use $||$ to denote the concatenation of two strings:

$$(a_{\ell_1-1}, \ldots, a_0) \,||\, (b_{\ell_2-1}, \ldots, b_0) := (a_{\ell_1-1}, \ldots, a_0, b_{\ell_2-1}, \ldots, b_0).$$

For length 1 strings in the context above we will often omit the parentheses, i.e., we write $a_0 \,||\, (b_{\ell_2-1}, \ldots, b_0)$ instead of $(a_0) \,||\, (b_{\ell_2-1}, \ldots, b_0)$.

Definition 2. *For $k \in \mathbb{Z}^+$ with $k \geq 2$, we define an integer-valued function $(-)_k$, called k-ary **value**, on the set of strings as $(-)_k : \mathcal{S} \to \mathbb{Z}$ by*

$$(a_{\ell-1}, a_{\ell-2}, \ldots, a_1, a_0) \mapsto \sum_{i=0}^{\ell-1} a_i k^i.$$

Remark 1. We note that Definition 2 yields a many-to-one function. In particular, the string obtained from the traditional k-ary representation of a positive integer z is included in the preimage set of z under our function in Definition 2.

The k-ary representation of any integer can be easily defined in terms of the above k-ary value function, which we include here for reference.

Definition 3. *For $k \geq 2$, the k-ary representation of a positive integer $a \in \mathbb{Z}^+$ is defined to be the unique ℓ-String $S = (a_{\ell-1}, \ldots, a_0)$ such that $\ell = \lceil \log_2(a) \rceil$, $a_i \in \{0, 1, \ldots, (k-1)\}$ for $0 \leq i < \ell$ with $a_{\ell-1} \neq 0$, and $(S)_k = a$.*

One of the ultimate goals of our Recode algorithm will be to give a nonzero representation of an integer as an ℓ-String. Therefore it makes sense to give specific attention to strings which contain 0 and strings which do not.

Definition 4. *An ℓ-String is* Good *if all of its entries are non-zero. An ℓ-String is* Bad *if it has the form $(0, 0, \ldots, 0, j)$ where j is nonzero and $\ell \geq 2$. For simplicity we will often refer to these as* GoodStrings *and* BadStrings*, respectively. We note that an ℓ-String could be neither good nor bad, and that the empty string is vacuously a* GoodString*.*

Lemma 1. *Let $S_\ell = (a_{\ell-1}, ..., a_0)$ be an ℓ-String with $\ell \geq 1$ and $a_0 \neq 0$. Then S_ℓ can be written as a concatenation of* GoodStrings *and* BadStrings*.*

Proof. We apply induction on ℓ. For the base case, $\ell = 1$ implies $S_\ell = (a_0)$ where $a_0 \neq 0$ which is clearly a concatenation of 0 many BadStrings and a single GoodString. While doing the induction step we start tracing the entries towards its decreasing index. If S_ℓ is a GoodString then we are done; otherwise, let i be the largest integer such that $a_i = 0$ with $1 \leq i \leq \ell - 1$. Since $a_0 \neq 0$, there exists an integer $0 \leq j < i$ which is the largest integer such that $a_j \neq 0$. By such choice of i and j, the substring $B := (a_i, ..., a_j)$ is a BadString and the substring

$$G := \begin{cases} (a_{\ell-1}, ..., a_{i+1}) & \text{if } i < \ell - 1 \\ \text{empty string} & \text{if } i = \ell - 1 \end{cases}$$

is a GoodString since $a_t \neq 0$ for all $t = i+1, i+2, \ldots, \ell - 1$. Now we have

$$S_\ell = (a_{\ell-1}, \ldots, a_{i+1}) \,||\, (a_i, \ldots, a_j) \,||\, S'_{\ell'} = G \,||\, B \,||\, S'_{\ell'},$$

where the ℓ'-String $S'_{\ell'}$ is the remaining part of S_ℓ with $0 \leq \ell' < \ell$. When $\ell' = 0$ we have that $S'_{\ell'}$ is the empty string, and we can conclude $S_\ell = G||B$ so that the result holds. Otherwise, for $\ell' > 0$, we apply our inductive hypothesis on $S'_{\ell'}$ and conclude that S_ℓ is a concatenation of GoodStrings and BadStrings.

Remark 2. In Lemma 1 we separate a GoodString from a BadString by taking all the non-zero entries within a GoodString from the end of a BadString until the beginning of the next BadString tracing from left to right. Thus in the representation of an ℓ-String we can never obtain two consecutive GoodStrings and as a result representing the ℓ-String as a concatenation of GoodStrings and BadStrings in this way is unique.

We now define a function which replaces BadStrings with equivalent GoodStrings.

Definition 5. *Let $k, \ell \geq 2$ and $\mathcal{S}_1$ and $\mathcal{S}_2$ be the set of* BadStrings *and* GoodStrings *respectively. We define a function $F : \mathcal{S}_1 \cup \mathcal{S}_2 \to \mathcal{S}_2$ as follows: for $S_\ell = (c_{\ell-1}, \ldots, c_0)$ with $c_i \in \{0, 1, \ldots, (k-1)\}$, set*

$$F(S_\ell) = \begin{cases} S_\ell & \text{if } S_\ell \text{ is a GoodString} \\ (1, -(k-1), \ldots, -(k-1), -(k-c_0)) & \text{if } S_\ell \text{ is a BadString} \end{cases}$$

where the lengths of S_ℓ and $F(S_\ell)$ are equal.

Notice that the output of F is indeed a GoodString.

Lemma 2. *If S_ℓ is any* GoodString *or* BadString*, then $(S_\ell)_k = (F(S_\ell))_k$.*

Proof. When S_ℓ is a GoodString we have $F(S_\ell) = S_\ell$ and the result is clear. Suppose now that S_ℓ is a BadString with $S_\ell = (0, 0, \ldots, 0, j)$. We have

$$(S_\ell)_k = (0, 0, \ldots, 0, j)_k = k^{\ell-1} \cdot 0 + k^{\ell-2} \cdot 0 + \cdots + k^1 \cdot 0 + k^0 \cdot j = j$$

and

$$\begin{aligned} (F(S_\ell))_k &= k^{\ell-1} \cdot 1 - k^{\ell-2} \cdot (k-1) - \cdots - k^1 \cdot (k-1) - k^0 \cdot (k-j) \\ &= k^{\ell-1} - (k-j) - (k-1)(k^{\ell-2} + \cdots + k) = j. \end{aligned} \tag{1}$$

Lemma 3. *Let S_{ℓ_u}, S_{ℓ_v}, and S_{ℓ_w} be ℓ_u, ℓ_v and ℓ_w-*Strings*, respectively, with S_{ℓ_v} being a* GoodString *or* BadString *and $\ell_u, \ell_w \geq 0,\ \ell_v > 0$. Then*

$$(S_{\ell_u}||S_{\ell_v}||S_{\ell_w})_k = (S_{\ell_u}||F(S_{\ell_v})||S_{\ell_w})_k.$$

Proof. By Lemma 2 we have

$$\begin{aligned}(S_{\ell_u}||S_{\ell_v}||S_{\ell_w})_k &= (S_{\ell_u})_k \cdot k^{\ell_v+\ell_w} + (S_{\ell_v})_k \cdot k^{\ell_w} + (S_{\ell_w})_k \\ &= (S_{\ell_u})_k \cdot k^{\ell_v+\ell_w} + (F(S_{\ell_v}))_k \cdot k^{\ell_w} + (S_{\ell_w})_k \\ &= (S_{\ell_u}||F(S_{\ell_v})||S_{\ell_w})_k.\end{aligned}$$

Theorem 1. *Let $S_\ell = (a_{\ell-1}, \ldots, a_1, a_0)$ be the k-ary representation of $N \in \mathbb{Z}^+$ with $k \nmid N$. Let $S_{\ell_n}||\cdots||S_{\ell_1}$ be the decomposition of S_ℓ into* GoodStrings *and* BadStrings *given by Lemma* 1*. Then*

$$N = (S_{\ell_n}||\cdots||S_{\ell_1})_k = (F(S_{\ell_n})||\cdots||F(S_{\ell_1}))_k.$$

Remark 3. For all $1 \leq i \leq n$, the entries of $F(S_{\ell_i})$ lie in the set $\{\pm 1, \ldots, \pm(k-1)\}$.

Proof. By definition of the a_i we have $N = (a_{\ell-1}, a_{\ell-2}, \ldots, a_1, a_0)_k = (S_{\ell_n}||\cdots||S_{\ell_1})_k$. We define a new ℓ-String from the S_{ℓ_i} as

$$S'_\ell := (F(S_{\ell_n})||\cdots||F(S_{\ell_1})) = (b_{\ell-1}, b_{\ell-2}, \ldots, b_1, b_0)$$

so that $b_i \neq 0$ for all i by to the definition of F. We apply Lemma 3 on each of the n GoodStrings or BadStrings in S_ℓ where at each time we replace S_{ℓ_i} by $F(S_{\ell_i})$. At the end of n number of steps the new recoded string is $(F(S_{\ell_n})||\cdots||F(S_{\ell_1}))$. At each of these steps the *k-ary* value of the updated string remains the same according to Lemma 3 by the equation

$$(S_{\ell_u}||S_{\ell_v}||S_{\ell_w})_k = (S_{\ell_u}||F(S_{\ell_v})||S_{\ell_w})_k.$$

While doing so we follow three different settings following Lemma 3

1. We begin with applying the function F according to Definition 5 on S_{ℓ_n} by setting $S_{\ell_v = s_{\ell_n}}$, S_{ℓ_u} to be an empty string and $S_{\ell_w} = S_{\ell_{n-1}}||\cdots||S_{\ell_1}$.
2. In the intermediate steps we set $S_{\ell_u} = F(S_{\ell_n})||\cdots||F(S_{\ell_{i+1}})$, $S_{\ell_v} = S_{\ell_i}$ and $S_{\ell_w} = S_{\ell_{i-1}}||\cdots||S_{\ell_1}$ for the intermediate applications $2 \leq i \leq n-1$
3. And at the final step we set $S_{\ell_u} = F(S_{\ell_n})||\cdots||F(S_{\ell_2})$, $S_{\ell_v} = S_{\ell_1}$and S_{ℓ_w} is an empty string

Therefore, we can conclude $N = (S_{\ell_n}||\cdots||S_{\ell_1})_k = (F(S_{\ell_n})||\cdots||F(S_{\ell_1}))_k$.

x_0 \ x_1	0	1	2	3	$\cdots$	m	$\cdots$	$k-1$
0	$-(k-1)$	$-(k-1)$	$-(k-2)$	$-(k-3)$	$\cdots$	$-(k-m)$	$\cdots$	$-(k-(k-1))$
1	1	1	2	3	$\cdots$	m	$\cdots$	$(k-1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	1	1	2	3	$\cdots$	m	$\cdots$	$(k-1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	1	1	2	3	$\cdots$	m	$\cdots$	$(k-1)$

Table 1. Lookup table where the entry in row x_0 and column x_1 represents the value of $M_{(x_0,x_1)}$.

2.2 Recoding Algorithm

In the previous section we have built a theory of replacing all zero entries within an ℓ-String with signed non-zero entries so that the k-*ary* value of the ℓ-String remains unchanged. While doing this we trace down all BadStrings, since only BadStrings contains zeroes in an ℓ-String, from left to right and replace each of them with a GoodString. Due to the fact that the length of each of the GoodStrings or BadStrings are very likely to be different makes this tracing process irregular. Therefore, building an algorithm following the tracing process that has such a lack of uniformity will make it vulnerable against simple power analysis attacks. Thus, in order to gain some degree of resistance against such attacks, we need to build an algorithm that replaces BadStrings by GoodStrings keeping the k-*ary* value unchanged in a uniform manner. To give it a regular structure, we use the lookup table given in Table 1, called the k-ary table. While generating the signed non-zero digit representation of a given ℓ-String $S_\ell = (a_{\ell-1}, \ldots, a_0)$, Algorithm 1 traces two entries a_i, a_{i-1} from left to right and replaces a_{i-1} by the (a_i, a_{i-1})-*th* entry of the k-*ary* table. Since at each step we are tracing two entries at a time and replace one entry by a table look up, this gives a regularity to the algorithm. We call this algorithm Recode. The output of Recode on input $(a_{\ell-1}, \ldots a_1, a_0)$ is exactly

$$\mathsf{Recode}(a_{\ell-1}, \ldots a_1, a_0) = (1, M_{(0,a_{\ell-1})}, M_{(a_{\ell-1},a_{\ell-2})}, \ldots, M_{(a_1,a_0)}). \qquad (2)$$

Note that all rows of Table 1 after the first are identical. Therefore it can be compressed to a $2 \times k$ table, where the two rows of the compressed table are identical with the first two rows of Table 1. Using the compressed k-ary table in the above context, one can instead replace a_{i-1} with the $(\chi(a_i), a_{i-1})$ entry of the compressed table, where χ is an *indicator function* (see Section 3.2). Alternatively, one might replace the entire table with an algebraic function which returns the appropriate value.

Before we state and prove the next Lemma we introduce the following notation. For a string $S_{\ell+1} = (c_\ell, c_{\ell-1}, c_{\ell-2}, \ldots, c_0)$, we define $S_{\ell+1} \sim (\ell, \ell-1)$ to be

Algorithm 1 Recode

Input: $k \in \mathbb{N}$ with $k \geq 2$, and a string of digits $(a_{\ell-1}, \ldots, a_1, a_0)$ with each $a_i \in \{0, 1, \ldots, k-1\}$ and $a_0 \neq 0$.

Output: A string $s = (s_\ell, s_{\ell-1}, \ldots, s_1, s_0)$ with $s_i \in \{\pm 1, \pm 2, \ldots, \pm(k-1)\}$, and $(s_\ell, s_{\ell-1}, \ldots, s_1, s_0)_k = (a_{\ell-1}, \ldots, a_1, a_0)_k$

1: $M \leftarrow \mathsf{GenTable}(k)$
2: **for** $i = 0$ to $\ell - 2$ **do** $s_i \leftarrow M_{(a_{i+1}, a_i)}$ **end for**
3: $s_{\ell-1} \leftarrow M_{(0, a_{\ell-1})}$ and $s_\ell \leftarrow 1$
4: **return** s

the string $S_{\ell+1}$ with the entries with index ℓ and $\ell - 1$ removed:

$$S_{\ell+1} \sim (\ell, \ell-1) := (c_{\ell-2}, \ldots, c_0).$$

Lemma 4. *For $1 \leq i \leq n$, let $S_{\ell_i} = (c^i_{\ell_i - 1}, \cdots, c^i_0)$ be either a* GoodString *or a* BadString*. Then*

$$\mathsf{Recode}(S_{\ell_n} || \cdots || S_{\ell_1}) = \mathsf{Recode}(S_{\ell_n}) \,||\, T_{\ell_n - 1} || \cdots || \, T_{\ell_1},$$

where

$$T_{\ell_i} = M_{(c_0^{i+1}, c^i_{\ell_i - 1})} || (\mathsf{Recode}(S_{\ell_i}) \sim (\ell_i, \ell_i - 1)), \text{ for all } 1 \leq i < n.$$

Note that each $\mathsf{Recode}(S_{\ell_i})$ is an (ℓ_i+1)-String, and so the removal operation $\sim$ is well-defined here.

Proof. Running Algorithm 1 with input $S_{\ell_n} || \cdots || S_{\ell_1}$ gives exactly the output

$$\mathsf{Recode}(S_{\ell_n} || \cdots || S_{\ell_1}) = S'_{\ell_n} || S'_{\ell_{n-1}} || \cdots || S'_{\ell_1},$$

where

$$\begin{aligned} S'_{\ell_n} &= (1, M_{(0, c^n_{\ell_n - 1})}, M_{(c^n_{\ell_n - 1}, c^n_{\ell_n - 2})}, M_{(c^n_{\ell_n - 2}, c^n_{\ell_n - 3})}, \ldots, M_{(c^n_1, c^n_0)}) \quad \text{and} \\ S'_{\ell_i} &= (M_{(c_0^{i+1}, c^i_{\ell_i - 1})}, M_{(c^i_{\ell_i - 1}, c^i_{\ell_i - 2})}, \;\; M_{(c^i_{\ell_i - 2}, c^i_{\ell_i - 3})}, \ldots, M_{(c^i_1, c^i_0)}) \quad \text{for } i < n. \end{aligned}$$

For $i < n$, we can directly compute

$$\begin{aligned} S'_{\ell_i} &= (M_{(c_0^{i+1}, c^i_{\ell_i - 1})}, M_{(c^i_{\ell_i - 1}, c^i_{\ell_i - 2})}, \ldots, M_{(c^i_1, c^i_0)}) \\ &= M_{(c_0^{i+1}, c^i_{\ell_i - 1})} || \Big(\big(1, M_{(0, c^i_{\ell_i - 1})}, M_{(c^i_{\ell_i - 1}, c^i_{\ell_i - 2})}, \ldots, M_{(c^i_1, c^i_0)}\big) \sim (\ell_i, \ell_i - 1) \Big) \\ &= M_{(c_0^{i+1}, c^i_{\ell_i - 1})} || \Big(\mathsf{Recode}(S_{\ell_i}) \sim (\ell_i, \ell_i - 1) \Big) = T_{\ell_i}. \end{aligned}$$

Finally, by Equation 2 we have $S'_{\ell_n} = \mathsf{Recode}(S_{\ell_n})$.

Lemma 5. *Let an ℓ-*String *S_ℓ be written as $S_{\ell_n} || \ldots || S_{\ell_1}$ where $S_{\ell_i} = (c^i_{\ell_i - 1}, \cdots, c^i_0)$ is either a* GoodString *or a* BadString *for all $1 \leq i \leq n$. We define*

$$T_{\ell_i} = M_{(c_0^{i+1}, c^i_{\ell_i - 1})} || (\mathsf{Recode}(S_{\ell_i}) \sim (\ell_i, \ell_i - 1))$$

for all $1 \leq i < n$ as in Lemma 4. Then for all $i \in \{1, 2, \ldots, n-1\}$ we have $T_{\ell_i} = F(S_{\ell_i})$, where F is given in Definition 5.

Proof. By Equation 2 the full output of the Recode algorithm on input S_{ℓ_i} is $\mathsf{Recode}(S_{\ell_i}) = \left(1, M_{(0,c^i_{\ell_i-1})}, \ldots, M_{(c^i_1,c^i_0)}\right)$. We therefore have

$$\begin{aligned} T_{\ell_i} &= \left(M_{(c^{i+1}_0,c^i_{\ell_i-1})} || (\mathsf{Recode}(S_{\ell_i}) \sim (\ell_i, \ell_i - 1))\right) \\ &= (M_{(c^{i+1}_0,c^i_{\ell_i-1})}, M_{(c^i_{\ell_i-1},c^i_{\ell_i-2})}, M_{(c^i_{\ell_i-2},c^i_{\ell_i-3})}, \ldots, M_{(c^i_1,c^i_0)}). \end{aligned}$$

Using the definition of M we can directly compute the M values above. Recall that each c^i_j is nonzero when S_{ℓ_i} is a GoodString, each c^i_j is zero for $j > 0$ when S_{ℓ_i} is a BadString, and every c^i_0 is always nonzero. For $1 < j < \ell_i$, we then have

$$\begin{aligned} M_{(c^{i+1}_0,c^i_{\ell_i-1})} &= \begin{cases} 1 & \text{if } S_{\ell_i} \text{ is a BadString} \\ c^i_{\ell_i-1} & \text{if } S_{\ell_i} \text{ is a GoodString,} \end{cases} \\ M_{(c^i_j,c^i_{j-1})} &= \begin{cases} -(k-1) & \text{if } S_{\ell_i} \text{ is a BadString} \\ c^i_{j-1} & \text{if } S_{\ell_i} \text{ is a GoodString,} \end{cases} \\ M_{(c^i_1,c^i_0)} &= \begin{cases} -(k-c^i_0) & \text{if } S_{\ell_i} \text{ is a BadString} \\ c^i_0 & \text{if } S_{\ell_i} \text{ is a GoodString.} \end{cases} \end{aligned}$$

By replacing each M table lookup with its value, we get

$$T_{\ell_i} = \begin{cases} (1, -(k-1), \ldots, -(k-1), -(k-c^i_0)) & \text{if } S_\ell \text{ is a BadString} \\ (c^i_{\ell_i-1}, c^i_{\ell_i-2}, \ldots, c^i_1, c^i_0) & \text{if } S_\ell \text{ is a GoodString.} \end{cases}$$

The right hand side in the above equation is exactly $F(S_{\ell_i})$.

Lemma 6. *If S_ℓ is any* GoodString *or* BadString*, then* $\mathsf{Recode}(S_\ell)_k = (S_\ell)_k$.

Proof. Suppose first that S_ℓ is a BadString of length ℓ with $S_\ell = (0, 0, \ldots, 0, a)$ for some $a \in \{1, 2, \ldots, k-1\}$. Since $M_{(0,a)} = -(k-a)$ and $M_{(0,0)} = -(k-1)$, the output of Recode is

$$\mathsf{Recode}(S_{\ell_n}) = \underbrace{(1, -(k-1), -(k-1), \ldots, -(k-1), -(k-a))}_{\ell+1}.$$

Computing the k-ary value results in a telescoping sum:

$$\begin{aligned} (\mathsf{Recode}(S_\ell))_k &= [1 \cdot k^\ell] + [-(k-1) \cdot k^{\ell-1}] + \cdots + [-(k-1) \cdot k] + [-(k-a)] \\ &= k^\ell + [-k^\ell + k^{\ell-1}] + \cdots + [-k^2 + k] + [-k + a] = a = (S_\ell)_k. \end{aligned}$$

This proves the statement when S_ℓ is a BadString. Suppose now that S_ℓ is a GoodString with $S_\ell = (a_{\ell-1}, a_{\ell-2}, \ldots, a_1, a_0)$ for some $a_i \in \{1, 2, \ldots, k-1\}$ for all $0 \le i \le \ell - 1$. Since $M_{(a_{i+1},a_i)} = a_i$ the output of Recode is mostly the same as the input, but we must account for the 0 that is initially prepended. We have $M_{(0,a_{\ell-1})} = -(k - a_{\ell-1})$, and so

$$\mathsf{Recode}(S_\ell) = (1, -(k - a_{\ell-1}), a_{\ell-2}, \ldots, a_1, a_0)$$

When taking the k-ary value, the "1" and the "k" in the expression above cancel each other:

$$\begin{aligned}(\mathsf{Recode}(S_\ell))_k &= [1 \cdot k^\ell] + [-(k - a_{\ell-1}) \cdot k^{\ell-1}] + (a_{\ell-2}, \ldots, a_1, a_0)_k \\ &= a_{\ell-1} \cdot k^{\ell-1} + (a_{\ell-2}, \ldots, a_1, a_0)_k = (S_\ell)_k.\end{aligned}$$

This proves the statement when S_ℓ is a GoodString and concludes the proof.

Theorem 2. *Let $S_\ell = (a_{\ell-1}, \ldots, a_0)$ be the k-ary representation of some $N \in \mathbb{N}$ with $k \nmid N$. Write $S_\ell = S_{\ell_n}||\cdots||S_{\ell_1}$ with each S_{ℓ_i} a* GoodString *or* BadString. *Then* $\mathsf{Recode}(S_\ell) = \mathsf{Recode}(S_{\ell_n})||F(S_{\ell_{n-1}})||\ldots||F(S_{\ell_1})$, *and*

1. $(\mathsf{Recode}(S_\ell))_k = N$,
2. $\mathsf{Recode}(S_\ell)$ *has entries in* $\{\pm 1, \ldots, \pm(k-1)\}$,
3. *the length of* $\mathsf{Recode}(S_\ell)$ *is* $\ell + 1$.

Proof. As in Lemma 4 we define

$$T_{\ell_i} = M_{(c_0^{i+1}, c_{\ell_i - 1}^i)}||(\mathsf{Recode}(S_{\ell_i}) \sim (\ell_i, \ell_i - 1))$$

for each $1 \leq i < n$. Applying Recode with Lemmas 4 and 5, we get

$$\begin{aligned}\mathsf{Recode}(S_\ell) &= \mathsf{Recode}(S_{\ell_n}||\cdots||S_{\ell_1}) = \mathsf{Recode}(S_{\ell_n})||T_{\ell_{n-1}}||\cdots||T_{\ell_1} \\ &= \mathsf{Recode}(S_{\ell_n})||F(S_{\ell_{n-1}})||\cdots||F(S_{\ell_1}). \qquad (3)\end{aligned}$$

Taking the k-ary value, we have

$$\begin{aligned}(\mathsf{Recode}(S_\ell))_k &= (\mathsf{Recode}(S_{\ell_n})||F(S_{\ell_{n-1}})||\cdots||F(S_{\ell_1}))_k \\ &= (\mathsf{Recode}(S_{\ell_n}))_k \cdot k^{\sum_{j=1}^{n-1} \ell_i + 1} + (F(S_{\ell_{n-1}})||\cdots||F(S_{\ell_1}))_k.\end{aligned}$$

Now we apply Theorem 1 on $(S_{\ell_{n-1}}||\cdots||S_{\ell_1})_k$ and Lemma 6 on $\mathsf{Recode}(S_{\ell_n})$ to get

$$(F(S_{\ell_{n-1}})||\cdots||F(S_{\ell_1}))_k = (S_{\ell_{n-1}}||\cdots||S_{\ell_1})_k, \quad (\mathsf{Recode}(S_{\ell_n}))_k = (S_{\ell_n})_k$$

and conclude $\mathsf{Recode}(S_\ell)_k = (S_\ell)_k = N$, proving the first claim.

To show claim 2, we examine Equation (3). Each $F(S_{\ell_n})$ has entries in $\mathcal{K} = \{\pm 1, \ldots, \pm(k-1)\}$ by definition, and the proof of Lemma 6 explicitly computes the value of $\mathsf{Recode}(S_{\ell_n})$, from which one can see it also has values in $\mathcal{K}$.

By examining Algorithm 1, it should be clear that the length of the output array B is exactly $\ell + 1$, which shows claim 3.

3 Generalized Sign Aligned Recoding Algorithm

In Section 2 we introduced an algorithm that recodes the *k-ary* representation of an integer into a signed non-zero digit representation. As was discussed in Section 1, it is desirable to have a sign aligned recoding algorithm which yields a better storage complexity for our scalar multiplication algorithm. This section will introduce a sign-alignment algorithm for a general base k and prove its correctness.

Algorithm 2 Align

Input: $k \in \mathbb{N}$ with $k \geq 2$; a sign sequence $s = (s_\ell, s_{\ell-1}, \ldots, s_0)$ with each $s_i \in \{-1, 1\}$ and $s_\ell = 1$; and $a = (a_{\ell-1}, \ldots, a_0)$ with each $a_i \in \{0, 1, \ldots, k-1\}$.

Output: A string $b = (b_\ell, b_{\ell-1}, \ldots, b_0)$ with $b_i \in \{0, \pm 1, \pm 2, \ldots, \pm(k-1)\}$, with $(b_\ell, b_{\ell-1} \ldots, b_0)_k = (a_{\ell-1}, \ldots, a_0)_k$, and either $b_i = 0$ or $\mathsf{Sign}(b_i) = s_i$ for all i.

```
1: b_i ← a_i for i = 0, 1, ..., ℓ − 1.
2: b_ℓ ← 0.
3: for i = 0 to ℓ − 1 do
4:    if b_i = k then
5:       b_i ← 0, b_{i+1} ← b_{i+1} + 1
6:    else
7:       if s_i = 1 then
8:          b_i ← b_i, b_{i+1} ← b_{i+1}
9:       else
10:         if b_i = 0 then
11:            b_i ← b_i, b_{i+1} ← b_{i+1}
12:         else
13:            b_i ← −(k − b_i), b_{i+1} ← b_{i+1} + 1
14:         end if
15:      end if
16:   end if
17: end for
18: return b
```

3.1 Basic Sign-Aligned Recoding

A *sign sequence* is any ℓ-String consisting of 1's and -1's. Suppose $S_\ell = (a_{\ell-1}, \ldots, a_0)$ is the k-ary representation of a positive integer N, and that S_ℓ is recoded using Recode to a new representation $(b_\ell, \ldots, b_0)$. In this section, we detail Algorithm 2, called Align, and its supporting mathematical proof. For a given length $\ell+1$ sign-sequence $(s_\ell, \ldots, s_0)$, the Align algorithm recodes an ℓ-String with entries in $\{0, 1, \ldots, k-1\}$ so that the sign of each of its the entries agrees with the sign sequence. In the scalar multiplication algorithm presented in Section 4, we will take $(\mathsf{Sign}(b_\ell), \ldots, \mathsf{Sign}(b_0))$ as the sign sequence. The following theorem makes this discussion more precise and serves as a correctness statement for Algorithm 2.

Theorem 3. *Let $\xi_{\ell+1} = (s_\ell, \ldots, s_0)$ such that $s_i \in \{1, -1\}$ with $s_\ell = 1$, and let $S_\ell = (a_{\ell-1}, \ldots, a_0)$ with $a_i \in \{0, 1, \ldots, k-1\}$. Let $\mathsf{Align}(\xi_{\ell+1}, S_\ell) = (b_\ell, \ldots, b_0)$. Then:*

1. *For $0 \leq i < \ell$ we have $b_i \in (-k, k) \cap \mathbb{Z}$, and $b_\ell \in \{0, 1\}$,*
2. *For $0 \leq i \leq \ell$, either $b_i = 0$ or $\mathsf{Sign}(b_i) = s_i$,*
3. *$(b_\ell, b_{\ell-1}, \ldots, b_0)_k = (a_{\ell-1}, \ldots, a_0)_k$.*

Proof. We first prove claims (1.) and (2.). Notice that during the algorithm every entry of b with the exception of b_0 is updated exactly twice (though sometimes

to the same value), with b_i being updated on iterations $i-1$ (by a "$b_{i+1} \leftarrow$" update rule) and i (by a "$b_i \leftarrow$" update rule).

We'll first prove the claims true for b_0, which is only modified on iteration $i = 0$. At the start of the algorithm b_0 is initialized with value a_0, which lies in the set $\{0, 1, \ldots, k-1\}$ by assumption, and so the conditional statement "$b_0 = k$" cannot evaluate to true. In the Else branch on line 6, if $s_0 = 1$ then b_0 is to remain unmodified from its value of a_0 and the claims are true. Alternatively we have $s_0 = -1$. Then either $b_0 = a_0 = 0$ and its value is left unchanged and the claims are true, or $b_0 = a_0 \in \{1, \ldots, k-1\}$ in which case the final value becomes $-(k - a_0) \in \{-1, -2, \ldots, -(k-1)\}$ and $\mathsf{Sign}(b_0) = -1 = s_0$. This proves claims (1.) and (2.) for $i = 0$.

Now fix an index $1 \leq j \leq \ell - 1$ to examine. As previously stated, b_j is modified only on iterations $i = j-1$ and $i = j$ using separate sets of rules. All of the "$b_{i+1} \leftarrow$" update rules either leave the variable unchanged or increment it by 1, and we will handle each case separately. First suppose on iteration $i = j - 1$ that b_j remains unchanged, so that at the start of iteration $i = j$ we have $b_j = a_j \in \{0, 1, \ldots, k-1\}$. Here we cannot have $b_i = k$, so we proceed into the Else branch of line 6. There are three separate cases to consider:

1. $s_j = 1$: then no update to b_i is performed and b_j's final value is a_j.
2. $s_j = -1$ and $b_j = 0$: then b_j is assigned final value 0.
3. $s_j = -1$ and $b_j \in \{1, 2, \ldots, k-1\}$: then b_j is assigned final value $-(k - a_j) \in \{-1, -2, \ldots, -(k-1)\}$.

In each case above claims (1.) and (2.) of the theorem are satisfied.

Now suppose that iteration $i = j - 1$ takes the action "$b_{i+1} \leftarrow b_{i+1} + 1$", so that at the start of iteration $i = j$ we have $b_j = a_j + 1 \in \{1, 2, \ldots, k\}$. Here it's possible that $b_i = k$, in which event b_j gets final value 0. Otherwise, the cases proceed identical to the previous paragraph except that the option $b_j = 0$ is eliminated. In all cases claims (1.) and (2.) are therefore satisfied for $b_0, \ldots, b_{\ell-1}$.

The final digit b_ℓ is initialized to 0 and is only modified on the final iteration $i = \ell - 1$ by one of the "$b_{i+1} \leftarrow$" update rules. Consequently it can only have value 0 or 1. Note that $s_\ell = 1$ by assumption. This proves claims (1.) and (2.) for all digits.

We prove claim (3.) by showing inductively that it holds true upon completion of each iteration of the main loop, and will therefore hold true at the end of the algorithm. The first line of the algorithm initializes b with value a, serving as the base case of the induction. The branching structure of the algorithm gives four possible cases to consider, two of which leave the value of b unmodified. We consider the two nontrivial cases below. In the following, we let b_j^i denote the value of the variable b_j at the time of completion of the i-th iteration of the algorithm. Let $0 \leq i \leq \ell - 1$ be fixed.

The first nontrivial case is if $b_i^{i-1} = k$. In this case the update rules give $b_i^i = 0$ and $b_{i+1}^i = b_{i+1}^{i-1} + 1$, and all other values of b remain unchanged from the

previous iteration giving $b_j^i = b_j^{i-1}$ for $j \neq i, i+1$. We then have

$$\begin{aligned}(b_\ell^i, \ldots, b_{i+1}^i, b_i^i, \ldots, b_0^i)_k &= (b_\ell^{i-1}, \ldots, b_{i+1}^{i-1} + 1, 0, \ldots, b_0^{i-1})_k \\ &= (b_\ell^{i-1}, \ldots, b_{i+1}^{i-1}, k, \ldots, b_0^{i-1})_k \qquad \text{(same } k\text{-ary value)} \\ &= (b_\ell^{i-1}, \ldots, b_{i+1}^{i-1}, b_i^{i-1}, \ldots, b_0^{i-1})_k \\ &= (a_{\ell-1}, \ldots, a_0)_k. \qquad \text{(by inductive hypothesis)}\end{aligned}$$

The remaining case is when $s_i = -1$ and $b_i^{i-1} \neq 0$, where the new values become $b_i^i = -(k - b_i^{i-1})$ and $b_{i+1}^i = b_{i+1}^{i-1} + 1$. Here we get

$$\begin{aligned}(b_\ell^i, \ldots, b_{i+1}^i, b_i^i, \ldots, b_0^i)_k &= (b_\ell^{i-1}, \ldots, b_{i+1}^{i-1} + 1, -(k - b_i^{i-1}), \ldots, b_0^{i-1})_k \\ &= (b_\ell^{i-1}, \ldots, b_{i+1}^{i-1}, b_i^{i-1}, \ldots, b_0^{i-1})_k \qquad \text{(same } k\text{-ary value)} \\ &= (a_{\ell-1}, \ldots, a_0)_k. \qquad \text{(by inductive hypothesis)}\end{aligned}$$

In all cases claim (3.) is satisfied. This completes the induction and concludes the proof.

3.2 Optimized Regular Sign-Alignment

As written, Algorithm 2 contains numerous branches which are heavily dependent upon the input string, which can potentially be exploited in side channel attacks. In this section, we present Algorithm 3, an alternate form of Algorithm 2, which is more resistant against side-channel attacks.

To get started, we notice that each of the conditional statements in Algorithm 2 are a check for equality between two values. All of these checks can be put under the same umbrella by introducing an *indicator function* $\chi : \mathbb{Z} \to \{0, 1\}$, whose value is defined to be

$$\chi(x) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ if } x \neq 0. \end{cases}$$

The function χ can be used to transform conditional statements of a certain form into arithmetic evaluations in the following manner:

$$\begin{array}{l}\textbf{if } a = b \textbf{ then} \\ \quad var \leftarrow X \\ \textbf{else} \\ \quad var \leftarrow Y\end{array} \qquad \longrightarrow \qquad var \leftarrow \chi(a-b) \cdot X + (1 - \chi(a-b)) \cdot Y$$

assuming that X and Y have numeric values. By repeatedly replacing the innermost conditional statements in Algorithm 2 with their equivalent arithmetic expressions as above and simplifying, we arrive at Algorithm 3. This version of Align makes three calls to the indicator function χ before computing the new value of b, and completely eliminates the branching based on secret data seen in Algorithm 2. Therefore, if both χ and integer arithmetic are implemented in a constant-time fashion, we obtain better side-channel resilience with Algorithm 3.

Algorithm 3 OptimizedAlign(s, a)

Input: $k \in \mathbb{N}$ with $k \geq 2$; a sign sequence $s = (s_\ell, s_{\ell-1}, \ldots, s_0)$ with each $s_i \in \{-1, 1\}$ and $s_\ell = 1$; and $a = (a_{\ell-1}, \ldots, a_0)$ with each $a_i \in \{0, 1, \ldots, k-1\}$.

Output: A string $b = (b_\ell, b_{\ell-1}, \ldots, b_0)$ with $b_i \in \{0, \pm 1, \pm 2, \ldots, \pm(k-1)\}$, with $(b_\ell, b_{\ell-1} \ldots, b_0)_k = (a_{\ell-1}, \ldots, a_0)_k$, and either $b_i = 0$ or $\mathsf{Sign}(b_i) = s_i$ for all i.

1: $b_\ell \leftarrow 0$ and $b_i \leftarrow a_i$ for $i = 0, 1, \ldots, \ell - 1$.
2: **for** $i = 0$ **to** $\ell - 1$ **do**
3: $\quad u_1 \leftarrow \chi(b_i - k)$, $u_2 \leftarrow \chi(s_i - 1)$, $u_3 \leftarrow \chi(b_i)$
4: $\quad v \leftarrow (1 - u_1) \cdot (1 - u_2) \cdot (1 - u_3)$
5: $\quad b_i \leftarrow (1 - u_1) \cdot b_i - v \cdot k$ and $b_{i+1} \leftarrow b_{i+1} + u_1 + v$
6: **end for**
7: **return** b

4 Cryptographic Applications

Algorithm 6 presents our scalar multiplication algorithm, which uses Recode (Algorithm 1) and OptimizedAlign (Algorithm 3) from the previous sections. Any cryptographic protocol which requires a VS-FB scalar multiplication computation is a suitable application for our algorithm; see Section 1 for examples of such protocols. If the precomputation stage has low cost (which may be the case on a curve with efficient endomorphisms, for instance), Algorithm 6 would also be suitable for applications in the VS-VB setting. In this section, we analyze Algorithm 6 and its cost in terms of both run time and storage. We also discuss choosing the parameters k and d, and the choice of curve and coordinate system which result in a more efficient algorithm.

Throughout this section we will use the following notation for the cost of various operations: $\mathbf{a}, \mathbf{m}$, $\mathbf{s}$, and $\mathbf{i}$ will respectively denote the costs of addition/subtraction, multiplication, squaring, and inversion in a fixed field $\mathbb{F}$; we will use $\mathbf{A}, \mathbf{D}, \mathbf{T}$, and $\mathbf{K}$ to respectively denote the elliptic curve operations of point addition, doubling, tripling, and general multiplication by k on some fixed elliptic curve E.

4.1 Explanation of Algorithm 4

Our scalar multiplication algorithm uses an integer parameter $k \geq 2$, for which the input scalar is written in base k and recoded, and an integer parameter $d \geq 2$, which is used to split the input scalar into d subscalars for sign alignment. The base point P of the algorithm should have order m relatively prime to k, where P is a point of an abelian group $\mathbb{G}$, with m having $\ell = \lceil \log_2(m) \rceil$ bits. The input scalar a is an integer in $[1, m)$, and the algorithm returns aP as output. We let $w := \left\lceil \frac{\lceil \ell \log_k(2) \rceil}{d} \right\rceil$, which will determine the length in digits of each subscalar. We briefly explain the details of Algorithm 4 as follows.

In the *Precomputation Stage* of Algorithm 4, specific multiples of P are computed which will be used in the main loop of the algorithm (on line 11). For

Algorithm 4 Scalar Multiplication

Input: An integer $k \geq 2$, a point P of order m relatively prime to k in an abelian group $\mathbb{G}$, with $\ell = \lceil \log_2(m) \rceil$; a scalar $a \in [1, m)$; an integer $d \geq 2$ determining the number of subscalars, with $w := \left\lceil \frac{\lceil \log_k(2^\ell) \rceil}{d} \right\rceil$.

Output: aP.

Precomputation Stage

1: $T[uk+v] \leftarrow (u_{d-1}k^{(d-1)w} + \cdots + u_2 k^{2w} + u_1 k^w + v)P$ for all $v \in [1, k)$, $u \in [0, k^{d-1})$ where $(u_{d-1}, \ldots, u_1)$ is the k-ary representation of u.

Recoding Stage

2: kMULT $\leftarrow a \bmod k$
3: **if** kMULT $= 0$ **then** $a \leftarrow m - a$
4: $(a_{dw-1}, \ldots, a_0) \leftarrow k$-ary representation of a, padded with sufficiently many zeroes on the left.
5: $(b_w^1, \ldots, b_0^1) \leftarrow \mathsf{Recode}(a_{w-1}, \ldots, a_0)$
6: $(b_w^i, \ldots, b_0^i) \leftarrow \mathsf{OptimizedAlign}(\mathsf{Sign}(b_w^1), \ldots, \mathsf{Sign}(b_0^1)), (a_{iw-1}, \ldots, a_{(i-1)w}))$ for $2 \leq i \leq d$.
7: $\mathbb{B}_i \leftarrow |b_i^d k^{d-1} + b_i^{d-1} k^{d-2} + \cdots + b_i^2 k + b_i^1|$ for $0 \leq i \leq w$.

Evaluation Stage

8: $Q \leftarrow T[\mathbb{B}_w]$
9: **for** $i = w - 1$ **to** 0 **by** -1 **do**
10: $\quad Q \leftarrow kQ$
11: $\quad Q \leftarrow Q + \mathsf{Sign}(b_i^1) \cdot T[\mathbb{B}_i]$
12: **end for**
13: **if** kMULT $= 0$ **then return** $-Q$
14: **return** Q

$0 \leq i < d$, let $P_i = k^{iw}P$. Then line 1 computes all points in the set

$$\{u_{d-1}P_{d-1} + \cdots + u_1 P_1 + vP_0 : 0 \leq u_i < k, 1 \leq v < k\} \tag{4}$$

and arranges them into a table T. For a fixed base algorithm (such as the first round of ECDH) this step will be performed offline at no cost, and so we do not go into details on how the computation of the points in this set is carried out.

In the *Recoding Stage*, the scalar a is recoded using Recode and OptimizedAlign from the previous sections, which gives our scalar multiplication algorithm its regular structure. Note that the input scalar a can be any integer in $[1, m)$, while the input to Recode requires k not to divide a. To address this restriction, we assign a variable kMULT to $a \bmod k$ and if this equals 0 we update scalar a to $m-a$. In this case, the return value should be corrected to $-Q$. We have included the restriction that k and m be relatively prime so that k cannot divide both a and $m-a$ simultaneously. Line 3 uses this approach to ensure that k does not divide a. Afterwards, a is written in k-ary as $a = (a_{dw-1}, \ldots, a_0)_k$ (padded with zeroes if necessary) and partitioned into d subscalars; let $a^{(i)} := (a_{iw-1}, \ldots, a_{(i-1)w})$ for $1 \leq i \leq d$. On line 5 the Recode algorithm is invoked on $a^{(1)}$ with base k (note that $a_0 \neq 0$ since by this step a is not divisible by k) to get an output $b^{(1)}$

with the properties of Theorem 2. Next a sign alignment is performed on each of the remaining subscalars $a^{(2)}, \ldots, a^{(d)}$ on line 6 to find $b^{(2)}, \ldots, b^{(d)}$, each of which satisfies the properties of Theorem 3 with respect to the signs of the digits of $b^{(1)}$. Finally line 7 collects the digits of the $b^{(j)}$ into new scalars $\mathbb{B}_i$ which give the entries of the lookup table T to be used in the evaluation stage.

The last lines 8–12 of Algorithm 4 define the *Evaluation Stage*. This stage proceeds by performing one point multiplication by k and one point addition in $\mathbb{G}$ on each iteration of the loop by use of the $\mathbb{B}_i$. Due to the sign alignment of the scalars the sum involved is either an addition or subtraction based on the sign of $b_i^{(1)}$.

4.2 General Cost Analysis

Here we derive general costs involved in Algorithm 4 for a general group $\mathbb{G}$ in terms of storage and computation. The main storage cost involved is due to line 1, where all points in the set given in Equation 4 are required to be stored in the table T. The number of points in this set is exactly $(k-1)k^{d-1}$. Storage costs involved during runtime are that of storing the digits of a, the digits of the recoded scalars $b^{(i)}$, the integers $\mathbb{B}_i$, and the point Q; we assume these costs are negligible in comparison to the size of the table T and therefore ignore them.

The computational costs involved in lines 3–7 deal mainly with small integer arithmetic (including the operations involved in Recode and OptimizedAlign), and we assume the construction of T in line 1 is performed offline at no cost, like in the VS-FB setting. Therefore, we only consider the computational costs involved in the evaluation stage. The loop contains w iterations, which each perform a single point multiplication by k and point addition (we assume inversion in $\mathbb{G}$ is negligible, such as on an elliptic curve). This gives a total cost of $w(\mathbf{K}+\mathbf{A}) = \left\lceil \frac{\lceil \ell \log_k(2) \rceil}{d} \right\rceil (\mathbf{K}+\mathbf{A})$. We summarize these costs as:

$$\text{Algorithm 4 Storage Cost:} \qquad (k-1)k^{d-1} \text{ points of } \mathbb{G}, \tag{5}$$

$$\text{Algorithm 4 Computation Cost:} \qquad \left\lceil \frac{\lceil \ell \log_k(2) \rceil}{d} \right\rceil (\mathbf{K}+\mathbf{A}). \tag{6}$$

The split and comb method is a well-known general-use VS-FB algorithm generally attributed to Yao [28] and Pippenger [24] in 1976. Lim and Lee improved upon this method in [19] to give an efficient special case in 1994. The split and comb method further subdivides each of the d subscalars into v subsubscalars, increasing the storage space linearly in v with the benefit of reducing the number of $\mathbf{K}$ operations required. Algorithm 4 can be modified to use the split and comb method by partitioning the $(b_w^i, \ldots, b_0^i)$ strings into v many blocks, padding with 0's if necessary. New integers $\mathbb{B}_{i,j}$ can be derived in a similar way, and the main loop performs $1\mathbf{K}$ and $v\mathbf{A}$ per iteration, with fewer iterations total. See [19] for further details on the split and comb method. The total costs for

Algorithm 4 modified with the split and comb method are:

$$\text{Storage Cost:} \qquad v(k-1)k^{d-1} \text{ points of } \mathbb{G}. \tag{7}$$

$$\text{Computational Cost:} \quad \left\lceil \frac{\lceil \ell \log_k(2) \rceil}{d} \right\rceil \mathbf{A} + \left(\left\lceil \frac{\lceil \lceil \ell \log_k(2) \rceil / d \rceil + 1}{v} \right\rceil - 1 \right) \mathbf{K}. \tag{8}$$

4.3 Concluding Remarks

An interesting question to ask is under what scenario will $k = 3$ outperform $k = 2$ in our algorithm (Algorithm 4 with the split and comb modification) at the same storage level?

To investigate this we now compare the computational costs for the $k = 2$ and $k = 3$ settings assuming approximately the same amount of total storage space. We choose 256-bit scalar and denote the cost of the point doubling, tripling and addition operations as $\mathbf{D}$, $\mathbf{T}$ and $\mathbf{A}$ respectively. We use parameters d_2, v_2 for the $k = 2$ setting and d_3, v_3 for $k = 3$ setting. This leads to solving the following inequality, subject to the equal storage constraint:

$$\left\lceil \frac{256 \log_3 2}{d_3} \right\rceil \mathbf{A} + \left(\left\lceil \frac{\left\lceil \frac{256 \log_3 2}{d_3} \right\rceil + 1}{v_3} \right\rceil - 1 \right) \mathbf{T} \leq \left\lceil \frac{256}{d_2} \right\rceil \mathbf{A} + \left(\left\lceil \frac{\left\lceil \frac{256}{d_2} \right\rceil + 1}{v_2} \right\rceil - 1 \right) \mathbf{D} \tag{9}$$

$$3^{d_3 - 1} 2 v_3 \approx 2^{d_2 - 1} v_2 \tag{10}$$

To derive a concrete solution, we specialize to the case of Twisted Edwards curves using points in projective coordinates. Here, we assume table elements are stored with Z value 1 and therefore use the cost values of $\mathbf{A} = 9\mathbf{m} + 1\mathbf{s}$, $\mathbf{D} = 3\mathbf{m} + 4\mathbf{s}$, and $\mathbf{T} = 9\mathbf{m} + 3\mathbf{s}$ (see [1]), and assume $1\mathbf{s} = 0.8\mathbf{m}$. We then solve Approximation 10 to obtain $v_3 \approx \frac{2^{d_2 - 2} v_2}{3^{d_3 - 1}}$, substitute this expression into inequality 9, and consider the two cases $d_2 = d_3 = 2$ and $d_2 = d_3 = 4$. For these cases, we find $v_2 \geq 6$ for $d_2 = d_3 = 2$ and $v_2 \geq 13$ for $d_2 = d_3 = 4$. Taking $d_2 = 2$ and $v_2 = 6$ yields a storage level of $2^1 \cdot 6 = 12$ points, while $d_2 = 4$ and $v_2 = 13$ gives a storage level of $2^3 \cdot 13 = 108$ points; these storage sizes are certainly feasible for implementation, and so $k = 3$ may indeed be more desirable than $k = 2$ in realistic scenarios. Table 2 lists some particular cases where $k = 3$ outperforms $k = 2$ at comparable storage levels when $d = 2, 4$.

We should note that our comparisons are made over a fixed choice of d (while varying v). This is the case in many cryptographic applications. For example, if a protocol is implemented using elliptic curves with endomorphisms of degree 4 [6], then d is naturally fixed to 4. Similarly, $d = 2$ is a common choice in applications where curves with endomorphisms of degree 2 are deployed [10, 11], or when double point multiplication is required such as in isogeny based cryptosystems [14].

	$k=2$			$k=3$		
d	v_2	Storage	Cost Per Bit	v_3	Storage	Cost Per Bit
2	6	12	5.41	2	12	4.88
	9	18	5.24	3	18	4.30
	12	24	5.14	4	24	3.99
	54	108	4.95	18	108	3.28
	108	216	4.92	36	216	3.19
4	13	104	2.55	2	108	2.46
	20	160	2.52	3	162	2.15
	27	216	2.50	4	216	2.01

Table 2. Fixing $d = 2$ and 4 the run time in terms of cost per bit unit for $k = 3$ is better than $k = 2$ when the storage *i.e* the number of precomputed points is equal (or approximately equal)

4.4 Future Work

We considered only Twisted Edwards curves and projective coordinates for representing points, and we focused on the VS-FB setting. Other curve and coordinate choices may be more suitable for certain scenarios, such as using a tripling-oriented Doche-Icart-Kohel curve in the $k = 3$ setting, or using a mixing of projective and extended coordinates on Twisted Edwards curves in the $k = 2$ setting. A careful analysis of the performance of Algorithm 4 will be required for both scenarios before a clear winner can be determined. A detailed C implementation would also be very insightful to see how our theoretical costs reflect the timings achieved in practice, for both VS-FB and VS-VB settings.

Algorithms we presented in this paper provide some degree of resistance against side-channel attacks thanks to their regular nature. However, further analysis and implementation would be required to evaluate their security in practice.

A Example of Algorithm 4

Let P be point in an abelian group, such as an elliptic curve over a finite field $\mathbb{F}_p$, and that $|P| = m$ with $\ell = \lceil \log_2(m) \rceil = 15$. Then scalars in $[1, m)$ are represented with length $\lceil \log_3(2^\ell) \rceil = 10$, appending leading 0s if necessary. Suppose we run Algorithm 4 with inputs P, $a = 39907$, $d = 4$ and $k = 3$. Notice k does not divide a, so the exact value of m is irrelevant. The length of each subscalar is determined as $w = \lceil \ell/d \rceil = \lceil 10/4 \rceil = 3$. Algorithm 4 computes $39907P$ as follows.

Precomputation Stage: According to Algorithm 4 we first precompute a table T of points of the form

$$T[uk + v] = (u_3, u_2, u_1, v)_{k^w} P = (u_3 k^{3w} + u_2 k^{2w} + u_1 k^w + v)P$$

for $v = 1, 2$ and $u \in [0, 3^3)$ where (u_3, u_2, u_1) is the k-ary representation of u.

Recoding Stage: The *k-ary* representation of $a = 39907$ is $(2, 0, 0, 0, 2, 0, 2, 0, 0, 1)$. We pad $dw - \ell = 4 \cdot 3 - 10 = 2$ many 0's on its left to get

i	b^1	b^2	b^3	b^4	$\mathbb{B}_i$	$T[\mathbb{B}_i]$	s_i	$Q \leftarrow 3Q$	$Q \leftarrow Q + s_i T[\mathbb{B}_i]$
3	1	1	0	1	31	$19711P$	—	—	$19711P$
2	−2	0	0	−2	56	$39368P$	−1	$59133P$	$59133P - 39368P = 19765P$
1	−2	−2	0	−2	62	$39422P$	−1	$59295P$	$59295P - 39422P = 19873P$
0	−2	−1	0	−1	32	$19712P$	−1	$59619P$	$59619P - 19712P = 39907P$

Table 3. Evaluation stage of Algorithm 4 for input $a = 39907$, $k = 3$, and $d = 4$.

$(0, 0, 2, 0, 0, 0, 2, 0, 2, 0, 0, 1)$. The string is then split into $d = 4$ many 3-Strings A_1, A_2, A_3, A_4, shown below. Recode is then applied to $A_1 = (0, 0, 1)$ to get the nonzero scalar $b^1 = (1, -2, -2, -2)$, and Align (or OptimizedAlign) is applied to A_2, A_3, A_4 with the sign sequence

$$\mathsf{Sign}(b^1) = (\mathsf{Sign}(1), \mathsf{Sign}(-2), \mathsf{Sign}(-2), \mathsf{Sign}(-2)) = (1, -1, -1, -1),$$

to get sign-aligned scalars b^2, b^3, and b^4. This process can be visualized in matrix form as:

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} 0\ 0\ 1 \\ 2\ 0\ 2 \\ 0\ 0\ 0 \\ 0\ 0\ 2 \end{bmatrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} \begin{matrix} \xrightarrow{\mathsf{Recode}} \\ \\ \xrightarrow{\mathsf{Align}} \\ \\ \end{matrix} \begin{bmatrix} 1 & -2 & -2 & -2 \\ 1 & 0 & -2 & -1 \\ 0 & 0 & 0 & 0 \\ 1 & -2 & -2 & -1 \end{bmatrix} = \begin{bmatrix} b^1 \\ b^2 \\ b^3 \\ b^4 \end{bmatrix}$$

Note that the above 4×3 matrix is the matrix used in Straus' algorithm. The final step of the recoding stage is to compute $\mathbb{B}_0, \mathbb{B}_1, \mathbb{B}_2, \mathbb{B}_3$. These are determined by interpreting the columns of the 4×4 matrix above in base k, with the top row being the least significant digit and $\mathbb{B}_0$ corresponding to the rightmost column. Specifically, the $\mathbb{B}_i$ are derived as follows. For convenience, the corresponding table entries and the signs $s_i = \mathsf{Sign}(b_i^1)$ are also listed.

$$\begin{aligned}
\mathbb{B}_0 &= |(-1, \quad 0, -1, -2)_3| = 32, & T[32] &= 19712P, & s_0 &= -1, \\
\mathbb{B}_1 &= |(-2, \quad 0, -2, -2)_3| = 62, & T[62] &= 39422P, & s_1 &= -1, \\
\mathbb{B}_2 &= |(-2, \quad 0, \quad 0, -2)_3| = 56, & T[56] &= 39368P, & s_2 &= -1, \\
\mathbb{B}_3 &= |(\ \ 1, \quad 0, \quad 1, \quad 1)_3| = 31, & T[31] &= 19711P.
\end{aligned}$$

Evaluation Stage: It can be easily verified that the following equality holds:

$$39907P = T[31] \cdot 3^3 + s_2 T[56] \cdot 3^2 + s_1 T[62] \cdot 3^1 + s_0 T[32].$$

In the evaluation stage Algorithm 4 computes $39907P$ in a triple-and-add manner using the above equality. We stress that in general the $T[\mathbb{B}_i]$ are never zero, so trivial additions never occur. We therefore initialize a point Q to have value $Q = T[\mathbb{B}_3] = T[31] = 19711P$, and then proceed in an alternating sequence of tripling Q and adding/subtracting the appropriate $T[\mathbb{B}_i]$ to Q. Each of the steps are given in Table 3.

References

1. Explicit Formulas Database. `https://hyperelliptic.org/EFD/`.
2. A.Avizienis. Signed-Digit Number Representations for Fast Parallel Arithmetic. *IRE Transactions on Electronic Computers*, EC-10:289–400, 1961.
3. R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2012.
4. A. Booth. A Signed Binary Multiplication Technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4:236–240, 1951.
5. J. Bos and M. Coster. Addition Chain Heuristics. *Advances in Cryptology - CRYPTO' 89. CRYPTO 1989. Lecture Notes in Computer Science*, 435:400–407, 1990.
6. P. Longa C. Costello. Four$\mathbb{Q}$: Four-Dimensional Decompositions on a $\mathbb{Q}$-curve over the Mersenne Prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 214–235, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
7. W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.
8. A. Faz-Hernández, P. Longa, and A. Sánchez. Efficient and Secure Algorithms for GLV-Based Scalar Multiplication and Their Implementation on GLV-GLS Curves. *Topics in Cryptology - CT-RSA 2014, Lecture Notes in Computer Science*, 8366:1–27, 2014.
9. M. Feng, B. Zhu, C. Zhao, and S. Li. Signed MSB-Set Comb Method for Elliptic Curve Point Multiplication. *Information Security Practise and Experience Conference - ISPEC 2005. Lecture Notes in Computer Science*, 3903:13–24, 2006.
10. S. Galbraith, X. Lin, and M. Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. *Advances in Cryptology - EUROCRYPT 2009. Lecture Notes in Computer Science*, 5479:518–535, 2009.
11. R. Gallant, J. Lambert, and S. Vanstone. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. *Advances in Cryptology - CRYPTO 2001. Lecture Notes in Computer Science*, 2139:190–200, 2001.
12. M. Hedabou, P. Pinel, and L. Bebeteau. Countermeasures for Preventing Comb Method Against SCA Attacks. *Information Security Practise and Experience Conference - ISPEC 2005, Lecture Notes in Computer Science*, 3439:85–96, 2005.
13. H. Hisil, A. Hutchinson, and K. Karabina. d-MUL: Optimizing and Implementing a Multidimensional Scalar Multiplication Algorithm over Elliptic Curves. *Security, Privacy, and Applied Cryptography Engineering. SPACE 2018. Lecture Notes in Computer Science*, 11348:198–217, 2018.
14. D. Jao and L. De Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. *Post-Quantum Cryptography - PQC 2011. Lecture Notes in Computer Science*, 7071:19–34, 2011.
15. J. Jedwab and C. Mitchell. Minimum Weight Modified Signed-Digit Representations and Fast Exponentiation. *Electronics Letters*, 25:1171–1172, 1989.
16. D. Johnson, A. Menezes, and S. Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1:36–63, 2001.
17. M. Joye and M. Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. *Progress in Cryptology - AFRICACRYPT 2009. Lecture Notes in Computer Science*, 5580:334–349, 2009.

18. K. Karabina. A Survey of Scalar Multiplication Algorithms. In F. Chung, R. Graham, F. Hoffman, R. C. Mullin, L. Hogben, and D. B. West, editors, *50 years of Combinatorics, Graph Theory, and Computing*, chapter 20, pages 359–386. Chapman and Hall/CRC, 2019.
19. C. Lim and P. Lee. More Flexible Exponentiation with Precomputation. *Advances in Cryptology - CRYPTO '94. Lecture Notes in Computer Science*, 839:95–107, 1994.
20. K. McCurley, D. Wilson, E. Brickell, and D. Gordon. Fast Exponentiation with Precomputation. *Advances in Cryptology - EUROCRYPT '92, Lecture Notes in Computer Science*, 658:200–207, 1993.
21. B. Möller. Securing Elliptic Curve Point Multiplication Against Side-Channel Attacks. *Information Security - ISC 2001. Lecture Notes in Computer Science*, 2200:324–334, 2001.
22. F. Morain and J. Olivos. Speeding Up the Computations on an Elliptic Curve Using Addition-Subtraction Chains. *Theoretical Informatics and Application*, 24:531–543, 1990.
23. K. Okeya and T. Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Curve Scalars Multiplications Against Side- Channel Attacks. *Topics in Cryptology - CT-RSA 2003. Lecture Notes in Computer Science*, 2612:328–342, 2003.
24. N. Pippenger. On the Evaluation of Powers and Related Problems. *17th Annual IEEE Symposium on Foundations of Computer Science*, pages 258–263, 1976.
25. J. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes, and Cryptography*, 19:195–249, 2000.
26. E. Straus. Addition Chains of Vectors (problem 5125). *American Mathematical Monthly*, 70:806–808, 1964.
27. M. Xu, M. Feng, B. Zhu, and S. Liu. Efficient Comb Methods for Elliptic Curve Point Multiplication Resistant to Power Analysis. Cryptology ePrint Archive, Report 2005/22, 2005.
28. A. Yao. On the Evaluation of Powers. *SIAM Journal on Computing*, 5:281–307, 1976.