Exp-HE: A Family of Fast Exponentiation Algorithms Resistant to SPA, Fault, and Combined Attacks

Carlos Moreno
Deptartment of Electrical and Computer Engineering
University of Waterloo.
Waterloo, Canada
cmoreno@uwaterloo.ca

M. Anwar Hasan
Department of Electrical and Computer Engineering
University of Waterloo.
Waterloo, Canada
ahasan@uwaterloo.ca

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo.
Waterloo, Canada
sfischme@uwaterloo.ca

ABSTRACT

Security and privacy are growing concerns in modern embedded software, given the increasing level of connectivity as well as complexity and features in embedded devices. Use of cryptographic techniques is often a requirement on which the security of the device relies. However, important challenges arise when potential attackers have physical access to the device. Side-channel analysis, including simple power analysis (SPA), is a class of powerful non-intrusive attacks that are suitable for adversaries with physical access to the device. Countermeasures exist, but they typically involve a considerable performance penalty, and some of them in turn introduce a vulnerability to induced fault attacks.

In this work, we present several new efficient cryptographic exponentiation algorithms that work by splitting the exponent in two halves for simultaneous processing while using special representations derived from signed-digit encoding that improve computational efficiency. A key detail in the design of these algorithms is that they are compatible with the idea of buffering the operations to provide resistance to SPA. Experimental results are presented, including implementations of the proposed methods with both modular integer exponentiation and elliptic curve (ECC) scalar multiplication. We also performed statistical analysis of the traces, showing that trace segments for different exponent bits are statistically indistinguishable. Our proposed techniques also exhibit better resistance against fault attacks and combined fault and side-channel attacks, compared to previous SPA-resistant techniques.

Keywords

Embedded systems security, Public-key cryptography, Elliptic curve cryptography, Side-channel attacks, Fault attacks

1. INTRODUCTION

Embedded developers face important challenges when developing systems with security requirements. This is in part due to the fact that adversaries with physical access to a device have powerful tools at their disposal, including invasive or semi-invasive reverse-engineering [28] and the non-invasive side-channel attacks [20]. One critical aspect that devices need to protect is the secrecy of cryptographic keys embedded in the device and necessary to meet the security requirements. Though the notion of physical unclonable functions (PUF) attempts to get around this issue, they do not always address the secrecy requirements of the necessary cryptographic primitives [16].

Some of the cryptographic primitives that are useful for embedded systems rely on exponentiation with large secret exponents [21]. Algorithms exist that provide secure and efficient ways to execute these required exponentiations [13]. However, *implementations* of these algorithms may be vulnerable to power analysis attacks [20], an efficient type of side-channel attack suitable when attackers have physical access to the device. Depending on the implementation, a single power trace may suffice to recover the secret parameters of the cryptosystem; this technique is known as simple power analysis (SPA). Multiple power traces may be required to extract the useful information, exploiting correlation between power consumption and the secret data, through a technique known as Differential Power Analysis (DPA) [20]. Though we mostly focus our attention on SPA, we will discuss some aspects related to DPA and its countermeasures.

SPA typically relies on data-dependent optimizations, and most SPA countermeasures introduce a performance penalty, since they typically remove some of these optimizations to avoid the vulnerability. Moreover, countermeasures that involve dummy operations for the purpose of SPA resistance in general introduce a vulnerability to induced fault attacks [6], a powerful class of attacks also suitable when potential attackers have physical access to the device.

In this work, we propose several new efficient exponentiation algorithms that work by splitting the exponent in two halves for simultaneous processing. One key detail to the algorithms' efficiency is the encodings of the exponents derived from signed-digit representations. Unlike general exponentiation algorithms that focus exclusively on execution speed, all of our algorithms are designed to be easily adapted to make them SPA-resistant. In their SPA-resistant form, our proposed algorithms also exhibit better resistance to fault and combined attacks [2] compared to existing SPA-resistant techniques. The exponent encodings include the Non-Adjacent Form (NAF) and Joint Sparse Form (JSF) for the representation of the two exponent halves. Then we adapt the technique to process blocks of multiple digits of the exponent. Unlike existing multi-digit exponentiation techniques, which focus on performance, our multi-digit approaches are also designed to provide resistance to SPA while maintaining the improvements in computational efficiency.

Our work includes implementations of several of the proposed methods, confirming their computational performance. We also extracted and statistically analyzed power traces to confirm the methods' resistance to SPA.

The remaining of this paper proceeds as follows: In Section 2, we review basic exponentiation algorithms and SPA vulnerabilities. Section 3 presents some of the existing countermeasures as well as a preliminary comparison between our work and existing techniques. Sections 4 and 5 describe our proposed techniques. We first show the methods in their SPA-vulnerable form, for the purpose of analyzing their functionality and computational efficiency, and then in Section 6, we show them in their SPA-resistant form. We also show a comparison between the various methods presented in this work and previous approaches (Section 7), with experimental results that confirm our analysis (Section 8). We then close with some concluding remarks in Section 9.

2. BINARY EXPONENTIATION AND SPA

Exponentiation algorithms — or scalar multiplication in the context of ECC [15] — exploit the properties of the exponent's binary representation. Right-to-left (R-T-L) binary exponentiation is one of the common methods [21], taking advantage of the property shown in Equation (1), for an ℓ-bits exponent e with binary representation $b_{\ell-1} \cdots b_1\, b_0$:

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

A commonly used and important optimization for this algorithm derives from the use of signed-digit representation of the exponent [3], with

$$e = \sum_{i=0}^{\ell-1} d_i 2^i, \qquad d_i \in \{\bar{1}, 0, 1\}$$

where $\bar{1} \triangleq -1$ when used to denote the value of a digit. In particular, the NAF representation minimizes the number of nonzero digits in the representation, with an average of one third of nonzero digits [3]. It is straightforward for R-T-L exponentiation to work with signed-digit exponent:

$$e = \sum_{d_i=1} 2^i - \sum_{d_i=\bar{1}} 2^i \;\Rightarrow\; x^e = \Big(\prod_{d_i=\bar{1}} x^{\left(2^i\right)}\Big) \cdot \Big(\prod_{d_i=\bar{1}} x^{\left(2^i\right)}\Big)^{-1} \quad (2)$$

Algorithm 1 shows the R-T-L algorithm with exponent in NAF representation.

Multiplications are executed conditionally on exponent digits, making Algorithm 1 vulnerable to SPA: given the multiplications' distinguishable power consumption profile, an attacker can recover the exponent from a single power trace of the operation (i.e., a "plot" of the device's power consumption) [20]. We observe that the use of NAF slows down the attack, since only nonzero vs. zero can be distinguished, but it does not entirely eliminate the vulnerability, since two thirds of the exponent bits are revealed.

Algorithm 1: R-T-L Exponentiation with NAF Exponent

```
Input: $x$;  $e = (d_{\ell-1} d_{\ell-2} \cdots d_1 d_0)_{\text{NAF}}$
Returns:  $x^e$

begin
    $S \leftarrow x$;  $R_1 \leftarrow 1$;  $R_{\bar{1}} \leftarrow 1$;
    for each digit $d_i$  ($i$ from 0 up to $\ell - 1$) do
        if $d_i \neq 0$ then
            $R_{d_i} \leftarrow R_{d_i} \times S$;
        end
        $S \leftarrow S^2$;
    end
    return $R_1 \times (R_{\bar{1}})^{-1}$;
end
```

Figure 1 shows a fragment of a power trace during an ECC scalar multiplication running on an AVR Atmega2560 microcontroller. In the context of ECC, we have doubling as the equivalent of squaring, and adding as the equivalent of multiplications. Between approx. time indexes 36 000 and 37 000 we observe two bumps (marked with arrows in the figure) separating the visually similar "dense" fragments about 5 000 samples long. Between approx. time indexes 42 000 and 44 000 we observe three bumps. Visual inspection on the complete trace showed that both doubling and adding produce a trace with distinctive sequences in the number of bumps that separate the dense fragments. In particular, only the adding exhibits three bumps. We could easily discern this pattern and read the first several bits of the exponent being used, starting with 1, 0, 1, 0, 0, 1, 1 (the figure shows only a small fragment to highlight this distinctive detail).

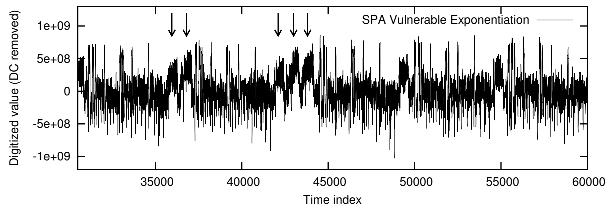


Figure 1: Fragment of Power Trace of SPA-Vulnerable Exponentiation

3. RELATED WORK

This section reviews existing work that relates to our proposed techniques, including existing SPA countermeasures, induced fault attacks and combined attacks, and differential power analysis (DPA).

As briefly mentioned in Section 2, straightforward implementations of binary exponentiation or ECC scalar multiplication algorithms are vulnerable to SPA. This is the case for both left-to-right and right-to-left forms. Coron [11] proposed the *square-and-always-multiply* (S-A-A-M) technique, in which multiplications are executed unconditionally, with the result discarded when the exponent bit is 0. In addition to the important performance penalty involved, this countermeasure introduces a vulnerability to induced fault attacks [6], as pointed out by Yen et al. [31]. Ha and Moon [14] proposed a method that exhibits SPA resistance; however, the SPA-resistant component is essentially equivalent to the S-A-A-M method, and the performance advantage is only observed when combined with the randomization approach used to introduce resistance to DPA. Even with this additional improvement, the performance of this method is below that of our proposed techniques, all of which can be easily combined with DPA countermeasures that add little or negligible computational overhead. Moreover, our techniques allow for randomization of the order of multiplications with negligible computational cost, potentially introducing some level of resistance to DPA; and through a more aggressive use of storage space, our techniques can be made fully resistant to DPA. Joye [17] (later generalized in Chevallier-Mames et al. [10]) proposed an approach where squarings are implemented as a multiplication where the two operands are the same value, effectively avoiding the vulnerability to SPA. In addition to the performance penalty due to the potential speedup in squaring operations being unused, this technique has been shown to be ineffective due to several reasons; [1] shows that the countermeasure can be bypassed through a DPA-like attack, even when blinding, one of the common DPA countermeasures, is used. The technique also introduces a vulnerability to fault attacks; more specifically, to a *combined* fault and side-channel attack, as shown by Amiel et al. [2]. The algorithms presented in [18] exhibit similar inefficiencies in terms of computational performance: all of the algorithms execute either one addition and one squaring per exponent bit (performance equivalent to S-A-A-M) or two additions per exponent bit (performance equivalent to [17]).

As we will discuss in Section 6.2, all of our techniques, which outperform the above existing techniques, introduce resistance to SPA without introducing vulnerabilities to fault or combined attacks. Any vulnerabilities to fault attacks that may come from specific implementation details such as choice of curves in ECC, initializations, etc., are independent of the implementation details of our techniques [9, 8, 5, 4]. This means that existing countermeasures against those specific attacks can be combined with the use of our proposed techniques.

The Montgomery Ladder was originally proposed by Montgomery [22] and later revisited by Joye and Yen [19] in the context of fault resistance. While Joye and Yen report important speedups due to algorithmic structure that facilitates some optimizations in the implementation, our proposed techniques outperform this technique, both from an analytic runtime complexity comparison, and in terms of real execution speed even when we incorporate the practical factors that lead to the speedup factors reported in [19]. Furthermore, Fouque et al. [12] presented a practical fault attack on this technique, in spite of the arguments presented in [19].

Moreno and Hasan [23] proposed the SABM method and combined it with the method by Sun et al. [29]; however, Sun's method is fundamentally incompatible with the use of signed-digit representation, which limits its computational efficiency. In the case of Sun's method in its original form, the presence of dummy multiplications implies that it is vulnerable to induced fault attacks, although in this case the attack is much less severe, since it only discloses pairs of zero-valued digits, which discloses no more than 25% of the exponent bits on average. All of our proposed techniques in this paper outperform these two techniques, either in their original form or in the combined form presented in [23].

4. SIMULTANEOUS PROCESSING OF HALF-EXPONENTS

We now present our proposed techniques. In this and next section we focus on their functionality and computational efficiency; in Section 6 we discuss their SPA-resistant form.

4.1 Exponent in Signed-Digit Representations

The central idea of simultaneous processing of half exponents is to split the set of values multiplied together into subsets. To this end, we use one accumulator R_{xy} for each combination xy of bits — one bit from each half-exponent in corresponding positions — where at least one of the bits is nonzero. For signed-digits $\{\bar{1},0,1\}$, we have accumulators $R_{\bar{1}\bar{1}}$, $R_{\bar{1}0}$, $R_{\bar{1}1}$, $R_{0\bar{1}}$, R_{01}, $R_{1\bar{1}}$, R_{10}, and R_{11}.

With this setup, the result of the exponentiation is computed as follows: Let e be the ℓ-digit exponent, with signed-digit representation $e = d_{\ell-1}\cdots d_1 d_0,\ \ 2\,|\,\ell$, let $\ell' = \ell/2$, let $e_{\mathrm{L}} = d_{\ell'-1}\cdots d_1 d_0$ and $e_{\mathrm{H}} = d_{\ell-1}d_{\ell-2}\cdots d_{\ell'+1}d_{\ell'}$. Then, $x^e = x^{e_{\mathrm{L}}}\cdot\left(x^{e_{\mathrm{H}}}\right)^{\left(2^{\ell'}\right)}$, with $x^{e_{\mathrm{L}}}$ and $x^{e_{\mathrm{H}}}$ computed as follows:

$$x^{e_{\mathrm{L}}} = R_{01}\cdot R_{\bar{1}1}\cdot R_{11}\cdot\left(R_{\bar{1}\bar{1}}\cdot R_{0\bar{1}}\cdot R_{1\bar{1}}\right)^{-1} \tag{3}$$
$$x^{e_{\mathrm{H}}} = R_{10}\cdot R_{1\bar{1}}\cdot R_{11}\cdot\left(R_{\bar{1}\bar{1}}\cdot R_{\bar{1}0}\cdot R_{\bar{1}1}\right)^{-1} \tag{4}$$

$$\text{where}\quad R_{\mathrm{d_H d_L}} = \prod_{\substack{i=0,\\ d_i=\mathrm{d_L},\, d_{\ell'+i}=\mathrm{d_H}}}^{\ell'-1} x^{\left(2^i\right)}$$

Algorithm Exp-HE (shown below as Algorithm 2) shows the details, with correctness asserted by Theorem 4.1.

Algorithm 2: Exp-HE – Simult. Processing of Half-Exponents

```
Input: $x$;  $e = (d_{\ell-1}\ \cdots\ d_1 d_0)_{\mathrm{S.D.}}$ with $2\,|\,\ell$, $\ell' \triangleq \frac{\ell}{2}$
Returns: $x^e$
begin
    $S \leftarrow x$;  $R_{\bar{1}\bar{1}}, R_{\bar{1}0}, R_{\bar{1}1}, R_{0\bar{1}}, R_{01}, R_{1\bar{1}}, R_{10}, R_{11} \leftarrow 1$;

    for each digit pair $d_{\ell'+i}d_i$ ($i$ from 0 up to $\ell'-1$) do
        if $d_{\ell'+i}d_i \neq 00$ then
            $R_{d_{\ell'+i}d_i} \leftarrow R_{d_{\ell'+i}d_i} \times S$;
        end
        $S \leftarrow S^2$;
    end
    $R_{01} \leftarrow R_{01}\times R_{\bar{1}1}\times R_{11}\times(R_{\bar{1}\bar{1}}\times R_{0\bar{1}}\times R_{1\bar{1}})^{-1}$;
    $R_{10} \leftarrow R_{10}\times R_{1\bar{1}}\times R_{11}\times(R_{\bar{1}\bar{1}}\times R_{\bar{1}0}\times R_{\bar{1}1})^{-1}$;

    repeat $\ell'$ times: $R_{10} \leftarrow (R_{10})^2$;
    return $R_{01}\times R_{10}$;
end
```

We observe that the combination 00 does not have a corresponding accumulator, since it does not incur a multiplication. We also notice that the products of values corresponding to positions where the digit is $\bar{1}$ need to be inverted.

Theorem 4.1: Given inputs x and ℓ-digit exponent e, with e in signed-digit representation (including NAF), Algorithm Exp-HE correctly computes the value of x^e.

Proof: Without loss of generality, we assume that $2 \mid \ell$ (we can pad with a leading zero digit as needed). Let $\ell' = \frac{\ell}{2}$, and consider the two ℓ'-digit exponent halves:

$$\begin{aligned} e_{\mathrm{H}} &= d_{\ell-1}d_{\ell-2}\cdots d_{\ell'+1}d_{\ell'} \\ e_{\mathrm{L}} &= d_{\ell'-1}d_{\ell'-2}\cdots d_2d_1d_0 \end{aligned}$$

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

$$e = e_{\mathrm{L}} + e_{\mathrm{H}}2^{\ell'} \quad \Rightarrow \quad x^e = x^{e_{\mathrm{L}}} \cdot (x^{e_{\mathrm{H}}})^{\left(2^{\ell'}\right)} \tag{5}$$

Since each of the half exponents are themselves numbers in signed-digit representation, the values $x^{e_{\mathrm{L}}}$ and $x^{e_{\mathrm{H}}}$ are given by:

$$x^{e_{\mathrm{L}}} = \left(\prod_{i \in \mathcal{D}_{\mathrm{L}}^{+}} x^{(2^i)}\right) \cdot \left(\prod_{i \in \mathcal{D}_{\mathrm{L}}^{-}} x^{(2^i)}\right)^{-1} \tag{6}$$

$$x^{e_{\mathrm{H}}} = \left(\prod_{i \in \mathcal{D}_{\mathrm{H}}^{+}} x^{(2^i)}\right) \cdot \left(\prod_{i \in \mathcal{D}_{\mathrm{H}}^{-}} x^{(2^i)}\right)^{-1} \tag{7}$$

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

Consider 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 $\mathcal{R}_{\mathrm{d_H d_L}}$ denotes the set $\{ i : 0 \leqslant i < \ell',\ d_i = \mathrm{d_L},\ d_{i+\ell'} = \mathrm{d_H} \}$.

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

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 $R_{\mathrm{d_H d_L}}$, since all of the $R_{\mathrm{d_H d_L}}$ defined are such that $\mathrm{d_H d_L}$ is not 00; this variable $R_{\mathrm{d_H d_L}}$ is precisely the one corresponding to the digit pair $\mathrm{d_H d_L}$. Thus, the values stored in each variable $R_{\mathrm{d_H d_L}}$ are:

$$R_{\mathrm{d_H d_L}} = \prod_{i \in \mathcal{R}_{\mathrm{d_H d_L}}} x^{(2^i)} \tag{8}$$

Therefore, we have

$$R_{01} \times R_{\bar{1}1} \times R_{11} = \prod_{i \in \mathcal{R}_{01} \cup \mathcal{R}_{\bar{1}1} \cup \mathcal{R}_{11}} x^{(2^i)} \tag{9}$$

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

$$R_{01} \times R_{\bar{1}1} \times R_{11} = \prod_{i \in \mathcal{D}_{\mathrm{L}}^{+}} x^{(2^i)} \tag{10}$$

We also have

$$R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}} = \prod_{i \in \mathcal{R}_{\bar{1}\bar{1}} \cup \mathcal{R}_{0\bar{1}} \cup \mathcal{R}_{1\bar{1}}} x^{(2^i)} \tag{11}$$

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

$$R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}} = \prod_{i \in \mathcal{D}_{\mathrm{L}}^{-}} x^{(2^i)} \tag{12}$$

Combining equations (10) and (12) with Equation (6), we see that the final value assigned to R_{01} is $x^{e_{\mathrm{L}}}$.

By an identical argument, we have that the value assigned to 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 R_{10} is $(x^{e_{\mathrm{H}}})^{2^{\ell'}}$. Since the output of the algorithm is the product of R_{01} and R_{10}, Equation (5) shows that the output is x^e, completing the proof. □

A fundamental distinction between our techniques and Sun's algorithm [29] is that the latter executes multiplications for all pairs of digits. This means that using signed-digits/NAF representation for the exponents with Sun's algorithm does not reduce the number of multiplications even when it reduces the number of nonzero digits. This aspect also limits the computational performance in [23]. Reducing the number of multiplications through the use of signed-digit representations is only possible if the multiplications are conditional to the exponent's digit pairs. These conditional multiplications can then be buffered to provide SPA resistance without sacrificing computational performance, as we will discuss in Section 6.1. This also leads to the possibility of multi-digit encodings, representing a substantial improvement over the results in [23] and [29].

4.2 Exponent in NAF Representation

The use of NAF for exponent representation reduces the number of multiplications in Algorithm Exp-HE:

Proposition 4.2: Given inputs x and ℓ-digit exponent e, with e in NAF representation, Algorithm Exp-HE executes $\frac{5}{18}\ell$ multiplications on average.

Proof: For each i, $0 \leqslant i < \ell'$, where $\ell' = \frac{\ell}{2}$, a multiplication takes place if $d_{i+\ell'}d_i \neq 0$. For large ℓ, we can assume $\mathrm{P}\{d_k = 0\} = \frac{2}{3}$ [3]. We can reasonably assume that digits d_i and $d_{i+\ell'}$ are independent $\Rightarrow \mathrm{P}\{d_{i+\ell'}d_i = 00\} = \frac{4}{9} \Rightarrow \mathrm{P}\{d_{i+\ell'}d_i \neq 00\} = \frac{5}{9}$. Thus, the average number of multiplications is $\frac{5}{9}\ell' = \frac{5}{18}\ell$ □

4.3 Exponent Halves in JSF Representation

Computational performance of Algorithm Exp-HE can be improved by observing that the two exponents halves can be jointly encoded using JSF. With this technique, we can obtain a joint representation for the exponent halves with one half of the columns being 00 on average [27].

Proposition 4.3: Given inputs x and ℓ-digit exponent e, with exponent halves e_{H} and e_{L} jointly represented in JSF, algorithm Exp-HE executes $\frac{1}{4}\ell$ multiplications on average.

Proof: The statement follows directly from the fact that JSF represents the exponent halves with one half of the digit pairs (columns) being 00 on average. □

5. PROCESSING MULTI-DIGIT BLOCKS

We now introduce two extensions to improve performance, where several columns are combined for simultaneous processing. As will become apparent, there are fundamental differences between these and existing techniques that use multi-digit processing for increased performance, such as those described in [21]; in our case, the characteristics of the algorithms are constrained to a structure that allows the multiplications to be buffered. Also, our proposed methods have an advantage over other multi-digit or windowing methods such as the method presented in [25] in that they do not require precomputations, making them equally well suited for the cases of fixed and non-fixed base.

5.1 Processing Two-digit Blocks with NAF

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 shown in Section 4.3.

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 $0\bar{1}$, 00, 01, 10, and $\bar{1}0$, 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.

The exponentiation algorithm is easily modified to work with this non-binary representation of ℓ_2-digit exponent $e = \sum_{i=0}^{\ell_2 -1} d_i 4^i$ with $d_i \in \{\bar{2}, \bar{1}, 0, 1, 2\}$, as shown below:

$$\begin{aligned} e &= \sum_{d_i=1} 4^i \;+\; 2\sum_{d_i=2} 4^i \;-\; \sum_{d_i=\bar{1}} 4^i \;-\; 2\sum_{d_i=\bar{2}} 4^i \\ \Rightarrow\;\; x^e &= \prod_{d_i=1} x^{4^i} \;\; \Big(\prod_{d_i=2} x^{4^i}\Big)^2 \Big(\prod_{d_i=\bar{1}} x^{4^i}\Big)^{-1} \Big(\prod_{d_i=\bar{2}} x^{4^i}\Big)^{-2} \\ &= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,(\mathbf{P}_{\bar{1}})^{-1}\,(\mathbf{P}_{\bar{2}})^{-2} \qquad (13) \end{aligned}$$

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

From Equation (13), it is clear that algorithm Exp-HE can be modified by adding additional accumulators R_{xy} for the additional digit values 2 and $\bar{2}$. That is, we need accumulators $R_{\bar{2}\bar{2}}$, $R_{\bar{2}\bar{1}}$, $R_{\bar{2}0}$, $R_{\bar{2}1}$, $R_{\bar{2}2}$, $R_{\bar{1}\bar{2}}$, etc.

Equation (13) can be rearranged to group operations depending on whether inversion is more expensive than squaring or vice-versa:

$$\begin{aligned} x^e &= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,(\mathbf{P}_{\bar{1}})^{-1}(\mathbf{P}_{\bar{2}})^{-2} \\ &= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,\big(\mathbf{P}_{\bar{1}}\,(\mathbf{P}_{\bar{2}})^2\big)^{-1} \qquad (14) \\ &= (\mathbf{P}_1)\,(\mathbf{P}_{\bar{1}})^{-1}\big(\mathbf{P}_2\,(\mathbf{P}_{\bar{2}})^{-1}\big)^2 \qquad (15) \end{aligned}$$

For example, in ECC, inversion is a virtually free operation, and thus the form in Equation (15) would be preferred; for cases where inversion is more expensive than squaring, Equation (14) is preferred, since it groups the terms to execute a single inversion.

We observe that asymptotic performance for this method is better than that obtained from the use of JSF as presented in Section 4.3. To show this, we first present the following lemma (proof shown in Appendix A), addressing adjacent digits in NAF representations:

Lemma 5.1: Let x be a randomly chosen ℓ-bit non-negative value (i.e., uniformly distributed in the interval $[0, 2^\ell - 1]$), and let $d_\ell d_{\ell-1} \cdots d_1 d_0$ be its NAF representation. For sufficiently large values of ℓ, the probability of contiguous zeros, $\mathrm{P}\{d_{k+1}d_k = 00\}$ for k even approaches $\frac{1}{3}$ as k becomes large.

Proposition 5.2: Given inputs x and ℓ-digit exponent e in NAF representation, algorithm Exp-HE-Base4 executes $\frac{2}{9}\ell$ multiplications on average, and never more than $\frac{1}{4}\ell$ multiplications.

Proof: The upper-bound of $\frac{1}{4}\ell$ follows directly from the base-4 signed-digit representation, where we never have more than one multiplication per two-digit block.

For the two-digit blocks that have four zeros, multiplication does not take place. From Lemma 5.1, we have that $\mathrm{P}\{d_{i+1}d_i = 00\} = \frac{1}{3}$. Assuming independence of the digits from the lower and upper halves of the exponent, we have that $\mathrm{P}\{d_{i+1}d_i = 00,\ d_{\ell'+i+1}d_{\ell'+i} = 00\} = \frac{1}{9}$ where $\ell' = \frac{\ell}{2}$. The average number of multiplications is $\frac{8}{9} \cdot \frac{\ell'}{2} = \frac{2}{9}\ell$. □

Due to space constraints, we omit any additional details and diagrams for this algorithm (Exp-HE-Base4); it follows the same idea as Algorithm Exp-HE, with modifications as per Equation (13).

5.2 Processing Three-digit Blocks with JSF

The technique can be extended to a base-8 procedure that deals with blocks of three digits. With a similar derivation as Equation (13), and taking into account the constraints for three-digit blocks in JSF (see below), we obtain

$$\begin{aligned} x^e \;=\; & \mathbf{P}_1(\mathbf{P}_2)^2(\mathbf{P}_3)^3(\mathbf{P}_4)^4(\mathbf{P}_5)^5(\mathbf{P}_6)^6 \\ & \big(\mathbf{P}_{\bar{1}}(\mathbf{P}_{\bar{2}})^2(\mathbf{P}_{\bar{3}})^3(\mathbf{P}_{\bar{4}})^4(\mathbf{P}_{\bar{5}})^5(\mathbf{P}_{\bar{6}})^6\big)^{-1} \end{aligned}$$

from which we see that the exponentiation procedure remains essentially the same, albeit with a higher cost in terms of storage and post-processing. However, the average number of multiplications is reduced to 0.165ℓ (one sixth due to the processing of three-bit blocks, minus the fraction of blocks that are all-zeros, which we experimentally determined to be approximately $\frac{1}{96}$). Also, the number of required accumulators is reduced from 255 (16 possible values for each digit, minus the combination 00) to 120 given that some combinations of values are not possible. This is due to the constraints in JSF representation: (1) at least one column is zero in any three contiguous columns, and (2) 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 $e_{\mathrm{H}\,i} = 0$ [27].

We do not include any additional details or diagrams since they follow the same idea as Exp-HE-Base4 and Exp-HE.

6. RESISTANCE TO PHYSICAL ATTACKS

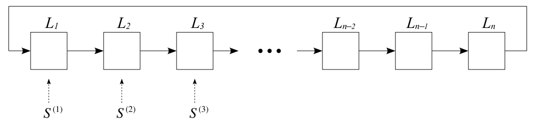
We now present the algorithms in their SPA-resistant form and discuss the methods' resistance to fault and combined attacks, as well as the potential for resistance to DPA.

6.1 Resistance to Simple Power Analysis

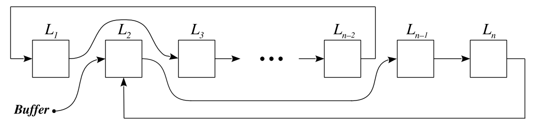
The algorithms are designed under the constraint of compatibility with the aspect of buffering the operations. This is the case because the result is obtained by multiplying a set of terms together. Thus, instead of executing the multiplication conditioned to the exponent bits, we buffer the term to be multiplied, and then execute the multiplications at a fixed rate. With a novel design of the algorithms with exponent encodings deriving from signed-digit representations, our proposed techniques achieve substantially better performance than the simple buffering technique in [23]. We also reduce the required storage space with respect to the space requirements in [23]; though we also require a buffer of size $\mathrm{O}(\sqrt{\ell}\,)$, where ℓ is the length of the exponent, the smaller variance due to signed-digits encodings leads to a smaller scaling factor (the multiplicative factor hidden in the big-Oh notation). This allows us to achieve the same probability of buffer failure with substantially lower size of the buffer. A detailed analysis is presented in [23]; notice that our techniques have the additional advantage that the buffer's size depends on *half* the exponent size, whereas the combined method in [23] cannot benefit from this aspect when combining their technique with Sun's method [29], since it cannot make use of signed-digit exponents.

Insertions in the buffer must avoid any conditional statements, since the conditional itself would leak information to the power side-channel. We observe that the actual elements (the values of S) need not be copied or moved into the buffer. Instead, insertion of a reference (e.g., a pointer in C or C++) to the element suffices. Given this low overhead in insertions, we use an additional small buffer for "fake" insertions, making the execution path of every iteration identical, regardless of whether the value of S is inserted in the buffer.

A simple data structure allows us to achieve efficient and leak-free buffer operations. The buffer storage space is represented as a linked list, as shown in Figure 2. In the diagram, L_i denotes the storage location for the (physical) i^{th} element of the buffer, and $S^{(k)}$ denotes the storage location for S at iteration k (notice that values are dynamically assigned to different elements in the buffer; the diagram shows one particular example). Contents of the buffer also use a



(a) Buffer Storage Space



(b) Buffer Usage

Figure 2: Data Structure for SPA-resistant Buffer.

linked-list structure; in Figure 2(b), storage locations L_2, L_{n-1}, and L_n correspond to the three elements that have been inserted in the buffer (to be multiplied together). Insertion and removal from the buffer is done by disconnecting the element from one sequence and connecting it to the other one (through reassignment of the appropriate pointers).

Algorithm SABM-HE (shown as Algorithm 3) shows a sketch of the complete procedure for the case of exponent in NAF representation. Notice that we need to pre-fill the buffer to half its capacity on average ($\mathrm{P}\{\text{Insertion}\} = \frac{5}{9}$) before extracting elements from it. This is to avoid buffer underflow. We also need to process any elements left in the buffer after the loop completes.

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.

6.2 Resistance to Fault and Combined Attacks

Our proposed methods also exhibit better resistance against fault attacks and combined fault and side-channel attacks compared to existing SPA countermeasures. Yen et al. [31] noted that the S-A-A-M countermeasure creates an easy-to-exploit vulnerability to fault attacks: since "dummy" operations are executed depending on exponent bits, injected faults allow an attacker to determine whether an operation was a dummy, thus revealing exponent bits. We notice

Algorithm 3: SABM-HE (NAF Exponent) – Sketch

```
begin
    Initialize variables and accumulators;
    for each digit pair $d_{\ell'+i}d_i$ ($i$ from 0 up to $\ell'-1$) do
        Insert or fake-insert element;
        $S \leftarrow S^2$;
        if $i > 9 \cdot |\mathrm{Buff_S}|/10$ and $i$ mod 9 is even then
            $\langle Tmp, d_{\mathrm{H}}d_{\mathrm{L}}\rangle \leftarrow \mathrm{Buff_S}$;
            $R_{d_{\mathrm{H}}d_{\mathrm{L}}} \leftarrow R_{d_{\mathrm{H}}d_{\mathrm{L}}} \times Tmp$;
        end
    end
    Process remaining elements in the buffer;
    Combine accumulators to produce result;
end
```

that our methods are naturally immune to this type of fault attack, since no dummy operations are executed. Furthermore, the fact that elements in the buffer can be multiplied in a randomized order somewhat enhances this immunity to this type of attack.

Amiel et al. [2] show a combined attack, where by injecting a single fault at the beginning of the exponentiation, a single power trace reveals the exponent in an otherwise SPA-resistant algorithm. Though this combined attack is specific to the types of algorithms in [10], where the use of R_1 introduces the vulnerability, the idea that they present is clever and should be considered when designing SPA-resistant techniques. Clearly, since our algorithms do not follow the pattern where a modified value of R_1 creates the SPA vulnerability, our techniques are not vulnerable to this combined attack in [2]. More in general, their combined attack is based on the following idea: the algorithms in [10] work by hiding the multiplications from the power trace, yet an injected fault makes the multiplications visible in the power trace. In contrast, our techniques do not hide the multiplications from the power trace; instead, they make the positions of these multiplications *independent* of the exponent bits. Thus, even the general idea presented in [2] is not applicable to our proposed algorithms. Furthermore, the fact that multiplications can be performed in a randomized order provides additional protection against combined attacks — even if an injected fault caused the power trace to reveal exponent bits, with our methods the bits would be revealed in a randomly permuted order.

As an additional aspect, our techniques have the natural advantage over existing methods that because they combine two exponent bits for simultaneous processing, the only information leaked is whether or not pairs of signed digits are both zero. Thus, any attacks that exploit this leak (even yet-to-be-discovered attacks) would be at the very least slowed down considerably.

6.3 Resistance to DPA

Though not the main focus in our work, it is important to highlight that our techniques exhibit the potential to be resistant to DPA. This is due to the possibility of randomizing the order of the multiplications in the buffer. This randomization removes any correlation between multiple traces, which is an essential aspect on which DPA techniques rely. In its space-optimized form, where the buffer size is $\mathrm{O}(\sqrt{\ell}\,)$, the scope of the randomization is limited, and thus the effectiveness of the randomization against DPA attacks is unknown. However, the designers of a system can decide to increase this buffer size, possibly up to the size corresponding to the average number of multiplications, thus making the operations fully resistant to DPA. This is a reasonable possibility, since storage space has increasingly become affordable in recent decades, and even low-cost microcontrollers nowadays include generous amounts of memory.

Unlike other forms of randomization proposed in the literature (for example, blinding countermeasures suggested in [20] or exponent randomization suggested in [11]), the form of randomization that our techniques allow involves negligible computational overhead. For embedded devices, savings in computational cost may be much more important than savings in storage space, especially for battery-powered devices, since increased computations translate into increased power consumption.

7. PERFORMANCE COMPARISON

Our methods achieve better computational performance than any existing SPA-resistant methods. While this improved computational performance comes at the cost of additional storage, this can be reasonable for embedded devices, especially those relying on battery-supplied power. This is the case because storage has become a less expensive commodity in modern systems, yet increased computations always translate into increased power consumption that reduces battery life.

Moreover, we observe that even though our exponentiation algorithms require a larger number of accumulators with respect to the work in [23], we actually reduce the total amount of storage, since the required size of the buffer is smaller in our case, as discussed in Section 6.1.

Our methods also outperform the Montgomery ladder [19], even with the optimizations suggested in that work. In particular, the most important speedup mentioned in [19] comes from the parallelizable nature of that method (up to two simultaneous operations); all of our methods are parallelizable up to two threads of execution (the buffered multiplications can be done in parallel), maintaining the advantage in the algorithmic run time.

Table 1 summarizes the differences in performance of the various proposed techniques and compares against existing solutions. We show the fully optimized (and thus SPA-vulnerable) R-T-L exponentiation performance as a baseline. Values shown are *average* number of multiplications required to execute an exponentiation with an ℓ-bit exponent (we recall that all the methods listed, except for Joye's method, require exactly ℓ squarings in addition to the number of multiplications shown), and *average* amount of units of time to execute, where our convention is that squaring routines execute in 1 unit of time and multiplication routines execute in 1.5 units of time (a typical ratio for practical implementations). We omit the Montgomery ladder in this analytic comparison since the algorithmic performance is not its strength, and it is rather the practical aspects derived from its structure that makes it attractive. Even with the speedups reported in [19], our methods outperform it.

8. EXPERIMENTAL RESULTS

We implemented several of the methods for the purpose of experimentally verifying their efficiency and also to better illustrate the effectiveness of the countermeasure.

	Multiplications		**Execution Time**	
	Binary	NAF/S.D.	Binary	NAF/S.D.
R-T-L	0.5ℓ	0.33ℓ	1.75ℓ	1.5ℓ
S-A-A-M	ℓ	ℓ	2.5ℓ	2.5ℓ
Joye	$--$	$--$	2.25ℓ	2ℓ
Sun et al.	$0.5\ell + O(1)$	$--$	$1.75\ell + O(1)$	$--$
SABM	0.5ℓ	0.33ℓ	1.75ℓ	1.5ℓ
SABM-HE $^{(*)}$	$0.375\ell + O(1)$	$0.275\ell + O(1)$	$1.56\ell + O(1)$	$1.416\ell + O(1)$
SABM-HE-JSF $^{(*)}$	$--$	$0.25\ell + O(1)$	$--$	$1.375\ell + O(1)$
SABM-HE-Base4 $^{(*)}$	$--$	$0.22\ell + O(1)$	$--$	$1.33\ell + O(1)$
SABM-HE-Base8 $^{(*)}$	$--$	$\mathbf{0.165\ell + O(1)}$	$--$	$\mathbf{1.247\ell + O(1)}$

$(*)$ This work

Table 1: Analytic performance comparison.

8.1 Performance

In addition to the fully optimized SPA-vulnerable R-T-L 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. The implementations include ECC over binary fields using the MIRACL library [26] and integer arithmetic using the GMP library [30]. For ECC, we used NIST curves [24] at 163, 283, and 571 bits sizes to cover the standard range. For integer arithmetic we included 1024 and 2048 bits, which are in the typical range of RSA applications [7], assuming a CRT-based implementation. We also included 4096-bit since it confirms the asymptotic behavior expected for larger exponents.

Table 2 shows the results; measurements are actual execution time of the exponentiation routines (excluding startup and initialization time for the library facilities). Multiple measurements (10 000) with randomly chosen exponents were performed, and Table 2 shows the average values along with 99.9% confidence intervals. The implementations were compiled and executed on a Raspberry Pi with a 700MHz ARMv6 processor, using the gcc 4.6.3 compiler present in the system.

$\mathbf{ECC - GF(2^m)}$	**163 bits**	**283 bits**	**571 bits**
R-T-L (vulnerable)	8.50 ± 0.009	26.26 ± 0.07	148.5 ± 0.08
Exp-HE (NAF)	7.99 ± 0.008	25.54 ± 0.07	136.6 ± 0.08
Exp-HE-JSF	7.72 ± 0.006	23.45 ± 0.05	130.7 ± 0.06
Exp-HE-Base4	7.99 ± 0.008	23.91 ± 0.05	129.8 ± 0.05

Integer Arithmetic	**1024 bits**	**2048 bits**	**4096 bits**
R-T-L (vulnerable)	15.61 ± 0.005	97.27 ± 0.02	584.99 ± 0.075
Exp-HE (NAF)	15.10 ± 0.005	93.13 ± 0.02	558.90 ± 0.077
Exp-HE-JSF	14.68 ± 0.004	90.72 ± 0.01	545.92 ± 0.05
Exp-HE-Base4	14.60 ± 0.004	89.07 ± 0.01	532.76 ± 0.04

All values in milliseconds. Values after the ± sign indicate 99.9% confidence interval.

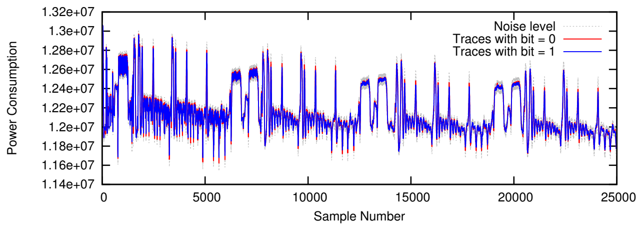
Table 2: Execution time of exponentiation.

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, but it is asymptotically more efficient, as shown by the results for larger exponent sizes.

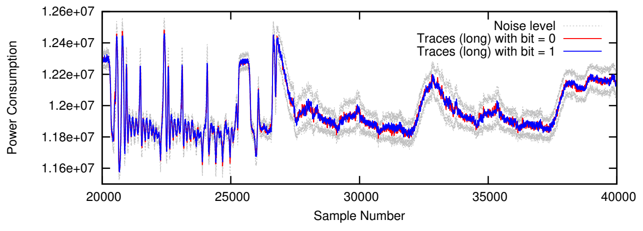
8.2 SPA Resistance

Our experiments also include capture and analysis of power traces to show that the implementation, including the conditional buffering aspect, is resistant to SPA. To this end, we implemented a scalar multiplication over GF (2^{163}) on an AVR Atmega2560 8-bit microcontroller at 1 MHz. We chose a different platform for this experiment since power analysis can be done with higher precision and less difficulty on a system running in the order of megahertz, compared to hundreds of megahertz. This way, we show that even with this advantage leading to a more powerful attack, we confirm that the countermeasure is effective against the attack.

Figure 3 shows a comparison of averaged traces for one iteration of the algorithm (one bit of the exponent/scalar), comparing traces where the bit is zero vs. those where the bit is nonzero (units are not indicated since these are the digitized values with a somewhat arbitrary scaling factor). The differences fall well below the noise level (notice that the blue traces, for bit=1, are plotted on top of the red ones, hiding them almost completely). With the buffering technique,



(a) Short Iterations (Doubling only).



(b) Long Iterations (Doubling + Addition).

Figure 3: Averaged Power Traces.

we have iterations where there is just one doubling and iterations with a doubling and an addition — "short" and "long" traces, respectively. The fragment shown in Figure 3(b) corresponds to the end of the squaring portion (approx. up to sample 27 000), followed by the addition. Only a fragment of the trace is shown since it is too long to properly display in these figures.

We would like to highlight the aspect that our technique proved effective even against an unrealistically powerful attack. The trace captures and stacking benefit from an artificial advantage given by an instrumented version of the scalar multiplication: an auxiliary signal is output to mark the beginning of each iteration. The two signals (power consumption and markers) are captured and the markers are used to segment the trace into the fragments corresponding to each exponent bit. Additionally, we used knowledge of the exponent to partition the traces based on the exponent bit being zero vs. exponent bit being nonzero. Clearly, this represents an unrealistic advantage (compared to a practical SPA attack) in terms of being able to align and combine the traces for different exponent bits and reduce the effect of the noise. The plots show that even with this advantage, the traces where the secret bit is zero are indistinguishable (to an SPA attacker) from those where the bit is nonzero.

In addition to comparing the traces as shown in Figure 3, we also performed statistical analysis on the power consumption. This analysis revealed no statistically significant difference between the different inputs. We collected about 300 samples for each of the 110 476 points (equally spaced, corresponding to all samples of the digitized signal) in the power trace. Since the points exhibited not normally distributed data (visual inspection on a q-q plot), we decided to use the non-parametric Mann-Whitney test with a Bonferroni correction ($p = 0.01/3$). None of the points show a statistically significant difference between the two inputs.

9. CONCLUSIONS

In this work, we have presented several new methods that efficiently perform binary exponentiation while exhibiting resistance to SPA. The methods provide substantial improvements in computational efficiency with respect to existing SPA-resistant methods. They also exhibit better resistance to fault attacks, and combined fault and side-channel attacks. Resistance to DPA can be added as well, through adjustments in the implementation parameters.

Experimental results were also presented, including power trace captures and analysis that provides solid evidence to our methods' resistance to SPA.

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. For applications requiring large exponent sizes, such as RSA with the currently NIST-recommended key sizes, these multi-digit methods can offer a considerable advantage in performance.

Acknowledgements

The authors would like to thank our anonymous reviewers for valuable feedback and suggestions.

This work was supported in part through an NSERC grant awarded to Dr. Hasan, and also supported in part by CFI 20314 and CMC, and the industrial partners associated with these projects.

10. REFERENCES

[1] F. Amiel, B. Feix, M. Tunstall, C. Whelan, and W. P. Marnane. Distinguishing Multiplications from Squaring Operations. In *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2009.

[2] F. Amiel, K. Villegas, B. Feix, and L. Marcel. Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In *Fault Diagnosis and Tolerance in Cryptography*, 2007.

[3] S. Arno and F. Wheeler. Signed Digit Representations of Minimal Hamming Weight. *IEEE Transactions on Computers*, 42(8), 1993.

[4] A. Barenghi, G. Bertoni, A. Palomba, and R. Susella. A Novel Fault Attack Against ECDSA. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 161–166, 2011.

[5] A. Barenghi, G. M. Bertoni, L. Breveglieri, G. Pelosi, and A. Palomba. Fault Attack to the Elliptic Curve

Digital Signature Algorithm with Multiple Bit Faults. In *Proceedings of the 4th International Conference on Security of Information and Networks*, SIN '11, pages 63–72. ACM, 2011.

[6] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11), Nov 2012.

[7] E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A, 2011.

[8] I. Biehl, B. Meyer, and V. Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In *Advances in Cryptology – CRYPTO 2000*, volume 1880, pages 131–146. Springer Berlin Heidelberg, 2000.

[9] J. Blömer, M. Otto, and J.-P. Seifert. Sign Change Fault Attacks on Elliptic Curve Cryptosystems. In *Fault Diagnosis and Tolerance in Cryptography*, volume 4236, pages 36–52. Springer Berlin Heidelberg, 2006.

[10] 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.

[11] J.-S. Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. *CHES-99*.

[12] P. Fouque, R. Lercier, D. Real, and F. Valette. Fault Attack on Elliptic Curve Montgomery Ladder Implementation. In *5th Workshop on Fault Diagnosis and Tolerance in Cryptography – FDTC '08.*, pages 92–98, Aug 2008.

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

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

[15] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

[16] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas. Physical Unclonable Functions and Applications: A Tutorial. *Proceedings of the IEEE*, 102(8), 2014.

[17] M. Joye. Recovering Lost Efficiency of Exponentiation Algorithms on Smart Cards. *Electronic Letters*, 2002.

[18] M. Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In *Cryptographic Hardware and Embedded Systems – CHES 2007*, pages 135–147. Springer Berlin Heidelberg, 2007.

[19] M. Joye and S.-M. Yen. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems – CHES 2002*, pages 291–302, 2002.

[20] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Advances in Cryptology*, 1999.

[21] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[22] P. L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[23] C. Moreno and A. Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *Journal of Cryptographic Engineering*, 2011.

[24] NIST. Recommended Elliptic Curves for Federal Government Use, 1999. `http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf`.

[25] K. Okeya and T. Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks. In *CT-RSA*, volume 2612, 2003.

[26] M. Scott. MIRACL: Multi-Precision Integer and Rational Arithmetic C/C++ Library. `http://www.certivox.com/miracl`.

[27] J. A. Solinas. Low-Weight Binary Representations for Pairs of Integers. *CACR 2001-41*, 2001.

[28] D. Strobel, D. Oswald, B. Richter, F. Schellenberg, and C. Paar. Microcontrollers as (In)Security Devices for Pervasive Computing Applications. *Proceedings of the IEEE*, 102(8), 2014.

[29] D.-Z. Sun, J.-P. Huai, J.-Z. Sun, and Z.-F. Cao. An Efficient Modular Exponentiation Algorithm against Simple Power Analysis Attacks. *IEEE Transactions on Consumer Electronics*, 53(4), 2007.

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

[31] S.-M. Yen, S. Kim, S. Lim, and S. Moon. A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In *Information Security and Cryptology*, pages 414–427, 2002.

APPENDIX

A. PROOF FOR LEMMA 5.1

We will prove the statement based on a procedure to construct the NAF representation from the standard binary representation by processing individual bits from right to left (LSB to MSB) to generate the digits d_i. Since the procedure is different from the standard, commonly known algorithm (shown for example as Algorithm 3.30 in [21]), we include a description in Appendix B.

The procedure works in an "online" manner, processing each input bit to generate new output digits, with the key detail that each bit in the standard binary representation is independent of every other bit (even for adjacent bits) and can take values 0 and 1 with equal probability.

This allows us to obtain the following recurrence relations for the possible outputs of the algorithm at each step (i.e., upon processing each input bit), taking into account that $\mathrm{P}\{b_n = 0\} = \mathrm{P}\{b_n = 1\} = \frac{1}{2}$; $P_0(k)$ denotes the probability of producing a digit 0 with no carry at iteration k (that is, $\mathrm{P}\{d_k = 0, \mathrm{C} = 0\}$), $P_1(k) = \mathrm{P}\{d_k = 1\}$, and $P_\mathrm{C}(k)$ denotes the probability of producing a carry at iteration k:

$$\begin{aligned} P_0(n) &= \mathrm{P}\{b_n = 0\}\, P_0(n-1) + \mathrm{P}\{b_n = 0\}\, P_1(n-1) \\ &= \tfrac{1}{2} P_0(n-1) + \tfrac{1}{2} P_1(n-1) \qquad (16) \\ P_\mathrm{C}(n) &= \mathrm{P}\{b_n = 1\}\, P_1(n-1) + \mathrm{P}\{b_n = 1\}\, P_\mathrm{C}(n-1) \\ &= \tfrac{1}{2} P_1(n-1) + \tfrac{1}{2} P_\mathrm{C}(n-1) \qquad (17) \\ P_1(n) &= 1 - P_0(n) - P_\mathrm{C}(n) \qquad (18) \end{aligned}$$

Equation (18) corresponds to the fact that these are the only three possibilities at each iteration, so the corresponding probabilities must add to 1.

Since we are interested in the probability of adjacent zeros, we only need to solve for $P_0(n)$ and $P_\mathrm{C}(n)$, so we rewrite Equation (18) at $n-1$ and substitute in the other two equations, obtaining

$$\begin{aligned} P_0(n) &= \tfrac{1}{2}(1 - P_\mathrm{C}(n-1)) \\ P_\mathrm{C}(n) &= \tfrac{1}{2}(1 - P_0(n-1)) \end{aligned}$$

Rewriting again for the left-hand sides at $n-1$ and substituting in each other, we finally obtain the following recurrence relations:

$$\begin{aligned} P_0(n) &= \tfrac{1}{4} + \tfrac{1}{4}P_0(n-2) \\ P_\mathrm{C}(n) &= \tfrac{1}{4} + \tfrac{1}{4}P_\mathrm{C}(n-2) \end{aligned}$$

With initial conditions being $P_0(0) = P_0(1) = \frac{1}{2}$, $P_\mathrm{C}(0) = 0$, and $P_\mathrm{C}(1) = \frac{1}{4}$ (these are trivially obtained by counting occurrences in the four possible two-bit combinations).

The above recurrence relations are easily solved by repeated substitution, obtaining identical solutions for both (the difference given by the different initial conditions):

$$P(n) = \tfrac{1}{3} + \left(\tfrac{1}{2}\right)^n \left(P(0) - \tfrac{1}{3}\right) \qquad \text{(for } n \text{ even)}$$

Since we are interested in processing pairs of bits, we want the probability of digits $d_{2k}d_{2k+1}$ being 00, and thus, we only need to obtain the above solution for n even.

$$\begin{aligned} P_0(n) &= \tfrac{1}{3} + \tfrac{1}{6}\left(\tfrac{1}{2}\right)^n \\ P_\mathrm{C}(n) &= \tfrac{1}{3}\left(1 - \left(\tfrac{1}{2}\right)^n\right) \end{aligned}$$

From this, we obtain, for n even:

$$\begin{aligned} \mathrm{P}\left\{d_{n+1} = 0, d_n = 0\right\} &= \mathrm{P}\left\{b_{n+1} = 0, d_n = 0\right\} \\ &\quad + \mathrm{P}\left\{b_{n+1} = 1, C = 0\right\} \\ &= \mathrm{P}\left\{b_{n+1} = 0\right\} \cdot P_0(n) \\ &\quad + \mathrm{P}\left\{b_{n+1} = 1\right\} \cdot P_\mathrm{C}(n) \\ &= \tfrac{1}{2}(P_0(n) + P_\mathrm{C}(n)) \\ &= \tfrac{1}{3}\left(1 - \left(\tfrac{1}{2}\right)^{n+2}\right) \qquad (19) \end{aligned}$$

From Equation (19), we clearly observe exponential convergence towards $\frac{1}{3}$.

To complete the proof, we should mention the fact that these probabilities correspond to the probabilities of 0 at the given positions for the NAF representations of large numbers. Indeed, the construction procedure (see Appendix B) is such that once the most-significant-digit at some iteration is 0, the next iteration can not make this change (and as a consequence, the same holds for the two most recent digits being 00), and this regardless of whether there is a carry or not at that iteration. Conversely, if we have a 1 at some iteration (the only possible nonzero as the most-significant digit), the next iteration can only make it change to $\bar{1}$, and not to 0. $\square$

B. ONLINE COMPUTATION OF THE NAF

We describe a simple online procedure to obtain the NAF representation of a non-negative integer with ℓ-bit binary representation, processing each input bit independently, updating the output accordingly. Thus, we treat the conversion from standard binary to NAF as a transformation applied to the output of a random source of independent and uniformly distributed bits.

After processing bit b_{n-1}, we have output $d_{n-1}d_{n-2}\cdots d_1 d_0$ with the possibility of a carry (at the very end, after processing bit $b_{\ell-1}$, this carry would correspond to digit d_ℓ). We observe that this output after processing bit b_{n-1} is the NAF representation of the n-bit non-negative integer $b_{n-1}b_{n-2}\cdots b_1 b_0$ that has been processed so far.

Since the values considered are always non-negative, the most-significant digit at the end of each iteration can not be $\bar{1}$ — if it was, then there would be no carry, since the output is a valid NAF representation, and NAF does not allow adjacent non-zero digits, and thus the represented value would be negative. Also, for the same reason, a carry can only occur if $d_{n-1} = 0$.

Thus, after processing bit b_{n-1}, the output digit d_{n-1} and carry c_{n-1} can only be

$$(d_{n-1}, c_{n-1}) = \begin{cases} (0,0) \\ (0,1) \\ (1,0) \end{cases}$$

We now consider the effect of processing bit b_n. If $b_n = 0$, then clearly $d_n = c_{n-1}$ (the carry from the previous iteration), and no carry can result from processing bit b_n. If $b_n = 1$, then, if $d_{n-1} = 0$ with no carry, we would have $d_n = 1$ with no carry produced. If there is a carry from the previous iteration, then we add b_n and the carry, obtaining a value 10_2 aligned at position n — that is, $d_n = 0$ with a carry produced at this iteration. Finally, if $d_{n-1} = 1$, then we have to substitute the resulting 11_2, since NAF precludes it. This is fixed by substituting 11_2 by its NAF equivalent, $10\bar{1}$ aligned at the same position; that is, we would replace the value of d_{n-1} with $\bar{1}$, $d_n = 0$ and a carry is produced at this iteration.

The procedure is necessarily correct given that: (1) it does output a valid signed-digit representation of the value represented by $b_{\ell-1}b_{\ell-2}\cdots b_1 b_0$ in standard binary — indeed, every operation that modifies the output replaces blocks of digits with a different block representing the same value and aligned at the same position; and (2) by construction, this output does not have adjacent non-zero digits. Since we know that NAF representation is unique [3], then the output of this procedure must be *the* NAF representation of the input value.