Power Attack on Small RSA Public Exponent

Pierre-alain Fouque[1], Sébastien Kunz-Jacques[1,2], Gwenaëlle Martinet[2], Frédéric Muller[3], and Frédéric Valette[4]

[1] École normale supérieure, 45 rue d'Ulm, 75005 Paris, France
Pierre-Alain.Fouque@ens.fr
[2] DCSSI Crypto Lab, 51 boulevard de La Tour-Maubourg
F-75700 Paris 07 SP, France
{Gwenaelle.Martinet;Sebastien.Kunz-Jacques}@sgdn.pm.gouv.fr
[3] HSBC, France, Frederic.Muller@m4x.org
[4] CELAR, 35 Bruz, France
Frederic.Valette@dga.defense.gouv.fr

Abstract. In this paper, we present a new attack on RSA when the public exponent is short, for instance 3 or $2^{16}+1$, and when the classical exponent randomization is used. This attack works even if blinding is used on the messages.

From a Simple Power Analysis (SPA) we study the problem of recovering the RSA private key when non consecutive bits of it leak from the implementation. We also show that such information can be gained from sliding window implementations not protected against SPA.

Keywords: RSA cryptosystem, sliding window methods, exponent randomization, Simple Power Analysis.

1 Introduction

Simple Power Analysis and Differential Power Analysis attacks are among the most efficient and devastating attacks on some RSA-based products. Many countermeasures have been proposed that prevent these attacks by securing the exponentiation algorithm which is usually targeted. This is the basis of a number of academic papers whose results are widely used in practice. However, such countermeasures often lead to a slower implementation and thus another area of research is the speedup of the exponentiation process. As we will show in this article, unfortunate interactions between side-channel countermeasures and optimized exponentiation algorithms may lead to insecure implementations.

The aim of this paper is to present a new attack on RSA in the special case where both a short public exponent and a randomization of the private exponent are used. In such a case, free information on the private exponent can be obtained from the public key and can be used to efficiently recover the whole private key. The attack studies the problem when non-consecutive bits of the private key can be found. It works on sliding window implementations not protected against SPA attack.

Known results on partially known information. Partial information on the RSA private key allows it to be recovered in some cases. This kind of attacks has experienced a revival since 1998 with the work of Boneh, Durfee and Frankel [1]. In their article, they give some results about the security of RSA schemes when some bits of the private key are exposed. However, the lattice technique used in such cases cannot be applied for non-consecutive bits. None of the previous papers have considered this particular case. Boneh *et al.* have even considered in [1] that *"[the authors] view attacks that require non-consecutive bits of d as artificial"*, showing the lack of interest for this topic at that time.

However, some practical attacks are now very efficient and allow the attacker to recover some bits of the private key, not necessarily consecutive. This is mainly due to the combination of very specialized attacks, based on side channel analysis, and of the various countermeasures based on algorithmic remarks.

Main idea of the attack. Here, we do not solve the open problem of recovering the whole private key from non-consecutive partial information on it. However, we focus on the special case where several non-consecutive bits of randomized versions of the private key are known.

Efficient countermeasures against SPA attack are often not perfect. It is classical that some information about the secret exponent leaks. For example, sliding window implementations can leak when consecutive bits are equal to zero. By randomly generating bitstring and applying the parsing exponent algorithm of the sliding window algorithm, either Constant Length Non-zero Window (CLNW) or Variable Length Non-zero Window (VLNW), one can observe that the information gained is 40% of the bits. This is not sufficient to recover the entire secret by using previous results such as those of [1].

The first part of the attack is to record some power curves C_i which correspond to the exponentiation of a message with the unknown private key $d_i = d + \lambda_i \times \varphi(N)$ associated to an unknown short value λ_i. Then, using the fact that the public exponent is small, we can consider that the most significant bits of the secret exponent d are known. With this information, we can try all the possible values of λ and check if the most significant bits of the value $\tilde{d} + \lambda \cdot N$, where $\tilde{d}$ equals d on the half bits of high order, are compatible with the partial information that can be recovered from the power curve by SPA. If we have enough information, we can associate a single λ_i to the curve C_i.

The second part of the attack is now to recover the least significant bits of d. Once we have enough curves with the known random value λ_i we can then use partial information on the least significant bits of the randomized exponent on all the curves to retrieve the least significant bits of the secret exponent. The principle is to guess the least significant bits of $\varphi(N)$ and so of d and of the secret exponent d_i. Then, we check if the guess is compatible with the partial information on the curve C_i. If we have enough curves, only one guess will be compatible. We can then guess the next bits and continue until we know enough bits of d.

Our results. We recall in section 2 how to get non-consecutive bits of the RSA secret exponent by using side channel attacks. Such leakage depends on the exponentiation algorithm used and on the various countermeasures implemented against side channel attacks.

Then, in section 3 we formally show how to recover the whole RSA secret key from such information in the case of the public exponent is 3. We extend this attack for $e = 2^{16} + 1$ in section 4, and we give practical results in section 5.

Related Works. A lot of work has already been done on the particular topic of attacks when countermeasures are implemented. Indeed, a countermeasure may allow or simplify a side channel attack.

Previous works have also been done to study the security of fast exponentiation algorithm. Walter, in [9], describes the Big Mac attack which works on sliding and m-ary window algorithms. He assumes that he can distinguish *squares and multiplies* and *operand of the multiplies*. Here, we only assume that we can distinguish *squares and multiplies* but we do not need to distinguish the different operands of the multiplications.

Walter has also see in [10] that *"in the classical m-ary and sliding windows exponentiation algorithms, the most significant half of the public modulus yields information which can be used to halve the number of key digits which need to be guessed."* Having reduce the key digit by half or a quarter is not sufficient for an 1024-bit value since 256 are missing.

The problem of computing the RSA private exponent from partial information on it has known only a little attention in the literature. In [8], Stinson presents two algorithms to compute discrete logarithms in a prime field, when the Hamming weight of the discrete log is small. As we want to recover d such that $s = f(H(m))^d \bmod N$, where f is the padding function, and we know s and $f(H(m))$, d can be viewed as the discrete log of s in basis $f(H(m))$ in $\mathbb{Z}_N^*$. In appendix A, we show that this algorithm can be used to recover d from non-consecutive bits of it if the number of missing bits is relatively small, 128 for instance. However the memory and time complexity of this algorithm is high compared to our algorithm and cannot recover a large number of bits.

2 Modular exponentiation and side channel attacks

2.1 Classical countermeasures against side channel attacks

To defeat DPA attacks, many protections methods have been suggested in the literature. The most secure and widely used is the exponent randomization [6] as it is very easy to implement and it comes at a reasonable computational cost. The idea of this countermeasure is to use a classical SPA-protected implementation of the exponentiation and to randomize the private exponent at each computation. This randomization is based on the fact that the private exponent d is defined modulo $\varphi(N)$ since for all $M \in \mathbb{Z}_N^\star$ and all $\lambda \in \mathbb{Z}$, $M^{\lambda \times \varphi(N)} = 1 \bmod N$. Figure 1 describes this randomized exponentiation algorithm.

- `Inputs`: a message M, an exponent d, a modulus N and $\varphi(N)$
- `Output`: $M^d \bmod N$

1. Pick at random $\lambda \in \{0, \ldots, 2^\ell - 1\}$
2. Compute $d' = d + \lambda \cdot \varphi(N)$
3. Return SPA protected exponentiation $M^{d'} \bmod N$

Fig. 1. The exponent randomization algorithm

The success of this countermeasure lies in its very good efficiency and the security it offers. Indeed, without randomization an attacker is able to guess the exponent bit per bit and his check would be confirmed with a DPA attack [7]. With the randomization, such a guess cannot be made anymore on the value d' since the attacker does not know the random value used.

2.2 Optimized exponentiation algorithms

Timing attacks or SPA attacks are known to be very efficient on RSA-based cryptosystems. In [6], Kocher has shown how to recover the whole private key from the power consumption of a single RSA signature or decryption. If the square-and-multiply algorithm is used for the exponentiation without any countermeasure, various side channel attacks may be used to compromise the private key.

More efficient exponentiation algorithms may be used. Some of them use a parsing of the private exponent into windows of constant or variable length. In that case, side channel attacks cannot recover the whole private exponent anymore. Only some bits of it leak from the implementation, whose distribution depends on technical details of the exact algorithm used. The m-ary or the sliding window techniques are such methods.

The sliding window methods. Such methods have been developed to speed up the exponentiation algorithm by searching in the exponent large windows of bits equal to zero. Contrary to the m-ary algorithm, sliding window methods relax the splitting of the exponent into *sliding windows*. There are two variants known as the Constant Length Non-zero Window (CLNW) technique due to Knuth [3] and the Variable Length Non-zero Window (VLNW) technique due to Bos and Coster in [2]. Both of these techniques try to minimize the number of multiplications in the square-and-multiply algorithm by performing some precomputations. These techniques have been described and analyzed by Koç in [4, 5] and allow 5 to 8% of the multiplications to be avoided compared to the binary exponentiation algorithm.

The CLNW method consists in splitting the exponent as follows: a non-zero window will always be of length m, for a given parameter m, often equal to 4 in practice, and the zero windows are of variable length. For the exponentiation, precomputations have to be done for all the 2^{m-1} values of the m non-zero windows (with a bit 1 in low order since the parsing is done from the least significant bit to the most significant one).

The Variable Length version is an optimization of it and is more tricky to detail. The rule is to split the exponent into zero windows of length at least a given value and non-zero windows of length at most another given parameter. We can show that the number of leaking bits during a SPA attack will be the same for both techniques and so we only focus on the CLNW variant.

Figure 2 details the sliding window exponentiation algorithm. Such a method assumes that the exponent is split into windows. This splitting may be done either with the m-ary, CLNW or VLNW method, depending on the splitting criteria used. The exponentiation just uses squarings and multiplications with precomputed values.

- `Inputs:` x, e, N, m
- `Output:` $y = x^e \bmod N$

1. Compute and store x^w for all **odd** integer $w \in \{1, \ldots, 2^m - 1\}$
2. Parse e into zero and non-zero windows F_i of length $L(F_i)$ at most m for the non-zero windows and for $i = 0, 1, \ldots k-1$. The parsing algorithm may be CLNW or VLNW.
3. $y \leftarrow x^{F_{k-1}} \bmod N$ (which is a precomputed value)
4. for $i = k-2$ downto 0
   - $y \leftarrow y^{2^{L(F_i)}} \bmod N$
   - if $F_i \neq 0$, then $y \leftarrow y \cdot x^{F_i} \bmod N$
5. `return` y

Fig. 2. The Sliding Window Algorithm

2.3 SPA Information leakage

To mount the attack described in section 3, the underlying assumption will be that the attacker knows partial information on the exponent used during the RSA signature or decryption. To this end, we will assume that we can distinguish squares from multiplies.

The optimized exponentiation algorithms such as m-ary, CLNW or VLNW leak some information about the exponent. For example, the CLNW algorithm, as described in section 2.2, consists in splitting the exponent into non-zero windows of fixed length. For all these windows, some precomputations are made to reduce the total cost of the exponentiation. If no protection against SPA attacks is used, an attacker may be able to distinguish the squaring operations from the multiplications. Each time the number of squarings is greater than the length of the window, the attacker can deduce that there are some 0 bits in the exponent. The position is deduced from the total number of previous multiplications and squarings. When a multiplication is detected, the attacker knows that there is a non-zero window of exactly m bits. In this window, the lowest order bit is 1 and the other ones are unknown. Thus, in the worst case, when there are are only non-zero windows, the attacker learns one bit of the exponent over m. These bits are the least significant ones (equal to 1) of the non-zero windows.

Thus, if $m = 4$, the attacker learns 25% of the bits of the exponent in the worst case. In practice, we obtain 40% of the bits of each randomized exponent. For $m = 3$, we obtain 50% of these bits.

For the attack to be successful in the case $e = 2^{16} + 1$, the attacker has to obtain a given number of windows of two bits. Note that if the attacker learns 3 consecutive bits, the two overlapping windows of two bits can be used in the attack. Simulations show that for a 1024-bit modulus, the CLNW methods for parameter $m = 4$ may leak 200 such windows if squarings and multiplies are distinguishable. For 2048-bit modulus, 400 windows are obtained. For $m = 3$, we obtain 250 2-bit windows for 1024-bit modulus and 500 for 2048 ones.

3 Recovering a private RSA exponent from partial information on randomized versions of it

We focus in this section on the special case $e = 3$. In this context, additional and free information can be deduced from the public key. This gives the attacker the knowledge of some bits on the private key "for free". Although this does not allow an adversary to break RSA cryptosystems in general, such information is very useful when combined with a side channel attack on some particular implementation of the exponentiation.

3.1 Free information

Let N be an RSA modulus, e the public exponent and d the private one.

The first remark is that the modulus N is a good approximation of $\varphi(N) = (p-1) \times (q-1) = N - p - q + 1$ on essentially the $n/2$ most significant bits. Since the number of these bits depends on a carry propagation, with high probability, the $n/2 - 10$ bits of high order of $\varphi(N)$ are those of N. To simplify, we consider that the $n/2$ bits of high order of $\varphi(N)$ are known and equal to those of N.

Secondly, when $e = 3$, the $n/2$ most significant bits of d are also known. Indeed, d satisfies the relation

$$ed = 1 + k\varphi(N) \tag{1}$$

for some positive integer k. Let us choose a representative of d in $[0, \varphi(N) - 1]$. Since $k \times \varphi(N) = ed - 1 < ed$, then $k < e$: if $e = 3$, then $k = 1$ or $k = 2$. In fact, 3 divides neither p nor q and since 3 is invertible $\mod \varphi(N)$, 3 also divides neither $p-1$ nor $q-1$. Thus, $p \neq 0 \bmod 3$, $p-1 \neq 0 \bmod 3$ (resp. for q), and finally, $p = 2 \bmod 3$ and $q = 2 \bmod 3$. Consequently, $\varphi(N) = 1 \bmod 3$. Finally, since $k = -1/\varphi(N) \bmod 3$, then $k = 2 \bmod 3$, and finally $k = 2$. Therefore,

$$3d - 2\varphi(N) = 1 \tag{2}$$

and

$$\tilde{d} = \left\lfloor \frac{1+kN}{e} \right\rfloor = \left\lfloor \frac{1+2N}{3} \right\rfloor$$

is a good approximation of d on the half bits of high order. Equation (2) will be extensively used in the cryptanalysis described above.

3.2 Recovering the RSA private key

We consider the countermeasure consisting in randomizing the private exponent d. Thus for each exponentiation, an equivalent exponent d_i is first computed as $d_i = d + \lambda_i \times \varphi(N)$, for a random value λ_i of ℓ bits. Typically, $\ell = 20$ or $\ell = 32$. Furthermore, an optimized exponentiation algorithm, such as the CLNW or the VLNW method, is supposed to be used. In this context, as described in section 2.3, we suppose that the attacker knows a fraction $1/r$ of the bits of the private exponents used, randomly distributed amongst the $n + \ell$ bits of each exponent. We also suppose that these bits are available for ω different exponents d_i. The position of the known bits differ from one exponent to another. These bits are obtained by signing or decrypting ω messages whose value does not matter for the attack. This is a model for the side channel attack.

In the rest of this paper, the following notations will be used:

- for an integer x of n bits, the i-th bit of x is denoted by $x[i]$. The integer x can then be written as an n-bit string $x = x[n-1]x[n-2]\ldots x[0]$;
- for a randomized exponent d_i of n bits, $[d_i]$ is a vector of length n such that for all $j \in \{0, \ldots, n-1\}$:

$$\begin{aligned}[d_i][j] &= d_i[j] \text{ if the bit } d_i[j] \text{ is known} \\ &= 2 \text{ otherwise}\end{aligned}$$

- for a vector $[d_i]$ of length n and integers a and b such that $0 \leq a \leq b \leq n-1$, $[d_i]_{a,b}$ is the extracted vector for the positions a to b;
- for integers x and d_i of n bits, we write $[d_i] \doteq x$ if d_i and x matches on all the known bits of d_i. That is, for all $0 \leq j \leq n-1$ such that $[d_i][j] \neq 2$, we have $d_i[j] = x[j]$.

For example, for an 8-bit value $x = 01010101 = x[7]x[6]\ldots x[1]x[0]$ for which the bits known are in positions 1, 4, 5 and 7, $[x] = [0, 2, 0, 1, 2, 2, 0, 2]$ and $[x]_{3,6} = [2, 0, 1, 2]$.

We first show how partial knowledge of the randomized exponents d_i allows the attacker to recover the λ_i values used to generate them from d. We then show in a second step how to recover the entire private exponent d from the partial leakage on the randomized exponent and from the known bits of d.

Step 1: recovering the λ_i. The strategy to recover the random values λ_i used to mask the private exponent d is to use the known approximation of d and $\varphi(N)$ to compute all the possible values

$$\tilde{d_j} = \tilde{d} + j \times N$$

for all $j \in [0, 2^\ell - 1]$. On the $n/2$ high order bits, $\tilde{d_j}$ is equal to $d_j = d + j \times \varphi(N)$. Indeed, $d_j = \tilde{d_j} + j(p + q - 1)$ and since $j(p + q - 1)$ is a number of at most $(n/2 + \ell)$ bits, the two $(n + \ell)$-bit values have near half of the most significant bits in common with high probability. Only if a carry propagates, some bits will not be equal. However, if two random bitstrings are added, a carry will be absorbed with probability $1/4$ at a step. Consequently, with probability $1 - (3/4)^{10} \approx 0.94$, the carry coming from the least significant half bits will not propagate after the $(n/2 + \ell + 10)$-th bit.

Given these 2^ℓ values and the known bits of the ω randomized exponents d_i, the attacker is now able to recover the corresponding λ_i values. For each value d_i, he knows a ratio $1/r$ of the bits. In particular this applies for the $n/2$ high order ones. He then looks for a matching value from the computed $\tilde{d_j}$ on these bits. When he finds j such that d_i equals $\tilde{d_j}$ on the known bits, he deduces that $\lambda_i = j$. The detailed algorithm is given in figure 3.

- `Inputs`: $[d_i]_{n/2+\ell, n+\ell}$ the known bits of high order of d_i for all $i \in \{1, \ldots, \omega\}$
- `Outputs`: the value λ_i such that $d_i = d + \lambda_i \times \varphi(N)$, for all $i \in \{1, \ldots, \omega\}$

1. For $j = 0$ to $2^\ell - 1$, $\tilde{d_j} \leftarrow \tilde{d} + j \times N$
2. For $i = 1$ to ω,
   $j \leftarrow 0$
   While $(j < 2^\ell)$
      If $[d_i]_{n/2+\ell+10, n+\ell} \doteq \tilde{d_j}_{n/2+\ell+10, n+\ell}$ then $\lambda_i \leftarrow j$, break;
      else $j \leftarrow j + 1$;
3. Return λ_i for all $i \in \{1, \ldots, \omega\}$.

Fig. 3. The attacker strategy to recover the λ_i corresponding to each d_i

Some optimizations may be implemented depending on the value ℓ, and the best time-memory trade-off for the attacker. However, as long as the random values λ are relatively small, for example of at most 20 bits, the exhaustive search of figure 3 is clearly practical.

At the end of this step, the attacker has thus recovered each random value used to randomize the private exponent for ω RSA executions.

Let us now present some analysis of the success probability of the first step. For each given d_i, we compare it with the 2^ℓ values of $\tilde{d_j}$. Let us denote by Bad_i the event "a bad value λ is associated with d_i". If we assume that the bitstrings d_i and $\tilde{d_j}$ are uniformly distributed, we match a false j to λ_i with probability less than $(1/2)^{(n/2-\alpha)/r}$ where α depends on the carry propagation during the computation of $\tilde{d}_j$. As seen above, α may be upper bounded by 10 with high probability.

As we have 2^ℓ comparisons corresponding to all the $\tilde{d}_j$, the probability of Bad_i is upper bounded by $2^\ell/2^{(n/2-\alpha)/r}$. Therefore, we get

$$\Pr[\exists i\ 1 \leq i \leq \omega : \mathsf{Bad}_i] \leq \frac{2^\ell \omega}{2^{(n/2-\alpha)/r}}$$

For $n = 1024$, $r = 5$, $\ell = 32$, $\alpha = 10$ and $\omega \approx 64$, we get a probability of a good association for each d_i of $1 - 1/2^{63}$. Such an estimation does not take into account imperfect input data.

Step 2: recovering $\varphi(N)$ and d. The attacker's goal is now to recover the entire private key. To this end, he makes an exhaustive search, 8 bits per step, on the bits of $\varphi(N)$. He will now use the known *least significant bits* of d_i to recover d.

First, the attacker recovers the 8 least significant bits of $\varphi(N)$. This is performed by guessing $\varphi(N) \bmod 2^8$. From this guess, he computes the corresponding guess for $d \bmod 2^8$ from the equation 2:

$$d \bmod 2^8 = \frac{1 + 2\varphi(N)}{3} \bmod 2^8$$

Then with high probability there exists i s.t. some of the 8 least significant bits of d_i are known from the side channel attack. For this value d_i, the corresponding λ_i gives us some constraints on the low order bits of the value $d + \lambda_i \times \varphi(N) \bmod 2^8$. If the constraints on the corresponding d_i value cannot be met, another guess for $\varphi(N) \bmod 2^8$ is made. Otherwise, if for all $i \in \{1, \ldots, \omega\}$, no incompatibility has been discovered, the guess is the good one with high probability. The attack can then be extended with a guess for $\varphi(N) \bmod 2^{16}$ and so on. Figure 4 details the algorithm to recover $\varphi(N) \bmod 2^{8k}$ from $\varphi(N) \bmod 2^{8(k-1)}$.

Note that in practice the attacker should deal with imperfect input data since these data are collected in a side channels context. Thus, candidates that match a sufficiently high fraction of these data should be accepted: this may be done by implementing a more complex version of the Boolean function OK.

We need to estimate the average number of false candidates at each step. We have 2^8 values for each $\bar{d}$ and we have $8/r$ bits on each 8-bit window for each $\bar{d}_i$ where $1/r$ is the ratio of known bits deduced from the side channel attack. As the correct value for $\bar{d}$ allows ω correct values $\bar{d}_i$ for all i to be computed, then on average the number of false candidates is $\left(1/2^{8/r}\right)^\omega \times 2^8$ if all the experiments are independent and the bitstrings uniformly distributed. Thus, the average number of candidates tends to 1 as the number of false candidates tends to 0 and the correct candidate matches the input data, or eventually almost fit the input data.

– Inputs:
  - $\{([d_i], \lambda_i)\}_{1 \le i \le \omega}$ the list of the known bits for each d_i and the corresponding λ_i value
  - a candidate for $\varphi(N) \bmod 2^{8(k-1)}$
– Output: a list of candidates for $\varphi(N) \bmod 2^{8k}$

1. For $j = 0$ to $2^8 - 1$,
   (a) $y_j \leftarrow \varphi(N) \bmod 2^{8(k-1)} + j \cdot 2^{8(k-1)}$;
   `/*` y_j `is a candidate value for` $\varphi(N) \bmod 2^{8k}$ `*/`
   (b) $\bar{d} \leftarrow \frac{1+2y_j}{3} \bmod 2^{8k}$;
   `/*` $\bar{d}$ `is the corresponding candidate for` $d \bmod 2^{8k}$ `*/`
   (c) $OK \leftarrow$ `true`;
   (d) $i \leftarrow 1$;
   (e) While ($OK =$ `true`) and ($i \le \omega$)
   $\bar{d}_i \leftarrow \bar{d} + \lambda_i \times y_j \bmod 2^{8k}$;
   if $[d_j]_{0,8k-1} \doteq \bar{d}_i$ then $i \leftarrow i + 1$;
   else $OK \leftarrow$ `false`;
   (f) if $OK =$ `true`, add y_j to the list of candidates for $\varphi(N) \bmod 2^{8k}$;
2. Return the list of candidates for $\varphi(N) \bmod 2^{8k}$

Fig. 4. The attacker strategy to recover $\varphi(N) \bmod 2^{8k}$ from $\varphi(N) \bmod 2^{8(k-1)}$

4 Extension for $e = 2^{16} + 1$

In case $e = 3$, one knows that $k = 2$. For each measure using a random value λ, as shown in previous section, some bits in the upper half of

$$\left\lfloor \frac{1 + kN}{e} \right\rfloor + \lambda N \qquad (3)$$

are known: this allows us to retrieve λ with an exhaustive search. For other values of e, this approach cannot work directly as k is not known anymore. In this section, we show how to extract k *and* λ from only **one** measure yielding some bits of the randomized exponent, when e is not too large, the typical case being $e = 2^{16} + 1$. Once k is found, the attack can proceed exactly as described in Step 2 of the attack of section 3.

4.1 Finding k and λ by exhaustive search

The value (3) can be used to perform a direct exhaustive search of k and λ from one exponentiation measure. One has $0 < k < e$ and $0 \le \lambda < 2^\ell$: if e is u-bit long, there are $2^{u+\ell}$ candidates to try, and the exhaustive search yields only the correct values with good probability if the number of known bits in equation (3) is above $u + \ell$. For the typical values of $u = 16$ and $\ell = 20$, this approach requires to perform 2^{36} additions of large integers, assuming the values of λN for $0 \le \lambda < 2^\ell$ and $\frac{kN}{e}$ for $0 < k < e$ are precomputed. The aim is to recover k more efficiently.

4.2 Finding Matching Pairs of Values of k and λ

From now on, we consider a unique measure of an RSA signature or decryption with randomized exponent. Let δ denote the randomized exponent used during the exponentiation considered. One has:

$$\delta = d + \lambda \times \varphi(N)$$

As before, the most significant half $U(d)$ of the private exponent d is equal to $U\left(\left\lfloor \frac{1+kN}{e} \right\rfloor\right)$ except maybe on a few least significant bits. $U(\delta)$ can likewise be approximated by $U\left(\left\lfloor \frac{1+kN}{e} \right\rfloor\right) + U(\lambda N)$. Our goal is to recover k and λ.

If $U(\delta)$ were completely known, the exhaustive search on both k and λ could be transformed it into a list matching problem: indeed, correct values of k and λ correspond to matching elements in the lists

$$L_1 = \left\{ U(\delta) - U\left(\left\lfloor \frac{1+kN}{e} \right\rfloor\right) \;\middle|\; 0 < k < e \right\} \text{ and } L_2 = \left\{ U(\lambda N) \mid 0 \le \lambda < 2^\ell \right\}$$

However, since only *some* bits of δ are known, some further work is required to find matching elements. In the next paragraphs, we show how to associate with each candidate value of k a partially known value for $\delta - \left\lfloor \frac{1+kN}{e} \right\rfloor$, and then how to find matches between the list of these partially known values and L_2.

Step 1a: Compute Partial Values for $\boldsymbol{\delta - \left\lfloor \frac{1+kN}{e} \right\rfloor}$. In the following, $b(k)$ denotes $\left\lfloor \frac{1+kN}{e} \right\rfloor$.

For each candidate value of k, some bits of $\delta - b(k)$ can be computed. Indeed, assume that two consecutive bits δ_i, δ_{i+1} of δ are known as a result a side-channel attack like the one of paragraph 2.3. Let b_i and b_{i+1} the corresponding bits of $b(k)$, and c_i, c_{i+1} the corresponding carry bits in $\delta - b(k)$. The bits δ_i, δ_{i+1}, b_i, b_{i+1} are known while the carries are unknown. The subtraction looks as follows :

$$\begin{array}{rcll} & \delta_{i+1} & & \delta_i \\ - & b_{i+1} & \overset{c_{i+1}}{\leftarrow} & b_i \overset{c_i}{\leftarrow} \\ \hline & \cdots & & \cdots \end{array}$$

Assume that $b_i = 1 \oplus \delta_i$. Then one has:

$$\begin{array}{rcccl} & & \delta_{i+1} & & 0 \\ - & \overset{\cdots}{\leftarrow} & b_{i+1} & \overset{1}{\leftarrow} & 1 \overset{c_i}{\leftarrow} \\ \hline & & 1 \oplus \delta_{i+1} \oplus b_{i+1} & & 1 \oplus c_i \end{array} \quad \text{or} \quad \begin{array}{rcccl} & & \delta_{i+1} & & 1 \\ - & \overset{\cdots}{\leftarrow} & b_{i+1} & \overset{0}{\leftarrow} & 0 \overset{c_i}{\leftarrow} \\ \hline & & \delta_{i+1} \oplus b_{i+1} & & 1 \oplus c_i \end{array}$$

Therefore **whenever $b_i = 1 \oplus \delta_i$, the $(i+1)$-th bit of $\delta - b$ is equal to $b_i \oplus \delta_{i+1} \oplus b_{i+1}$ which is a known value**.

To compute partial values for $\delta - b(k)$, first mark the (possibly overlapping) windows of two consecutive known bits in the upper half of δ. Assume there are v such windows.

For each value of k in $[1, e-1]$, compute the value $b(k) = \lfloor \frac{1+kN}{e} \rfloor$. Depending on the bits of $b(k)$ aligned with the marked windows in δ, some bits of $\delta - b(k)$ can be computed according to the rules above : in each 2-bit window, the most significant bit of $\delta - b(k)$ can be computed with probability $1/2$, according to $b_i = 1 \oplus \delta_i$. A sequence (s_k) of v known or unknown bits in $\delta - b(k)$ is therefore obtained.

Step 1b: Find Matches for Partial Values. Associate to each value of λ the sequence t_λ of the values of the most significant bits of λN in the targeted 2-bit windows of δ (remember that there are v such windows). The correct value for k and λ yields a match between t_λ and the known bits in s_k. If there are sufficiently many windows, only the correct values gives a match.

For each value of k, let $L(k)$ denote the set of all λ such that t_λ matches s_k. Assuming $L(k)$ can be built efficiently, the exhaustive search for (k, λ) proposed in subsection 4.1 can be improved by adding an early elimination step where pairs (k, λ) s.t. $\lambda \notin L(k)$ are discarded. If $L(k)$ is small enough, this reduces the complexity of the exhaustive search. We therefore focus on efficiently computing the set $L(k)$.

A direct approach to the construction of $L(k)$ consists in considering every possible value of the unknown bits of s_k. For each of them, the set of the matching t_λ can be computed. Since each of the v bits in s_k is known with probability $1/2$, the number of possible values for the unknown bits in s_k is the product of v independent random variables that are equal to 2 with probability $1/2$ and to 1 with probability $1/2$. This number is therefore equal on average to $(3/2)^v$.

For $u = 16$, $v = 40$, there are about $2^{16} \times (3/2)^{40} \approx 2^{39.4}$ completions of the s_k, for all values of k, $0 < k < e$. Therefore, whatever the method used to find the corresponding t_λ for each of these completions, at least $2^{39.4}$ operations are required to build all lists $L(k)$ this way.

This approach can however be refined by splitting s_k into subpieces before exploring the possible values of the unknown bits. From now on, we assume that $v = 2\ell$; the attack is even faster for higher values v which correspond to cases where more information is available.

First, precompute the lists $L_l(\alpha) = \{\lambda \mid \text{the left half of } t_\lambda \text{ is equal to } \alpha\}$ for $0 \leq \alpha \leq 2^\ell$ and the lists $L_r(\beta) = \{\lambda \mid \text{the right half of } t_\lambda \text{ is equal to } \beta\}$ for $0 \leq \beta \leq 2^\ell$. This requires $2 \times 2^\ell$ operations on large integers.

Then for a candidate k, if $\alpha_1, \ldots, \alpha_n$ (resp. $\beta_1, \ldots, \beta_m$) are the values of the left (resp. right) half of s_k obtained by filling the unknown bits with any possible value,

$$L(k) = \left[\bigcup_{i=1}^{n} L_l(\alpha_i) \right] \cap \left[\bigcup_{i=1}^{m} L_r(\beta_i) \right]$$

On average, $n \approx m \approx (3/2)^{v/2}$, and for any i, $\#L_l(\alpha_i) \approx \#L_r(\beta_i) \approx 2^{\ell - v/2} = 1$. Using suitable data structure (of size $2^{v/2}$) to be able to compute an intersection in constant time, the formula above can therefore be evaluated using $2 \times (3/2)^{v/2}$

constant-size operations. This means that all the lists $L(k)$ can be built using only $2^u \times 2 \times (3/2)^{v/2}$ operations.

For $u = 16$, $\ell = 20$, $v = 2\ell = 40$, the total complexity is $2^{28.7}$ operations. One can show that cutting s_k in two halves is optimal when $v = 2\ell$.

Assuming that the t_λ and the completions of the s_k are random, the birthday paradox shows that the average number of elements in $L(k)$ is $\frac{(3/2)^v \times 2^\ell}{2^v}$. With $\ell = 20$, $v = 2\ell$, $\#L(k) \approx 2^{3.4}$. Therefore after the above early elimination step, $2^{19.4}$ pairs (k, λ) must be considered if $u = 16$, compared to 2^{36} pairs before the elimination step.

Overall, this improved attack retrieves k using $2^{28.7}$ constant-size operations and around 2^{20} operations on large integers. We implemented the attack; it runs in about one minute on an average PC.

In practice, as shown in section 2.3, the number of windows available is far above what is needed: with a 1024-bit exponent and the exponentiation algorithm of figure 2 with windows of size 4, one has approximately $v = 100$ windows of two known consecutive bits in the upper half of δ. This extra information can be taken into account to eliminate more pairs (k, λ). With v large enough, this filters out all the wrong pairs, thereby eliminating the need for an exhaustive search phase. On the other side, the complexity of the pair elimination phase is linear in v.

5 Practical results

There are two limits for this attack: the first one is to have enough information on one curve to be able to recover only one λ for each curve, the second one is to have enough information on all the curves to recover only one possibility for the least significant bits. The following tables will give some examples where the attack is feasible or not. We can note that the attack is more efficient on larger modulus size. Table 5 gives the results in the case $e = 3$. In that case, only a ratio of bits has to be known in each randomized exponent. In practice, if side channel attack is possible on a CLNW or VLNW splitting method, we may obtain a better ratio than detailed in the table.

Table 6 gives the results for $e = 2^{16} + 1$. In that case, as explained in section 4, the attack has better complexity if 2-bit windows are obtained for one randomized exponent. Such information may be obtained with the CLNW or VLNW splitting algorithms. For the optimized method to be more efficient than exhaustive search, the number of 2-bit windows should be twice the length of the random value λ used to randomized the private exponent.

The fifth column is computed by using the formula $n/(2r) \gg \ell$. For each parameter, 50 curves are sufficient in practice, without considering imperfect input data.

In conclusion we can see that for classical size and reasonable information leaking, the attack is feasible and of low complexity.

Modulus size	ℓ, size of random	$1/r$, ratio of partially known information	attack success
512	20	1/16	no
1024	20	1/16	yes
	32	1/16	no
2048	20	1/32	yes
	32	1/64	yes

Fig. 5. Practical results for $e = 3$.

Modulus size	ℓ, size of random	number of 2-bit windows	$1/r$, ratio of partially known information	attack success
512	20	40	1/16	no
1024	20	40	1/16	yes
	32	64	1/16	yes
2048	32	64	1/32	yes

Fig. 6. Practical results for $e = 2^{16} + 1$.

6 Acknowledgments

The authors would like to thank the anonymous referees for many useful comments on the first version of this paper.

References

1. D. Boneh, G. Durfee, and Y. Frankel. An attack on RSA given a fraction of the private key bits. In K. Ohta and D. Pei, editors, *Advances in Cryptology – Asiacrypt'98*, volume 1514 of *LNCS*, pages 25 – 34. Springer-Verlag, 1998.
2. J. Bos and M. Coster. Addition Chain Heuristics. In G. Brassard, editor, *Advances in Cryptology – Crypto 1989*, volume 435 of *LNCS*, pages 400 – 407. Springer Verlag, 1989.
3. D. E. Knuth. *The Art of Computer Programming, Vol 2: Semi Numerical Algorithms*. Addison Wesley, 1969.
4. C. K. Koç. High Speed RSA Implementation. Technical report, Tech Rep. 201, RSA Laboratories, 1994.
5. C. K. Koç. Analysis of Sliding Window Technique for Exponentiation. *Computers and Mathematics with Applications*, 10(30):17 – 24, 1995.
6. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Others Systems. In N. Koblitz, editor, *Advances in Cryptology – Crypto '96*, volume 1109 of *LNCS*, pages 104 – 113. Springer-Verlag, 1996.
7. T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcard. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, volume 1717 of *LNCS*, pages 144 – 157. Springer-Verlag, 1999.

8. D. R. Stinson. Some Baby-Step Giant-Step Algorithms for the Low Hamming Weight Discrete Logarithm Problem. *Mathematics of Computation*, 71:379 – 391, 2002.
9. C. D. Walter. Sliding Windows Succumbs to Big Mac Attack. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 286 – 299. Springer-Verlag, 2001.
10. C. D. Walter. Seeing through MIST Given a Small Fraction of an RSA Private Key. In M. Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 391 – 402. Springer-Verlag, 2003.

A When few bits are missing

In this appendix, we show that when the number of missing bits is small, we can recover missing bits of d by using a discrete log based algorithm. Stinson describes and analyzes several algorithms due to Heiman and Odlyzko and Coppersmith in [8]. Let m be the number of missing bits of $x = \log_\alpha \beta$ and t is the Hamming weight of x. Heiman and Odlyzko describe a meet-in-the-middle attack. We search Y_1 and $Y_2 \subseteq \mathbb{Z}_m$ such that $\alpha^{\mathsf{val}(Y_1)} = \beta(\alpha^{\mathsf{val}(Y_2)})^{-1} \bmod N$ where $\mathsf{val}(Y_i) = \sum_{j \in Y_i} 2^j$ by ranging through all Y_1 and Y_2 such that $|Y_1| = |Y_2| = t/2$. As there are $\binom{m}{t/2}$ such sets Y_1 and Y_2, the space and time complexity of the attack is of order $O(\binom{m}{t/2})$. Moreover, if we have only an upper bound t' on t, we have to run through all $t = 1$ to $t = t'$ and the time complexity becomes $O(\sum_{t=1}^{t'} \binom{m}{t/2}) = O(t' \binom{m}{t'/2})$.

Coppersmith's algorithm, described in [8], allows one to lower the time complexity to $O(m\binom{m/2}{t/2})$ and the space complexity to $O(\binom{m/2}{t/2})$. The idea is to use an (m,t)-splitting system for $\mathbb{Z}_m$. Such combinatorial structure is a pair $(X, \mathcal{B})$ with the following properties:

1. $|X| = m$, and $\mathcal{B}$ is a set of subsets of size $m/2$ of X, called *blocks*
2. for every $Y \subseteq X$ s.t. $|Y| = t$, there exists a block $B \in \mathcal{B}$ s.t. $|B \cap Y| = t/2$

Coppersmith shows that there exists an (m,t)-splitting system of size $m/2$. Therefore by picking $Y_1 \subseteq B_i$ for all $B_i \in \mathcal{B}$ and $Y_2 \in \mathbb{Z}_m \setminus B_i$ for any t-set Y_2 in $\mathbb{Z}_m \setminus B_i$ the same algorithm finds the matching in time $O(\binom{m/2}{t/2})$.

The last algorithm can be adapted to work $\bmod N$ where N is a RSA modulus and when the missing bits are not consecutive. The memory complexity is $O(m\binom{m/2}{t/2})$ where t is the number of 1 bits among m bits.

Consequently, if we assume that in the m missing bits, one of two are a one, then $t = m/2$. Therefore, the complexity is $O(m^2\binom{m/2}{m/4})$. Since, $\binom{N}{N/2} \approx \sqrt{2/\pi} \cdot 2^N$, then the complexity becomes $O(m^2 \cdot 2^{m/2})$, and in practice, we can only deal with $m \approx 128$.