$wNAF^*$, an Efficient Left-to-Right Signed Digit Recoding Algorithm

Brian King

Indiana University Purdue University Indianapolis
briking@iupui.edu

Abstract. Efficient implementations of cryptosystems are important in order to conserve resources, memory, power, etc., which will enable resource-limited devices to compute necessary cryptographic operations. One technique that successfully reduces the number of necessary operations is the use of a signed digit representation for the key, because it reduces the nonzero density of the representation. One such signed digit representation is the *non-adjacent form* or NAF. Moreover, one can make more reductions in the number of nonzero symbols of the key by expressing the key with a w-ary NAF or $wNAF$ form. A drawback is that one needs to parse the key twice, once to construct the $wNAF$ representation and the second time to perform the necessary cryptographic operation. At Crypto 2004 [10], Okeya et. al. introduced a w-ary representation $wMOF$, which possess the same nonzero density as $wNAF$, as well as an algorithm that computes $wMOF$ in a left-to-right manner utilizing very little memory ("memory-less"). At that time, the authors noted that a left-to-right "memory-less" algorithm that computes $wNAF$ is an open problem. In this work, we define $wNAF^*$, a generalization of $wNAF$. Further, we construct a left-to-right "memory-less" algorithm that computes the w-ary $wNAF^*$ representation of a key and demonstrate that $wNAF^*$ is as efficient as $wNAF$. Our work will demonstrate that the left-to-right $wNAF^*$ recoding algorithm closely resembles the right-to-left $wNAF$ recoding algorithm.

1 Introduction

It is well known that if one uses a signed digit representation of the cryptographic key then one can reduce the number of complex computations that are needed to perform the cryptographic operation. One such signed digit representation is the *non-adjacent form* or NAF. By expressing a key in NAF form, one will reduce the number of nonzero symbols, however this will introduce signed digits. Fortunately, in many algebraic settings an inverse is very efficient to compute. This is especially true when using elliptic curve cryptosystems (ECC), where the inverse (negative) of an ECC point can be trivially computed from the EC point. When using the NAF form of a key, the computational complexity of computing the scalar multiple kP will be very efficient, since the hamming weight of a key in NAF form is less than the hamming weight of a key. One can make further

S.M. Bellovin et al. (Eds.): ACNS 2008, LNCS 5037, pp. 429–445, 2008.

reductions in the number of nonzero symbols of the key by expressing it with a w-ary NAF or wNAF form [2]. RSA and discrete-log cryptosystems would also benefit by expressing a key in NAF or wNAF form because of the reduction of the hamming weight (number of nonzero bits) of the key. In order to be more succinct, we will emphasize the use of NAF and wNAF with ECC, however our work will impact efficient implementations of all cryptosystems.

Algorithms demonstrating how to compute the NAF form of a key can be found in [2,12,4] and algorithms to compute the wNAF form are provided in [2,4]. In addition to NAF, there are other alternative ways to reduce the number of nonzero symbols in the key representation by using techniques such as *sliding window with NAF* [3].

At Crypto 2004 Okeya et. al. [10] introduced a left-to-right binary signed digit recoding algorithm called Mutual Opposite Form or MOF. The goal of the authors of [10] was to develop an algorithm that constructs an "efficient signed digit representation that could be computed in a left-to-right manner without requiring additional storage". That is, a signed digit representation that can be computed in a left-to-right manner on-the-fly. The importance of such a construction is that as the key bits are generated (in a left-to-right manner), one can immediately construct the signed digit representation. Further, one can immediately start the computation of the cryptographic operations as the most significant digits of the signed digit representation become available. This is a good security practice, in that the secret key should only be available to software modules that require it. Moreover, it is good practice to limit the number of software modules that use/possess key. Consequently, if the cryptographic computation can be computed simultaneously as the signed digits are generated, this would limit the access of the key to software modules.

A left-to-right construction of a signed digit representation NAF was first constructed in [5], however this construction was limited to $w = 2$ and the authors did not provide a left-to-right algorithm of the more efficient representation $wNAF$. In [10], it was noted that the construction of a memory-less left-to-right algorithm for $wNAF$ is an open problem. In this work, we examine the open problem posed in [10], and provide an algorithm that can construct a left-to-right $wNAF^*$ algorithm, where $wNAF^*$ is a left-to-right generalization of $wNAF$.

Summary of our results. We will address an open problem by defining a signed digit representation $wNAF^*$ and constructing a left-to-right "memory-less" method for computing $wNAF^*$. Our work will establish the relationship between left-to-right w-ary signed digit recoding and right-to-left w-ary signed digit recoding. We will also provide several algorithms, including an algorithm that computes the ECC scalar multiple kP, processing the key k using the left-to-right $wNAF^*$ recoding.

2 Background

In [11], Reitwiesner introduced the signed digit representation referred to as NAF. The definition that an integer k is written in NAF form is:

Definition 1 (NAF). *[2] A nonnegative integer* $k = \sum_{i=0}^{n-1} k_i 2^i$ *where* $k_i \in \{-1, 0, 1\}$ *is said to be in non-adjacent form (NAF) provided* $k_i \cdot k_{i+1} = 0$ *for* $i = 0, ..., n-2$.

Reitwiesner's algorithm [11] efficiently converted a binary number to signed-digit NAF form using a right-to-left method, see Algorithm 1.

Throughout this paper we will abbreviate -1 as $\overline{1}$. The following are some well-known results concerning the NAF form of an integer. Every positive integer k has a unique NAF representation [1,2]. The length of k written in NAF-form is at most one bit longer than the length of k. In [1], it was proven that the expected number of nonzero symbols in a NAF representation was 1/3 times the length of k. In general, one would expect a random k to have an equal number of 1's as 0's.

Joye and Yen proposed two left-to-right binary signed-digit recording algorithms in [5]. Based on Reitwiesner's algorithm and the left-to-right addition algorithm [9], Joye and Yen developed the first left-to-right recoding algorithm which preserves the NAF property. Joye and Yen also developed a more efficient left-to-right signed digit recoding algorithm whose output possessed the same nonzero density properties as a NAF representation.

One can make further reductions in the number of nonzero symbols by utilizing a w-ary NAF form or $wNAF$. Formally the definition of $wNAF$ is:

Definition 2 (wNAF). *[4] Let* $w > 2$ *and* k *a positive integer. We say that* $k = \sum_{i=0}^{n-1} k_i 2^i$ *is a wNAF representation of integer k provided (i)* $k_{n-1} \neq 0$*, (ii) for all nonzero* k_i*,* k_i *is an odd integer with* $|k_i| < 2^{w-1}$*, and (iii) at most one of any w consecutive digits* $k_i, k_{i+1}, ..., k_{i+w-1}$ *is nonzero.*

The term 2NAF is often used to describe a NAF representation. The ratio of nonzero symbols to symbols in a wNAF representation has been shown to be on average $1/(w+1)$ [2,10].

At Crypto 2004, Okeya, et.al. [10] created a sparse representation of a key, which can be constructed using a left-to-right pass through the key, utilizing little memory. The representation they introduced was the *Mutual Opposite Form (MOF).*

Definition 3 (MOF). *[10] A n-bit integer is represented using the mutual opposite form (MOF) provided:*

1. the signs of adjacent nonzero bits are opposite sign

2. the most significant bit is 1 *and the least significant bit is* -1*, unless all bits are zero*

Okeya et. al. [10] then generalized this to construct wMOF.

Definition 4 (wMOF). *[10] A signed digit representation satisfies wMOF provided*

1. The most significant nonzero digit is positive.

2. All but the least significant nonzero digit x are adjoint by $w-1$ *zeros as*

(i) if $2^{k-1} < |x| < 2^k$ *for* $2 \leq k \leq w-1$ *the pattern is* 0...0x0...0 *(k leading zeros and* $w-k-1$ *trailing zeros),*

(ii) if $|x| = 1$ *we have either the pattern* $x000...0$ *(*$w - 1$ *trailing zeros) and the next lower nonzero digit has opposite sign to* x *or the pattern* $0x0...0$ *(*$w - 2$ *trailing zeros) and the next lower digit has the same sign as* x*, and*

(iii) if x *is the least significant nonzero digit, it is possible that the number of right-hand adjacent zeros is smaller than the stated above. It is not possible that the last nonzero digit is a 1 following any nonzero digit.*

3. Each nonzero digit is odd and less that 2^{w-1} *in absolute value.*

In [10], the authors provided an algorithm that constructs the $wMOF$ representation in a left-to-right manner and showed that the nonzero density of a $wMOF$ representation was $1/(w+1)$, the same as $wNAF$.

The amount of precomputations represent both a computational resource requirement as well as a memory requirement. Because the inverse (negative) of the group operation in an Elliptic Curve Cryptosystem is trivial to compute, one does not need to pre-compute nor store negative multiples of the EC point, since the negative of an EC point can be computed requiring very little resources and can be computed as needed. In addition to the left-to-right algorithm which computes the $wMOF$ representation, Okeya et. al. also constructed a left-to-right algorithm that computes $wNAF$. Unfortunately their algorithm requires additional memory and hence it is not a memoryless left-to-right algorithm. In their left-to-right $wNAF$ algorithm, they require $O(\frac{n}{w})$ memory (see Table 5). In practice since w will be constant, this is $O(n)$ amount of memory. In our algorithm that computes $wNAF^*$, we require no additional memory. The nonzero density for $wNAF$ is the optimal value $1/(w+1)$. We will establish that the nonzero density of $wNAF^*$ is also $\frac{1}{w+1}$ and the precomputations needed by $wNAF^*$ is identical to $wNAF$. Reminder, other cryptosystems like RSA and discrete-log cryptosystems would realize improved efficiency when using a key in $wNAF$ form.

3 *NAF* and *NAF**

Algorithm 1 illustrates Reitweisner's canonical recoding of the key k, computed in a right to left manner. In this case $k = (d_{n-1}, d_{n-2}, \ldots, d_2, d_1, d_0) = \sum_{j=0}^{n-1} d_j \cdot 2^j$. This algorithm saves each carry c_j that is produced during each iteration, though one only needs to know the current carry-in, the current symbol d_i and the lookahead symbol d_{i+1}.

Algorithm 1 can be expressed in a table format, see Table 1. Here d_i denotes the current symbol, d_{i+1} denotes the lookahead symbol and C denotes the carry-in. The output symbol is δ_i. A second output symbol is C which is the carry-out and becomes the next carry-in symbol.

In addition to constructing a left-to-right wMOF representation, Okeya, et. al. constructed a left-to-right NAF algorithm (see Algorithm 5 in [10]) but this construction required external memory. They needed to track the number of consecutive ones that are visited. Because the number of consecutive ones could be on the order of n, at least $O(\log_2 n)$ memory is needed. By applying a innovated technique we can create a left-to-right binary encoding which we call NAF^*, which

Algorithm 1. Reitwiesner's canonical recoding [5, 2, 11]

1: **INPUT:** $d_n, ..., d_0$
2: **OUTPUT:** $\delta_{n+1}, ..., \delta_0$
3: $c_0 \leftarrow 0$
4: **for** $j = 0$ to $n + 1$ **do**
5: $\quad c_{j+1} \leftarrow \lfloor (d_j + d_{j+1} + c_j)/2 \rfloor$
6: $\quad \delta_j \leftarrow d_j + c_j - 2c_{j+1}$
7: **return** $\delta_{n+1}\delta_{n-1} \ldots \delta_0$

Table 1. Reitwiesner's algorithm to compute right-to-left NAF

Carry-in C	Current Symbol d_i	Lookahead Symbol d_{i+1}	Output result δ_i
0	0	x	$\delta_i = 0$, $i \leftarrow i + 1$, and set $C = 0$
0	1	0	$\delta_i = 1$, $i \leftarrow i + 1$, and set $C = 0$
0	1	1	$\delta_i = -1$, set $C = 1$ and $i \leftarrow i + 1$
1	0	0	$\delta_i = 1$, $i \leftarrow i + 1$, and set $C = 0$
1	0	1	$\delta_i = -1$ and set $C = 1$ and $i \leftarrow i + 1$
1	1	x	$\delta_i = 0$ and set $C = 1$, $i \leftarrow i + 1$

Here x represents a don't care, it can be either 0 or 1.

requires no additional storage. We amend the definition of NAF to form our definition of NAF^*. Our technique mimics the right-to-left NAF computation. For example when encountering a sequence 11 (in a right-to-left manner) the NAF computation makes the replacement of 11 with $0\overline{1}$ and a left carry of one. When we encounter 11 in a left-to-right manner, then we have already processed the bit prior to the sequence 11 (to the left). Since we have already processed the bits to the left, we can only create "carries to the right". Hence we replace 11 by 20 with a carry-to-the right of $\overline{2}$ (here $\overline{2}$ denotes negative 2). Of course the sequence 11 represents 3=2*1+1, whereas the sequence 20 with a carry-to-the right of $\overline{2}$ represents $2 * 2 + 1 * 0 + \frac{1}{2} * \overline{2} = 3$. It is of course straightforward to handle sequences 01, 10, and 00. What remains to be considered are the cases when we have carry-in's from the left, which can occur, such as in the case described above. The only two possible carry-in symbols will be 0 and $-2 = \overline{2}$. Recall, we are examining current symbol d_i while the lookahead symbol is d_{i-1}. The possible values that these two consecutive symbols can represent are determined by $2 * d_i + d_{i-1}$ (integers between 0 and 3). If we have a carry-in (from the left) of -2, then this would be placed in the d_i place. This would be equivalent to a value of $-2 * 2 = -4$, thus we must add -4 to the possible values, resulting in a sequence of integers between -4 to -1. In NAF^*, we use an extended set of symbols that consist of 0, ± 1, and ± 2. Table 2 illustrates how we handle the cases of a left carry-in of $-2 = \overline{2}$. The definition of NAF^* is identical to that of NAF except the symbols can consist of 0, ± 1, and ± 2 (whereas NAF restricts them to 0 and ± 1). An integer $k = \sum_{i=0}^{n-1} k_i 2^i$ is expressed in NAF^* form provided $k_i \in \{0, \pm 1, \pm 2\}$ and $k_i \cdot k_{i-1} = 0$ for all $i = 1, \ldots, n - 1$.

The computation of the NAF^* recoded key can be generated using Table 2, where the initial value for C is 0 and $i = n - 1$. Here the key $k = d_{n-1} \dots d_1 d_0$. Further, for obvious reasons we use $d_{-1} = d_{-2} = 0$.

Table 2. Left-to-right NAF^*

Carry-in C	Current Symbol d_i	LookaHead Symbol d_{i-1}	result δ_i
0	0	x	$\delta_i = 0$, $i \leftarrow i - 1$, and set $C = 0$
0	1	0	$\delta_i = 1$, $i \leftarrow i - 1$, and set $C = 0$
0	1	1	$\delta_i = 2$, $i \leftarrow i - 1$, and set $C = -2$
-2	0	0	$\delta_i = -2$, $i \leftarrow i - 1$, and set $C = 0$
-2	0	1	$\delta_i = -1$, $i \leftarrow i - 1$ and set $C = -2$
-2	1	x	$\delta_i = 0$, $i \leftarrow i - 1$ and set $C = -2$

Here x represents a *don't care*, it can be either 0 or 1.

We now describe NAF^* in algorithm form. That is the following algorithm is an implementation of Table 2.

Algorithm 2. NAF^*

1: **INPUT:** $d_n, ..., d_0$
2: **OUTPUT:** $\delta_n, ..., \delta_0 \delta - 1$
3: $c_n \leftarrow 0$
4: **for** $j = n$ downto 0 **do**
5: $\quad c_{j-1} \leftarrow -2 \cdot (\lfloor (d_j + d_{j-1} + \frac{|c_j|}{2})/2 \rfloor)$
6: $\quad \delta_j \leftarrow d_j + c_j - \frac{1}{2} \cdot c_{j-1}$
7: $\delta_{-1} \leftarrow c_{-1}$
8: **return** $\delta_n \delta_{n-1} \dots \delta_0 \delta_{-1}$

In Algorithm 2, because we execute left-to-right, carries are either -2 or 0. This explains the use of the absolute value and the fraction $\frac{1}{2}$ in line 5 and the use of the fraction $\frac{1}{2}$ in line 6.

Example 1. The following example illustrates both the NAF recoding and the NAF^* recoding for a key k.

Key k	$1\,0\,0\,1\,1\,0\,1\,1\,0\,1\,0\,1\,1\,1\,1$	
NAF recoding of k	$1\,0\,1\,0\,0\,\bar{1}\,0\,0\,\bar{1}\,0\,\bar{1}\,0\,0\,0\,\bar{1}$	
NAF^* recoding of k	$1\,0\,0\,2\,0\,\bar{1}\,0\,0\,\bar{1}\,0\,\bar{1}\,0\,0\,0\,0$	$\bar{2}$

The reason for the $\bar{2}$ symbol in the NAF^* recoding (see the far right symbol), is that this occurs in the case where $i = -1$, i.e. $\delta_{-1} = \bar{2}$. That is, the length of the NAF^* recoded key is one symbol longer than that of the key $k = d_{n-1}, \ldots, d_1, d_0$. Much like NAF, the recoding NAF^* may have a length of one symbol longer than the length of the key but the extra symbol occurs in the far right place (where $i = -1$).

Theorem 1. *The NAF^* algorithm, as described by Table 2, when applied to k will produce a recoded sequence of symbols satisfying NAF^*, such that the sequence is equivalent to k*

Proof. To establish this theorem we are left to show that after each iteration i of applying Table 2, we have a sequence of symbols which is equivalent to the key k. We will assume that prior to the i^{th} iteration the result computes the key. Prior to the i^{th} iteration we have completed the $i + 1^{st}$ iteration. Thus we have computed $\delta_{n-1}, \ldots, \delta_{i+1}$, the carry C, which we will denote as C_{IN} (since this is the carry-in to the i^{th} iteration), as well as the symbols $d_i, \ldots, d_0$ which have not been processed. Thus

$$k = C_{IN} \cdot 2^i + \sum_{j=i+1}^{n-1} \delta_j \cdot 2^j + \sum_{j=0}^{i} d_j \cdot 2^j. \tag{1}$$

Now after the i^{th} iteration we have processed d_i based on C_{IN} and d_{i-1}. The result is that we have now computed δ_i and C_{OUT} (the C value which is the carry-out after the i^{th} iteration) based on Table 2. Consider $C_{OUT} \cdot 2^{i-1} + \delta_i \cdot 2^i + \sum_{j=i+1}^{n-1} \delta_j \cdot 2^j + \sum_{j=0}^{i-1} d_j \cdot 2^j$. We need to show that this value is equivalent to equation (1). By cancelling the common terms, we are left to show $C_{IN} \cdot 2 + d_i \cdot 2 = C_{OUT} + \delta_i \cdot 2$. By examining each row of Table 2, we see that this equation is valid for each possible input and so this establishes the theorem.

•

Observe that there is a natural mapping between the execution of the NAF algorithm and the execution of the NAF^* algorithm. This is clearly demonstrated by a comparison of Table 1 and Table 2. The mapping is implied by the correspondence between rows of the tables. This is a one-to-one mapping such that NAF outputs a nonzero symbol iff NAF^* outputs a nonzero symbol and NAF outputs a carry term that is nonzero iff NAF^* outputs a carry term that is nonzero. By [1], the expected hamming weight of a key, of length n, recoded in NAF is $\frac{n}{3}$. This mapping, together with the result by [1], establishes the following.

Theorem 2. *The expected hamming weight of a NAF^* recoded key of length n is $\frac{n}{3}$.*

Proof. This theorem follows from the result that the expected hamming weight of a key recoded in NAF form is $\frac{n}{3}$, and the fact that there exists a natural one-to-one mapping between NAF recoding and a NAF^* recoding (as described above). Since the input distribution, as well as the distribution of zero vs. nonzero

carry symbols are identical, we see that under this mapping and due to the result by [1], the expected hamming weight of a NAF^* recoded key is $\frac{n}{3}$.

•

Consequently NAF^* is as "efficient" as NAF except NAF^* allows the use of the symbols 2 and -2. Thus when applying NAF^* to perform an ECC scalar multiple, it appears that a precomputation of $2P$ will need to be performed, however $2P$ always must be computed. Consequently, no additional precomputation is needed. The ECC scalar multiple algorithm is provided in the Appendix, see Algorithm 4.

4 *wNAF* and *wNAF**

The following is a right-to-left calculation for wNAF as provided in [12,10]. Note that the term "mods", as used in Algorithm 2, is defined as: x mods y returns a value j such that $-y/2 \leq j < y/2$ and $j \mod y = x \mod y$. For example 5 mods $16 = 5$ and 11 mods $16 = -5$.

Algorithm 3. $wNAF$ [12, 10]

Input: width w and n-bit integer d
Output: wNAF $\delta_n\delta_{n-1}\ldots\delta_0$ of d
$i \leftarrow 0$
while $d \geq 1$ **do**
  if d is even **then**
    $\delta_i \leftarrow 0$
  else
    $\delta_i \leftarrow d$ mods 2^w
    $d \leftarrow d - \delta_i$
  $d \leftarrow d/2$; $i \leftarrow i + 1$
return $\delta_n\delta_{n-1}\ldots\delta_0$

Algorithm 3, which computes the $wNAF$ recoding, can be rewritten in table form, see Table 3. The correctness of the table can be established by considering the various cases and applying the above algorithm. We omit the proof of correctness for the table.

4.1 *wNAF**

We modify the NAF^* construction and apply techniques that mimic a right-to-left wNAF construction to construct a $wNAF^*$ definition and recoding algorithm. Again as our algorithm parses the key in a left-to-right fashion we will restrict ourselves to a carry-in of 0 or -2. Observe that in the definition of $wNAF$, a nonzero integer k_i is such that it is a most $w-1$ bits, where the low-bit is a one (i.e. it is an odd integer) and the high bit is zero (thus the absolute

Table 3. Right-to-left $wNAF$

Carry-in C	Input $R = (d_{i+w-1}, \ldots, d_{i+1}, d_i)$	Output δ_i result
0	$d_i = 0$	$\delta_i = 0$, $i \leftarrow i+1$, and $C = 0$
0	$d_i = 1$ and $d_{i+w-1} = 0$	$(\delta_{i+w-1}, \ldots, \delta_{i+1}, \delta_i) = (0, 0, \ldots, 0, R)$ $i \leftarrow i+w$, and $C = 0$
0	$d_i = 1$ and $d_{i+w-1} = 1$ $\exists j\, d_j = 0$	$(\delta_{i+w-1}, \ldots, \delta_{i+1}, \delta_i) = (0, 0, \ldots, 0, -(2^w - R)$ $i \leftarrow i+w$, $C = 1$
0	$d_i = 1 = d_{i+1} = \ldots = d_{i+w-1} = 1$	$(\delta_{i+w-1}, \ldots, \delta_{i+1}, \delta_i) = (0, 0, \ldots, 0, -1)$ $i \leftarrow i+w$, $C = 1$
1	$d_i = 1$	$\delta_i = 0$ $i \leftarrow i+1$, $C = 1$
1	$d_i = 0$ and $d_{i+w-1} = 0$	$(\delta_{i+w-1}, \ldots, \delta_{i+1}, \delta_i) = (0, 0, \ldots, 0, R+1)$ $i \leftarrow i+w$, and $C = 0$
1	$d_i = 0$ and $d_{i+w-1} = 1$	$(\delta_{i+w-1}, \ldots, \delta_{i+1}, \delta_i) = (0, 0, \ldots, 0, -(2^w - (R+1))$ $i \leftarrow i+w$, $C = 1$

value of k_i is less than 2^{w-1}). We need to provide a recoding definition for our left-to-right construction that mimics these properties. In our definition, if the symbol is nonzero the left-most bit must be nonzero and the rightmost w^{th} bit will be zero. We now formalize the definition of $wNAF^*$

Definition 5. wNAF* *Let $w > 2$ and k a positive integer. Then we say that $\sum_{i=-1}^{n-1} a_i 2^i$ is a $wNAF^*$ representation of k provided $k = \sum_{i=-1}^{n-1} a_i 2^i$ and*

(i) a_i is a rational number for all i,
(ii) at most one of any w consecutive symbols $a_i, a_{i+1} \ldots, a_{i+w-1}$ is nonzero,
(iii) for all i, if a_i is nonzero then $1 \leq |a_i| \leq 2$,
(iv) for all i, the product $a_i \cdot 2^{w-2} \in \mathbb{Z}$, and
(v) for all i, with $-1 \leq i \leq w-1$ we have $a_i \cdot 2^i \in \mathbb{Z}$.

First observe that if $a_{-1} \neq 0$ then $1 \leq |a_{-1}| \leq 2$ and $a_{-1} \cdot 2^{-1} \in \mathbb{Z}$. Since $1 \cdot 2^{-1} \leq |a_{-1}| \cdot 2^{-1} \leq 2 \cdot 2^{-1} = 1$ we see that $|a_{-1}| = 2$. So if $a_{-1} \neq 0$ then either $a_{-1} = -2$ or $a_{-1} = 2$.

Let us now consider a_i where $i \geq w-2$. Suppose a_i is nonzero, without loss of generality assume that $a_i > 0$. Thus $1 \leq a_i \leq 2$. Then $a_i = 1 + \epsilon$ where $\epsilon \in [0, 1]$. Therefore we can express a_i as

$$a_i = 1 + \sum_{j=1}^{\infty} a_{i,j} \frac{1}{2^j} \text{ where } a_{i,j} \in \{0, 1\} \ . \tag{2}$$

Now $a_i \cdot 2^{w-2} \in \mathbb{Z}$. Consequently,

$$a_i \cdot 2^{w-2} = 1 \cdot 2^{w-2} + \sum_{j=1}^{w-2} a_{i,j} \frac{2^{w-2}}{2^j} + \sum_{j=w-1}^{\infty} a_{i,j} \frac{2^{w-2}}{2^j}$$

where $a_{i,j} \in \{0, 1\}$. Observe that $1 \cdot 2^{w-2} + \sum_{j=1}^{w-2} a_{i,j} \frac{2^{w-2}}{2^j} \in \mathbb{Z}$. Thus as $a_i \cdot 2^{w-2} \in \mathbb{Z}$ we have $\sum_{j=w-1}^{\infty} a_{i,j} \frac{2^{w-2}}{2^j}$ is an integer. Now $0 \leq \sum_{j=w-1}^{\infty} a_{i,j} \frac{2^{w-2}}{2^j} \leq 1$.

Therefore since $\sum_{j=w-1}^{\infty} a_{i,j}\frac{2^{w-2}}{2^j}$ is an integer, there are two possible cases, it is either 0 or 1.

Case 1. Suppose $\sum_{j=w-1}^{\infty} a_{i,j}\frac{2^{w-2}}{2^j} = 0$. Then $a_{i,j} = 0$ for all $j = w-1, \ldots$. So in this case $a_i = 1 + \sum_{j=1}^{w-2} a_{i,j}\frac{1}{2^j}$.

Case 2. Suppose $\sum_{j=w-1}^{\infty} a_{i,j}\frac{2^{w-2}}{2^j} = 1$. Then $a_{i,j} = 1$ for all $j = w-1, \ldots$ and $\sum_{j=w-1}^{\infty} a_{i,j}\frac{2^{w-2}}{2^j} = \frac{1}{2} + \frac{1}{4} + \cdots = 1$. Thus in this case $\sum_{j=w-1}^{\infty} a_{i,j}\frac{1}{2^j} = \frac{1}{2^{w-2}}$. Recall $a_i = 1 + \sum_{j=1}^{w-2} a_{i,j}\frac{1}{2^j} + \sum_{j=w-1}^{\infty} a_{i,j}\frac{1}{2^j}$, so $a_i = 1 + \sum_{j=1}^{w-2} a_{i,j}\frac{1}{2^j} + \frac{1}{2^{w-2}}$.

We now examine this latter case, case 2, more closely. Suppose $a_i = 1 + \sum_{j=1}^{w-2} a_{i,j}\frac{1}{2^j} + \frac{1}{2^{w-2}}$. Observe that if $a_{i,j} = 1$ for all $j = 1, \ldots, w-2$, then $a_i = 2$. Suppose there exists a j with $1 \le j \le w-2$ such that $a_{i,j} = 0$. Since $a_i = 1 + \sum_{j=1}^{w-2} a_{i,j}\frac{1}{2^j} + \frac{1}{2^{w-2}}$, and there exists some $a_{i,j}$ equal to zero, we can simplify this as $a_i = 1 + \sum_{j=1}^{w-2} b_{i,j}\frac{1}{2^j}$ where $b_{i,j} \in \{0,1\}$. Thus there are three possible representations for a_i, either it is 0, ± 2 or $\pm(1 + \sum_{j=1}^{w-2} b_{i,j}\frac{1}{2^j})$ where $b_{i,j} \in \{0,1\}$.

For all cases, when a_i is expressed using the representation given in (2), we see that $a_{i,j} = 0$ for $j = w-1, \ldots$. Further, if $a_i \neq 0$, nd $a_i \neq \pm 2$, then we can express a_i as $|a_i| = (1, a_{i1}, \ldots a_{i,w-2}, 0)$. Consequently either $a_i = 0$, $a_i = \pm 2$ or it can be interpreted as a $w-1$ bit rational number (either positive or negative). The interpretation of a_i as a $w-1$ bit rational number comes from $|a_i| = (1, a_{i,1}, \ldots, a_{i,w-2}, 0) = 1 + \frac{a_{i,1}}{2} + \frac{a_{i,2}}{2^2} + \cdots + \frac{a_{i,w-2}}{2^{w-2}}$ where $a_{i,j} \in \{0,1\}$.

Obviously the definition of $wNAF^*$ mimics the definition of wNAF. That is, one can view a_i as a w-bit symbol such that if a_i is non zero, and if $a_i \neq \pm 2$ then the high bit $a_{i,w-1}$ is one (this is analogous to the case that in wNAF the k_i is odd) and the far right w^{th} bit is zero. This implies that $|a_i|$ is between 1 and 2, such that $|a_i| = 1 + \frac{a_{i,1}}{2} + \frac{a_{i,2}}{2^2} + \cdots + \frac{a_{i,w-2}}{2^{w-2}}$.

Let R be a $w-1$-bit (or less) nonzero positive integer. $R = r_j, \ldots, r_1$ where $r_j = 1$ and $1 \le j \le w-1$. Then we define R_{frac} as

$$R_{frac} = (1, r_{j-1} \ldots, r_1)_{frac} = \sum_{i=0}^{j-1} r_{j-i} \cdot \frac{1}{2^i} = 1 + r_{j-1} \cdot \frac{1}{2} + \cdots + r_1 \cdot \frac{1}{2^{j-1}}. \quad (3)$$

4.2 *wNAF** Recoding Algorithm

When computing the $wNAF$ representation of a key, we may have carry-in's from the right that are 0 or 1. In the left-to-right calculation of $wNAF^*$ we could have carry-in's (from the left) of either -2 or 0. Table 4 describes how to construct the $wNAF^*$ recoding of key k. We initialize $C = 0$ and $i = n-1$ and process and compute each output symbol δ_i in a left-to-right manner. We will continue until $i < -1$. It is possible that the last symbol generated δ_{-1} may be nonzero.

Observe that termination is guaranteed, because $k = \sum_{j=0}^{n-1} d_j 2^j$, so we interpret input values $d_{-1} = d_{-2} = 0$ (since they do not have any assigned values, they are correctly treated as zero). Also observe that it is possible that we have an output result of δ_{-1} being nonzero (this is illustrated in the Example 2).

Table 4. Left-to-right $wNAF^*$

Carry-in C	Input $R = (d_i, d_{i-1}, \ldots, d_{i-w+2}, d_{i-w+1})$	Output result
0	$d_i = 0$	$\delta_i = 0$, $i \leftarrow i-1$, and $C = 0$
0	$d_i = 1$ and $d_{i-w+1} = 0$	$(\delta_i, \delta_{i-1}, \ldots, \delta_{i-w+2}, \delta_{i-w+1}) = (R_{frac}, 0, 0, \ldots, 0)$, $i \leftarrow i-w$ and $C = 0$
0	$d_i = 1$ and $d_{i-w+1} = 1$ $\exists j\, d_j = 0$	$(\delta_i, \delta_{i-1}, \ldots, \delta_{i-w+2}, \delta_{i-w+1}) = (R+1)_{frac}, 0, 0, \ldots, 0)$, $i \leftarrow i-w$, $C = -2$
0	$d_i = 1 = \ldots = d_{i-w+1} = 1$	$(\delta_i, \delta_{i-1}, \ldots, \delta_{i-w+2}, \delta_{i-w+1}) = (2, 0, \ldots, 0)$, $i \leftarrow i-w$, $C = -2$
-2	$d_i = 1$	$\delta_i = 0$, $i \leftarrow i-1$, $C = -2$
-2	$d_i = 0$ and $d_{i-w+1} = 0$	$(\delta_i, \delta_{i-1}, \ldots, \delta_{i-w+2}, \delta_{i-w+1}) = (-(2^w - R)_{frac}, 0, 0, \ldots, 0)$, $i \leftarrow i-w$ and $C = 0$
-2	$d_i = 0$ and $d_{i-w+1} = 1$	$(\delta_i, \delta_{i-1}, \ldots, \delta_{i-w+2}, \delta_{i-w+1}) = (-(2^w - (R+1))_{frac}, , 0, 0, \ldots, 0)$, $i \leftarrow i-w$, $C = -2$

However if $\delta_{-1} \neq 0$ then δ_{-1} is either 2 or -2. Further $\delta_{-2} = \delta_{-3} = ... = 0$. In Theorem 3 we show that the $wNAF^*$ recoding is correct.

Example 2. We illustrate the different recodings systems for a fixed key k. For a key $k = (1,1,1,0,1,0,0,1,1,0,0,1,0,0,0,1,0,1,0,1,1,1,0,1,0,1,0,1,0,1,1,1) = 3,918,617,943$. We express k utilizing the following representations: NAF, NAF^*, 3NAF, $3NAF^*$, 4NAF, $4NAF^*$, and we provide a generalization of $4NAF^*$ which uses integer symbols.

key$k$		1 1 1 0 1 0 0 1 1 0 0 1 0 0 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1	
NAF form of k	1	0 0 $\bar{1}$ 0 1 0 1 0 $\bar{1}$ 0 0 1 0 0 1 0 $\bar{1}$ 0 $\bar{1}$ 0 0 0 $\bar{1}$ 0 $\bar{1}$ 0 $\bar{1}$ 0 $\bar{1}$ 0 0 $\bar{1}$	
NAF^* form of k		2 0 0 $\bar{1}$ 0 $\bar{2}$ 0 2 0 $\bar{2}$ 0 1 0 0 0 1 0 1 0 2 0 0 $\bar{1}$ 0 $\bar{1}$ 0 $\bar{1}$ 0 $\bar{1}$ 0 0 0	$\bar{2}$
3NAF form of k	1	0 0 0 0 $\bar{3}$ 0 0 0 3 0 0 1 0 0 0 1 0 0 3 0 0 0 $\bar{1}$ 0 0 $\bar{3}$ 0 0 3 0 0 $\bar{1}$	
$3NAF^*$ form of k		2 0 0 $\frac{\bar{3}}{2}$ 0 0 0 $\frac{3}{2}$ 0 0 0 1 0 0 0 $\frac{3}{2}$ 0 0 $\bar{1}$ 0 0 0 $\frac{\bar{3}}{2}$ 0 0 $\frac{3}{2}$ 0 0 $\bar{1}$ 0 0 0	$\bar{2}$
$4NAF$ form of k	0	0 0 7 0 0 0 5 0 0 0 $\bar{3}$ 0 0 0 $\bar{7}$ 0 0 0 $\bar{5}$ 0 0 0 0 $\bar{3}$ 0 0 0 5 0 0 0 7	
$4NAF^*$ form of k		$\frac{7}{4}$ 0 0 0 $\frac{5}{4}$ 0 0 0 0 $\frac{\bar{7}}{4}$ 0 0 0 0 0 $\frac{5}{4}$ 0 0 0 $\frac{7}{4}$ 0 0 0 $\frac{5}{4}$ 0 0 0 $\frac{3}{2}$ 0 0 0 0	$\bar{2}$
Modified version [a] $4NAF^*$ form of k		0 0 7 0 0 0 5 0 0 0 0 $\bar{7}$ 0 0 0 0 0 5 0 0 0 7 0 0 0 5 0 0 3 0 0 0	$\bar{2}$

[a] In this representation we express entries as integers rather than rational numbers. Observe that we do not necessarily satisfy wNAF properties.

Theorem 3. *Let* $\Delta = \sum_{j=-1}^{n-1} \delta_j 2^j$ *be the* $wNAF^*$ *recoding of key* k *then* $\Delta = k$.

Proof. Our goal is to show that after each iteration we replace key bits d_j with symbols from the $wNAF^*$ recoding δ_j such that result after replacement still equals k. Assume that prior to the i^{th} iteration we have $C_{IN} \cdot 2^i + \sum_{j=0}^{i} d_j \cdot 2^j + \sum_{j=i+1}^{n-1} \delta_J \cdot 2^j = k$.
We now consider each of the seven possible cases, a case for each of the seven rows of Table 4.

First case, suppose $C_{IN} = 0$ and $d_i = 0$, then trivially after executing the i^{th} iteration the resulting replacement still equals k.

Now suppose $C_{IN} = 0$ and $d_i = 1$ and $d_{i-w+1} = 0$. Then $R = d_i \dots d_{i-w+1} = 1xx \cdots x0$. We replace $\sum_{j=i-w+1}^{i} d_j \cdot 2^j$ by $R_{frac} \cdot 2^i$ which are equal. Again trivially after executing the i^{th} iteration the result still equals k.

Now suppose $C_{IN} = 0$ and $d_i = 1$, $d_{i-w+1} = 1$ and there exists j with $i - w + 1 < j < i$ such that $d_j = 0$. Then $R = d_i \dots d_{i-w+1} = 1x \cdots x01 \cdots 1$. Thus $R + 1 = 1x \cdots x10 \cdots 0$. In this case $C_{OUT} = -2$. Now $(R+1)_{frac} \cdot 2^i + C_{OUT} \cdot 2^{i-w} = R_{frac} \cdot 2^i + 1 \cdot \frac{2^i}{2^{-w+1}} + -2 \cdot 2^{i-w} = R_{frac} \cdot 2^i = \sum_{j=i-w+1}^{i} d_j \cdot 2^j$. So the use of $(R+1)_{frac}$ with C_{OUT} of -2 will provide a valid replacement.

Now suppose $C_{IN} = 0$ and $d_i = 1 = \dots = d_{i-w+1} = 1$. Then $R = d_i \dots d_{i-w+1} = 11 \cdots 111 \cdots 1$. Thus $R + 1 = 2^{w+1}$. Note $(R+1)_{frac} = 2$. In this case $C_{OUT} = -2$. With an argument similar to the above case, the use of $(R+1)_{frac}$ with C_{OUT} of -2 will provide a valid replacement.

Now suppose $C_{IN} = -2$ and $d_i = 1$. Then $-2 + 1 = -1$ so the use of $\delta_i = 0$ and $C_{OUT} = -2$ will provide a valid replacement.

Now suppose $C_{IN} = -2$ and $d_i = 0$ and $d_{i-w+1} = 0$. Then $R = d_i \dots d_{i-w+1} = 1x \cdots x0$. Thus $-(2^w - R)$ represents the sum of C_{IN} with R. The use of $-(2^w - R)_{frac}$ with C_{OUT} of 0 will provide a valid replacement.

Now suppose $C_{IN} = -2$ and $d_i = 0$ and $d_{i-w+1} = 1$. Then $R = d_i \dots d_{i-w+1} = 0x \cdots x1$. Thus $R+1 = x'x' \cdots x'0$. Consequently $-(2^w - (R+1))$ represents the sum of C_{IN} with R producing a $C_{OUT} = -2$. The use of $-(2^w - (R+1))_{frac}$ with C_{OUT} of -2 will provide a valid replacement.

Thus all seven cases produce a valid replacement.

•

Theorem 3 established that the $wNAF^*$ representation is correct (equivalent to k). We now discuss the "efficiency" of the $wNAF^*$ recoding.

Theorem 4. *The average nonzero density of a* $wNAF^*$ *recoding is* $\frac{1}{w+1}$.

Proof. The exists a natural mapping between $wNAF$ and $wNAF^*$. This can be seen by examining each row of Table 3 with the corresponding row of Table 4. Under this mapping $wNAF$ produces a nonzero output iff $wNAF^*$ produces a nonzero output and $wNAF$ produces a nonzero carry-out iff $wNAF^*$ produces a nonzero carry out. Since the nonzero density of $wNAF$ is $\frac{1}{w+1}$ (see [10]), then the nonzero density of $wNAF^*$ must be $\frac{1}{w+1}$.

•

Remark. In the appendix we illustrate how to compute the ECC scalar multiple (see Algorithm 5 in Appendix). In our algorithm, precomputations of the form $R_{frac} \cdot P$ will need to take place. This can be done by using a halving a point technique. For example, over binary elliptic curves the halving point algorithm [8] is very efficient, see [6, 4]. However we could avoid the use of fractional symbols and use integers (see Example 2, row labeled with [a]). By doing so we benefit from the fact we have odd integers less than 2^{w-1} (in absolute value). Further this generalization of $wNAF^*$ will have the same nonzero density as $wNAF^*$. Thus this generalization has the same efficiency as $wNAF^*$ (in the appendix we construct the scalar multiple algorithm using this technique, see Algorithm 5). In general, $wNAF^*$ would require one more precomputation than $wNAF$ since $2P$ would need to be computed and stored, but again $2P$ would always have to be computed. Thus no extra precomputations are needed.

Table 5 illustrates the resource requirement of various signed digit recoding representations. This table was provided in [10], we have added the additional entry for the resource requirements of our algorithm $wNAF^*$. Only wMOF, left-to-right wNAF [b] and $wNAF^*$ are left-to-right w-ary recoding. However left-to-right wNAF [b], has additional memory requirements. Most importantly $wNAF^*$ is much more straightforward to implement than wMOF.

Table 5. Table of various signed digit representations as provided in [10]

Scheme	Precomputations	density	additional memory
wNAF [12, 2]	2^{w-2}	$\frac{1}{w+1}$	$O(n)$
Koyama [7]	$2^{w-1}-1$	$\frac{1}{w+3/2}$	$O(n)$
NAF +SW [3]	$1/3(2^w+(-1)^{w+1})$	$\frac{1}{w+4/3-\nu(w)}$	$O(n)$
wMOF [10]	2^{w-2}	$\frac{1}{w+1}$	$O(w)$
left-to-right wNAF [b] [10]	2^{w-2}	$\frac{1}{w+1}$	$O(\frac{\log(w)}{w} \cdot n)$ when $w > 2$ $O(\log n)$ when w=2
$wNAF^*$	2^{w-2}	$\frac{1}{w+1}$	$O(w)$

Here $\nu(w) = \frac{(-1)^w}{3 \cdot 2^{w-2}}$.

5 Conclusion

We have examined an open problem concerning the creation of a left-to-right on-the-fly implementation of $wNAF$ by defining $wNAF^*$ and creating the left-to-right $wNAF^*$ recoding algorithm. Though the $wNAF^*$ recoding definition is not the same as the $wNAF$ definition, it closely mimics the definition. Further the left-to-right recoding algorithm for $wNAF^*$ closely mimics the right-to-left recoding algorithm of $wNAF$. Our work has provided the relationship between a right-to-left $wNAF$ calculation and the left-to-right w-ary non adjacent form calculation.

When applying the $wNAF^*$ representation to the key k in the computation of the ECC scalar multiple, it at first appears to require the use of halving of a point computation [8], which is very efficient. However it is possible to modify

our algorithm to create integer symbols, yet still reaping the same efficiency as $wNAF^*$. Our algorithm provides the optimal nonzero density (which implies less additions need to take place) while also maintaining the optimal minimal amount of precomputations (as illustrated in Table 5).

References

1. Arno, S., Wheeler, F.: Signed Digit Representations of Minimal Hamming Weight. IEEE Transactions on Computers 42(8), 1007–1009 (1993)
2. Blake, I., Seroussi, G., Smart, N.: Elliptic Curve Cryptography. Cambridge University Press, Cambridge (1999)
3. De Win, E., Mister, S., Preneel, B., Wiener, M.: On the performance of signature schemes based on elliptic curves. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 252–266. Springer, Heidelberg (1998)
4. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, New York (2004)
5. Joye, M., Yen, S.-M.: Optimal left-to-right binary signed-digit recoding. IEEE Transactions on Computers 49(7), 740–748 (2000)
6. King, B., Rubin, B.: Improvements to the halving a point algorithm. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 262–276. Springer, Heidelberg (2004)
7. Koyama, K., Tsuruoka, Y.: Speeding up Elliptic Cryptosystems by Using a Signed Binary Window Method. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 345–357. Springer, Heidelberg (1993)
8. Knudsen, E.W.: Elliptic Scalar Multiplication Using Point Halving. In: Lam, K.-Y., Okamoto, E., Xing, C. (eds.) ASIACRYPT 1999. LNCS, vol. 1716, pp. 135–149. Springer, Heidelberg (1999)
9. Knuth, D.: Art of Programming, 3rd edn., NY (1998)
10. Okeya, K., Schmidt-Samoa, K., Spahn, C., Takagi, T.: Signed binary representations revisited. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, Springer, Heidelberg (2004)
11. Reitwiesner, G.W.: Binary arithmetic. Advances in Computers 1, 231–308 (1960)
12. Solinas, J.: Efficient Arithmetic on Koblitz Curves. Des. Codes Cryptography 19(2/3), 195–249 (2000)

Appendix

We have modified the algorithm from Table 2 slightly to provide a more straightforward implementation of the algorithm, collapsing more than one case (i.e. more than one row of Table 2) into one logical statement.

We have modified the algorithm from Table 4 slightly to provide a more straightforward implementation of the algorithm, collapsing more than one case (i.e. more than one row of Table 2) into one logical statement, this is provided in Algorithm 5. In Algorithm 6, we have modified Algorithm 5, so that the symbols produced will be integers. We are using a technique illustrated in Example 2, row denoted by ^{a} and then incorporate this technique to compute the scalar multiple of an EC point.

Algorithm 4. Using NAF^* to compute the scalar multiple in a left-to-right fashion

INPUT: $k = d_{n-1}, ..., d_0$, and base point P belonging to elliptic curve
OUTPUT: kP
Compute $2P$ and store it
$c \leftarrow 0$
$Q \leftarrow \mathcal{O}$ { $\mathcal{O}$ is the point of infinity}
$j \leftarrow n - 1$
while $j \geq 0$ **do**
  if $2 * d_j + c = 0$ **then**
    $Q \leftarrow 2Q$ {the next carry-in is the same as the previous carry-in}
  else
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + (c + d_j + d_{j-1})P$
    if exactly one of c, d_j, d_{j-1} is nonzero **then**
      $c = 0$
    else
      $c = -2$
  $j \leftarrow j - 1$
{**Comment** we now handle the case if δ_{-1} is nonzero}
if $c \neq 0$ **then**
  $Q \leftarrow Q + (c/2)P$
return Q

Algorithm 5. Computing the EC scalar multiple using the $wNAF^*$ algorithm

INPUT: $k = d_{n-1}, ..., d_0$, and base point P belonging to elliptic curve
OUTPUT: kP
FORALL nonzero w bit integers with a leading one and trailing zero $r = (1, x_1, \ldots, x_{w-2}, 0)$
    Compute $r_{frac}P$ and store in table
Compute $2P$ and store in table
$c \leftarrow 0$
$Q \leftarrow \mathcal{O}$ { $\mathcal{O}$ is the point of infinity}
$j \leftarrow n - 1$
$\rho \leftarrow -1$
while $j \geq 0$ **do**
    $r = (d_j, d_{j-1}, \ldots, d_{j-w+1})$
    if $2 * d_j + c = 0$ **then**
        $Q \leftarrow 2Q$, $j \leftarrow j - 1$
    else
        if $c = 0$ **then**
            if $d_j \neq d_{j-w+1}$ **then**
                $t \leftarrow r_{frac}$
            else
                $t \leftarrow (r + 1)_{frac}$
                $c = -2$
        else
            if $d_j = d_{j-w+1}$ **then**
                $t \leftarrow -(2^w - r)_{frac}$
                $c = 0$
            else
                $t \leftarrow -(2^w - (r + 1))_{frac}$
        if $j \geq w$ **then**
            $Q \leftarrow 2^w Q$
            $Q \leftarrow Q + tP$ {use precomputed table to find tP}
            $j \leftarrow j - w$
        else
            $\rho \leftarrow j$
            $j \leftarrow j - w$
if $0 \leq \rho < w$ **then**
    $Q \leftarrow 2^\rho Q$
    $Q \leftarrow Q + tP$ {use precomputed table to find tP}
else
    if $j = -1$ and $c = -2$ **then**
        $Q \leftarrow Q + -P$
return Q

Algorithm 6. Modified version of scalar multiple calculation using $wNAF^*$ and integer constants

INPUT: $k = d_{n-1}, ..., d_0$, and base point P belonging to elliptic curve
OUTPUT: kP
FORALL nonzero $w-1$ (or less) bit odd integers r
    Compute $r \cdot P$ and store in table
Compute $2P$ and store in table {This was actually done some time in line 2}
$c \leftarrow 0$
$Q \leftarrow \mathcal{O}$ { $\mathcal{O}$ is the point of infinity}
$j \leftarrow n-1$
$\rho \leftarrow -1$
$\tau \leftarrow 0$
while $j \geq 0$ **do**
    $Q \leftarrow 2^{\tau}Q$
    $r = (d_j, d_{j-1}, \ldots, d_{j-w+1})$
    if $2 * d_j + c = 0$ **then**
        $Q \leftarrow 2Q$, $j \leftarrow j-1$, $\tau \leftarrow 0$
    else
        if $c = 0$ **then**
            if $d_j \neq d_{j-w+1}$ **then**
                Let τ denote the least significant nonzero bit of r (label the LSB of a w bit integer as the 0 bit)
                $t \leftarrow (r/2^{\tau})$
            else
                Let τ denote the least significant nonzero bit of $r+1$ (label the LSB of a w bit integer as the 0 bit)
                $t \leftarrow (r+1)/2^{\tau}$
                $c = -2$
        else
            if $d_j = d_{j-w+1}$ **then**
                Let τ denote the least significant nonzero bit of $2^w - r$ (label the LSB of a w bit integer as the 0 bit)
                $t \leftarrow -(2^w - r)/2^{\tau}$
                $c = 0$
            else
                Let τ denote the least significant nonzero bit of $2^w - (r+1)$ (label the LSB of a w bit integer as the 0 bit)
                $t \leftarrow -(2^w - (r+1)/2^{\tau}$
        if $j \geq w$ **then**
            $Q \leftarrow 2^{w-\tau}Q$
            $Q \leftarrow Q + tP$ {use precomputed table to find tP}
            $j \leftarrow j - w$
        else
            $\rho \leftarrow j$
            $j \leftarrow j - w$
if $0 \leq \rho < w$ **then**
    $Q \leftarrow 2^{\rho-\tau}Q$
    $Q \leftarrow Q + tP$ {use precomputed table to find tP}
    $Q \leftarrow 2^{\tau}Q$
else
    if $j = -1$ and $c = -2$ **then**
        $Q \leftarrow Q + -P$
return Q