On Randomizing Private Keys to Counteract DPA Attacks

Nevine Ebeid and M. Anwar Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
nebeid@uwaterloo.ca
ahasan@ece.uwaterloo.ca

Abstract. Differential power analysis (DPA) attacks can be of major concern when applied to cryptosystems that are embedded into small devices such as smart cards. To immunize elliptic curve cryptosystems (ECCs) against DPA attacks, recently several countermeasures have been proposed. A class of countermeasures is based on randomizing the paths taken by the scalar multiplication algorithm throughout its execution which also implies generating a random binary signed-digit (BSD) representation of the scalar. This scalar is an integer and is the secret key of the cryptosystem. In this paper, we investigate issues related to the BSD representation of an integer such as the average and the exact number of these representations, and integers with maximum number of BSD representations within a specific range. This knowledge helps a cryptographer to choose a key that provides better resistance against DPA attacks. Here, we also present an algorithm that generates a random BSD representation of an integer starting from the most significant signed bit. We also present another algorithm that generates all existing BSD representations of an integer to investigate the relation between increasing the number of bits in which an integer is represented and the increase in the number of its BSD representations.

Keywords: Differential power analysis, elliptic curve cryptosystems, binary signed-digit representation, scalar multiplication, smart cards.

1 Introduction

Smart cards and wireless devices are vulnerable to a special type of attacks, those are side-channel attacks. For these systems, a correct implementation of a strong protocol is not necessarily secure if this implementation does not take into account the leakage of secret key information through side channels. Examples of side channels are execution time [8], computational faults [1] and power consumption [9, 12, 13]. The mentioned systems are vulnerable since the task of monitoring the power consumption or the execution time of cryptographic protocols running on them is relatively easy.

Power Analysis attacks are those based on monitoring and analyzing the power consumption of devices while executing a cryptographic algorithm in order to obtain significant information about the secret key. Kocher et al. were the first to present attacks based on *simple* and *differential* power analysis (referred to as SPA and DPA respectively). In SPA, a single power trace of a cryptographic execution is measured and analyzed to identify large features of a cryptographic algorithm and specific operations that are typical of the underlying cryptosystem. On the other hand, to mount a DPA attack, an attacker collects hundreds of power signal traces, and manipulates them with statistical techniques to extract the *differential* signal. Therefore, DPA is in general more powerful than SPA.

Coron [3] has explained how power analysis attacks can be mounted on ECCs – which are more suitable for memory limited devices because of the small size of their keys – and suggested countermeasures against both types of attacks. Other authors have proposed DPA countermeasures for both general and specific types of elliptic curves [4, 5, 6, 10, 15, 16]. Specifically, the approach followed by the authors in [4, 16] was based each on randomizing the number and the sequence of execution of operations in the scalar multiplication algorithm. This randomization consists of inserting a random decision in the process of building the representation of the scalar k. The algorithms to which this randomization is applied were originally proposed to speed up the elliptic curve (EC) scalar multiplication. They are based on replacing the binary representation of the *scalar* k by another representation with a fewer number of nonzero symbols by allowing negative symbols to be inserted (e.g. *binary signed-digit* (BSD) representation of an integer). This yields a fewer number of executions of the point addition (or subtraction) in the EC scalar multiplication algorithm. This speedup was possible using the fact that the negative of a point on an elliptic curve is available at no extra computational cost. The algorithms proposed were computing what we referred to as the *canonical* BSD (also known as the *sparse* or NAF) representation of the integer k by scanning the bits of its binary representation starting from the least significant bit. A brief overview of the EC scalar multiplication and the possible power attacks on it is presented in Sect. 2.

When we consider the DPA countermeasures based on randomizing the BSD representation of the integer k some natural questions arise, those are: What is the average number of BSD representations for an integer k represented in n bits, where the BSD representations are to be of length l signed bits – which we will refer to as *sbits* – for both cases $l = n$ and $l = n+1$? For integers represented in n bits, which one has the maximum number of BSD representations since this may affect the choice of the secret scalar for cryptosystems that will adopt this countermeasure? In this paper, we are interested in answering those questions and the answers are presented in Sect. 3. Also, in the same section, we present an algorithm that calculates the exact number of those representations for any integer k in $\mathcal{O}(n)$.

In Sect. 4, we present an algorithm that generates a random BSD representation for an integer k by scanning its bits starting from the most significant bit in $\mathcal{O}(n)$ and another algorithm that generates all BSD representations for an integer k in $\mathcal{O}(3^{\lfloor \frac{n}{2} \rfloor})$ in the worst case. We also demonstrate the effect of increasing n on the number of BSD representations of k. Many of the proofs and examples have been omitted due to the space limitations. For a complete version of this article, we refer the reader to the corresponding CACR technical report published at `http://www.cacr.math.uwaterloo.ca/techreports/2003/corr2003-11.ps`

2 EC Scalar Multiplication and Power Analysis Attacks

In the context of ECCs, the EC scalar multiplication is the core cryptographic operation performed. This operation computes $Q = kP$, i.e. the addition of P to itself k times, where Q and P are two points on a predefined elliptic curve over a finite field and are public, and k is a secret integer. The EC scalar multiplication is conventionally performed using the *double-and-add* algorithms which are also referred to as the *binary algorithms* [7, Sect. 4.6.3]. The doubling of a point and the addition of two points on an elliptic curve are operations performed using basic finite field operation as explained in [11].

The side channel attacks on the EC scalar multiplication aim to discover the value of k. We assume that the attacker gets hold of the cryptographic token performing the EC scalar multiplication. Coron [3] has explained the possible SPA and DPA attacks that can be mounted on this token. He suggested a reasonable SPA countermeasure at the expense of the execution time of the algorithm. We will focus here on DPA countermeasures.

To mount a DPA attack, the attacker inputs to the token a large number of points and collects the corresponding power traces. As Coron [3] and Hasan [5] have explained, knowing the input point to the algorithm, the attacker starts by guessing the bits of the scalar k starting by the first bit involved in the computation, and accordingly computes the intermediate value for each point he inputs to the algorithm after the bit considered is processed. Based on the representation of the intermediate results, he chooses some partitioning function to partition the power traces he collects and processes them with statistical techniques, such as averaging, to conclude the real value of the bit considered. The attacker then repeats this procedure with every bit of the scalar.

Many of the countermeasures proposed to defeat DPA where based mainly on some form of randomization. They suggested either randomizing (or *blinding*) the point P or the scalar k. For example, one of the countermeasures proposed by Coron [3] suggested randomizing the value of k modulo the order of the point P, this is done as the first step in the scalar multiplication algorithm. Another type of countermeasures, that is of interest to us, is based on modifying the representation of the scalar k throughout the execution of the scalar multiplication algorithm. The representation of k in [4, 16] on which this approach is applied is the BSD representation. The underlying idea is as follows. The right-to-left EC scalar multiplication can be speed up by computing the NAF of k along with performing the corresponding doubling and addition (or subtraction) of points [14]. Random decisions can be inserted in the NAF generating algorithms so that they do not generate the NAF of k but any of the possible BSD representations of k (including the NAF). A different representation for the scalar k is chosen, and hence a different sequence of EC operations is followed, every time the algorithm is executed. We refer the reader to the paper by Oswald and Aigner [16] and the one by Ha and Moon [4] for the details since they are not relevant to the work presented here.

3 Number of Binary Signed Digit Representations

Considering the binary representation of k as one of its BSD representations, different BSD representations for k can be obtained by replacing 01 with $1\overline{1}$ and vice versa and by replacing $0\overline{1}$ with $\overline{1}1$ and vice versa [17, equations 8.4.4- and 8.4.5-]. The binary representation of k, must include at least one 0 that precedes a 1, so that starting from it we can obtain other BSD representations.

We consider in this case positive integers k, i.e. $0 \leq k < 2^n$. In general, the integers x, represented in $l = n$ sbits in BSD system, would be in the range $-2^l < x < 2^l$. There are 3^l different combinations for x. Also, in this system, for every positive integer corresponds a negative one obtained by exchanging the 1s with $\overline{1}$s and vice versa. Hence, the total number of non-negative combinations is $\frac{3^l+1}{2}$. This result can be confirmed using the lemmas below.

3.1 Useful Lemmas

In this subsection, we present a number of lemmas related to the number of BSD representations of an integer k. These lemmas will be used to derive the main results of this article.

Let $\lambda(k, n)$ be the number of BSD representations of k for $0 \leq k < 2^n$ that are $l = n$ sbits long. Then the following lemmas hold.

Lemma 1. (i) $\lambda(0, n) = 1$, (ii) $\lambda(1, n) = n$, (iii) $\lambda(2^i, n) = n - i$.

Lemma 2. *For* $2^{n-1} \leq k \leq 2^n - 1$, $\lambda(k, n) = \lambda(k - 2^{n-1}, n - 1)$.

Lemma 3. *For* k *even,* $\lambda(k, n) = \lambda(\frac{k}{2}, n - 1)$.

Lemma 4. *For* k *odd,*

$$\lambda(k, n) = \lambda(k - 1, n) + \lambda(k + 1, n) \ ,$$

or

$$\lambda(k, n) = \lambda\left(\left\lfloor \frac{k}{2} \right\rfloor, n - 1\right) + \lambda\left(\left\lfloor \frac{k}{2} \right\rfloor + 1, n - 1\right) \ .$$

3.2 Number of BSD Representations of Length $l = n + 1$

The algorithms that we are concerned with as DPA countermeasures are based on algorithms that generate the NAF representation of an integer. Since the NAF of an integer may be one sbit longer than its binary representation, we are interested in knowing the number of BSD representations of an integer k that are l sbits long, where in this case $l = n + 1$.

A part of these representations are the ones that are n sbits long and their number is $\lambda(k,n)$. For $l = n+1$, those representations will have a 0 as the most significant sbit. If we were to change this 0 to a 1, i.e. add 2^n to the value of k, we should subtract $2^n - k$ in the remaining n sbits, that is the 2's complement of k. $2^n - k$ has $\lambda(2^n - k, n)$ BSD representations of length n. The negative of these representations is obtained by interchanging 1s and $\bar{1}$s. Thus, if we let $\delta(k,n)$ represent the number of BSD representations of k in $n+1$ sbits for $1 \le k < 2^n$, we have

$$\delta(k,n) = \lambda(k,n) + \lambda(2^n - k, n) \ . \tag{1}$$

The same argument applies to the 2's complement of k

$$\delta(2^n - k, n) = \lambda(2^n - k, n) + \lambda(k,n) = \delta(k,n) \ . \tag{2}$$

For $k = 0$, $\delta(0,n) = \lambda(0,n) = 1$. From (2), we conclude that, for k in the defined range, the distribution of $\delta(k,n)$ is symmetric around $k = 2^{n-1}$.

Let $\varsigma(n)$ be the total number of BSD representations of length $n+1$ for all integers $k \in [0, 2^n - 1]$, we can prove that

$$\varsigma(n) = 3^n \ . \tag{3}$$

Remark 1. *We conclude that, for any n-bit integer, the average number of its – $(n+1)$ sbits long – BSD representations, i.e. of possible random forms it can take, is roughly $\left(\frac{3}{2}\right)^n$.*

It is clear from the definitions of $\lambda(k,n)$ and $\delta(k,n)$ that, for $0 \le k < 2^{n-1}$,

$$\lambda(k,n) = \delta(k, n-1) \tag{4}$$

since in this range, the binary representation of k has a 0 as the leftmost bit. In other words, the algorithm that computes $\lambda(k,n)$ presented in the following section can be used to compute $\delta(k,n)$ as

$$\delta(k,n) = \lambda(k, n+1) \ . \tag{5}$$

3.3 Algorithm to Compute the Number of BSD Representations for an Integer

Here we will present an algorithm that computes $\lambda(k,n)$ for any integer $k \in [0, 2^n - 1]$. This algorithm is based on the lemmas presented in Sect. 3.1.

Algorithm 1. Number of BSD representations of an integer k in $l = n$ sbits

```
INPUT: $k \in [0, 2^n - 1]$, $n$
OUTPUT: $C = \lambda(k,n)$
**external** $\lambda 2(k_e, k_o, w_e, w_o, n)$ /*computed by Algorithm 2 that follows*/
 1. if $(k = 0)$ then
        $C \leftarrow 1$
 2. else if $(k = 1)$ then
        $C \leftarrow n$
```

```
3. else if ($k \geq 2^{n-1}$) then
         $C \leftarrow \lambda(k - 2^{n-1}, n-1)$
4. else if ($k$ is even) then
         $C \leftarrow \lambda(\frac{k}{2}, n-1)$
5. else
   5.1 if ($k \equiv 1 \pmod 4$) then
            $C \leftarrow \lambda2(\lfloor \frac{k}{2} \rfloor, \lfloor \frac{k}{2} \rfloor + 1, 1, 1, n-1)$
   5.2 else
            $C \leftarrow \lambda2(\lfloor \frac{k}{2} \rfloor + 1, \lfloor \frac{k}{2} \rfloor, 1, 1, n-1)$
6. return $C$
```

Algorithm 1 uses Lemma 1(i) and 1(ii) to return the value of $\lambda(k, n)$ directly if the value of k is either 0 or 1. Otherwise, it uses Lemmas 2 and 3 to trim k recursively from any leading 1's or trailing 0's since they don't add to the number of BSD representations of k. Then this algorithm calls Algorithm 2 to find the actual number of BSD representations of k which is then an odd integer in the range $[0, 2^{n'-1} - 1]$, for some $n' \leq n$. Hence, Lemma 4 is applicable to this k.

Algorithm 2. Auxiliary algorithm used by Algorithm 1 to compute $\lambda(k, n)$

```
INPUT: $k_e$, $k_o$, $w_e$, $w_o$, $n$
OUTPUT: $c = \lambda2(k_e, k_o, w_e, w_o, n)$
 1. if ($k_o = 1$ AND $k_e = 2$) then
          $c \leftarrow n * w_o + (n-1) * w_e$
 2. else
    2.1 if ($k_e \equiv 0 \pmod 4$) then
       2.1.1 if ($k_o \equiv 1 \pmod 4$) then
                $c \leftarrow \lambda2(\frac{k_e}{2}, \frac{k_e}{2} + 1, w_o + w_e, w_o, n-1)$
       2.1.2 else
                $c \leftarrow \lambda2(\frac{k_e}{2}, \frac{k_e}{2} - 1, w_o + w_e, w_o, n-1)$
    2.2 else
       2.2.1 if ($k_o \equiv 1 \pmod 4$) then
                $c \leftarrow \lambda2(\frac{k_e}{2} - 1, \frac{k_e}{2}, w_o, w_o + w_e, n-1)$
       2.2.2 else
                $c \leftarrow \lambda2(\frac{k_e}{2} + 1, \frac{k_e}{2}, w_o, w_o + w_e, n-1)$
 3. return $c$
```

Using Lemma 4, $\lambda(k, n)$ for k odd constitutes of two other evaluations of the same function λ; one is for an even integer, k_e which is the closest even integer to $k/2$, and the other is for the preceding or the following odd integer, k_o. If we start using Lemmas 3 and 4 recursively to evaluate λ for k_e and k_o respectively, at each iteration there will be always two terms for the λ function multiplied by a certain weight each, w_e and w_o. In general, at the i^{th} iteration

$$\lambda(k, n) = w_{e,n-i}\ \lambda(k_{e,n-i}, n-i) + w_{o,n-i}\ \lambda(k_{o,n-i}, n-i)\ .$$

At the beginning, $w_{e,n} = w_{o,n} = 1$. It can be shown that when $k_{e,n-i} \equiv 0 \pmod 4$, the weights are updated as follows

$$w_{e,n-i-1} = w_{o,n-i} + w_{e,n-i}\ , \qquad w_{o,n-i-1} = w_{o,n-i}\ ,$$

and when $k_{e,n-i} \equiv 2 \pmod 4$, they are updated as

$$w_{e,n-i-1} = w_{o,n-i} \ , \qquad w_{o,n-i-1} = w_{o,n-i} + w_{e,n-i} \ .$$

For the complexity of Algorithm 1 – including its usage of Algorithm 2 – it is clear that it runs in $O(n)$ time and occupies $O(n)$ bits in memory. This time complexity results from the fact that both algorithms deal with the integer k one bit at a time. For the space complexity of Algorithm 1, even in its recursive form, the new values that it generates for k and n can replace the old values in memory. The same argument is valid for Algorithm 2 regarding the new values of k_e, k_o, w_e, w_o and n.

3.4 Integer with Maximum Number of BSD Representations

From the view point of DPA countermeasure, while choosing a secret key k, it is desirable to know which integer $k \in [0, 2^n - 1]$, for a certain number of sbits $l = n$ or $l = n + 1$, has the maximum number of BSD representations. We note from (4) that this integer with the largest $\delta(k, n)$ is the same one with the largest $\lambda(k, n+1)$. Also from the symmetry of $\delta(k, n)$ around 2^{n-1}, we note that there are two values of k for which $\delta(k, n)$ is the maximum value. We will denote them as $k_{max_1,n}$ and $k_{max_2,n}$.

From Lemma 3 and Lemma 4, $k_{max_1,n}$ and $k_{max_2,n}$ must be odd integers. There are two cases:

Case 1: $k_{max_1,n} \equiv 1 \pmod 4$
In this case, $\delta(k_{max_1,n} + 1, n) > \delta(k_{max_1,n} - 1, n)$, since $k_{max_1,n} - 1$ has two zeros as the least significant sbits while $k_{max_1,n} + 1$ has only one zero. Those rightmost zeros are trimmed by Algorithm 1.
Thus, from Lemma 4, we have

$$k_{max_1,n+1} = k_{max_1,n} + k_{max_1,n} + 1 = 2\ k_{max_1,n} + 1 \equiv 3 \pmod 4. \quad (6)$$

Case 2: $k_{max_1,n} \equiv 3 \pmod 4$
In this case, $\delta(k_{max_1,n} - 1, n) > \delta(k_{max_1,n} + 1, n)$. Thus, we have

$$k_{max_1,n+1} = k_{max_1,n} + k_{max_1,n} - 1 = 2\ k_{max_1,n} - 1 \equiv 1 \pmod 4. \quad (7)$$

With every increment of n, cases 1 and 2 alternate. Case 1 occurs when n is even and Case 2 occurs when n is odd.

We can use recursive substitution to find that, for n even,

$$k_{max_1,n} = \frac{1}{3}(2^n - 1) \quad (8)$$

and for n odd,

$$k_{max_1,n} = \frac{1}{3}(2^n + 1)\ . \quad (9)$$

Since $k_{max_1,n}$ and $k_{max_2,n}$ are equidistant from 2^{n-1}, we can obtain $k_{max_2,n}$ for n even as follows

$$\begin{aligned} k_{max_2,n} &= 2^{n-1} + 2^{n-1} - k_{max_1,n} = 2^n - \frac{1}{3}(2^n - 1) \\ &= \frac{1}{3}(2^{n+1} + 1) = k_{max_1,n+1} \end{aligned} \tag{10}$$

where the last equality follows from (9).

Similarly, for n odd, we have

$$k_{max_2,n} = \frac{1}{3}(2^{n+1} - 1) = k_{max_1,n+1} \quad . \tag{11}$$

Finally we should notice that, for n even, $k_{max_1,n}$ is of the form $(\langle 0\ 1\rangle^{\frac{n}{2}})_2$ and, for n odd, $k_{max_1,n}$ is of the form $(\langle 0\ 1\rangle^{\frac{n-1}{2}} 1)_2$.

Knowing the integer with maximum number of BSD representations can help the cryptographer choose a secret key as follows. He can run Algorithm 1 with that integer as input to know the maximum number of representations. Then, he can pick an integer at random and run the same algorithm again to compare the number of BSD representations of that integer with the maximum one and accept that integer as the secret key if the number of its representations is within a predefined range of the maximum number. In general, from Lemma 4, odd integers have a number of BSD representations of order of twice that of their neighboring even integers. We have also observed that integers having a good random distribution of their bits are more probable to have larger number of BSD representations.

4 Left-to-Right BSD Randomization and Generation Algorithms

The algorithms mentioned in Sect. 2 that are used to generate a random BSD representation of k are modified versions of NAF generation algorithms. Those latter ones scan the bits of the integer k from right-to-left and hence, perform the EC scalar multiplication from right to left.

Sometimes, it is advantageous to perform the EC scalar multiplication from left to right, especially when a mixed coordinate (projective and affine) system is used for saving computational cost in scalar multiplication [2].

In this section, we will present an algorithm that generates a random – $(n+1)$ sbits long – BSD representation of k while scanning it from left to right. Also we will present another algorithm that generates *all* of these possible BSD representations of k.

4.1 Left-to-Right Randomization Algorithm

The underlying idea of the algorithm is that the binary representation of k is subdivided into groups of bits – of different lengths – such that each group is formed by a number of consecutive 0s ending with a single 1. For each of these groups a random BSD representation is chosen. Whenever the choice yields a $\overline{1}$ at the end of a group – which happens when any representation for that group other than the binary one is chosen – and a 1 at the beginning of the next group – which happens when the representation chosen for the next group is the one that has no 0s – another random decision is taken so as whether to leave those two sbits (i.e., $\overline{1}1$) as they are or to change them to $0\overline{1}$.

The choice of a random representation for a certain group is done by counting the number of 0s in it, say z, and choosing a random integer $t \in [0, z]$ which will be the number of 0s to be written to the output. If t is equal to z, the last sbit to be written to the output for this group is 1, and it is actually written. Otherwise, after writing the t 0s, a 1 and only $z - t - 1$ $\overline{1}$s are written, that is the last $\overline{1}$ is not written, but saved as the value of a variable labeled as *last*. We then do the same for the next group. If for the next group $t = 0$, we take a random decision whether to write $\overline{1}1$ to the output or $0\overline{1}$ at the boundary of the two groups. This leads to the following algorithm.

Algorithm 3. Left-to-Right Randomization of an integer's BSD representation

```
INPUT: $k = (k_{n-1} \dots k_0)_2$
OUTPUT: $k' = (k'_n \dots k'_0)_{\mathrm{BSD}}$, a random BSD representation of $k$
 1. Set $k_n \leftarrow 0$; $i \leftarrow n + 1$; $last \leftarrow 1$
 2. for $j$ from $n$ down to 0 do
      if $(k_j = 1)$ then
    2.1 $t \leftarrow_R [0, i - j - 1]$
    2.2 if $(t = 0$ AND $last \neq 1)$ then
       2.2.1 $c \leftarrow_R \{0, 1\}$
       2.2.2 if $(c = 0)$ then
                 $k'_i \leftarrow \overline{1}$; $i \leftarrow i - 1$; $k'_i \leftarrow 1$
       2.2.3 else
                 $k'_i \leftarrow 0$
    2.3 else
       2.3.1 if $(last \neq 1)$ then
                 $k'_i \leftarrow \overline{1}$
       2.3.2 while $(t > 0)$ do
                 $i \leftarrow i - 1$; $k'_i \leftarrow 0$; $t \leftarrow t - 1$
       2.3.3 $i \leftarrow i - 1$; $k'_i \leftarrow 1$
    2.4 if $(i = j)$ then
              $last \leftarrow 1$
    2.5 else
       2.5.1 while $(i > j + 1)$ do
                 $i \leftarrow i - 1$; $k'_i \leftarrow \overline{1}$
       2.5.2 $i \leftarrow i - 1$; $last \leftarrow \overline{1}$
 3. if $(last \neq 1)$ then
        $k'_i \leftarrow \overline{1}$
 4. while $(i > 0)$ do
        $i \leftarrow i - 1$; $k'_i \leftarrow 0$
```

The algorithm runs in $O(n)$ time. The bits of k' can be used as they are generated in left-to-right scalar multiplication algorithms in EC cryptosystems or in left-to-right exponentiation algorithms in RSA or ElGamal cryptosystems. This means that there is no need to store k'.

4.2 Left-to-Right Generation Algorithm

The algorithm presented here is a modified version of Algorithm 3 that recursively and extensively substitutes every group of 0s ending with a 1 with one of its possible forms. It also takes into consideration the alternative substitutions at the group boundary when the representation of a group ends with a $\bar{1}$ and that of the next group starts with 1. This algorithm can be used as a straight-forward check that Algorithm 3 is capable of generating any possible – $(n+1)$ sbits long – BSD representation of an integer k in the range $[1, 2^n - 1]$, i.e. there is no BSD representation of k that cannot be generated by that algorithm. It was tested on small values of n.

Algorithm 4. Left-to-Right Generation of all BSD representations of k

```
INPUT: $k = (k_{n-1} \dots k_0)_2$
OUTPUT: all possible strings $k' = (k'_n \dots k'_0)_{\text{BSD}}$
  1. Subdivide $k$ from left to right into groups of consecutive 0s each ending with a single
     1. Store the length of each group in a look-up table $G$. Let $g$ be the index of the
     table.
  2. Set $g \leftarrow 0$; $i \leftarrow n+1$;
     $last \leftarrow 1$; $j \leftarrow i - G[g]$; $k' \leftarrow \langle\rangle$
  3. for $t = 0$ to $i - j - 1$ do
         ChooseForm($k, g, t, i, j, last, k'$)
```

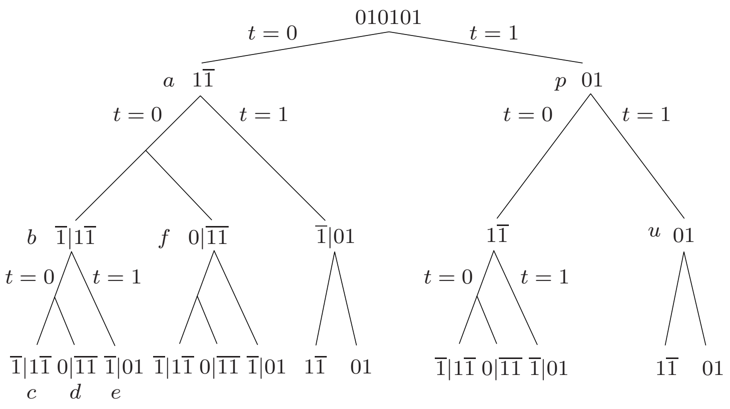
Algorithm 5. ChooseForm($k, g, t, i, j, last, k'$), a recursive procedure employed by Algorithm 4

```
INPUT: $k$, $g$, $t$, $i$, $j$, $last$, $k'$
OUTPUT: returns $k'$, a string of sbits, as a possible BSD representation of $k$.
  1. if ($t > 0$) OR ($last = 1$) then      //this step is equivalent to step 2.3 in
     Algorithm 4
     1.1 if ($last \neq 1$) then
             $k' \leftarrow k'|\bar{1}$                  //concatenate $k'$ with $\bar{1}$
     1.2 while ($t > 0$) do
             $i \leftarrow i-1$; $k' \leftarrow k'|0$; $t \leftarrow t-1$
     1.3 $i \leftarrow i-1$; $k' \leftarrow k'|1$
  2. if ($i = j$) then
         $last \leftarrow 1$
  3. else
     3.1 while ($i > j+1$) do
             $i \leftarrow i-1$; $k \leftarrow k'|\bar{1}$
     3.2 $i \leftarrow i-1$; $last \leftarrow \bar{1}$
  4. if ($j = 0$) then
     4.1 if ($last \neq 1$) then
             $k' \leftarrow k'|\bar{1}$
```

```
    4.2 return $k'$
 5. $g \leftarrow g+1;\ j \leftarrow j - G[g]$
 6. if $(j = 0)$ AND ($k$ is even) then
    6.1 if $(i > 0)$ then
       6.1.1 if $(last \neq 1)$ then
                $k' \leftarrow k'|\overline{1}$
       6.1.2 while $(i > 0)$ do
                $i \leftarrow i-1;\ k' \leftarrow k'|0$
    6.2 return $k'$
 7. $t = 0$
 8. if $(last \neq 1)$ then
          ChooseForm$(k, g, t, i-1, j, last, k'|\overline{1}1)$
          ChooseForm$(k, g, t, i, j, last, k'|0)$
 9. else
          ChooseForm$(k, g, t, i, j, last, k')$
10. for $t = 1$ to $i - j - 1$ do
          ChooseForm$(k, g, t, i, j, last, k')$
```

To better explain how this algorithm works we present in Fig. 1 the tree explored by the algorithm for $k = 21$ and $n = 5$. This tree is explored by the algorithm in a *depth-first* fashion. That is, the recursive function *ChooseForm* is first called from Algorithm 4 at node a in the figure. Then this function calls itself at node b and then at node c where it returns with the first BSD representation for $k = 21$ which is $1\overline{1}1\overline{1}1\overline{1}$. With the flow control at node b the function calls itself at node d where the second BSD representation is generated and so forth.

The algorithm works as follows. The integer k is first subdivided into groups of bits where each group consists of consecutive 0s and a single 1 at the end as for Algorithm 3. We should mention that a 0 is prepended to the binary representation of k, so that the BSD representations, k' of k are $(n+1)$ sbits long. Starting from the leftmost group, a particular BSD representation for that



010101
$t=0$
$t=1$
a $1\overline{1}$
p 01
$t=0$
$t=1$
$t=0$
$t=1$
b $\overline{1}|1\overline{1}$
f $0|\overline{1}\overline{1}$
$\overline{1}|01$
$1\overline{1}$
u 01
$t=0$
$t=1$
$t=0$
$t=1$
$\overline{1}|1\overline{1}$ $0|\overline{1}\overline{1}$ $\overline{1}|01$ $\overline{1}|1\overline{1}$ $0|\overline{1}\overline{1}$ $\overline{1}|01$ $1\overline{1}$ 01
$\overline{1}|1\overline{1}$ $0|\overline{1}\overline{1}$ $\overline{1}|01$
$1\overline{1}$ 01
c d e

Fig. 1. The tree explored by Algorithm 4 for $k = 21$ and $n = 5$

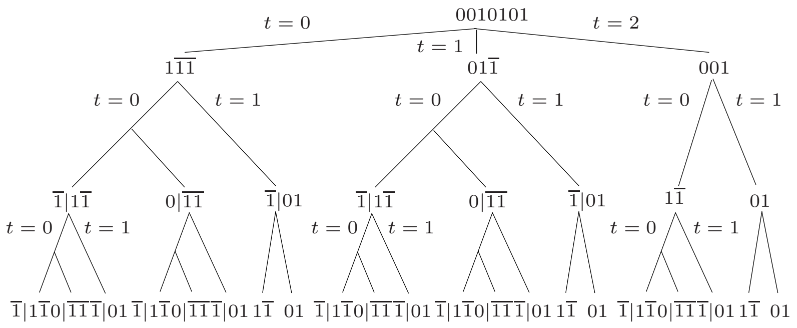

0010101
t = 0
t = 1
t = 2
$1\overline{1}\overline{1}$
$01\overline{1}$
001
$\overline{1}|1\overline{1}$
$0|\overline{1}\overline{1}$
$\overline{1}|01$
$1\overline{1}$
01
$\overline{1}|1\overline{1}0|\overline{1}\overline{1}\overline{1}|01$
$1\overline{1}$
01

Fig. 2. The tree explored by Algorithm 4 for $k = 21$ and $n = 6$

group is chosen starting with the one that contains no 0s (i.e. t, the number of 0s to be written to the output for that group, is equal to 0). The representation is formed inside the function *ChooseForm* which takes t as one of its arguments. In turn, this function goes through the possible values of t for the following group and, for each one, calls itself to form the corresponding BSD representation of that group. When t is equal to 0, the two possible alternatives at the group boundary are considered as was explained in Sect. 4.1. For example, in Fig. 1, the last sbit in the group at node a may remain $\overline{1}$ or change to 0 depending on the random decision taken at the group boundary when $t = 0$ for the next group. This is why this last sbit is written at nodes b and f before the symbol '|' which designates the boundary between the groups.

The *worst-case* complexity analysis of Algorithm 4 is presented in the following. The worst case occurs for the integer with the maximum number of BSD representations in the range $[1, 2^n - 1]$. There are actually two integers with this property for any given n, which we referred to as $k_{max_1,n}$ and $k_{max_2,n}$ in Sect. 3.4. We mentioned that, for n even, $k_{max_1,n}$ is of the form $(\langle 0\ 1\rangle^{\frac{n}{2}})_2$. For example, $k_{max_1,6} = 21 = (010101)_2$ (see Fig. 2). We also mentioned that, for any n, $k_{max_2,n} = k_{max_1,n+1}$. For example, $k_{max_1,5} = 11 = (01011)_2$ and $k_{max_2,5} = 21 = (10101)_2$ (see Fig. 2). Therefore, our analysis is conducted on those integers k of the binary form $(\langle 0\ 1\rangle^{\frac{n}{2}})_2$ for n even and $(1\langle 0\ 1\rangle^{\frac{n-1}{2}})_2$ for n odd. In the following discussion, we will drop the subscript n from $k_{max_1,n}$ and $k_{max_2,n}$ for simplicity, since it will be obvious from the context.
For n even, we have the following.

$n = 2$: $k_{max_1} = k_{max_2} = 1 = (01)_2$, $\delta(1, 2) = \lambda(1, 3) = 3$.
The tree explored here is the same as the one having as root the node b in Fig. 1. The difference is that in this case t can take the values 0, 1 and 2 with only one representation for $t = 0$.

$n = 4$: $k_{max_1} = 5 = (0101)_2$, $\delta(5, 4) = \lambda(5, 5) = 8 = 3 \cdot 3 - 1$.
The tree explored for this integer is the same as the one having as root a in Fig. 1.

$n = 6$: $k_{max_1} = 21 = (010101)_2$, $\delta(21, 6) = \lambda(21, 7) = 21 = 3 \cdot 3 \cdot 3 - (3.1 + 3)$.
The tree explored for this integer is illustrated in Fig. 2.

Let $m = \frac{n}{2}$. By induction we can deduce the following

$$\begin{aligned}\lambda(k_{max_1}, n+1) = 3^m - (m-1)3^{m-2} + \left(\sum_{i_1=1}^{m-3} i_1\right) 3^{m-4} \\ - \left(\sum_{i_1=1}^{m-5}\sum_{i_2=1}^{i_1} i_2\right) 3^{m-6} + \left(\sum_{i_1=1}^{m-7}\sum_{i_2=1}^{i_1}\sum_{i_3=1}^{i_2} i_3\right) 3^{m-8} - \cdots \ .\end{aligned} \tag{12}$$

Also, for n odd, by induction we can deduce the following

$$\begin{aligned}\lambda(k_{max_2}, n+1) = 2\cdot 3^m - \left[3^{m-1} + 2(m-1)3^{m-2}\right] \\ + \left[(m-2)3^{m-3} + 2\left(\sum_{i_1=1}^{m-3} i_1\right)3^{m-4}\right] \\ - \left[\left(\sum_{i_2=1}^{m-4} i_2\right)3^{m-5} + 2\left(\sum_{i_1=1}^{m-5}\sum_{i_2=1}^{i_1} i_2\right)3^{m-6}\right] \\ + \left[\left(\sum_{i_2=1}^{m-6}\sum_{i_3=1}^{i_2} i_3\right)3^{m-7} + 2\left(\sum_{i_1=1}^{m-7}\sum_{i_2=1}^{i_1}\sum_{i_3=1}^{i_2} i_3\right)3^{m-8}\right] \\ - \cdots\end{aligned} \tag{13}$$

where $m = \frac{n-1}{2}$.
From this discussion, we state the following theorem.

Theorem 1. *For any n, the number of BSD representations generated by Algorithm 4 is, in the worst case, $\mathcal{O}(3^{\lfloor \frac{n}{2} \rfloor})$.*

4.3 Effect of Increasing n on the Number of BSD Representations of an Integer k

In an attempt to improve DPA countermeasure, one may increase the length of the binary representation of the integer k. Below we show how the number of BSD representations of k increases if we lengthen its binary representation by adding 0s at the most significant end.

If we compare Fig. 1 with Fig. 2, we see that for the same integer $k = 21$, increasing n from 5 to 6 had the effect of increasing the number of branches emerging from the root by one. The added branch has the same tree structure as the leftmost branch a in Fig. 1. This is because the number of BSD representations of the first group – recall how the integer is subdivided into groups – has increased by one. Since all representations of a group, except for the original binary representation, end with a $\overline{1}$, the added representation would generate two alternatives when $t = 0$ for the next group. If we increase n to 7, another subtree like the one having as root a will be added to the tree. The same subtree is repeated with every 0 prepended to the binary representation of k. It is easy to verify that this is true for any integer k.

As was mentioned before, the subtree having as root the node a is the tree explored for $k = 5$ and $n = 4$. In general, the subtree that is repeated is the one formed for the integer with the binary representation having the same groups as k except for the leftmost group. That is

$$\Delta(k) = \lambda(k, n+1) - \lambda(k, n) = \lambda(k - 2^{\lfloor \log_2 k \rfloor}, \lfloor \log_2 k \rfloor + 1) \tag{14}$$

where $\Delta(k)$ is the number of leaves in the repeated subtree.

We notice that $\Delta(k)$ does not depend on n and hence, based on (14), we have the following theorem.

Theorem 2. *The number of BSD representations of an integer increases linearly with the number of bits in which it is represented in its binary form.*

5 Conclusion

In this paper, we have presented a number of cryptographically important issues related to the binary-signed digit (BSD) representation of integers, such as the average number of BSD representations of an integer k in the range $[0, 2^n - 1]$, the integer in this range that has maximum number of BSD representations. Our results provide a mechanism for choosing integers (i.e. cryptographic keys) with larger space of BSD representations and hence, with better resistance to DPA attacks. We have presented an algorithm that calculates for any integer k represented in n bits the exact number of BSD representations with the same length in $\mathcal{O}(n)$. We have also presented an algorithm that generates a random BSD representation of an integer by scanning its bits starting from the most significant end which can be used where the EC scalar multiplication is to be performed from left to right to reduce computational cost using a mixed coordinate system. This algorithm runs in $\mathcal{O}(n)$. In addition, we presented an algorithm that can generate all BSD representations of an integer, which runs in $\mathcal{O}(3^{\lfloor \frac{n}{2} \rfloor})$ in the worst case, i.e. in the case of the integer with maximum number of BSD representations in the range $[0, 2^n - 1]$. We have also proved that the number of BSD representations of an integer increases linearly with the number of bits in which it is represented.

Acknowledgements

This work has been supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada), OCIPEP (Office of Critical Infrastructure Protection and Emergency Preparedness) and RIM (Research in Motion) through an NSERC strategic project.

References

[1] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14:101–119, 2001. This is an expanded version of an earlier paper that appeared in *Proc. of EUROCRYPT '97*. 58

[2] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology – ASIACRYPT '98*, volume 1514 of *LNCS*, pages 51–65. Springer-Verlag, 1998. 65

[3] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES '99*, volume 1717 of *LNCS*, pages 292–302. Springer-Verlag, 1999. 59, 60

[4] JaeCheol Ha and SangJae Moon. Randomized signed-scalar multiplication of ECC to resist power attacks. In *Cryptographic Hardware and Embedded Systems – CHES '02*, volume 2523 of *LNCS*, pages 551–563. Springer-Verlag, 2002. 59, 60, 61

[5] M. A. Hasan. Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems. *IEEE Transactions on Computers*, 50(10):1071–1083, October 2001. 59, 60

[6] M. Joye and J. J. Quisquater. Hessian elliptic curves and side-channel attacks. In *Cryptographic Hardware and Embedded Systems – CHES '01*, volume 2162 of *LNCS*, pages 402–410. Springer-Verlag, 2001. 59

[7] D. E. Knuth. *The Art of Computer Programming/Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1973. 60

[8] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, August 1996. 58

[9] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*. Springer-Verlag, 1999. 58

[10] P. Y. Liardet and N. P. Smart. Preventing SPA/DPA in ECC systems using the Jacobi form. In *Cryptographic Hardware and Embedded Systems – CHES '01*, volume 2162 of *LNCS*, pages 391–401. Springer-Verlag, 2001. 59

[11] Alfred J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993. 60

[12] T. Messerges, E. Dabbish, and R. Sloan. Investigations of power analysis attacks on smart cards. In *USENIX Workshop on Smart-card Technology*, pages 151–161, May 1999. 58

[13] T. Messerges, E. Dabbish, and R. Sloan. Power analysis attacks of modular exponentiation in smart cards. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES '99*, volume 1717 of *LNCS*, pages 144–157. Springer-Verlag, August 1999. 58

[14] François Morain and Jorge Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Informatique théorique et Applications/Theoritical Informatics and Applications*, 24(6):531–544, 1990. 60

[15] K. Okeya and K. Sakurai. Power analysis breaks elliptic curve cryptosystems even secure against the timing attack. In *Advances in Cryptology – INDOCRYPT '00*, volume 1977 of *LNCS*, pages 178–190. Springer-Verlag, 2000. 59

[16] Elisabeth Oswald and Manfred Aigner. Randomized addition-subtraction chains as a countermeasure aginst power attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES '01*, volume 2162 of *LNCS*, pages 39–50. Springer-Verlag, 2001. 59, 60, 61

[17] G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960. 61