Timing Attack against RSA Signature Generation – Draft

F. De Santis
Sichere Implementierung kryptographischer Verfahren WS 2015-2016
Technische Universität München **November 11, 2015**

1 Timing Analysis

The implementation of cryptographic algorithms often leads to non-constant execution times depending on the input data values, *e.g.*due to conditional instructions, cache mechanisms or compiler level optimizations. These differences in the time behaviour characteristic of cryptographic implementations may leak information about the secret material involved in the computations. The attacks which exploit the time behaviour characteristic of cryptographic implementations to recover the secret key are generally referred to as *timing attacks* in the literature. Timing attacks have been firstly proposed by P. Kocher in [Koc96] and subsequently developed in many other papers such as [DKL+98, HKQ99].

2 Timing Attack against RSA Signature Generation

RSA can be used for digital signature schemes by reversing the role of the encryption and decryption as shown in Figure 2.

`Signing Entity` $\mathcal{A}$		`Verifying Entity` $\mathcal{B}$
$y = \mathrm{sig}^{\mathcal{A}}_{k_{pr}}(x) = x^d \bmod n$	$\xrightarrow{\text{send}(x,y)}$	
	$\xleftarrow{\text{accept or reject}}$	$\mathrm{ver}^{\mathcal{B}}_{k_{pub}}(x,y) = \begin{cases} \texttt{accept if} & y^e \bmod n = x \\ \texttt{reject if} & y^e \bmod n \neq x \end{cases}$

Figure 1: The RSA Signature Scheme

For performance reasons the modular exponentiation is often implemented using the Square-and-Multiply algorithm and the Montgomery multiplication algorithm, especially on constrained devices. The Montgomery multiplication and the Left-to-Right Square-and-Multiply algorithms are recapped in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Montgomery Multiplication Algorithm

Input: a, b, n, z
Output: $abz^{-1} \bmod n$
1: $n' \leftarrow (-n)^{-1} \bmod z$ (precomputation)
2: $c \leftarrow ab$ (integer multiplication)
3: $d \leftarrow cn' \bmod z$ (reduction is a truncation to k bits being $z = 2^k$)
4: $e \leftarrow c + nd$ (integer multiplication and addition)
5: $f \leftarrow e/z$ (right shift by k bits being $z = 2^k$)
6: **if** $f \geq n$ **then**
7: $\quad f \leftarrow f - n$ (integer substraction)
8: **end if**
9: **return** f

Assuming that the execution of lines $1-5$ in Algorithm 1 takes a certain time τ and the execution of lines $6-8$ takes a certain time δ, then the overall execution time of the Left-to-Right Square-and-Multiply algorithm using the Montgomery multiplication takes a certain time $t = (k + w_H(d))\tau + R\delta$, where R is the number of required ER steps and $w_H(d)$ is the Hamming weight of secret exponent d. Therefore, the overall computational time depends on both the number of reductions and the Hamming weight of the secret exponent.

Algorithm 2 Left-to-Right Square-and-Multiply Algorithm

Input: $y, n, d = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0)_2$
Output: $x = y^d \pmod{n}$
1: $x \leftarrow 1$
2: **for** $i = k-1$ downto 0 **do** (scan through the key bits d_i from MSB to LSB)
3: $\quad x \leftarrow x^2 \pmod{n}$ (do the squaring always)
4: $\quad$ **if** $d_i = 1$ **then** (if the key bit d_i is set)
5: $\quad\quad x \leftarrow xy \pmod{n}$ (do the conditional multiplication)
6: $\quad$ **end if**
7: **end for**
8: **return** x

One way of exploiting the non-constant timing behaviour of such unprotected implementations of the modular exponentiation is to take advantage of the time variations due the reduction in the Montgomery multiplication (*cf.*, line 6 – 8 Algorithm 1) during the squaring operation of Left-to-Right Square-and-Multiply algorithm (*cf.*, line 3 Algorithm 2). Hereinafter, the conditional reduction will be referred to as the *Extra-Reduction* (ER) step.

Let $\mathsf{er} : \mathbb{Z}_2^k \times \mathbb{Z}_2^i \to \mathbb{Z}_2$ be the function $(x, D_{i-1}) \mapsto \{0, 1\}$ for x being an arbitrary integer of size k and $D_{i-1} = (d_{k-1}, \ldots, d_{k-i})_2$:

$$\mathsf{er}(x, D_{i-1}) = \begin{cases} 1 & \text{if the reduction in the squaring operation for the bit } d_{k-i} \text{ is performed} \\ 0 & \text{otherwise} \end{cases}$$

The timing attack proceeds iteratively by recovering one bit of the secret exponent d at a time, starting by the most significant key bit d_{k-1}. The working principle of the attack is to assume that $D_{i-1} = (d_{k-1}, \ldots, d_{k-i})_2$ bits of the secret exponent d are known, then perform a look-ahead in the next squaring operation according to a key hypothesis κ on the bit d_{k-i-1} and verify which key hypothesis leads to a distinguishable characteristic in the sampling distributions of timings. The original timing attack can be described as follows:

1. $x_0, \ldots x_{N-1}$ input messages are uniformly chosen at random for a sufficiently large N
2. The timings $t_0, \ldots, t_{N-1}$ corresponding to the signature generations on inputs $x_0, \ldots x_{N-1}$ are measured
3. Assuming $D_{i-1} = (d_{k-1}, \ldots, d_{k-i})_2$ bits of the secret exponent are known, a key hypothesis $\kappa \in \{0, 1\}$ on the next key bit d_{k-i-1} is done such that two hypothetical exponents are created, namely:
$$D_i^0 = (d_{k-1}, \ldots, d_{k-i}, 0)_2 \text{ and } D_i^1 = (d_{k-1}, \ldots, d_{k-i}, 1)_2$$
4. For each key hypothesis κ on the key bit d_{k-i-1}, the timings $t_0, \ldots, t_{N-1}$ are classified into two sets $\mathcal{T}_0$ and $\mathcal{T}_1$ according to the results of the er function as follows:
$$\begin{array}{lll} \kappa = 0 & \Rightarrow & \mathcal{T}_0^0 = \{t_j : \mathsf{er}(x_j, D_i^0) = 0\} \text{ and } \mathcal{T}_1^0 = \{t_j : \mathsf{er}(x_j, D_i^0) = 1\} \\ \kappa = 1 & \Rightarrow & \mathcal{T}_0^1 = \{t_j : \mathsf{er}(x_j, D_i^1) = 0\} \text{ and } \mathcal{T}_1^1 = \{t_j : \mathsf{er}(x_j, D_i^1) = 1\} \end{array}$$
5. For each key hypothesis κ on the key bit d_{k-i-1}, the absolute difference of means $|\tau_0^\kappa - \tau_1^\kappa|$ is computed, where $\tau_0^\kappa = \frac{1}{|\mathcal{T}_0^\kappa|} \sum_j t_j$ with $t_j \in \mathcal{T}_0^\kappa$ and $\tau_1^\kappa = \frac{1}{|\mathcal{T}_1^\kappa|} \sum_j t_j$ with $t_j \in \mathcal{T}_1^\kappa$
6. The key hypothesis κ leading to the largest absolute difference of means is kept as candidate for the key bit d_{k-i-1}. Therefore, the procedure resume from 3 till the second to last bit is attacked. The last key bit must always be guessed, since no look-ahead for the least significant bit is possible.

In practice, the sample Pearson's correlation coefficient r^κ is often used in place of the absolute difference of means for its improved robustness to noise and computed as follows:

$$r^\kappa = \frac{\sum_{j=0}^{N-1} (t_j - \frac{1}{N} \sum_{j=0}^{N-1} t_j)(e_j^\kappa - \frac{1}{N} \sum_{j=0}^{N-1} e_j^\kappa)}{\sqrt{\sum_{j=0}^{N-1} (t_j - \frac{1}{N} \sum_{j=0}^{N-1} t_j)^2 \sum_{j=0}^{N-1} (e_j^\kappa - \frac{1}{N} \sum_{j=0}^{N-1} e_j^\kappa)^2}},$$

where $e_j^\kappa = \mathsf{er}(x_j, D_i^\kappa)$.

3 Assignment

The goal of the assignment is to implement the timing attack against RSA signature generation in Python 2.x *and* run the attack to recover the secret exponent d from the timings of 5000 signature generations.

The framework for the assignment can be downloaded from https://tueisec-sica.sec.ei.tum.de/file.php?f=SIKA_Aufgabe_3.zip, while the input messages with the corresponding timing measurements and a testing pair can be downloaded from https://tueisec-sica.sec.ei.tum.de/rsa/. The testing pair consists of an input message x and the corresponding signature $x^d \bmod n$. The testing pair can be used to verify the correctness of the recovered secret exponent by running a signature generation on the given input message x with the recovered exponent and verify the result against the provided signature. For the purpose of the assignment 64-bits integers are considered.

The framework contains the files `project.nja`, `main.py` and `student.py`. The `project.nja` is the project file for the Ninja integrated development environment, cf. http://ninja-ide.org/, while the files `main.py` and `student.py` must be used as skeleton for the implementation of the timing attack. The file `main.py` loads the input messages, the corresponding timings and the testing pair and calls the function `perform_timing_attack`. Eventually, it writes the recovered secret exponent to the file `key.txt`, as shown in the Listing 1.

Listing 1: main.py

```python
########################### IMPORT MODULES ############################

import sys, time, glob, os, csv, student

######################### DEFINE PARAMETERS ###########################

timings_csv_file      = './timings.csv'
inputs_csv_file       = './inputs.csv'
testing_pair_csv_file = './testing_pair.csv'

############## LOAD TIMINGS INFORMATION AND TEST PAIR #################

csv_reader = csv.reader(open(timings_csv_file, 'rb'), delimiter=',')
timings = [int(element) for element in csv_reader.next()]

csv_reader = csv.reader(open(testing_pair_csv_file, 'rb'), delimiter=',')
testing_pair = [long(element) for element in csv_reader.next()]

csv_reader = csv.reader(open(inputs_csv_file, 'rb'), delimiter=',')
inputs = [long(element) for element in csv_reader.next()]

##################### PERFORM TIMING ATTACK ###########################

key = student.perform_timing_attack(inputs, timings, testing_pair)

######################## OUTPUT RESULTS ###############################

keyhex = ",".join(["%02X" % (key >> 64-(8*(i+1)) & 0x00000000000000FF ) \
         for i in range(64/8) ])
print keyhex

################### WRITE RESULTS TO A FILE ###########################

keyF = open( "./key.txt", "w" )
keyF.write(keyhex)
keyF.close()
```

The function `perform_timing_attack` must be implemented in the file `student.py` to perform the actual timing attack and return the recovered secret exponent. In order to facilitate the development, the implementation of the following functions is provided in the file `student.py`: `modInvEuclid` to compute the modular multiplicative inverse, `testBit` to verify whether a particular bit for a given value is set and `testPair` to verify the correctness of the recovered secret exponent from the provided testing pair. The `student.py` is shown in the Listing 2.

Listing 2: student.py

```
######################## IMPORT MODULES ############################

import os, sys, math, csv, time, numpy, math

##################### USEFUL ROUTINES ##############################

def testBit(val, offset):
    """
    Test whether bit at position offset is set or not
    Returns: 1 if it is set, 0 otherwise
    """
    if (val & (1 << offset))!=0:
        return bool(1)
    else:
        return bool(0)

def testPair(testing_pair, d, n):
    """
    Test whether the provided secret key d is correct or not
    Returns: 1 if it is correct, 0 otherwise
    """
    y = testing_pair[0]
    S = testing_pair[1]

    S1 = long(pow(y,d,n))

    if S == S1:
        return bool(1)
    else:
        return bool(0)

def extEuclideanAlg(a, b) :
    """
    Extended Euclidean algorithm
    """
    if b == 0 :
        return 1,0,a
    else :
        x, y, gcd = extEuclideanAlg(b, a % b)
        return y, x - y * (a // b),gcd

def modInvEuclid(a,n) :
    """
    Computes the modular multiplicative inverse using
    the extended Euclidean algorithm
    Returns: multiplicative inverse of a modulo n
    """
    x,y,gcd = extEuclideanAlg(a,n)
    if gcd == 1 :
```

```
        return x % n
    else :
        return None

############# IMPLEMENT YOUR TIMING ATTACK BELOW ##############

def MontgomeryMul(a,b,n,n1,z):
    """
␣␣␣␣Montgomery␣Multiplication
␣␣␣␣Returns:␣f=a*b␣and␣er=1␣if␣reduction␣is␣done,␣er=0␣otherwise
␣␣␣␣"""

    # ... WRITE YOUR CODE HERE ...

    return f,er

def perform_timing_attack(inputs, timings, testing_pair):
    """
␣␣␣␣Timing␣attack
␣␣␣␣Returns:␣Secret␣key␣d␣(64␣bit␣integer␣number)
␣␣␣␣"""

    n = long(0xB935E2B84B83E9EB) # modulus
    z  = long(pow(2,64))
    z2 = long(pow(z,2,n))
    n1 = modInvEuclid(-n, z)

    d = long(0)

    # ... WRITE YOUR CODE HERE ...


    # brute force last bit
    d1 = (d ^ 1)
    if testPair(testing_pair, d1, n):
        d=d1

    return d
```

References

[DKL+98] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems, *A practical implementation of the timing attack*, Proceedings of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998) (Jean-Jacques Quisquater and Bruce Schneier, eds.), LNCS, vol. 1820, Springer-Verlag, 1 1998.

[HKQ99] Gaël Hachez, François Koeune, and Jean-Jacques Quisquater, *Timing attack: what can be achieved by a powerful adversary?*, Proceedings of the 20th symposium on Information Theory in the Benelux (E.C. van der Meulen A. Barbé and P. Vanroose, eds.), 1 1999, pp. 63–70.

[Koc96] Paul C. Kocher, *Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems*, Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (London, UK, UK), CRYPTO '96, Springer-Verlag, 1996, pp. 104–113.