Fast SPA-Resistant Exponentiation Through Simultaneous Processing of Half-Exponents

Carlos Moreno and M. Anwar Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Canada

cmoreno@uwaterloo.ca, ahasan@uwaterloo.ca

2011-05-12

Abstract

Straightforward implementations of binary exponentiation algorithms make the cryptographic system vulnerable to side-channel attacks; specifically, to Simple Power Analysis (SPA) attacks. Most solutions proposed so far introduce a considerable performance penalty. A method exists that introduces SPA-resistance to certain types of binary exponentiation algorithms while introducing zero computational overhead, at the cost of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent. In this work, we present several new SPA-resistant algorithms that result from combining that technique with an alternative binary exponentiation algorithm where the exponent is split in two halves for simultaneous processing, showing that by combining the two techniques, we can make use of signed-digit representations of the exponent to further improve performance while maintaining SPA-resistance. In particular, we combine this idea with the use of Joint Sparse Form (JSF) for the representation of the two exponent halves, as well as signed-digit base-4 representation derived from the Non-Adjacent Form (NAF) representation of the exponent, and base-8 derived from the JSF representation. Experimental results are presented as well, confirming our performance analysis for the various methods presented.

Keywords. Public-key cryptography, ellipctic curve cryptography, binary exponentiation, signed-digit representation, Non-Adjacent Form (NAF), side-channel attacks, simple power analysis, SPA-resistant implementations.

1 Introduction

Binary exponentiation is a fundamental operation in many cryptographic systems, specifically, in public-key cryptosystems [4], [15], [5], [10], [13]. Algorithms that exploit the binary representation of the exponent have been presented, and are widely used [7]. However, it has been shown that a straightforward implementation of these algorithms is vulnerable to the so-called Power Analysis attacks [11]. 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 [11]. We focus our attention to SPA.

SPA-resistant algorithms have been proposed, but they typically 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 a previous work, we presented an SPA-resistant technique, namely, Square-and-buffered-multiplications (SABM), that is optimal in execution time, at the cost of a small amount of storage required [14]. In that work, we showed that the method can be combined with alternative algorithms as the underlying binary exponentiation. In particular, we demonstrated this aspect by combining our technique with that proposed by Sun et al. [17]; however, the method by Sun et al., as originally proposed, is fundamentally incompatible with the use of signed-digit representation of the exponent, in particular the Non-Adjacent Form (NAF), which limits its computational efficiency; in this work, we extend our previous results by adapting their algorithm to make use of signed-digit representation of the exponent, which becomes feasible when combining their method with our SABM technique. We demonstrate the technique with several representations of the exponent, showing further improvements in performance with respect to our previous results.

The remaining of this report proceeds as follows: we first review the existing exponentiation algorithms, their vulnerabilities and existing countermeasures. We then present our proposed extensions to our previous work; we first show the various proposed methods in their SPA-vulnerable form, for the purpose of analyzing their computational efficiency, and then combine them with our SABM buffering technique to provide SPA resistance. We also show a comparison between the various methods presented in this work and previous approaches (including our previous work), including experimental results that confirm our analysis. Finally, a brief discussion and concluding remarks are presented.

2 Background

In this section, we briefly review the mathematical background as well as existing techniques related to this work.

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 them traverses the exponent bits left-to-right (MSB to LSB), and one traverses the bits right-to-left. We will focus our attention to the latter, which takes advantage of the following property, for an ℓ-bits exponent e with binary representation $b_{\ell-1}\, b_{\ell-2} \cdots b_2\, b_1\, b_0$:

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

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

```
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 1: Right-to-Left exponentiation algorithm.

2.2 Signed-Digit NAF Representation of the Exponent

An important extension for the algorithm above comes from the use of signed-digit representation of the exponent [1]. Of interest to us is the case of expressing an exponent using signed digits representation as follows:

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

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

Signed-digit representation is redundant; i.e., multiple representations for the same value can be found (for example, 111 and $100\overline{1}$ are both valid representations of the value 7).

[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.

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 [1]. This leads to a reduction in the number of multiplications, which constitutes the main advantage of using NAF representation for the exponent.

Extending the algorithm in Figure 1 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 with the inverse of the value, which is in general easy and in certain cases very inexpensive to compute [12]. 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 2 shows the right-to-left exponentiation algorithm with exponent in NAF representation.

```
Input: x;  e = (d_ℓ d_{ℓ-1} ··· d_1 d_0)_NAF
Output: x^e

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

Figure 2: 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; 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 SPA and Existing Countermeasures

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 [11]. 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.

The simplest solution to the vulnerability described above is to execute the multiplication *unconditionally*, and discard the result when it is not needed [3]. This way, we achieve a *constant execution path* — that is, a sequence of executed instructions that is independent of the secret data. Figure 3 shows an example of this technique, applied to the right-to-left algorithm — when the exponent bit is 0, the multiplication is done using accumulator $\mathtt{R}_0$, with its value being discarded:

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

S ← x
R_0 ← 1, R_1 ← 1
For each bit b_i (i from 0 up to ℓ − 1)
{
    R_{b_i} ← R_{b_i} × S
    S ← S^2
}
return R_1
```

Figure 3: SPA-resistant exponentiation algorithm.

We will refer to the algorithm in Figure 3 as *square-and-always-multiply* (SAAM), even though this term is commonly used for the technique applied to the left-to-right algorithm; still, both cases are equivalent in terms of performance and in terms of resistance to SPA.

The disadvantage of this technique is obvious: a strong performance penalty is imposed on the algorithm, as a considerable number of unnecessary multiplications is executed.[2] 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 [9] (later generalized in [2]), 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 [9] is based on a rearrangement of the loops that avoids operations where the result is

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

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 [9] and [2].

Sun et al. [17] 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; this is due to the fact that 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.

Through the use of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent — the square-and-buffered-multiplications (SABM) method [14] achieves resistance to SPA with zero computational overhead. The method, in its basic form, is an extension of the right-to-left binary exponentiation algorithm, with exponent in either standard binary or NAF representation; the technique 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 (or left-to-right) algorithm that is vulnerable to SPA. It was also demonstrated that the technique can be combined with other exponentiation algorithms (instead of right-to-left binary exponentiation), which was shown to lead to further improvements in the execution time. In particular, the method was combined with the method proposed by Sun et al., showing that the performance obtained is better than the performance of their original method and also better than the performance of SABM in its basic form (using the right-to-left binary exponentiation) [14].

However, as mentioned above, the technique by Sun et al. is restricted to binary representation for the exponent. This limits the potential for improvement, in that making use of signed-digit representation, we could further reduce the number of multiplications required; of course, this is not possible with their technique in its original form; adapting their algorithm to NAF only translates into improved performance when combining it with the SABM technique [14].

3 Simultaneous Processing of Half-Exponents

We now present our proposed extension of the algorithm proposed by Sun et al. to make use of signed-digit representations, which, when combined with the SABM technique, leads to improved efficiency in terms of execution time while exhibiting resistance to SPA. We

emphasize the detail that in this section, all of the algorithms will be presented in their SPA-vulnerable form, for the purpose of analyzing their functionality and their computational efficiency. In §5, we will discuss the details of these methods combined with the SABM technique, providing SPA resistance while introducing zero computational overhead.

3.1 Exponent in Signed-Digit Representations

The central idea of simultaneous processing of half exponents is the use of multiple accumulators, to hold the product of subsets of the set of values that produce the correct result; these are then multiplied together to obtain the correct result with the product of the entire set of values. In particular, it uses one accumulator for each combination of bits (one bit from each half-exponent in corresponding positions) where at least one of the bits is nonzero. Thus, for the binary case we have $\mathtt{R}_{01}$, $\mathtt{R}_{10}$, and $\mathtt{R}_{11}$. Similarly, when extending this algorithm to work with signed-digit representations, we still have one accumulator for each combination of digits from each exponent half; thus, in the case of signed-digit exponent, we have accumulators $\mathtt{R}_{\bar{1}\bar{1}}$, $\mathtt{R}_{\bar{1}0}$, $\mathtt{R}_{\bar{1}1}$, $\mathtt{R}_{0\bar{1}}$, $\mathtt{R}_{01}$, $\mathtt{R}_{1\bar{1}}$, $\mathtt{R}_{10}$, and $\mathtt{R}_{11}$.

With this setup, the required products of values are spread across three different accumulators, since for each possible value of one digit, the other digit could have three different values. We also notice that the products of values corresponding to positions where the digit is $\bar{1}$ need to be inverted. Figure 4 shows the details of algorithm Exp-HE. A proof of correctness is presented for this algorithm.

Theorem 1: Given inputs x and exponent e, with e in signed-digit representation, Algorithm Exp-HE correctly computes the value of x^e.

***Proof*:** The assumption that $2 \mid \ell$ is made without loss of generality, as padding with one leading zero digit as needed can be made without changing the value of the exponent. Let $\ell' = \frac{\ell}{2}$, and consider the two ℓ'-digit exponent halves:

$$\begin{aligned} e_{\text{H}} &= d_{\ell-1} d_{\ell-2} \cdots d_{\ell'+1} d_{\ell'} \\ e_{\text{L}} &= d_{\ell'-1} d_{\ell'-2} \cdots d_2 d_1 d_0 \end{aligned}$$

The required value x^e can be obtained in terms of $x^{e_{\text{H}}}$ and $x^{e_{\text{L}}}$ as follows:

$$\begin{aligned} e = e_{\text{L}} + e_{\text{H}} 2^{\ell'} \implies x^e &= x^{e_{\text{L}}} \cdot x^{e_{\text{H}} 2^{\ell'}} \\ &= x^{e_{\text{L}}} \cdot \left(x^{e_{\text{H}}}\right)^{\left(2^{\ell'}\right)} \end{aligned} \tag{3}$$

Since each of the half exponents are themselves numbers in signed-digit representation, the

Algorithm Exp-HE

Input: x; $e = (d_{\ell-1}\ d_{\ell-2}\ \cdots\ d_1\ d_0)_{\text{S.D.}}$ with ℓ divisible by 2
Output: x^e

$S \leftarrow x$
$R_{\bar{1}\bar{1}} \leftarrow 1,\ R_{\bar{1}0} \leftarrow 1,\ R_{\bar{1}1} \leftarrow 1$
$R_{0\bar{1}} \leftarrow 1,\ R_{01} \leftarrow 1$
$R_{1\bar{1}} \leftarrow 1,\ R_{10} \leftarrow 1,\ R_{11} \leftarrow 1$
$\ell' \leftarrow \ell/2$
For each digit pair $d_{\ell'+i}d_i$ (i from 0 up to $\ell'-1$)
{
    if ($d_{\ell'+i}d_i$ is not 00)
    {
        $R_{d_{\ell'+i}d_i} \leftarrow R_{d_{\ell'+i}d_i} \times S$
    }
    $S \leftarrow S^2$
}
$R_{01} \leftarrow R_{01} \times R_{\bar{1}1} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}})^{-1}$
$R_{10} \leftarrow R_{10} \times R_{1\bar{1}} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{\bar{1}0} \times R_{\bar{1}1})^{-1}$

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

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

Figure 4: Simultaneous processing of half-exponents – S.D. exponent.

values $x^{e_{\text{L}}}$ and $x^{e_{\text{H}}}$ are given by:

$$x^{e_{\text{L}}} = \left(\prod_{i \in \mathcal{D}_{\text{L}}^{+}} x^{(2^i)}\right) \cdot \left(\prod_{i \in \mathcal{D}_{\text{L}}^{-}} x^{(2^i)}\right)^{-1} \tag{4}$$

$$x^{e_{\text{H}}} = \left(\prod_{i \in \mathcal{D}_{\text{H}}^{+}} x^{(2^i)}\right) \cdot \left(\prod_{i \in \mathcal{D}_{\text{H}}^{-}} x^{(2^i)}\right)^{-1} \tag{5}$$

where $\mathcal{D}_{\text{L}}^{+}$ denotes the set $\{\, i \ :\ 0 \leqslant i < \ell',\ d_i = 1 \,\}$, $\mathcal{D}_{\text{L}}^{-}$ the set $\{\, i \ :\ 0 \leqslant i < \ell',\ d_i = \overline{1} \,\}$, $\mathcal{D}_{\text{H}}^{+}$ the set $\{\, i \ :\ 0 \leqslant i < \ell',\ d_{i+\ell'} = 1 \,\}$, and $\mathcal{D}_{\text{H}}^{-}$ the set $\{\, i \ :\ 0 \leqslant i < \ell',\ d_{i+\ell'} = \overline{1} \,\}$.

Consider now the sets $\mathcal{R}_{\bar{1}\bar{1}}$, $\mathcal{R}_{\bar{1}0}$, $\mathcal{R}_{\bar{1}1}$, $\mathcal{R}_{0\bar{1}}$, $\mathcal{R}_{01}$, $\mathcal{R}_{1\bar{1}}$, $\mathcal{R}_{10}$, and $\mathcal{R}_{11}$, where $\mathcal{R}_{\text{d}_\text{H}\text{d}_\text{L}}$ denotes the set $\{\, i \ :\ 0 \leqslant i < \ell',\ d_i = \text{d}_\text{L},\ d_{i+\ell'} = \text{d}_\text{H} \,\}$.

Clearly, $\mathcal{R}_{\bar{1}1} \subset \mathcal{D}_{\text{L}}^{+}$, $\mathcal{R}_{01} \subset \mathcal{D}_{\text{L}}^{+}$, and $\mathcal{R}_{11} \subset \mathcal{D}_{\text{L}}^{+}$. Furthermore, $\mathcal{R}_{\bar{1}1} \cup \mathcal{R}_{01} \cup \mathcal{R}_{11} = \mathcal{D}_{\text{L}}^{+}$, since $\overline{1}$, 0 and 1 are the only possible values for d_H. Similarly, we have $\mathcal{R}_{\bar{1}\bar{1}} \cup \mathcal{R}_{0\bar{1}} \cup \mathcal{R}_{1\bar{1}} = \mathcal{D}_{\text{L}}^{-}$.

In Algorithm Exp-HE, S is initialized with the value of x, and at the end of each iteration it is squared; this means that at the beginning of iteration i, the value in S is $x^{(2^i)}$. This value of S will be included in the product of values stored in one of the variables $\text{R}_{\text{d}_\text{H}\text{d}_\text{L}}$, since all of the $\text{R}_{\text{d}_\text{H}\text{d}_\text{L}}$ defined are such that $\text{d}_\text{H}\text{d}_\text{L}$ is not 00; this variable $\text{R}_{\text{d}_\text{H}\text{d}_\text{L}}$ is precisely the one

corresponding to the digit pair $\mathrm{d_H d_L}$. Thus, the values stored in each variable $\mathtt{R}_{\mathrm{d_H d_L}}$ are:

$$\mathtt{R}_{\mathrm{d_H d_L}} = \prod_{i \in \mathcal{R}_{\mathrm{d_H d_L}}} x^{(2^i)} \tag{6}$$

Therefore, we have

$$\mathtt{R}_{01} \times \mathtt{R}_{\bar{1}1} \times \mathtt{R}_{11} = \prod_{i \in \mathcal{R}_{01} \cup \mathcal{R}_{\bar{1}1} \cup \mathcal{R}_{11}} x^{(2^i)} \tag{7}$$

But $\mathcal{R}_{01} \cup \mathcal{R}_{\bar{1}1} \cup \mathcal{R}_{11} = \mathcal{D}_{\mathrm{L}}^{+}$, and thus

$$\mathtt{R}_{01} \times \mathtt{R}_{\bar{1}1} \times \mathtt{R}_{11} = \prod_{i \in \mathcal{D}_{\mathrm{L}}^{+}} x^{(2^i)} \tag{8}$$

We also have

$$\mathtt{R}_{\bar{1}\bar{1}} \times \mathtt{R}_{0\bar{1}} \times \mathtt{R}_{1\bar{1}} = \prod_{i \in \mathcal{R}_{\bar{1}\bar{1}} \cup \mathcal{R}_{0\bar{1}} \cup \mathcal{R}_{1\bar{1}}} x^{(2^i)} \tag{9}$$

With $\mathcal{R}_{\bar{1}\bar{1}} \cup \mathcal{R}_{0\bar{1}} \cup \mathcal{R}_{1\bar{1}} = \mathcal{D}_{\mathrm{L}}^{-}$, and thus

$$\mathtt{R}_{\bar{1}\bar{1}} \times \mathtt{R}_{0\bar{1}} \times \mathtt{R}_{1\bar{1}} = \prod_{i \in \mathcal{D}_{\mathrm{L}}^{-}} x^{(2^i)} \tag{10}$$

Combining Equations (8) and (10) with Eq.(4), we see that the final value assigned to $\mathtt{R}_{01}$ is $x^{e_{\mathrm{L}}}$.

By an identical argument, we have that the value assigned to $\mathtt{R}_{10}$ by the end of the ℓ' iterations of the loop is $x^{e_{\mathrm{H}}}$. This value is then repeatedly squared ℓ' times, meaning that the final value stored in $\mathtt{R}_{01}$ is $(x^{e_{\mathrm{H}}})^{(2^{\ell'})}$. Since the output of the algorithm is the product of $\mathtt{R}_{01}$ and $\mathtt{R}_{10}$, Eq.(3) shows that the output is x^e, completing the proof. □

Corollary: Given the same inputs, with e in NAF representation, Algorithm Exp-HE correctly computes the value of x^e.

***Proof*:** The statement directly follows from the fact that NAF is a particular case of signed-digit representation, and thus Theorem 1 applies. □

3.2 Exponent in NAF Representation

The use of NAF for exponent representation exhibits the same advantage as for the case of straightforward binary exponentiation: with lowest Hamming Weight among all possible signed-digit representations for a given value, we reduce the number of multiplications; for the straightforward binary exponentiation, one third of the bit length of the exponent on average. In our case, under the assumption that digits d_i and $d_{i+\ell'}$ are independent, we have that each pair $d_{i+\ell'} d_i$ has a probability $\frac{4}{9}$ of being 00, reducing the average number of multiplications to five ninths of the (half-)exponent bit length (thus, $\frac{5}{18}$ of the exponent bit length).

3.3 Exponent Halves in Joint Sparse Form Representation

Further improvement is obtained by observing that this technique of simultaneously processing half exponents is similar to, or at least shares certain aspects with, multi-exponentiation; Joint Sparse Form (JSF) representation of exponent pairs has been suggested as a mean to optimize the signed-digit representation so that as many bit pairs as possible are 00. It has been shown that with JSF, we can obtain representations for each of the exponents (in our case, each of the exponent halves) with one half of the bit pairs being 00 on average [16]. This represents a non-negligible improvement over the use of NAF, since the number of multiplications with JSF is half the bit length of the exponents, compared to five ninths for NAF (thus, the average number of multiplications is $\frac{1}{4}$ of the bit length of the exponent).

Notice that with JSF, the representation for each half exponent remains a legitimate signed-digit representation. Thus, Theorem 1 holds, and algorithm Exp-HE remains valid when using JSF representation for the exponent halves.

4 Processing Multi-digit Blocks

We now discuss two extensions to the methods presented in the previous sections, where several columns are combined for simultaneous processing.

4.1 Processing Two-digit Blocks with NAF Representation

We first present an extension of the use of NAF for the half exponents that allows us to obtain an even better performance than that obtained through the use of JSF.

If we split the exponent into blocks of two digits (two columns, since we do this for the two exponent halves), the basic property of NAF guarantees that for each block of each half-exponent, at least one of the two digits must be 0. Thus, the only possible values for the digit pair are $0\overline{1}$, 00, 01, 10, and $\overline{1}0$, corresponding to numeric values -1, 0, 1, 2, and -2. Thus, if we take two-digit blocks and think of these blocks as digits in base-4 signed-digit representation, we can adapt the algorithm to work with these parameters. As an example to illustrate this, let us consider the 16-bit exponent with NAF representation $e = 10\overline{1}00\overline{1}00\overline{1}0001000$. We split it into two halves,

$$\begin{aligned} e_{\text{H}} &= 10\overline{1}00\overline{1}00 \\ e_{\text{L}} &= \overline{1}0001000 \end{aligned}$$

Grouping into two-digit blocks and expressing as base-4 signed-digit representations:

$$\begin{aligned} e_{\text{H}} &\rightarrow | 10 | \overline{1}0 | 0\overline{1} | 00 | \rightarrow 2\ \ \overline{2}\ \ \overline{1}\ \ 0 \\ e_{\text{L}} &\rightarrow | \overline{1}0 | 00 | 10 | 00 | \rightarrow \overline{2}\ \ 0\ \ 2\ \ 0 \end{aligned}$$

The exponentiation algorithm is easily modified to work with this non-binary representation of exponent $e = \sum_{i=0}^{\ell-1} d_i 4^i$ with $d_i \in \{\overline{2}, \overline{1}, 0, 1, 2\}$, as shown below:

$$\begin{aligned}
x^e &= x^{\left(\sum_{i=0}^{\ell-1} d_i 4^i\right)} \\
&= x^{\left(\sum_{\substack{i=0 \\ d_i=1}}^{\ell-1} 4^i \; + \; 2\sum_{\substack{i=0 \\ d_i=2}}^{\ell-1} 4^i \; - \; \sum_{\substack{i=0 \\ d_i=\overline{1}}}^{\ell-1} 4^i \; - \; 2\sum_{\substack{i=0 \\ d_i=\overline{2}}}^{\ell-1} 4^i\right)} \\
&= \left(\prod_{\substack{i=0 \\ b_i=1}}^{\ell-1} x^{(4^i)}\right) \cdot \left(\prod_{\substack{i=0 \\ b_i=2}}^{\ell-1} x^{(4^i)}\right)^2 \cdot \left(\prod_{\substack{i=0 \\ b_i=\overline{1}}}^{\ell-1} x^{(4^i)}\right)^{-1} \cdot \left(\prod_{\substack{i=0 \\ b_i=\overline{2}}}^{\ell-1} x^{(4^i)}\right)^{-2} \\
&= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,(\mathbf{P}_{\overline{1}})^{-1}\,(\mathbf{P}_{\overline{2}})^{-2} \qquad (11)
\end{aligned}$$

Where $\mathbf{P}_{\mathrm{B}}$ denotes the product corresponding to $b_i = \mathrm{B}$.

From Eq.(11), it is clear that algorithm Exp-HE can be modified by adding additional accumulators R_{xy} for the additional digit values 2 and $\overline{2}$. That is, we need accumulators $\mathrm{R}_{\overline{2}\overline{2}}$, $\mathrm{R}_{\overline{2}\overline{1}}$, $\mathrm{R}_{\overline{2}0}$, $\mathrm{R}_{\overline{2}1}$, $\mathrm{R}_{\overline{2}2}$, $\mathrm{R}_{\overline{1}\overline{2}}$, $\mathrm{R}_{\overline{1}\overline{1}}$, etc.

Equation (11) can be rearranged in two different ways for actual computation, as shown below:

$$\begin{aligned}
x^e &= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,(\mathbf{P}_{\overline{1}})^{-1}(\mathbf{P}_{\overline{2}})^{-2} \\
&= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\left(\mathbf{P}_{\overline{1}}\,(\mathbf{P}_{\overline{2}})^2\right)^{-1} \qquad (12) \\
&= (\mathbf{P}_1)\,(\mathbf{P}_{\overline{1}})^{-1}\left(\mathbf{P}_2\,(\mathbf{P}_{\overline{2}})^{-1}\right)^2 \qquad (13)
\end{aligned}$$

An implementation would choose which one depending on which cost is higher, inversion or squaring. For example, in ECC, inversion is a virtually free operation, and thus the form in Eq.(13) would be preferred; for cases where inversion is more expensive than squaring, Eq.(12) is preferred, since it groups the terms to execute a single inversion.

We observe that performance is better than with the use of JSF, since for JSF, half the columns are 00 on average, meaning that the number of multiplications is half the bit length of the exponent on average. When processing two-digit blocks as a base-4 signed-digit representation, we never have more than one multiplication per two-digit block, meaning that we guarantee the number of multiplications to be half the bit length of the (half-)exponent in the worst case. We have an additional performance gain in that a fraction of the two-digit blocks have four zeros, meaning that no multiplication is required for those blocks. Under the assumption of independence of the digits from the lower and upper halves of the exponent, and probability $\frac{1}{3}$ of two contiguous digits in a randomly chosen number represented in NAF being 00,[3] we have that, on average, $\frac{1}{9}$ of the two-digit blocks are all-zeros and thus do not require a multiplication.

[3] This is the case on average, for large bit lengths of the exponent, and not necessarily for every pair of contiguous digits; for example, for the two least-significant digits, the probability of being 00 is $\frac{1}{4}$.

Thus, the average number of multiplications is $\frac{2}{9}$ (approximately 22.2%) of the bit length of the exponent, and can never exceed $\frac{1}{4}$ of the bit length of the exponent. We notice that we only have extra digits 2 and -2, as a consequence of the exponent being in NAF representation; thus, the additional power requires a single squaring, as opposed to having powers 3 and -3, requiring one squaring and one multiplication for the exponentiation corresponding to those accumulators.

An additional advantage of this approach is that when grouping two-bit blocks for processing, instead of executing two squarings, we can obtain the fourth power directly, without having to explicitly compute an intermediate result, potentially obtaining better performance than with two successive squarings. This has been shown to be the case for ECC, where, under certain conditions, computing $4P$ directly can be faster than doubling twice [8].

Figure 5 shows algorithm Exp-HE-Base4, for the case of inversion being less expensive than squaring. Following the example from earlier in this section, with exponent $e = 10\bar{1}00\bar{1}00\bar{1}0001000$ in NAF representation, algorithm Exp-HE-Base4 would iterate four times, skipping the multiplication for the first iteration (since the digits from both rows are zero), then multiply into accumulator $\mathtt{R}_{\bar{1}2}$, then $\mathtt{R}_{\bar{2}0}$, then $\mathtt{R}_{2\bar{2}}$.

4.2 Processing Three-digit Blocks with JSF Representation

Algorithm Exp-HE-Base4 takes advantage of the constraints that NAF representation imposes on adjacent digits, and therefore on any two-digit blocks; following the same idea, we observe that JSF representation does impose constraints for three-digit blocks, which allows us to extend the method to a base-8 processing while taking advantage of the reduced number of base-8 digit combinations due to the constraints present in a three-digit block.

For this base-8 representation of exponent $e = \sum_{i=0}^{\ell-1} d_i 8^i$ with $d_i \in \{0, \pm1, \pm2, \pm3, \pm4, \pm5, \pm6\}$, we have:

$$
\begin{aligned}
x^e &= x^{\left(\sum_{i=0}^{\ell-1} d_i 8^i\right)} \\
&= x^{\left(\sum_{\substack{i=0 \\ d_i=1}}^{\ell-1} 8^i \;+\; 2\sum_{\substack{i=0 \\ d_i=2}}^{\ell-1} 8^i \;+\; 3\sum_{\substack{i=0 \\ d_i=3}}^{\ell-1} 8^i \;+\; \cdots\right)} \\
&= \left(\prod_{\substack{i=0 \\ b_i=1}}^{\ell-1} x^{(8^i)}\right) \cdot \left(\prod_{\substack{i=0 \\ b_i=2}}^{\ell-1} x^{(8^i)}\right)^2 \cdot \left(\prod_{\substack{i=0 \\ b_i=3}}^{\ell-1} x^{(8^i)}\right)^3 \cdot \;\cdots \\
&= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,(\mathbf{P}_3)^3\,(\mathbf{P}_4)^4\,(\mathbf{P}_5)^5\,(\mathbf{P}_6)^6 \\
&\quad (\mathbf{P}_{\bar{1}})^{-1}\,(\mathbf{P}_{\bar{2}})^{-2}\,(\mathbf{P}_{\bar{3}})^{-3}\,(\mathbf{P}_{\bar{4}})^{-4}\,(\mathbf{P}_{\bar{5}})^{-5}\,(\mathbf{P}_{\bar{6}})^{-6}
\end{aligned}
\tag{14}
$$

From Eq.(14), we see that the exponentiation procedure remains essentially the same, with a higher cost in terms of storage and post-processing (the step where the various accumulators are combined into the exponentiation results). However, the constraints given by the JSF

Algorithm Exp-HE-Base4

Input: x; e = $(d_{\ell-1}\ d_{\ell-2}\ \cdots\ d_1\ d_0)_{\mathrm{NAF}}$
with ℓ divisible by 4
Output: x^e

$R_{\bar{2}\bar{2}} \leftarrow 1,\ R_{\bar{2}\bar{1}} \leftarrow 1,\ R_{\bar{2}0} \leftarrow 1,\ R_{\bar{2}1} \leftarrow 1,\ R_{\bar{2}2} \leftarrow 1$
$R_{\bar{1}\bar{2}} \leftarrow 1,\ R_{\bar{1}\bar{1}} \leftarrow 1,\ R_{\bar{1}0} \leftarrow 1,\ R_{\bar{1}1} \leftarrow 1,\ R_{\bar{1}2} \leftarrow 1$
$R_{0\bar{2}} \leftarrow 1,\ R_{0\bar{1}} \leftarrow 1,\ R_{01} \leftarrow 1,\ R_{02} \leftarrow 1$
$R_{1\bar{2}} \leftarrow 1,\ R_{1\bar{1}} \leftarrow 1,\ R_{10} \leftarrow 1,\ R_{11} \leftarrow 1,\ R_{12} \leftarrow 1$
$R_{2\bar{2}} \leftarrow 1,\ R_{2\bar{1}} \leftarrow 1,\ R_{20} \leftarrow 1,\ R_{21} \leftarrow 1,\ R_{22} \leftarrow 1$
$S \leftarrow x$
$\ell' \leftarrow \ell/2$

For i from 0 up to $\ell'-2$ in steps of 2)
{
  $D_L \leftarrow d_i + 2 \cdot d_{i+1},\ D_H \leftarrow d_{\ell'+i} + 2 \cdot d_{\ell'+i+1}$
  if ($D_H D_L$ is not 00)
  {
    $R_{D_H D_L} \leftarrow R_{D_H D_L} \times S$
  }
  $S \leftarrow S^4$
}

$$R_{02} \leftarrow R_{02} \times R_{\bar{2}2} \times R_{\bar{1}2} \times R_{12} \times R_{22} \times (R_{\bar{2}\bar{2}} \times R_{\bar{1}\bar{2}} \times R_{0\bar{2}} \times R_{1\bar{2}} \times R_{2\bar{2}})^{-1}$$
$$R_{01} \leftarrow R_{01} \times R_{\bar{2}1} \times R_{\bar{1}1} \times R_{11} \times R_{21} \times (R_{\bar{2}\bar{1}} \times R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}} \times R_{2\bar{1}})^{-1} \times (R_{02})^2$$

$$R_{20} \leftarrow R_{20} \times R_{2\bar{2}} \times R_{2\bar{1}} \times R_{21} \times R_{22} \times (R_{\bar{2}\bar{2}} \times R_{\bar{2}\bar{1}} \times R_{\bar{2}0} \times R_{\bar{2}1} \times R_{\bar{2}2})^{-1}$$
$$R_{10} \leftarrow R_{10} \times R_{1\bar{2}} \times R_{1\bar{1}} \times R_{11} \times R_{12} \times (R_{\bar{1}\bar{2}} \times R_{\bar{1}\bar{1}} \times R_{\bar{1}0} \times R_{\bar{1}1} \times R_{\bar{1}2})^{-1} \times (R_{20})^2$$

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

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

Figure 5: Simultaneous processing of half-exponents – base-4 mode.

representation of the exponent halves offset this increase, since the number of accumulators is reduced. Specifically, we have the following properties for JSF [16] that affect our method:

- At least one column is zero in any three contiguous columns — this means that a value 7 can not occur for any of the two digits, and also, combinations such as 5-2 or 1-6 can not occur.
- If $e_{\mathrm{H}i+1}e_{\mathrm{H}i} \neq 0$, then $e_{\mathrm{L}i+1} \neq 0$ and $e_{\mathrm{L}i} = 0$, and if $e_{\mathrm{L}i+1}e_{\mathrm{L}i} \neq 0$, then $e_{\mathrm{H}i+1} \neq 0$ and $e_{\mathrm{H}i} = 0$ — this means that combinations such as 6-6, or 6-2 can not occur.

The number of accumulators is reduced from 224 (15 possible values for each digit, minus the combination 00) to 120 — an important reduction, but still leaving a considerably large number of accumulators, given that they incur both additional storage, and computational cost due to increased post-processing.

The interesting aspect of this method is its (asymptotic) computational efficiency; three digits of each half-exponent are processed with a single multiplication, bringing the average number of multiplications down to one sixth of the length of the exponent. We need to additionally factor in the fraction of blocks that are all-zeros (i.e., the base-8 digit pair is 00), such that no multiplication is required for those digit pairs. This fraction is of course lower than for the previous cases, since we're dealing with triplets of signed binary digits, with the constraints given by the JSF. We experimentally obtained this fraction to be approximately $\frac{1}{96}$, with which we obtain an average number of multiplications of 0.165ℓ.

We omit any additional details or a step-by-step diagram for the algorithm, since the details follow the same idea as algorithm Exp-HE-Base4.

5 SABM with Simultaneous Processing of Half-Exponents

In the previous sections, we discussed various algorithms that are the subject of this work, but we presented them in their SPA-vulnerable version, for the purpose of discussing the computational efficiency of each of the variants. All of the algorithms presented in the previous sections can be combined with the SABM technique to add resistance to SPA while introducing zero computational overhead. The idea is similar for all the variants of the algorithm — instead of conditionally executing the multiplication, we buffer the term to be multiplied, and then execute the multiplications at a fixed rate [14].

We recall that this can be easily done in a way that prevents Power Analysis on the buffer insertions (i.e., the possibility that the conditional insertion could be visible to an SPA attack) by noticing that the actual elements (the values of `S`) need not be actually copied or moved into the buffer, and that inserting a reference suffices, making the computational cost of the operation truly negligible. This reference to the actual element can be easily implemented as a pointer in languages such as C or C++, or implicitly as a reference in languages such as Java. Furthermore, given that the cost of buffer operations is negligible, it is reasonable to set up an additional buffer for "fake" insertions, or simply an additional set of pointers so that the exact same sequence of operations is done regardless of the value of the exponent bit — when the bit is 0, insertion into the "fake" buffer is done; alternatively, the same sequence of pointer assignments could be executed with the extra set of pointers when the exponent bit is 0. See [14] for more details.

When simultaneously processing half-exponents, the algorithm requires a selection of the accumulator where the multiplication should be performed; thus, in this case, we need to buffer the combination of the value to be multiplied and the digit pair for the selection of the accumulator (or equivalently, a direct reference, e.g., in the form of a pointer, to the selected accumulator could be used).

Figure 6 shows the details for the case of exponent in NAF representation; notice the pre-filling of the buffer to half its capacity on average: the probability of insertion on the buffer

is $p = \frac{5}{9}$, so we process $p^{-1}|\mathtt{Buff_S}|/2$ exponent digits before starting to extract elements from the buffer; at the end of the first loop, the buffer will be at half capacity on average, so we need to process any remaining elements.

Algorithm SABM-HE

Input: x; e = $(d_{\ell-1}\ d_{\ell-2}\ \cdots\ d_1\ d_0)_{\mathrm{NAF}}$
with ℓ divisible by 2
Output: x^e

$S \leftarrow x$
$R_{\bar{1}\bar{1}} \leftarrow 1,\ R_{\bar{1}0} \leftarrow 1,\ R_{\bar{1}1} \leftarrow 1$
$R_{0\bar{1}} \leftarrow 1,\ R_{01} \leftarrow 1$
$R_{1\bar{1}} \leftarrow 1,\ R_{10} \leftarrow 1,\ R_{11} \leftarrow 1$
$\ell' \leftarrow \ell/2$

For each digit pair $d_{\ell'+i}d_i$ (i from 0 up to $\ell'-1$)
{
    if ($d_{\ell'+i}d_i$ is not 00)
    {
        <S,$d_{\ell'+i}d_i$> $\rightarrow$ $\mathrm{Buff_S}$
    }
    $S \leftarrow S^2$
    if (i > 9·$|\mathrm{Buff_S}|$/10 and i mod 9 is even)
    {
        <Tmp,$d_H d_L$> $\leftarrow$ $\mathrm{Buff_S}$
        $R_{d_H d_L} \leftarrow R_{d_H d_L} \times$ Tmp
    }
}

While $\mathrm{Buff_S}$ is not empty
{
    <Tmp,$d_H d_L$> $\leftarrow$ $\mathrm{Buff_S}$
    $R_{d_H d_L} \leftarrow R_{d_H d_L} \times$ Tmp
}

$R_{01} \leftarrow R_{01} \times R_{\bar{1}1} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}})^{-1}$
$R_{10} \leftarrow R_{10} \times R_{1\bar{1}} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{\bar{1}0} \times R_{\bar{1}1})^{-1}$

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

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

Figure 6: SABM with simultaneous processing of half-exponents – NAF exponent.

The idea is almost identical and directly applicable to the other forms discussed in the previous section. In particular, for JSF representation of the exponent halves, the algorithm remains the same (except for a pre-processing stage to convert the exponent representation to JSF, or the precondition that the input be in this form).

We present a proof of correctness for the case of NAF exponent, but it should be clear that the idea holds for the other cases as well.

Theorem 2: Given inputs x and exponent e, with e in NAF representation, Algorithm SABM-HE correctly computes the value of x^e.

***Proof*:** (sketch)

The structure and sequence of computations in algorithm SABM-HE are identical to those in algorithm Exp-HE; since the input is in NAF representation, Theorem 1's Corollary applies. We only need to show that the buffering aspect does not affect this structure or the values being used in the computations.

Each computation requires the value of `S` at the given pass of the `For` loop, and it also requires the digit pair, so that the correct accumulator is selected. These two items are stored in the buffer; no other instruction in algorithm SABM-HE stores any values in the buffer. Each computation involves extracting an element from the buffer, which guarantees that no element will be used more than once. In the second loop (the `While` loop), every elment in the buffer is removed and used for the corresponding computation. Thus, every element for which a computation is required is indeed subject of said computation. Thus, algorithm SABM-HE executes the exact same set of computations on the same accumulators with respect to algorithm Exp-HE, completing the proof. □

6 Performance Comparison

We now focus on the performance of the various methods proposed in this work, and present experimental results that confirm our analysis.

6.1 Analytic Comparison

Tables 1 and 2 summarize the differences in performance of the various techniques described in this section, and compares against existing solutions, showing the Square-and-Multiply (SAM) performance as a baseline. Table 1 shows the *average* number of multiplications required to execute an exponentiation with an ℓ-bit exponent; we recall that all the methods listed require exactly ℓ squarings in addition to the number of multiplications shown.

	Binary	NAF/S.D.
S-A-M	0.5ℓ	0.33ℓ
S-A-A-M	ℓ	ℓ
Sun et al.	$0.5\ell + O(1)$	$--$
SABM [(*)]	0.5ℓ	0.33ℓ
SABM-HE [(**)]	$0.375\ell + O(1)$	$0.275\ell + O(1)$
SABM-HE-JSF [(**)]	$--$	$0.25\ell + O(1)$
SABM-HE-Base4 [(**)]	$--$	$0.22\ell + O(1)$
SABM-HE-Base8 [(**)]	$--$	$\mathbf{0.165\ell + O(1)}$

[(*)] Our previous work; [(**)] This work

Table 1: Number of multiplications for exponentiation algorithms.

Table 2 shows the *average* amount of units of time to execute an exponentiation with an ℓ-bit exponent. 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.

	Binary	NAF/S.D.
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	1.5ℓ
SABM-HE (**)	$1.56\ell + O(1)$	$1.416\ell + O(1)$
SABM-HE-JSF (**)	$--$	$1.375\ell + O(1)$
SABM-HE-Base4 (**)	$--$	$1.33\ell + O(1)$
SABM-HE-Base8 (**)	$--$	$\mathbf{1.2475\ell + O(1)}$

(a) Multiplications in 1.5 units of time

	Binary	NAF/S.D.
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	1.67ℓ
SABM-HE (**)	$1.75\ell + O(1)$	$1.55\ell + O(1)$
SABM-HE-JSF (**)	$--$	$1.5\ell + O(1)$
SABM-HE-Base4 (**)	$--$	$1.44\ell + O(1)$
SABM-HE-Base8 (**)	$--$	$\mathbf{1.33\ell + O(1)}$

(b) Multiplications in 2 units of time

Table 2: Performance comparison of exponentiation algorithms.
(*) Our previous work; (**) This work

6.2 Experimental Results

As part of this study, we implemented several of the methods for the purpose of experimentally verifying their efficiency; in particular, the implementations did not include the buffering aspect, since in our view, it seems rather clear that this aspect does not affect the computational performance of the methods. In addition to the basic RTL exponentiation with NAF exponent, used as a baseline for comparison, we implemented the methods Exp-HE (using NAF exponent), Exp-HE-JSF, and Exp-HE-Base4. All the implementations are

based on the GMP library (version 5.0.1, the latest version at the time of this work) for the underlying arithmetic operations [6]. We tested the methods with exponent lengths of 256 and 512 bits, in the range of typical ECC applications,[4] and also 2048 and 4096, in the typical range of RSA applications.

Table 3 shows the results; measurements are actual execution time of the exponentiation routines (excluding startup and initialization time for the library facilities). Multiple measurements (1000) with randomly chosen exponents were performed, and Table 3 shows the average value. The implementations were compiled and executed on an Ubuntu Linux 9.04 system, running on an AMD Quad-Core Phenom processor at 2.5GHz. CPU frequency scaling was disabled, as well as the graphical interface and all other applications, to avoid any disruption on the measurements.

	256 bits	512 bits	2048 bits	4096 bits
R-T-L (NAF)	31.1μs	93.9μs	2.19ms	13.23ms
Exp-HE (NAF)	35.5μs	96.5μs	2.12ms	12.66ms
Exp-HE-JSF	35.1μs	96.9μs	2.07ms	12.33ms
Exp-HE-Base4	36.6μs	95.9μs	2.06ms	12.08ms

Table 3: Execution time of exponentiation algorithms.

The results are consistent with the expected execution times of the various methods; the measurements confirm that the Base-4 method is actually at disadvantage for short keys, such as those typically used in ECC, but it is asymptotically more efficient, as shown by the results for larger exponent sizes.

7 Discussion

Several extensions to our previous results were presented, with various degrees of improvement and various trade-offs. The additional storage required for the extra accumulators with respect to the method in our previous work (eight accumulators for signed-digit/NAF exponent vs. three accumulators for standard binary exponent in our previous work) should be offset by the fact that the variance for the distribution of nonzeros is smaller when using signed-digit representations, thus leading to a reduced buffer size requirement for a given probability of buffer failure [14].

More importantly, the improvement in performance is substantial when using signed-digit representations for the exponent, presenting an interesting situation where we improve (or perhaps just maintain) the storage requirements and still observe a considerable improvement in terms of computational cost. We recall that this reduction in computational cost also

[4] Though all the implementations use exponentiation based on modular integer arithmetic, performance comparisons should still be meaningful when combining with exponent lengths typical for ECC, illustrating the usefulness of the various methods for ECC applications.

leads to a reduction in power consumption, which could be a critical aspect for systems relying on battery power, such as hand-held mobile devices.

In the various methods proposed in this work, performance in terms of asymptotic computational cost has distinct values and the various methods are clearly ranked by their efficiency; however, the cost hidden in the $O(1)$ expression (the constant number of operations in the post-processing) is different for each of the various methods, with the optimal trade-off — at least for the typical exponent bit lengths used in ECC — possibly being the method using JSF; the improvement derived from using JSF for the exponent halves comes at no cost whatsoever, since the algorithm is identical with respect to the version that uses NAF. However, the method processing two-digit blocks incurs additional computational cost in terms of post-processing: multiplications, squarings, and inversions to combine the various accumulators into the required results. Given the higher number of multiplications in the post-processing stage, for exponents below 1000 bits, the NAF or JSF methods actually require fewer multiplications, making this Base-4 method suitable for systems where the bit lengths are in the thousands of bits, such as RSA. In particular, for ECC, the Base-4 method is not particularly attractive. The same holds for the Base-8 method, where the number of operations in the post-processing stage makes it unattractive for the typical exponent bit lengths used in ECC, even though its asymptotic performance is considerably better than any of the other methods.

8 Conclusions

In this work, we have presented several new methods that efficiently perform binary exponentiation while exhibiting resistance to SPA. The methods extend the results obtained in our previous work, and provide substantial improvements in computational efficiency with respect to our previous results, or with respect to any other existing SPA-resistant methods.

The extensions derive from the use of signed-digit representations for the exponent, which was fundamentally incompatible with the technique of processing half-exponents in its original form; only when combining the method with the SABM buffering technique [14] is it possible to adapt the technique to the use of signed-digit/NAF exponent representation to obtain improvements in the computational cost. It was also noted that these improvements in computational cost can lead to a reduction in power consumption, a critical aspect for systems relying on battery power such as hand-held mobile devices. Even for the methods where the amount of storage is larger, power consumption is not affected, since the amount of computations and write operations are the same, spread across a larger number of storage locations.

From the various methods proposed, the method using JSF representation for the exponent halves is perhaps the better suited for typical cryptosystems based on ECC, since the exponent lengths are relatively small (in the hundreds of bits), and the lower post-processing cost of this method means that the total amount of operations is lower than for the other methods, even if some of these other methods are superior in terms of *asymptotic* performance.

Acknowledgements

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

References

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

[2] 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.

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

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

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

[6] Torbjörn Granlund et al. GNU Multi-Precision Arithmetic Library (GMP). `http://www.gmplib.org`.

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

[8] Jorge Guajardo and Christof Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In *Advances in Cryptology – CRYPTO '97*, volume 1294, pages 342–356. Springer Berlin / Heidelberg, 1997.

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

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

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

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

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

[14] Carlos Moreno and M. Anwar Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *Journal of Cryptographic Engineering*, pages 1–13, 2011. 10.1007/s13389-011-0008-9.

[15] 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.

[16] Jerome A. Solinas. Low-Weight Binary Representations for Pairs of Integers. *Centre for Applied Cryptographic Research Technical Report CORR 2001-41*, 2001.

[17] 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.