SPA-Resistant Binary Exponentiation with Optimal Execution Time

Carlos Moreno and M. Anwar Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Canada

cmoreno@uwaterloo.ca, ahasan@uwaterloo.ca

2011-02-09

Abstract

Straightforward implementations of binary exponentiation algorithms make the cryptographic system vulnerable to side-channel attacks; specifically, to Simple Power Analysis (SPA) attacks. Solutions proposed so far introduce a considerable performance penalty. In this report, we present a new method that implements an SPA-resistant binary exponentiation exhibiting optimal execution time at the cost of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent. The technique is optimal in the sense that it adds SPA-resistance to an underlying binary exponentiation algorithm while introducing zero computational overhead. Furthermore, we show that for practical applications, the same optimal execution time can be achieved with much less storage space, without noticeably sacrificing security or any other aspect of the cryptosystem's performance. We also discuss the possibility of our method being implemented in a way that a certain level of resistance against Differential Power Analysis may be obtained.

Keywords. Public-key cryptography, binary exponentiation, square-and-multiply, side-channel attacks, simple power analysis, differential power analysis, SPA-resistant implementations.

1 Introduction

Given the increase in usage of mobile, hand-held devices that make use of cryptography as one of the main aspects in the security of the involved systems, it becomes more and more important to protect cryptographic systems from attackers with physical access to the device holding secret parameters required to execute the cryptographic operations. At the same time, with increased interest in more sophisticated functionality while operating with the important constraint of power supplied by battery, it is equally important to maintain a good level of computational efficiency in all subsystems of these hand-held devices.

Of particular interest is the efficient execution of binary exponentiation, a fundamental operation in many cryptographic systems, specifically, in public-key cryptosystems [1], [2],

[3], [4], [5]. Algorithms that exploit the binary representation of the exponent have been presented, and are widely used [6]. However, it has been shown that a straightforward implementation of these algorithms is vulnerable to the so-called Power Analysis attacks [7]. Two types of Power Analysis attacks are of interest: Simple Power Analysis (SPA) and Differential Power Analysis (DPA). In SPA, a single power trace during the execution of the exponentiation can be used to recover the secret parameters of the cryptosystem. With DPA, statistical and signal processing is used to amplify the small variations in power consumption that are correlated to the secret data [7]. We focus our attention to SPA.

SPA-resistant algorithms have been proposed, but they all exhibit considerable performance penalties, since SPA relies on coarse-level data-dependent optimizations, and existing solutions remove some of these optimizations to avoid the problem. In this work, we present an SPA-resistant algorithm that is optimal in execution time, at the cost of a small amount of storage required.

The remaining of this report proceeds as follows: we first review the existing exponentiation algorithms, their vulnerabilities and the existing countermeasures; we then present our proposed technique, discuss some important characteristics, and compare it to prior approaches. We also discuss the possibility of parallelizing the algorithm, randomizing execution as a means of providing resistance against DPA, as well as some practical considerations. Finally, a brief discussion and suggested work is presented.

2 Review of Binary Exponentiation Algorithms

In this section, we review the main existing techniques as well as their vulnerability to SPA.

2.1 Binary Exponentiation

Two main types of algorithms have been proposed that exploit the binary representation of the exponent to provide efficient exponentiation — or, in the context of Elliptic Curve Cryptography (ECC), scalar multiplication.[1] Both algorithms traverse the exponent bits in sequence, and execute a square operation and a conditional multiplication for each exponent bit. One of the algorithms traverses the exponent bits left-to-right (MSB to LSB) using the property that, given the result of x^a, we can easily obtain $x^{a'}$, where a' is obtained by adding bit b at the right (LSB) of the exponent a; we notice that $a' = 2a + b$, and thus

$$x^{a'} = x^{2a+b} \;=\; (x^a)^2 \cdot x^b \tag{1}$$

[1] Through the rest of this paper, we will refer to exponentiation in general, including the case of ECC's scalar multiplication as an equivalent operation, since the difference is simply notational. Similarly, we will refer to multiplication or squaring of elements in general, which includes the case of ECC's point addition or doubling, respectively, as the equivalent operation.

We observe that x^b can only be 1 or x, if the value of bit b is 0 or 1, respectively. This means that this term can contribute with either a multiplication by the base, or with a null operation (a multiplication by 1), and leads to the iterative square-and-multiply algorithm shown in figure 1, for an exponent of ℓ bits.

```
Input: x;  e = (b_{ℓ−1} b_{ℓ−2} ··· b_1 b_0)_2
Output: x^e

R ← 1
For each bit b_i (i from ℓ − 1 down to 0)
{
    R ← R^2
    if (bit b_i is 1)
    {
        R ← R × x
    }
}
return R
```

Figure 1: Left-to-Right Exponentiation Algorithm.

The right-to-left variant of this algorithm simply takes advantage of the following property, for an ℓ-bits exponent e with binary representation $b_{\ell-1}\, b_{\ell-2} \cdots b_2\, b_1\, b_0$:

$$x^e \;=\; x^{\left(\sum_{i=0}^{\ell-1} b_i \cdot 2^i\right)} \;=\; x^{\left(\sum_{\substack{i=0 \\ b_i=1}}^{\ell-1} 2^i\right)} \;=\; \prod_{\substack{i=0 \\ b_i=1}}^{\ell-1} x^{\left(2^i\right)} \tag{2}$$

We observe that $x^{(2^{i+1})} = \left(x^{(2^i)}\right)^2$, leading to the iterative algorithm shown in figure 2.

```
Input: x;  e = (b_{ℓ−1} b_{ℓ−2} ··· b_1 b_0)_2
Output: x^e

S ← x
R ← 1
For each bit b_i (i from 0 up to ℓ − 1)
{
    if (bit b_i is 1)
    {
        R ← R × S
    }
    S ← S^2
}
return R
```

Figure 2: Right-to-Left Exponentiation Algorithm.

2.2 Signed-Digit NAF Representation of the Exponent

An important extension for both algorithms above comes from the use of signed-digit representation of the exponent [8]. Of interest to us is the case of expressing an exponent using signed digits representation as follows:

$$e = \sum_{i=0}^{\ell} d_i 2^i \qquad d_i \in \{\overline{1}, 0, 1\} \tag{3}$$

where, for convenience $\overline{1} \triangleq -1$ when used to denote the value of a digit.

Signed-digit representation is redundant, and thus, multiple representations for the same value can be found (for example, 111 and $100\overline{1}$ are both valid representations of the value 7). If we introduce the constraint that no two contiguous digits can be nonzero, we obtain the Non-Adjacent Form (NAF) representation, which is unique for every represented value. Furthermore, this representation may require $\ell + 1$ bits to represent an ℓ-bit binary number, but it has lowest Hamming Weight among all signed-digit representations, with one third of the digits being nonzero on average [8]. This leads to a reduction in the number of multiplications, which constitutes the main advantage of using NAF representation for the exponent.

Extending the algorithms in figures (1) and (2) to work with signed-digit exponent is straightforward: a digit $\overline{1}$ in the exponent involves a *division* instead of a multiplication; equivalently, we could think of a multiplication by the inverse of the value, which is in general easy and in certain cases very inexpensive to compute [9]. For example, in ECC, inversion of elements is essentially free, as it only involves changing the sign in the y-coordinate of the point. Assuming that the cost of inversion is negligible, figure 3 shows the right-to-left exponentiation algorithm with exponent in NAF representation.

```
Input: x;  e = $(d_\ell\ d_{\ell-1}\ \cdots\ d_1\ d_0)_{NAF}$
Output: $x^e$

S ← x
R ← 1
For each digit $d_i$ (i from 0 up to $\ell$)
{
    if (digit $d_i$ is 1)
    {
        R ← R × S
    }
    else if (digit $d_i$ is $\overline{1}$)
    {
        R ← R × $S^{-1}$
    }
    S ← $S^2$
}
return R
```

Figure 3: Right-to-Left Exponentiation with NAF Exponent.

For cases where computation of inverses is not particularly inexpensive, we can always implement the algorithm with a single inverse computation; for the left-to-right version, this is trivial, since it is always the inverse of the same value; for the right-to-left version, we could use two accumulators; one that would hold the product of all values corresponding to positive digits, and one to hold the product of values corresponding to negative digits — thus, a single inverse computation for the latter accumulator is required after all the digits have been processed.

2.3 Vulnerability to Simple Power Analysis

Both algorithms, in either standard binary or NAF forms, exhibit the same problem from the point of view of side-channel analysis; in particular, SPA: the multiplication, with a distinct and easily identifiable power consumption profile, is executed conditionally on bits of the exponent, making it possible for an attacker to recover the exponent by observing a single power trace of the device (i.e., a “plot” of the device’s measured power consumption as a function of time) while it executes the exponentiation [7]. We recall that in some operations of public-key cryptosystems, the exponent is a parameter that must be kept secret to ensure the security of the system.

We observe that the use of NAF does not eliminate the vulnerability to SPA; though a power trace only allows the attacker to distinguish nonzero digits, without knowing whether they are $\overline{1}$ or 1, the fact that only one third of the exponent bits are nonzero on average means that exhaustive search for the exponent value is within reach.

3 Existing SPA-Resistant Algorithms

The simplest solution to the vulnerability described in the previous section is to execute the multiplication *unconditionally*, and discard the result when it is not needed [10]. This way, we achieve a *constant execution path* — that is, a sequence of executed instructions that is independent of the secret data. Figure 4 shows an example of this technique, applied to the left-to-right algorithm, to obtain an algorithm known as *square-and-always-multiply*:

```
Input: x;  e = ($b_{\ell-1}$ $b_{\ell-2}$ $\cdots$ $b_1$ $b_0$)$_2$
Output: $x^e$

R ← 1
For each bit $b_i$ (i from $\ell-1$ down to 0)
{
    TMP[0] ← $R^2$
    TMP[1] ← TMP[0] × x
    R ← TMP[$b_i$]
}
return R
```

Figure 4: SPA-Resistant Exponentiation Algorithm.

The disadvantage is obvious: a strong performance penalty is imposed on the algorithm, as a considerable number of unnecessary multiplications[2] are executed — in the case of a standard binary representation of the exponent, $\ell/2$ extra multiplications are executed on average; if using NAF representation for the exponent, $2\ell/3$ extra multiplications are executed on average.

Marc Joye proposed a scheme that reduces this penalty [11] (later generalized in [12]), but it still exhibits a potentially high performance penalty, depending on the implementation of the underlying multiplication and squaring procedures. Indeed, the scheme proposed in [11] is based on a rearrangement of the loops that avoids operations where the result is discarded — however, this is achieved by implementing the squaring operations as a multiplication where the two operands are the same value. Depending on the implementation, there is a potential for a considerable speedup in squarings with respect to multiplication; with integer arithmetic, a factor of up to 2 (typically in the order of 1.5 in actual implementations), given that redundancy in the operands can be exploited. This potential speedup is completely unutilized in [11] and [12].

An additional problem with the scheme presented in [11] is that in the context of ECC, adding two different points and adding two points that are the same (i.e., doubling) are two different operations in practice [13], and thus, a difference is expected to be observed on power traces during the execution of scalar multiplications, making the technique less effective in the context of ECC operations.

Ha and Moon [14] presented a scheme resistant to DPA, and combined it with a simple SPA-resistant approach; the SPA-resistant algorithm is essentially equivalent to the square-and-always-multiply technique, and the benefit that they obtain is only observed when combined with the DPA-resistance component; still, the efficiency that they report for an exponent in NAF representation is comparable with the efficiency when using standard binary representation for the exponent, clearly indicating that their SPA-resistant component is sub-optimal.

Sun et al. [15] proposed a novel and very ingenious scheme resistant to SPA, in which the exponent is split in two halves and blocks of two bits — one bit from each half — are combined together for processing; however, the algorithm does involve operations where the result is discarded (albeit, a smaller fraction than in the case of the square-and-always-multiply technique), which necessarily means that is not optimal. Furthermore, their method is specific to standard binary representation of the exponent, which is considerably less efficient than using NAF representation; though it can be adapted to NAF, the complexity of the implementation would increase; more importantly, the fraction of operations where the result is discarded is considerably higher when using NAF representation for the exponent, making their method fundamentally incompatible with NAF. Incidentally, our proposed method can be combined with the method proposed in [15], as we will discuss in §4.3.

[2] Unnecessary from the point of view of performance, in that these multiplications are not required to obtain the correct result.

4 Our Proposed Method: SABM

We now present our proposed approach and discuss some of its aspects, as well as a comparison to previous solutions.

4.1 Square and Buffered Multiplications

In the right-to-left algorithm described in §2, a set of values are multiplied together to obtain the result. The key observation that allows us to achieve a constant execution path at coarse-level while maintaining optimality in execution time is the fact that these values need not be multiplied at the time that they are obtained. This allows us to *buffer* the execution of these multiplications, to hide any temporal patterns that would be visible through a power trace of the execution.

In its basic form, our proposed algorithm is an extension of the right-to-left algorithm; the difference being a buffer placed between the source of values to be multiplied together and the component that actually performs the multiplications. Whenever the exponent bit is 1, we send the value of S through a buffer; values enter at the times the corresponding bit is processed, but they exit at a constant rate, making power traces for different exponents identical. We thus obtain a *square-and-buffered-multiplications* (SABM) algorithm.

In an extreme (and simplified) scenario, one could store all the values that need to be multiplied together, and then, after all the squarings are done, one executes all the multiplications; this, of course, would require an unnecessarily high amount of storage. Instead, as we will discuss in §5, a small amount of storage, $O(\sqrt{\ell}\,)$, is necessary to implement a buffer so that the multiplications are interleaved with the squarings and executed at a constant rate — the average rate of ones in the exponent.

Figure 5 shows a sketch of our proposed SABM algorithm for exponent in standard binary representation. For NAF, the average rate of nonzeros would be one every three bits; thus, the condition for executing the multiplication or division would be `i` being divisible by 3.

It is assumed that the cost, in terms of execution time, to perform buffer operations is negligible compared to the operations involved in the exponentiation; this assumption is reasonable, since squarings and multiplications can have execution cost proportional to the square of the bit length of the elements (which is in general large), whereas operations on buffers typically can be implemented in constant time, with very short actual execution time. This has two important implications; one of them relates to the issue of preventing information leakage to the power trace by the buffer operations, which will be discussed in the next section. And also, for the purpose of evaluating the computational cost of our method, only the number of squarings and the number of multiplications are considered — a standard assumption in the context of cryptographic computations.

We notice that the algorithm, as shown in Fig. (5), does not address the possibility of buffer underflow; to avoid this issue, the buffer is pre-filled to half its capacity before starting the

```
Input: x;  e = $(b_{\ell-1}\ b_{\ell-2}\ \cdots\ b_1\ b_0)_2$
Output: $x^e$

S ← x
R ← 1
For each bit $b_i$ (i from 0 up to $\ell - 1$)
{
    if (bit $b_i$ is 1)
    {
        S → $Buff_S$
    }
    S ← $S^2$
    if (i is even)
    {
        Tmp ← $Buff_S$
        R ← R × Tmp
    }
}
return R
```

Figure 5: Square-and-Buffered-Multiplications (SABM) Algorithm.

multiplications (this will be discussed more in detail in §5, and was omitted in Fig. (5) for simplicity).

4.2 Avoiding Power Analysis on Buffer Operations

We now discuss the strategy and the assumptions that guarantee that no information is leaked to power traces for SPA to exploit the buffer operations. We notice that DPA would be successful at exploiting this aspect, since different operations, with some correlation between multiple power traces, is involved for the cases where exponent bits are 0 vs. cases of exponent bits being 1. However, we observe that standard DPA countermeasures (which would be needed in most cases anyway) would suffice to avoid this problem.

The key observation is that not only buffer operations are very inexpensive, in that they typically execute in constant time, with a very low number of operations; additionally, we notice that the actual data (the value being inserted in the buffer, e.g., a large integer, or a point on an elliptic curve) does not have to be copied or moved; only a reference to it (implemented for instance as a pointer in languages such as C or C++, or as a reference in languages such as Java) needs to be inserted in the buffer, making the operation truly negligible in terms of computational cost and power consumption.

If we statically allocate the storage space for the buffer, and set it up as a structure similar to a circular linked list, then only a handful of pointer assignments is required — typically word or double-word data that processors natively handle (i.e., typically fit within the processor's bus).

This allows us to assert the guarantee that no leakage to power traces occur due to the buffer operations, based on either one of the following premises:

- If we can make the assumption that the cost of a few *pointer assignments* is truly negligible and infeasible to observe through Analysis on a single power trace, then this directly translates into the guarantee that no information is leaked.

- If the above assumption does not hold for the particular architecture or attack model (for example, if the memory is external to the CPU and draws power from a separate line to which the attacker has access), then the fact that the computational cost of the pointer operations for buffer insertions is negligible makes it reasonable to set up an additional buffer for "fake" insertions — when the exponent bit is 0, we insert into this "fake" buffer. Since these values are discarded, this additional buffer does not have to be large (capacity for just one or two elements could suffice). Alternatively, we could simply set up a small set of pointer variables so that the exact same sequence of pointer assignments can be executed when the exponent bit is 0. Either way, we make each iteration *strictly identical* regardless of the value of the exponent bit. We could set up the two buffers (or the two sets of pointers) as an array of two elements, so that we use the exponent bit for subscripted (indexed) access, thus guaranteeing that the sequence of executed instructions is identical, simply operating with different data, as opposed to conditionally executing one path or the other. We omit any implementation details, since this is a somewhat trivial issue dealing with Data Structures. Also, for simplicity, we omit this detail in all remaining algorithms and figures where the buffering is involved, and simply show it as a conditional insertion on the buffer.

4.3 Simultaneous Processing of Half-Exponents

As mentioned in §3, our SABM algorithm can be combined with the method proposed by Sun et al. [15]. In their method, the exponent is split into two halves, and these are processed simultaneously. There are two key details in their method: (a) Each two bits involve a single multiplication, since they use four "accumulators" (one for each combination of the two bits); and (b) the results of multiplications where the two exponent bits are zero are discarded; but the fraction of two-bit combinations that are both zeros is $\frac{1}{4}$ of the length of the exponent on average, reducing the impact of this inefficiency on the overall performance. Furthermore, we observe that a non-SPA-resistant version of their method (one in which operations with no effect are not executed, instead of executed to then discard the result) is more efficient (in terms of execution time) than the standard binary exponentiation. Combining their algorithm with our proposed technique would provide resistance to SPA without any single operation where the result is discarded, providing increased efficiency with respect to the basic form of SABM algorithm.

If we implement our proposed method using the algorithm by Sun at al. as a replacement for the right-to-left binary exponentiation, we avoid the multiplications corresponding to the two bits being zero; instead, we buffer *all* multiplications, thus making the entire execution independent of the exponent bits, yet optimizing away the unnecessary multiplications. An additional advantage when combining Sun's method with our proposed technique is that with this method, the exponent length is reduced to half, which means that our storage requirement is reduced by a factor of $\sqrt{2}$. Figure 6 shows the details of this alternative implementation of our method; notice that the average rate of multiplications is $\frac{3}{4}$, since a multiplication occurs for every bit pair with at least one nonzero.

```
Input: x;  e = $(b_{\ell-1}\ b_{\ell-2}\ \cdots\ b_1\ b_0)_2$
           with $\ell$ divisible by 2
Output: $x^e$

S $\leftarrow$ x
$R_{01}$ $\leftarrow$ 1
$R_{10}$ $\leftarrow$ 1
$R_{11}$ $\leftarrow$ 1
$\ell'$ $\leftarrow$ $\ell/2$
For each bit pair $b_{\ell'+i}b_i$ (i from 0 up to $\ell'-1$)
{
    if ($b_{\ell'+i}b_i$ is not 00)
    {
        <S,$b_{\ell'+i}b_i$> $\rightarrow$ $Buff_S$
    }
    S $\leftarrow$ $S^2$
    if (i is not divisible by 4)
    {
        <Tmp,$b_H b_L$> $\leftarrow$ $Buff_S$
        $R_{b_H b_L}$ $\leftarrow$ $R_{b_H b_L}$ $\times$ Tmp
    }
}
$R_{01}$ $\leftarrow$ $R_{01}$ $\times$ $R_{11}$
$R_{10}$ $\leftarrow$ $R_{10}$ $\times$ $R_{11}$

Repeat $\ell'$ Times:
{
   $R_{10}$ $\leftarrow$ $(R_{10})^2$
}

return $R_{01}$ $\times$ $R_{10}$
```

Figure 6: SABM with Simultaneous Processing of Half-Exponents.

4.4 Comparison to Existing Solutions

Algorithm SABM executes in optimal time (optimal number of operations) in the sense that no unnecessary operations are executed; indeed, no result of any operation is discarded, meaning that the minimum number of operations required to obtain the correct result is performed. Also importantly, every operation is executed in its optimal form; that is, every square operation is done with an optimized procedure for squaring, and not as a multiplication routine where the two operands are equal. This constitutes an important advantage with respect to existing solutions, where a performance overhead exists with respect to the optimized (and vulnerable to SPA) forms of the algorithm.

The optimality of our method is of course relative to the underlying exponentiation technique — when used in combination with the standard binary exponentiation, our algorithm is optimal in the sense that it adds resistance to SPA with zero computational overhead;

that is, it executes the exact same number of squarings and the exact same number of multiplications as either the left-to-right or right-to-left optimized versions that are vulnerable to SPA attacks. Similarly, when used in combination with the technique proposed in [15], it maintains resistance to SPA while eliminating every unnecessary operation in their algorithm.

Table 1 summarizes the differences in performance with respect to existing solutions, showing the Square-and-Multiply (SAM) performance as a baseline. By convention, squaring routines execute in 1 unit of time; results are shown for the assumptions that multiplication routines execute in 2 units of time, and in 1.5 units of time. Values in table 1 are *average* amount of units of time to execute an exponentiation with an ℓ-bit exponent.

	Binary	NAF
S-A-M	1.75ℓ	1.5ℓ
S-A-A-M	2.5ℓ	2.5ℓ
Joye	2.25ℓ	2ℓ
Sun et al.	$1.75\ell + O(1)$	$--$
SABM	1.75ℓ	$\boldsymbol{1.5\ell}$
SABM + Sun	$1.56\ell + O(1)$	$--$

(a) Multiplications in 1.5 Units of Time

	Binary	NAF
S-A-M	2ℓ	1.67ℓ
S-A-A-M	3ℓ	3ℓ
Joye	3ℓ	2.67ℓ
Sun et al.	$2\ell + O(1)$	$--$
SABM	2ℓ	$\boldsymbol{1.67\ell}$
SABM + Sun	$1.75\ell + O(1)$	$--$

(b) Multiplications in 2 Units of Time

Table 1: Performance Comparison of Exponentiation Algorithms.

4.5 A Side-Effect of Optimal Execution

We observe that in the presence of SPA, our method reveals the number of nonzero bits in the exponent. This is a natural side-effect of the method being optimal while processing individual exponent bits, and should not be a reason for concern — the amount of leaked information is small enough that it should not compromise the security of the system.[3] Consider, in the binary case, what happens if the attacker learns that k bits are nonzero; there are $\binom{\ell}{k}$ possible exponent values, instead of 2^ℓ, reducing the entropy of the exponent by the log of the fraction (negative log, if we talk about a reduction by a positive amount).

$$\Delta H_{e;k} \;=\; -\log_2\left(\binom{\ell}{k}\,2^{-\ell}\right) \tag{4}$$

We consider the weighted average of this reduction in entropy (weighted by the probability of each number of nonzero bits) to determine the reduction in the exponent entropy ΔH_e as:

$$\Delta H_e \;=\; -\sum_{k=0}^{\ell} \mathrm{P}\{k\}\;\log_2\left(\binom{\ell}{k}\,2^{-\ell}\right) \tag{5}$$

[3] This assumes that for an attacker with a given level of computing power, the key sizes are high enough that after reducing the search space by this small amount, exhaustive search is still out of reach.

We observe that for the binary case, with $p = \frac{1}{2}$, we have

$$\mathrm{P}\{k\} \;=\; \binom{\ell}{k}\left(\tfrac{1}{2}\right)^{k}\left(1-\tfrac{1}{2}\right)^{\ell-k} \;=\; \binom{\ell}{k}\, 2^{-\ell} \tag{6}$$

From Eq.(6), we see that the argument of $\log_2$ in Eq.(5) is precisely the probability of k nonzero bits, and we easily recognize the sum in Eq.(5) as the entropy of the probability mass function [16]. Since the values of ℓ that we consider are large, the distribution is closely approximated by a normal distribution $\mathcal{N}(\ell p, \ell p(1-p))$ [17]. In the case of standard binary exponent, $\mathcal{N}(\ell/2, \ell/4)$, with entropy H_Φ in bits given by

$$\begin{aligned}
H_\Phi \;&=\; -\sum_k \Phi(k)\, \log_2 \Phi(k) \\
&=\; -\sum_k \Phi(k)\, \log_2\left(\tfrac{1}{\sqrt{2\pi\sigma^2}}\, \mathrm{e}^{-\frac{(k-\mu)^2}{2\sigma^2}}\right) \\
&=\; \sum_k \Phi(k)\left(\tfrac{1}{2}\log_2(2\pi\sigma^2) + \tfrac{(k-\mu)^2}{2\sigma^2}\cdot \log_2 \mathrm{e}\right) \\
&=\; \frac{1}{2}\log_2\left(2\pi\mathrm{e}\sigma^2\right) \;=\; \frac{1}{2}\log_2\left(\frac{\pi\mathrm{e}\,\ell}{2}\right)
\end{aligned} \tag{7}$$

As an example, for a 1024-bit exponent in standard binary representation, we see that leaking the number of nonzero bits in the exponent reduces its entropy by just 6 bits. The amount of information leaked when using NAF is even smaller, since the variance for the case of NAF is lower, as we will discuss in the next section.

5 Storage Requirements

We now discuss the storage requirements for our method to work without producing buffer overflows or underflows. Both conditions are critical for the security of the system; an instance of buffer overflow would require that a multiplication takes place immediately, when in principle one should not have taken place. Buffer underflow would make the algorithm skip a multiplication when one should have taken place. In both cases, we would leak partial information about the exponent to the power trace, since the attacker can determine the number of nonzero bits in the exponent portion that has been processed.

Let the exponent be an ℓ-bit random variable (either in binary or NAF representation), with probability p that a bit is nonzero — in the case of binary representation, p is $\frac{1}{2}$; with NAF, p is $\frac{1}{3}$. Let $\mathbf{k}$ be the number of nonzero bits in the exponent; clearly, for the binary case this random variable $\mathbf{k}$ follows the binomial distribution $\mathcal{B}(\ell, p)$, which, for sufficiently large values of ℓ can be approximated by $\mathcal{N}(\ell p, \ell p(1-p))$ [17].

In the case of NAF representation, the distribution is not binomial, since constraints between different bits exist. However, for large values of ℓ, it can also be approximated by a normal distribution. A closed-form description of this distribution is given in [8], from which it can be observed that for large values of ℓ, the variance is $\frac{2\ell}{27}$. For convenience, we use $\mathcal{N}(\ell p, \ell\zeta_p)$ as the normal approximation covering both cases; for standard binary, $\zeta_p = \frac{1}{4}$; for NAF, $\zeta_p = \frac{2}{27}$:

$$\mathrm{P}\left\{\mathbf{k} = k\right\} \approx \frac{1}{\sqrt{2\pi\ell\zeta_p}}\,\mathrm{e}^{-\frac{(k-\ell p)^2}{2\ell\zeta_p}} \tag{8}$$

In Equation (8), the term $(k - \ell p)$ represents deviation from the mean — multiplications are done at the average rate of nonzero bits, for a total of ℓp multiplications on average.

Let $\boldsymbol{\delta}$ denote the random variable corresponding to this deviation, and let us consider the probability of a deviation $\delta = c\sqrt{\ell}$, where c is a positive real number:

$$\begin{aligned}\mathrm{P}\left\{\boldsymbol{\delta} = c\sqrt{\ell}\right\} &\approx \frac{1}{\sqrt{2\pi\ell\zeta_p}}\,\mathrm{e}^{-\frac{(c\sqrt{\ell})^2}{2\ell\zeta_p}} \\ &= \frac{1}{\sqrt{2\pi\ell\zeta_p}}\,\mathrm{e}^{-\frac{c^2}{2\zeta_p}} \end{aligned}\tag{9}$$

If we set up a buffer of size $c\sqrt{\ell}$, then the above probability corresponds to the probability that the processing ends with the buffer at its full capacity. Thus, the probability of buffer overflow P_{BOF} is given by all deviations above this value:

$$\begin{aligned}P_{\text{BOF}} &\approx \sum^{\delta > c\sqrt{\ell}} \frac{1}{\sqrt{2\pi\ell\zeta_p}}\,\mathrm{e}^{-\frac{\delta^2}{2\ell\zeta_p}} \\ &\approx \int_{c\sqrt{\ell}}^{\infty} \frac{1}{\sqrt{2\pi\ell\zeta_p}}\,\mathrm{e}^{-\frac{\delta^2}{2\ell\zeta_p}}\,d\delta\end{aligned}$$

Making the substitution $t = \frac{\delta}{\sqrt{2\ell\zeta_p}}$, we obtain:

$$\begin{aligned}P_{\text{BOF}} &\approx \frac{1}{\sqrt{\pi}}\int_{\frac{c}{\sqrt{2\zeta_p}}}^{\infty} \mathrm{e}^{-t^2}\,dt \\ &= \frac{1}{2}\,\mathrm{erfc}\left(\frac{c}{\sqrt{2\zeta_p}}\right)\end{aligned}\tag{10}$$

where erfc is the complementary error function [18], defined as

$$\mathrm{erfc}\,(x) = \frac{2}{\sqrt{\pi}}\int_{x}^{\infty} \mathrm{e}^{-t^2}\,dt$$

Equation (10) considers buffer overflows that occur at the end of the processing of the exponent bits; buffer overflow could occur earlier in the processing; however, we notice that some cases overlap — for example, if the buffer size is exceeded by two units at the end of the processing, then overflow must have occurred one bit before completing the processing, and so we should not add the probability corresponding to this "early overflow", since it has been already accounted for. There are, however, instances where overflow occurs early but it is then "corrected" by the time the ℓ bits are processed (i.e., deviation exceeds the buffer size, but the bits that follow make the deviation fall back within the buffer size). We will refer to these cases as *legitimate early overflows.* The probabilities corresponding to these legitimate early overflows should be added to our estimate of the probability of buffer overflow. To this end, we make use of the following lemma:

Lemma 5.1: Legitimate early overflows occur with lower probability than non-legitimate early overflows for $p = \frac{1}{2}$ and $p = \frac{1}{3}$. (Proof included in the appendix).

From Lemma 5.1, it follows that a conservative estimate is obtained if we assume that these cases of legitimate early overflow occur with the same probability as non-legitimate early overflows. This means that the estimate of the probability of buffer overflow *within* ℓ bits should be twice the probability of overflow at exactly ℓ bits, which is what Eq.(10) represents.

The above deals with buffer overflow; to consider buffer underflow, we recall that the distribution (for the deviation δ) is assumed to be symmetric (we are using the normal distribution as an acceptable approximation for large values of ℓ). This means that buffer underflow occurs with the same probability as buffer overflow, provided that we set up a buffer of size $2c\sqrt{\ell}$ and fill it to half its size before we start removing items to perform the multiplications. Of course, we have to do this step without leaking information about exponent bits to the power traces; that is, we process the first $p^{-1}c\sqrt{\ell}$ bits before starting to remove items — this way, we fill the buffer to half its size *on average*, and the procedure is independent of the exponent bits, which ensures that no information about the exponent is leaked.

Thus, we finally obtain the probability of buffer failure P_{BF} (including overflow and underflow):

$$P_{\text{BF}} \;\approx\; 2\ \text{erfc}\left(\frac{c}{\sqrt{2\zeta_p}}\right) \tag{11}$$

From Equation (11), we see that with a buffer of size $O(\sqrt{\ell}\,)$ we can make the probability of success (i.e., probability of no buffer failure) arbitrarily close to 1, as a function of the security parameter c, corresponding to the multiplicative constant in the $O(\sqrt{\ell}\,)$ measure.

As an example, for ECC with a 256-bit exponent (i.e., $\ell = 256$) in NAF representation ($\zeta_p = \frac{2}{27}$), setting the buffer size to $4\sqrt{256} = 64$, corresponding to $c = 2$, gives us:

$$P_{\text{BF}} \approx 2\ \text{erfc}\left(\frac{2}{\sqrt{2\cdot\frac{2}{27}}}\right) \approx 2\ \text{erfc}\left(\sqrt{27}\right) \;\approx\; 4\cdot 10^{-13}$$

Buffer Structure to Prevent Buffer Underflow

As mentioned in the previous section, we need to start by filling the buffer to half its capacity before starting the multiplications, to prevent buffer underflow. However, doing this without leaking information about exponent bits requires that we process the first $\frac{1}{2}\,p^{-1}|\,\mathrm{B}\,|$ bits (where $|\,\mathrm{B}\,|$ denotes the size of the buffer) so that we fill the buffer to half its capacity *on average*, and the procedure is independent of the exponent bits.

This of course means that the algorithm will finish the processing of the exponent bits with $\frac{1}{2}\,p^{-1}|\,\mathrm{B}\,|$ elements remaining in the buffer on average, pending processing. These remaining elements would then be multiplied together before producing the result. This would also apply, though not as a requirement, to the parallelized implementation (discussed in §6); in this case, since the multiplications are being done in parallel (i.e., simultaneously) with the squarings, other adjustments could be made to avoid this issue, or at least reduce its impact.

Thus, the algorithms shown in figures (5), (6), and possibly (8), in either binary or NAF versions, would be adjusted as shown in Figure (7) — the algorithm shown corresponds to algorithm SABM from Fig.(5); however, the modification is directly applicable to all other forms of the algorithm shown in the other figures.

```
Input: x;  e = ($b_{\ell-1}$ $b_{\ell-2}$ $\cdots$ $b_1$ $b_0$)$_2$
Output: $x^e$

S $\leftarrow$ x
R $\leftarrow$ 1
For each bit $b_i$ (i from 0 up to $\ell-1$)
{
    if (bit $b_i$ is 1)
    {
        S $\rightarrow$ $\mathrm{Buff}_S$
    }
    S $\leftarrow$ $S^2$
    if (i > |$\mathrm{Buff}_S$|/(2p)
        and i is divisible by 1/p)
    {
        Tmp $\leftarrow$ $\mathrm{Buff}_S$
        R $\leftarrow$ R $\times$ Tmp
    }
}
While $\mathrm{Buff}_S$ is not empty
{
    Tmp $\leftarrow$ $\mathrm{Buff}_S$
    R $\leftarrow$ R $\times$ Tmp
}
return R
```

Figure 7: Buffer Pre-Filling and Post-Processing for SABM.

6 Parallelized Version of Algorithm SABM

Our proposed SABM algorithm has the additional benefit of being easily parallelizable while maintaining resistance to SPA attacks. This comes as a simple extension of the idea that the right-to-left exponentiation algorithm is naturally parallelizable up to two threads — as exponent bits are tested and operands for the multiplications are produced, these multiplications can be done in parallel, simultaneously to the squarings.

Incidentally, a two-thread parallel version of the right-to-left exponentiation algorithm requires a buffer for the multiplication operands. This is due to the fact that multiplication takes longer than squaring, and thus, if several contiguous nonzero bits are found, the new multiplication operands will be ready before the previous multiplications have completed.

Thus, the parallelized version of our SABM algorithm is similar to a straightforward parallelized version of the right-to-left exponentiation algorithm — the latter being vulnerable to SPA attacks. Thus, the main difference relates to the rate of execution of the multiplications; the second thread, responsible for the multiplications, should extract operands from the buffer *at a fixed rate*, taking advantage of the buffer that accommodates for the difference. The first thread adds elements to the buffer at the times that they are produced, and the second thread extracts elements from the buffer at the average rate of nonzero bits (one multiplication every two squarings for standard binary representation, or one every three squarings for NAF representation). Figure 8 shows the details of the parallel version of SABM algorithm.

```
Input: x;  e = $(b_{\ell-1}\ b_{\ell-2} \cdots b_1\ b_0)_{2/NAF}$
Output: $x^e$
R, $Buff_S$ shared between threads
```

```
Thread 1 (Squarings)

S ← x
R ← 1
For each bit $b_i$ (i from 0 up to $\ell - 1$)
{
    if (bit $b_i$ is 1)
    {
        S → $Buff_S$
    }
    S ← $S^2$
    if (i > 0 and divisible by 1/p)
    {
        Signal Thread 2
    }
}
return R
```

```
Thread 2 (Multiplications)

On Signal from Thread 1:
{
    Tmp ← $Buff_S$
    R ← R × Tmp
}
```

Figure 8: Two-Thread Parallel version of SABM Algorithm.

The thread synchronization details are omitted in Fig.(8); roughly speaking, access to the buffer `Buff`$_S$ should be synchronized, depending on how it is implemented — since insertion and removal operations are done at opposite ends of the sequence, it might be possible to implement them in such a way that no common data is accessed. In that case, simultaneous access to $\texttt{Buff}_\text{S}$ by both threads may not be a problem. The result `R`, however, needs synchronization, since the last operation of thread 1 (to return the result of the exponentiation) and operations from thread 2 might be subject to a race condition, since no guarantee should be assumed about relative order of execution between threads [19]. In particular, an additional shared (and synchronized) flag might be needed, so that thread 2 can communicate to thread 1 the fact that the last multiplication has completed, and thus the result is now available.

7 Randomized Execution of Multiplications

Given an SPA-resistant cryptosystem, [7] showed that correlation between power consumption and the data that the device is working with at a particular time can be exploited through the use of statistical and digital signal processing techniques on multiple power traces. As countermeasures, several techniques have been suggested that introduce randomization to eliminate this exploitable correlation. For example, [7] suggests blinding to randomize the data and eliminate any correlation between traces for different executions; [10] suggests several randomization techniques, some of them specific to ECC, as well as [20], which proposes techniques specific to Koblitz curve cryptosystems. These proposed techniques involve a small performance penalty, since they transform the data, so that the operations in different traces correspond to different actual data.

An interesting aspect of our proposed method — in both its parallel and non-parallel forms — is that it introduces the opportunity for randomization in the execution, with virtually no performance penalty; in particular, two different aspects can be randomized: timing and order of execution of the multiplications. Indeed, with respect to the timing of the multiplications, the detail that makes the technique work as countermeasure against SPA is not that the multiplications be executed at a fixed rate or with some fixed timing pattern: the important detail is to execute them at times independent or uncorrelated from the times at which the operands are produced. As much as we can extract elements from the buffer at a fixed rate corresponding to the average rate of nonzero bits, we can also extract them at random times, as long as the average rate at which they are extracted is the same average rate at which they are inserted into the buffer.

We can randomize this timing in several possible ways. For example, we could use a source of randomness to decide whether or not to execute a multiplication after each squaring operation: with probability p, we execute the multiplication ($p = \frac{1}{2}$ for standard binary exponent, $\frac{1}{3}$ for NAF). We could also do a random permutation of the bits of the exponent, such that we have the exact same number of nonzero bits as in the exponent, but at random positions, uncorrelated from the positions of the nonzero bits in the exponent; thus, we execute the multiplications conditioned on the bit of this permuted value. In the parallel version of the method, we could introduce randomness simply by adding small delays (idle waits) of random length after receiving the signal from the other thread indicating that a multiplication should take place.

We have to be careful with the way that this may affect the requirements on the buffer size; in particular, if we extract elements from the buffer with a random pattern, the buffer size has to be increased by a factor of $\sqrt{2}$. This is the case since now the condition for buffer failure is given by deviation from one random value to another random value. Let $\boldsymbol{\delta_e}$ be the deviation from the mean for the times of extractions from the buffer, and $\boldsymbol{\delta}$ for insertions, as defined in §5. Since by assumption these two random variables are uncorrelated, then

$$\mathrm{Var}(\boldsymbol{\delta} - \boldsymbol{\delta_e}) = \mathrm{Var}(\boldsymbol{\delta}) + \mathrm{Var}(\boldsymbol{\delta_e}) \ = \ 2\,\mathrm{Var}(\boldsymbol{\delta})$$

(since both variables have the same distribution).

From equations (11) and (8), and given that the value of ℓ does not change, we conclude that the buffer size has to be increased by a factor of $\sqrt{2}$ to maintain a given probability of buffer failure.

Notice that this is not the case if we randomly permute the bits of the exponent — on the contrary, the buffer size could be reduced and still maintain a given probability of failure. This is the case since with the bits of the exponent permuted, we guarantee that at the end of the processing — which is where buffer failure occurs with higher probability — deviation from the mean is zero.

As for randomization in the order of execution, it is clear that the order in which the multiplications are executed need not be the same as the order in which the operands are generated, since the final result is simply the product of the set of values; thus, we can permute the values in the buffer at random, so that they are extracted in a random order. This, however, provides a limited level of randomization, in that only $O(\sqrt{\ell}\,)$ elements are in the buffer at a given time, so only that many at a time can be randomized in the average case.

Furthermore, we do not even need to multiply the partial result times each value and reassign the partial result: we can, as long as there are enough values in the buffer, take any two random elements from the buffer, multiply them together, remove them from the buffer and insert the result of that multiplication back into the buffer (or remove one of the values and replace the other one with the result). Clearly, the final result will be the same. Also, the number of multiplications is necessarily the same, since each multiplication, whichever way is done, reduces the number of elements in the buffer, and thus the number of multiplications remaining, by exactly one. We notice that this compensates for the aspect mentioned in the previous paragraph about the limited level of randomization; indeed, the number of possible combinations for the sequence of operations greatly increases if we randomly choose pairs of elements from the buffer.

All of these aspects apply equally to the parallel and non-parallel versions of the method.

8 Practical Considerations

In §5, we discussed the storage requirements for our solution to work with probability arbitrarily close to 1. Several aspects can contribute to a substantial reduction in the amount of storage required in practice when implementing our proposed solution. In terms of asymptotic (big-Oh) notation, the requirement remains the same: we still require $O(\sqrt{\ell}\,)$ storage; the improvements relate to a substantial reduction in the multiplicative constant that asymptotic notation hides, but that plays an important role when it comes to an actual, practical implementation.

Buffer Underflow

The impact of buffer underflows could be entirely avoided; if the multiplications buffer is empty at the time that a multiplication must take place, a "dummy" multiplication can be performed, such as a multiplication by 1, or in the case of integer modular multiplications, by $m+1$, where m is the modulus; if a multiplication by 1 is suspected to have an identifiable power consumption profile, or in cases where multiplication by the identity is by nature a null procedure, we could multiply by a random value, as long as each of these dummy multiplications by a random value is matched by another dummy multiplication by its inverse. This can be done efficiently, since pseudo-random values suffice, and a stock of these pseudo-random values can be pre-computed. By eliminating the impact of buffer underflows, we can reduce the storage requirement to half the original amount. However, we must keep in mind that if we introduce "dummy" operations, execution time is no longer optimal; still, the fraction of "dummy" operations may be reasonably low.

Avoiding "Bad" Exponents

Perhaps a more important practical consideration is the fact that we can restrict the use of exponents to those that do not lead to buffer overflows or underflows for a given amount of storage, without noticeably sacrificing the security of the system, or any other aspect of the system's performance. We recall that for commonly used exponentiation based cryptosystems such as RSA and Diffie-Hellman/ElGamal, the secret exponent is randomly chosen when generating the key pair, or randomly chosen for each session. In both cases, the choice can be constrained to avoid buffer overflows or underflows for a given amount of storage. A crucial aspect is the fact that this constraint can be introduced without causing a noticeable reduction in the entropy of the secret exponent, as explained below.

If a buffer overflow or underflow occurs with probability P_{F}, then restricting the random choice of exponent to the subset of values that do not produce overflow or underflow reduces the entropy H_{e} of the secret exponent by $\Delta H_{\text{e}} = \log_2(1-P_{\text{F}})$. This can be easily seen, since a value of entropy of N bits is in this case associated to a uniformly distributed random choice in a space of size 2^N [16]; if invalid values occur with probability P_{F}, then the size is reduced to $2^N(1 - P_{\text{F}})$, and the random selection is still uniformly distributed, which means that the entropy is $H'_{\text{e}} = \log_2 \left(2^N(1 - P_{\text{F}})\right) \; = \; N + \log_2(1 - P_{\text{F}})$.

As an example to illustrate the validity of this argument; a probability of buffer overflow or underflow of 0.1 corresponds with a reasonably small buffer size (a value of $c \approx 0.5$ for NAF, as per Eq.(11)); yet, restricting the choice of exponents produces a change in the entropy of only $\log_2 0.9 = -0.152$ bits — a negligible reduction, as we usually consider exponents in the order of at least hundreds of bits. Even a more aggressive setting with a probability of failure close to 0.3 — which at first glance may seem alarmingly high — corresponds to a reduction in the entropy of only $\log_2 0.7 \approx -0.5$ bits, which could still be considered a negligible reduction, depending on the application. For these cases, however, it would be important to take into consideration the cost of generating random values — if the cost is too high, we may rather want to avoid low buffer sizes that would lead to discarding a high fraction of random bits.

Secure Validation of Exponents

Another crucial aspect to notice is the fact that the above constraint in the choice of exponents can be implemented without introducing any new vulnerabilities to side-channel analysis (or any other type of cryptanalysis). The validation of the exponent can be easily done with a constant execution path procedure, to avoid exposing the exponent to any side-channels; we notice that we only need to have a constant execution path until the point where it is determined that the exponent is invalid (that is, the procedure can use a conditional "early exit" as an optimization). This means that an attacker could still observe the fact that exponent values are being tested and discarded, but this constitutes useless information, since an invalid exponent is a random event that is independent from the next choice. The "early exit" optimization would leak information about the exponent, but only about exponents *being discarded*. If the exponent is valid, then the procedure will exhibit constant execution path, so no information will be leaked related to the exponents that pass the test.

9 Discussion and Suggested Work

One important aspect to take into account when considering the use of our proposed SABM algorithm is the issue of trading storage in exchange for optimal execution time. Though computing power in mainstream devices has greatly increased, and thus one could be inclined to somewhat dismiss the importance of a good execution time, the fact remains that public-key cryptographic computations — in particular exponentiation with large exponents — is usually the performance bottleneck in the security-related aspects of a device. Thus, the impact of improvement in this area on the overall performance is greater than that of improvement in any other areas. On the other hand, for storage, the cost and requirements in terms of area have decreased dramatically over the recent past, and one could reasonably expect them to continue to decrease. Even for embedded applications, where resistance to power analysis is usually a critical requirement, sufficient storage to implement our proposed method is easily available. This is certainly the case for hand-held mobile devices; maybe a little less for smart cards and almost certainly not the case for RFID devices.

Incidentally, for embedded devices — often relying on battery power — there is an additional subtle advantage with our proposed method: the savings in computations translate not only into better execution time, but also into decreased power consumption, which is often another critical requirement for these types of devices.

For most applications, the use of our technique may not even introduce the need for additional memory in the device, but simply make use of available memory that is present anyway for other reasons. Indeed, one could reasonably claim that systems for which our proposed method is suitable (including hand-held mobile devices) usually are sophisticated enough that multitasking is certainly an included feature, with other tasks that will require storage.

For situations where this is not the case, there is certainly the disadvantage that the use of additional storage introduces manufacturing cost *per unit*, unlike with techniques based on algorithmic improvement without extra storage; however, the fact remains that our proposed technique exhibits better execution performance than any of the existing techniques so far, possibly making the additional cost justifiable, depending on the situation.

Another important aspect to consider is the fact that the main idea of our technique — namely, buffering to execute multiplications at a constant rate — can be combined with other exponentiation techniques, providing optimal execution time with respect to the underlying exponentiation technique. This was shown to be the case with the technique proposed by Sun et al. [15], in which combination with our technique avoids any unnecessary operations, yielding an even more efficient method than when combining with basic right-to-left exponentiation. It was noted that the method by Sun et al. is specific to binary exponent representation, as both the complexity and the fraction of unnecessary operations increase if their method is adapted to NAF representation. However, additional work might prove useful in adapting it to NAF in combination with our technique, further improving execution time. We also emphasize the perhaps subtle detail that with their method, exponent size is reduced to half, which means that when combined with our technique, the amount of buffer storage required decreases by a factor of $\sqrt{2}$.

Additional work is also suggested to further investigate the prospect of resistance to DPA through randomization of timing pattern and order of multiplications. Though intuition suggests that it may be the case that DPA attacks could be defeated, or at least slowed down to a point where they are impractical, further research is necessary to determine whether such technique can be implemented in a way that attacks can not bypass or compensate for such randomization.

10 Conclusions

In this work, we have presented a method to perform binary exponentiation, namely, Square-and-Buffered-Multiplications, that is both resistant to SPA and optimal with respect to execution time; this is achieved through the use of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent. Storage is a commodity that most systems today, including embedded systems such as hand-held mobile devices, can afford. In exchange, we achieve execution time faster than any other SPA-resistant algorithms proposed so far. Additionally, it was noted that the reduction in computations also translates into a reduction in power consumption — a critical aspect for devices relying on battery power.

The method is an extension of the right-to-left binary exponentiation algorithm, with exponent in either standard binary or NAF representation; it is optimal in the sense that it introduces resistance to SPA while executing the exact same number of squarings and the same number of multiplications as the fully-optimized right-to-left algorithm that is vulnerable to SPA. We also demonstrated the aspect of combining our SABM technique with other exponentiation algorithms (instead of right-to-left binary exponentiation), which was shown to lead to further improvements in the execution time.

The possibility of implementing our technique in a way that it exhibits resistance to DPA was also presented. Such implementation would be centered around the idea of randomizing both the order and the timing of executions of multiplications. It was noted, however, that further research would be necessary to determine if the method can indeed be implemented in a way that DPA attacks are completely defeated or at least slowed down to a point where they are rendered impractical.

Acknowledgements

This work was supported in part through an NSERC grant awarded to Dr. Hasan.

References

[1] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[2] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[3] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, IT-31(4), 1985.

[4] Neil Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[5] Victor S. Miller. Use of Elliptic Curves in Cryptography. *Advances in Cryptology*, 1986.

[6] Daniel M. Gordon. A Survey of Fast Exponentiation Methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[7] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. *Advances in Cryptology – CRYPTO' 99*, pages 388–397, 1999.

[8] Steven Arno and Ferrell Wheeler. Signed Digit Representations of Minimal Hamming Weight. *IEEE Transactions on Computers*, 42(8):1007–1010, 1993.

[9] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[10] Jean-Sébastien Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. *Workshop on Cryptographic Hardware and Embedded Systems*, 1999.

[11] Marc Joye. Recovering Lost Efficiency of Exponentiation Algorithms on Smart Cards. *Electronic Letters*, 38(19):1095–1097, 2002.

[12] B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.

[13] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

[14] Jae Cheol Ha and Sang Jae Moon. Randomized Signed-Scalar Multiplication of ECC to Resist Power Attacks. *Workshop on Cryptographic Hardware and Embedded Systems*, 2002.

[15] Da-Zhi Sun, Jin-Peng Huai, Ji-Zhou Sun, and Zhen-Fu Cao. An Efficient Modular Exponentiation Algorithm against Simple Power Analysis Attacks. *IEEE Transactions on Consumer Electronics*, 53(4):1718–1723, 2007.

[16] T. M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley-Interscience, second edition, 2006.

[17] Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes.* McGraw-Hill, fourth edition, 2002.

[18] Milton Abramowitz and Irene A. Stegun (Editors). *Handbook of Mathematical Functions.* Dover Publications, 1965.

[19] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[20] M.A. Hasan. Power Analysis Attacks and Algorithmic Approaches to Their Countermeasures for Koblitz Curve Cryptosystems. *IEEE Transactions on Computers*, 50:1071–1083, 2001.

Appendix

Proof for Lemma 5.1

For convenience, we use the abbreviations O (Overflow), EO (Early Overflow), and prefixes L or NL (Legitimate or Non-Legitimate). The probabilities of legitimate and non-legitimate early overflow are given by

$$\begin{aligned} \mathrm{P}\{\text{LEO}\} &= \mathrm{P}\{\text{Not O} \mid \text{EO}\}\ \mathrm{P}\{\text{EO}\} \\ \mathrm{P}\{\text{NLEO}\} &= \mathrm{P}\{\text{O} \mid \text{EO}\}\ \mathrm{P}\{\text{EO}\} \end{aligned}$$

Clearly, $\mathrm{P}\{\text{Not O} \mid \text{EO}\} + \mathrm{P}\{\text{O} \mid \text{EO}\} = 1$, so it suffices to show that the conditional probability of overflow (at ℓ bits) given that early overflow occurs is greater than $\frac{1}{2}$.

Let m denote the bit at which early overflow occurs (thus, $m < \ell$), and let $\boldsymbol{\delta}_m$ denote the random variable corresponding to deviation from the mean for the remaining $\ell - m$ bits (that is, if $\mathbf{k}_m$ is the random variable representing the number of nonzero bits in the remaining $\ell - m$ bits, then $\boldsymbol{\delta}_m = \mathbf{k}_m - p(\ell - m)$).

We note that overflow occurs at ℓ bits if and only if $\boldsymbol{\delta}_m \geqslant 0$, so we have:

$$\begin{aligned} \mathrm{P}\{\text{O at } \ell \mid \text{O at } m\} &= \mathrm{P}\{\boldsymbol{\delta}_m \geqslant 0\} \\ &= \mathrm{P}\{\boldsymbol{\delta}_m = 0\} + \mathrm{P}\{\boldsymbol{\delta}_m > 0\} \end{aligned}$$

If $p = \frac{1}{2}$, then the probability mass function for $\boldsymbol{\delta}_m$ is symmetric; thus:

$$\mathrm{P}\{\boldsymbol{\delta}_m > 0\} = \mathrm{P}\{\boldsymbol{\delta}_m < 0\}$$

but

$$\mathrm{P}\{\boldsymbol{\delta}_m < 0\} + \mathrm{P}\{\boldsymbol{\delta}_m = 0\} + \mathrm{P}\{\boldsymbol{\delta}_m > 0\} = 1$$

and

$$\mathrm{P}\{\boldsymbol{\delta}_m = 0\} \quad \begin{cases} > 0 & \text{if } \ell - m \text{ is even} \\ = 0 & \text{if } \ell - m \text{ is odd} \end{cases}$$

Thus, on average (taken over m),

$$\mathrm{P}\{\boldsymbol{\delta}_m = 0\} > 0 \quad \Longrightarrow \quad \mathrm{P}\{\boldsymbol{\delta}_m \geqslant 0\} > \frac{1}{2}$$

For $p = \frac{1}{3}$, the distribution is, even for small values of $\ell - m$, near-symmetric (we recall that for reasonably large values of $\ell - m$, the distribution is closely approximated by a Gaussian, which is symmetric). This means that the argument used for the case $p = \frac{1}{2}$ is valid for $p = \frac{1}{3}$ as well — at least in the context of deriving an approximation for the probability of buffer failure. $\square$

The statement does hold in a strict and more rigorous sense, but the details of the proof get unnecessarily long and involved, given the context in which we make use of the lemma. A sketch of the necessary steps to complete the proof covering the case where $\ell - m$ takes small values is as follows: For $p = \frac{1}{3}$, we have to consider three different cases; m being congruent to 0, 1, or 2 (mod 3). That is, we consider the cases where m has the form $3n$, $3n+1$, and $3n+2$. Since we are interested in the average case (average taken over m), we work with the terms $\mathrm{P}\left\{\mathbf{k} \geqslant n;\ 3n\right\}$, $\mathrm{P}\left\{\mathbf{k} \geqslant n + \frac{1}{3};\ 3n+1\right\}$, and $\mathrm{P}\left\{\mathbf{k} \geqslant n + \frac{2}{3};\ 3n+2\right\}$ (where $\mathrm{P}\left\{\mathbf{k} \geqslant x;\ y\right\}$ denotes the probability that x or more nonzero bits occur in y bits), to show that their average is greater than $\frac{1}{2}$.

We observe that for the cases $3n+1$ and $3n+2$, since $\mathbf{k}$ is integer, equality to the mean can not occur, and therefore $\mathrm{P}\left\{\mathbf{k} \geqslant n + \frac{1}{3};\ 3n+1\right\} = \mathrm{P}\left\{\mathbf{k} \geqslant n+1;\ 3n+1\right\}$ and $\mathrm{P}\left\{\mathbf{k} \geqslant n + \frac{2}{3};\ 3n+2\right\} = \mathrm{P}\left\{\mathbf{k} \geqslant n+1;\ 3n+2\right\}$

We can either proceed by induction on n, or obtain a recurrence formula for the average of the above expressions, considering the effect of adding three additional random bits; the probabilities of adding $0, 1, 2$, and 3 nonzero bits are $\frac{8}{27}$, $\frac{12}{27}$, $\frac{6}{27}$, and $\frac{1}{27}$, respectively. This allows us to express the probabilities for $m = 3n+3$, $3n+4$, and $3n+5$ in terms of the above probabilities for $3n$, $3n+1$, and $3n+2$, and obtain the required result.