Recovering Lost Efficiency of Exponentiation Algorithms on Smart Cards

[Published in *Electronics Letters* **38**(19):1095–1097, 2002.]

Marc Joye

Gemplus Card International
B.P. 100, 13881 Gémenos Cedex, France
E-mail: `marc.joye@gemplus.com`

Abstract. When it comes to implementation, a major security concern is the resistance against the so-called side-channel attacks. Solutions are known but they increase the overall complexity by a non-negligible factor (typically, a protected RSA exponentiation is 133% slower). For the first time, this Letter proposes protected solutions that do not penalize the running time of an exponentiation.

Keywords. Information theory, cryptography, exponentiation, RSA, side-channel attacks, SPA, elliptic curves, smart cards.

1 Introduction

The basic operation of most public-key cryptosystems is the exponentiation (or the scalar multiplication for additively written sets, such as the points on an elliptic curve). This is for example the case of the widely-used RSA cryptosystem. When properly used, it can be shown that the RSA achieves indistinguishability against adaptive chosen-ciphertext attacks [1]. This is the strongest security notion one can hope for a public-key encryption scheme. However, in an unskilled implementation, a power attack [2] can easily recover a whole RSA secret key. To thwart such kinds of attacks, it is recommended to avoid (secret-)data-dependent executions of a given crypto-algorithm. There are known solutions but they increase the running time by a non-negligible factor. This Letter, rather, presents efficient and virtually free solutions towards resistance against power-like attacks for exponentiation-based cryptosystems.

2 Review of Exponentiation Algorithms

For smart cards, the most commonly-used algorithms for computing $y = x^r$ are based on *binary methods* [3, Section 4.6.3]. These algorithms come in two flavors

Input: $x, r = (r_{m-1}, \ldots, r_0)_2$
Output: $y = x^r$

$R_0 \leftarrow 1$; $R_1 \leftarrow x$
for $i = 0$ to $m-1$ do
  if $(r_i = 1)$ then $R_0 \leftarrow R_0 \cdot R_1$
  $R_1 \leftarrow (R_1)^2$
return R_0

(a) Right-to-left

Input: $x, r = (r_{m-1}, \ldots, r_0)_2$
Output: $y = x^r$

$R_0 \leftarrow 1$
for $i = m-1$ down to 0 do
  $R_0 \leftarrow (R_0)^2$
  if $(r_i = 1)$ then $R_0 \leftarrow R_0 \cdot x$
return R_0

(b) Left-to-right (a.k.a. square-and-multiply)

Fig. 1. Binary algorithms

according to the bits of the exponent are scanned from the right to left or from the left to the right.

We remark that the right-to-left algorithm (Fig. 1-a) needs two temporary registers whereas the left-to-right algorithm (Fig. 1-b), also known as *square-and-multiply algorithm*, just needs one. Assuming that a squaring is as costly as a multiplication, both algorithms require $\frac{3}{2}\,m = 1.5\,m$ multiplications, on average.

When the computation of an inverse is free, as is the case for elliptic curves, the expected number of operations can be lowered to $\frac{4}{3}\,m \approx 1.33\,m$ from the value of x^{-1} [4]. This is a straightforward generalization of the square-and-multiply algorithm.

3 Power-like Attacks

At Crypto '99, Kocher *et al.* [2] introduced the so-called *power analysis attacks*. By measuring the power consumption, they were able to find the secret keys embedded in tamper-resistant devices. When only a single measurement is performed the attack is referred to as an *SPA attack*, and when they are several correlated measurements it is referred to as a *DPA attack*. The main concern for public-key cryptography is the SPA-like attack since a DPA-like attack against an exponentiation operation can easily be avoided by randomizing the operands. We refer the reader to [2] for further details.

The algorithms presented in Figure 1 are trivially susceptible to this type of attacks since the operations depends on the bits of the (secret) exponent. To avoid SPA-like attacks, programmers suggested to replace the square-and-multiply algorithm by the *square-and-multiply-always algorithm* (see Fig. 2).

In this algorithm, a dummy multiplication is performed when the bit-value is '0'. Unfortunately, the performances of the resulting algorithm drop down to $2\,m$ multiplications instead of $1.5\,m$ multiplications. Moreover, it requires an additional temporary register, R_1.

The next section investigates new ways to recover the efficiency of the original algorithms, that is, implementations resistant against SPA-like attacks without using dummy multiplications.

```
Input: x, r = (r_{m-1}, ... , r_0)_2
Output: y = x^r
  R_0 ← 1
  for i = m − 1 down to 0 do
    R_0 ← (R_0)^2
    b ← ¬r_i; R_b ← R_b · x
return R_0
```

Fig. 2. Square-and-multiply-*always* algorithm

4 New Proposals

If we take a closer look at the (standard) right-to-left binary algorithm (Fig. 1-a), we see that there is first a multiplication if the value of the scanned bit is 1, always followed by a squaring. So the idea to make this code constant is to scan twice a bit when its value is 1 and then to rewrite it to 0: as before a '1' corresponds to a multiplication and a '0' to a squaring. They are several possible implementations of this idea. An example is given in Figure 3. As a side effect, we remark that after the execution of the algorithm, the whole value of the exponent is zero-ified. Because the exponent is usually first recopied in RAM memory and represents a secret data, this is a highly desired property.

```
Input: x, r = (r_{m-1}, ... , r_0)_2
Output: y = x^r
  R_0 ← 1; R_1 ← x; i ← 0
  while (i ≤ m − 1) do
    b ← ¬r_i
    R_b ← R_b · R_1; r_i ← 0; i ← i + b
return R_0
```

Fig. 3. SPA-protected right-to-left binary algorithm

The right-to-left algorithm has the disadvantage that the value of x is lost after the computation of $y = x^r$. This is not the case with the left-to-right algorithm. Transposing the algorithm of Figure 1-b is nevertheless less trivial because the data-independent operation (i.e., the squaring) is performed *prior to* the data-dependent operation (i.e., the multiplication by x). However, remarking that the first squaring yields $R_0 = 1^2 = 1$ and neglecting the last bit (r_0), the order of the square and the multiply operations can be exchanged. The resulting algorithm is given in Figure 4.
One could argue that this is not code-constant because of the last "if-then" instruction. However, in the case of RSA (in both standard and CRT modes) this

```
Input: x, r = (r_{m-1}, ... , r_0)_2
Output: y = x^r

  R_0 ← 1; R_1 ← x; i ← m − 1
  while (i ≥ 1) do
    b ← ¬r_i
    R_0 ← R_0 · R_{r_i}; r_i ← 0; i ← i − b
  if (r_0 = 1) then R_0 ← R_0 · R_1
return R_0
```

Fig. 4. SPA-protected square-and-multiply algorithm

value is always 1 (even if the exponent is randomized!). For other cryptosystems, implementation-dependent tricks may be used to avoid the leakage of the value of bit r_0.

In certain implementations, result-in-place is not allowed: operations such as $R_0 \leftarrow R_0 \cdot R_b$ are forbidden. We present in Figure 5 an SPA-protected implementation of the square-and-multiply algorithm without result-in-place. The right-to-left binary algorithm is adapted similarly.

```
Input: x, r = (r_{m-1}, ... , r_0)_2
Output: y = x^r

  R_1 ← 1; R_2 ← x; t ← 0; i ← m − 1
  while (i ≥ 1) do
    b ← ¬r_i; t ← ¬t
    R_{¬t} ← R_t · R_{2r_i+tb}; r_i ← 0; i ← i − b
  R_t ← R_{¬t} · R_2
return R_{¬t⊕r_0}
```

Fig. 5. SPA-protected square-and-multiply algorithm w/o result-in-place

Some words of explanation are needed. At each iteration, registers R_0 and R_1 successively (i.e., $t \leftarrow \neg t$) contain the result of the multiplication. When $r_i = 0$ then $R_{\neg t} \leftarrow R_t \cdot R_t$ (square) and when $r_i = 1$ then $R_{\neg t} \leftarrow R_t \cdot R_2$ (multiply). Finally, if $r_0 = 0$, the final result is in $R_{\neg t}$; otherwise, one has to multiply $R_{\neg t}$ by R_2. Remark that, without result-in-place, the value of the last bit of the exponent, r_0, does not leak and that the value of x is still available in register R_2.

Another case of interest in exponentiation techniques is when the computation of the inverse of an element is virtually free, as is the case for elliptic curves [4]. The basic square-and-multiply algorithm can then be advantageously replaced by a *square-and-multiply-or-divide method*. Making such an algorithm

```
Input: x, r = (r_{m-1}, ..., r_0)_SD2
Output: y = x^r
  R_0 ← 1; R_1 ← x; R_2 ← x^{-1}; i ← m − 1
  while (i ≥ 1) do
    b ← ¬r_{i,L}
    R_0 ← R_0 · R_{r_{i,H}+r_{i,L}}; r_i ← 0; i ← i − b
  R_2 ← R_0 · R_{r_{0,H}+r_{0,L}}
return R_{2r_{0,L}}
```

Fig. 6. SPA-protected square-and-multiply-or-divide algorithm

```
Input: x, r = (r_{m-1}, ..., r_0)_SD2
Output: y = x^r
  R_1 ← 1; R_2 ← x; t ← 0; i ← m − 1
  while (i ≥ 1) do
    b ← ¬r_{i,L}; t ← ¬t
    g ← 2 · r_{i,H} + ¬t · ¬r_{i,H}; R_g ← (R_g)^{-1}
    R_{¬t} ← R_t · R_{2r_{i,L}+tb}
    g ← 2 · r_{i,H} + t · ¬r_{i,H}; R_g ← (R_g)^{-1}
    r_i ← 0; i ← i − b
  g ← 2 · r_{0,H} + t · ¬r_{0,H}; R_g ← (R_g)^{-1}
  R_t ← R_{¬t} · R_2
  g ← 2 · r_{0,H} + ¬t · ¬r_{0,H}; R_g ← (R_g)^{-1}
return R_{¬t⊕r_{0,L}}
```

Fig. 7. Memory-efficient SPA-protected square-and-multiply-or-divide algorithm

resistant against SPA-like attacks is a straightforward generalization of our algorithm given in Figure 4. We assume that exponent r is given in a binary signed-digit representation (SD2), that is, with digits r_i in the set $\{-1, 0, 1\}$. We further assume that the digits -1, 0 and 1 are represented as 11, 00 and 01, respectively. The lower bit (bit of value) representing r_i is denoted by $r_{i,L}$ and its higher bit (bit of sign) by $r_{i,H}$. If exponent r is given in its binary representation then one can apply the algorithm of [5] to obtain, digit-by-digit, a minimal binary signed-digit representation for r from the left to the right.

The resulting algorithm (Fig. 6) requires $\frac{4}{3}\,m \approx 1.33\,m$ multiplications, on average. We can, however, further improves the algorithm, memory-wise, by using the same register for x and x^{-1}. Remember that we made the assumption that the computation of x^{-1} is very cheap. We give in Figure 7 the trick for an implementation without result-in-place. Suppose that register R_2 initially contains x. If $r_i = -1$ then we replace the value of register R_2 by its inverse, namely x^{-1}; otherwise we invert the content of the register that will be overwritten. Next, after the multiplication, we re-put x into register R_2.

5 Conclusion

This Letter presented detailed implementations towards resistance against SPA-like attacks. The main advantage of our solutions is that the overall complexity of the resulting algorithms is broadly the same as that of the classical (i.e., unprotected) implementations.

References

1. PKCS #1 v2.0 RSA cryptography standard. RSA Laboratories, October 1998. Available at `<http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-1.html>`.
2. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, vol. 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
3. Donald E. Knuth. *The art of computer programming/Seminumerical algorithms*, vol. 2, Addison-Wesley, 2nd edition, 1981.
4. François Morain and Jorge Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications* **24**:531–543, 1990.
5. Marc Joye and Sung-Ming Yen. Optimal left-to-right binary signed-digit recoding. *IEEE Transactions on Computers* **49**:740–748, 2000.