Rapport attaque CPA sur RSA

Alexandre LADRIERE - Hugo SERIEYS

22-29/01/2020

Introduction

L'objectif de ces travaux pratiques est de réaliser une attaque type CPA (Correlation Power Analysis) sur un algorithme cible RSA non protégé.

Pour réaliser cette attaque, il nous est fourni des traces de mesures de signaux électromagnétiques relevées lors de l'exécution de l'algorithme sur différents plaintexts (une trace par plaintext) également fournis.

Notre numéro étudiant était le **12.**

Procédé

Le principe de l'attaque sur RSA est le suivant:

- On fait l'hypothèse que le bit de poids fort de la clé est à 1 (premier bit non nul)
- On initialise une variable k à 1
- On itère les étapes suivantes pour chacun des autres bits de la clé (notre boucle s'arrête dès que les points dans les traces valent -1000):
    - On fait l'hypothèse de 0 et 1 sur le prochain bit à trouver, qu'on ajoute à la sous-clé déjà trouvée à l'itération précédente
    - On réalise l'exponentiation modulaire avec les 2 hypothèses de clé pour chacun des plaintexts disponibles
    - On calcule le poids de Hamming du résultat de l'exponentiation modulaire faite précédemment pour chacun des plaintexts disponibles
    - On calcule la corrélation entre les poids de Hamming calculés et les traces à l'instant k
    - On choisit l'hypothèse de clé pour laquelle la corrélation est la plus haute (0 ou 1)
    - On incrémente k :
        - De 1 si l'hypothèse retenue sur le bit est 0
        - De 2 si l'hypothèse retenue sur le bit est 1.

La variable k est essentielle car elle permet de "synchroniser" la corrélation.

Remarques sur notre code

Notre code calcul la clé en utilisant la CPA, mais également en utilisant la factorisation de N qui est possible ici car de petite taille. Cela lui permet de comparer directement en sortie si la clé trouvé par CPA est la bonne. A la fin de l'exécution du script, un fichier texte, nommé `d_NUMERO_ELEVE.txt`, est généré. La clé trouvée par CPA et celle trouvée par factorisation sont écrites dans ce fichier au format décimal mais aussi au format binaire.

Contenu du fichier de sortie `d_12.txt` :

```
KEY CALCULATED BY FACTORING N
Key (bin): 0b10110110000000101111010000001
Key (dec): 95426433

KEY CALCULATED BY CPA
Key (bin): 0b10110110000000101111010000001
Key (dec): 95426433
```

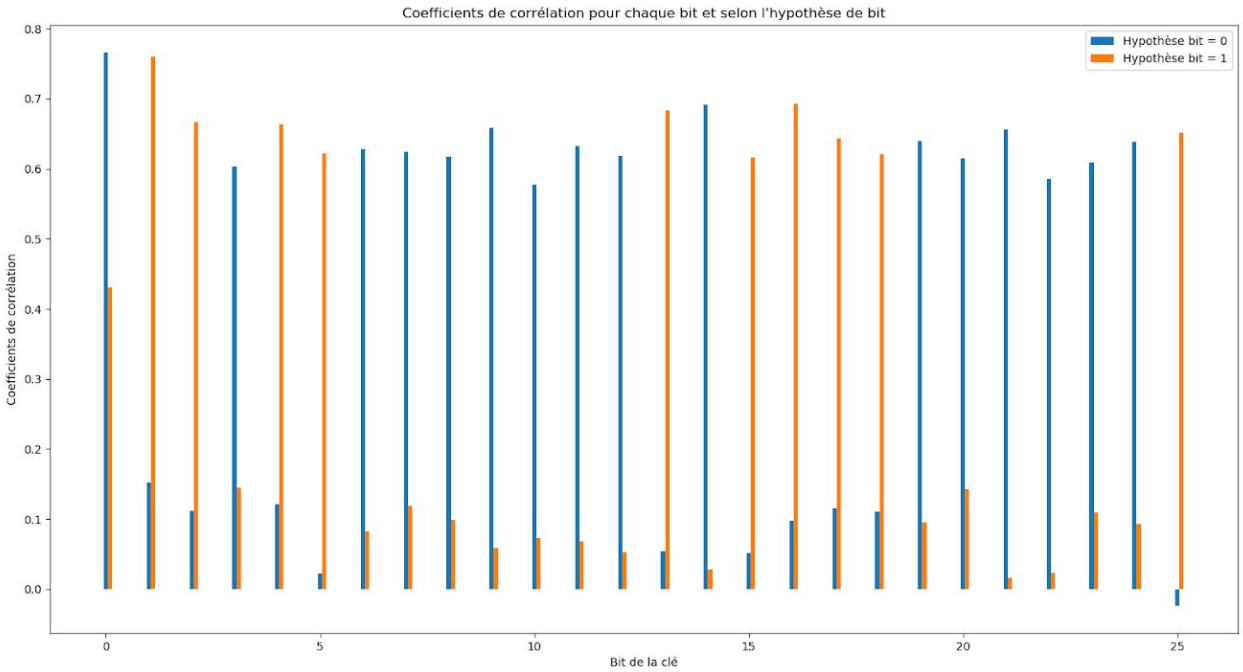

Figure 1 : Relevé des coefficients de corrélation pour chaque bit selon l'hypothèse sur le bit

Notre implémentation contient deux fichiers python:

- `cpa_rsa.py` qui contient les fonctions principales de calcul de clé ainsi que la fonction main

- `utils.py` qui contient toutes les fonctions annexes ainsi que les constantes utilisées (chemins des fichiers, numéro du data set, …)

Notre code est disponible en annexe de ce rapport, mais également au lien suivant: https://github.com/AlexandreLadriere/CPA-Attack-on-RSA

Protection sur l'exponentiation avec l'ajout d'une variable fantôme

La question posée est de savoir si l'ajout d'une telle variable dans le calcul de l'exponentiation modulaire comme décrite ci-dessous en rouge permet de protéger l'algorithme RSA d'une attaque CPA.

```
If(d[i]==1)
      T:=T*M mod N
Else
      T':=T*M mod N
End If
```

L'ajout d'une telle variable, si l'on considère qu'elle n'est *pas ignorée par le compilateur* car algorithmiquement inutile et que l'opération est donc bien exécutée en pratique certains compilateurs pourraient en effet supprimer d'eux-même cette opération), *permet de protéger l'algorithme RSA d'une attaque CPA*. En effet, l'attaque se base sur le fait que les traces sont différentes lors d'une itération de l'exponentiation modulaire car, par exemple, si d[i] :

- Vaut 1, il y aura deux opérations (carré et multiplication)
- Vaut 0, il n'en y aura qu'une.

En ajoutant le else, cela annulera l'aspect déséquilibré de la consommation en fonction de la valeur de d[i] et trompera l'attaque.

Annexe 1 - cpa_rsa.py

```
import numpy as np
from utils import *

corr_coeff_y1 = [] # for report purpose
corr_coeff_y0 = [] # for report purpose

def M_d_mod_N(M, d, N):
	"""
	Return the hamming weight of T at the end of the RSA exponentiation
	:param M: Massage (integer)
	:param d: Key (array of integer (0, 1))
	:param N: n parameter (n = p*q) (integer)
	:return: Hamming weight of the exponentiation result (integer)
	"""
	T = M
	hw = 0 # Hamming Weight of T
	for i in range(len(d) - 2, -1, -1):
	T = (T**2) % N
	if (d[i] == 1):
		T = (T*M) % N
		if i == 0:
			# The end
			hw = hamming_weight(T)
	else:
		if i == 0:
			# The end
			T = (T**2) % N
			hw = hamming_weight(T)
	return hw

def keyCalculationByCPA(n, traces, messages):
	"""
	Compute the secret key from cypher msg and traces
	:param n: the modulus (integer)
	:param traces: Traces retrieved when the RSA computation was made
(list of list of foat)
	:param messages: Cyper msg (list of integer)
	:return: Secret key (array of integer (0, 1))
	"""
	d_hyp = [1] # key hypothesis initialization
	array_hw_zeros = np.zeros((NB_MEASURES, 1))
	array_hw_ones = np.zeros((NB_MEASURES, 1))
	cpt = 1
```

```
	while traces[0][cpt] != -1000:
	for k in range(len(messages)):
		d_tmp = [0] + d_hyp # 0 hypothesis
		array_hw_zeros[k] = M_d_mod_N(messages[k], d_tmp, n)
		d_tmp = [1] + d_hyp # 1 hypothesis
		array_hw_ones[k] = M_d_mod_N(messages[k], d_tmp, n)
	mat_corr_zeros = np.corrcoef(array_hw_zeros, traces[:, cpt:cpt + 1],
False)
	mat_corr_ones = np.corrcoef(array_hw_ones, traces[:, cpt:cpt + 1],
False)
	corr_coef_zeros = mat_corr_zeros[1][0]
	corr_coef_ones = mat_corr_ones[1][0]
	corr_coeff_y1.append(corr_coef_ones)
	corr_coeff_y0.append(corr_coef_zeros)
	if (corr_coef_ones <= corr_coef_zeros): # it is highly possible that
it is a 0
		d_hyp = [0] + d_hyp
		cpt += 1
	else: # it is highly possible that it is a 1
		d_hyp = [1] + d_hyp
		cpt += 2
	d_hyp.reverse()
	return d_hyp

def keyCalculationByFactoring():
	"""
	Compute the secret key by factoring the mod
	:return: The secret key d (integer)
	"""
	n = getModulo(PATH + N_FILE_PATH)
	(p, q) = prime_factors(n)
	fi_n = (p-1) * (q-1)
	d = invmod(E, fi_n)
	return d

def saveKey(file_path, key_cpa, key_fac):
	"""
	Save the keys (in bin and dec format) found by factoring and CPA
calculation in a file
	:param file_path: Path for the output file (string)
	:param key_cpa: Key found by CPA (integer)
	:param key_dec: Key found by factoring (integer)
	:return: None
	"""
	f = open(file_path, 'w+')
	f.write("KEY CALCULATED BY FACTORING N")
```

```
        f.write("\nKey (bin): " + str(bin(key_fac)))
        f.write("\nKey (dec): " + str(key_fac))
        f.write("\n\nKEY CALCULATED BY CPA")
        f.write("\nKey (bin): " + str(bin(key_cpa)))
        f.write("\nKey (dec): " + str(key_cpa))
        f.close()

if __name__ == "__main__":
        n = getModulo(PATH + N_FILE_PATH)
        trace_t = np.asarray(read_entries(TRACE_TITLE, NB_MEASURES))
        msg_t = read_entries(MSG_TITLE, NB_MEASURES)
        key_cpa = keyCalculationByCPA(n, trace_t, msg_t)
        key_cpa_int = convertBinListToInt(key_cpa)
        key_fac_int = keyCalculationByFactoring()
        print("Key (by factoring) =", bin(key_fac_int))
        print("Key (by CPA) =", bin(key_cpa_int))
        print("Equal Keys: ", key_fac_int == key_cpa_int)
        saveKey(KEY_PATH, key_cpa_int, key_fac_int)
        plot_correlation_histogram(corr_coeff_y0, corr_coeff_y1)
```

Annexe 2 - utils.py

```
DATA_SET = "12" # Data set number
NB_MEASURES = 999 # Number of measures
KEY_LENGTH = 32 # Length of the key
E = 65537 # e parameter of the RSA algorithm
FILE_FORMAT = ".txt" # File format used for measures and results output
MSG_TITLE = "msg_" # File name format for messages
TRACE_TITLE = "curve_" # File name format for traces
N_FILE_PATH = "N" + FILE_FORMAT # File path for the parameter N
PATH = "./EMSE/etudiant - " + DATA_SET + "/" # Path of the data set used
KEY_PATH = "./d_" + DATA_SET + FILE_FORMAT # Path of the output key file

def prime_factors(n):
      """
      Compute the prime factors of the given number
      :param n: Number you want to compute the prime factors (intger)
      :return: Prime factors of the given number (list of integer)
      """
      i = 2
      factors = []
      while i * i <= n:
      if n % i:
            i += 1
      else:
            n //= i
            factors.append(i)
      if n > 1:
      factors.append(n)
      return factors

def extended_gcd(a, b):
      """
      Get the gcd of the given numbers
      :param a: First number (integer)
      :param b: Second number (integer)
      :return: gcd of a and b (integer)
      """
      x, lastx, y, lasty = 0, 1, 1, 0
      while b != 0:
      a, (quotient, b) = b, divmod(a, b)
      x, lastx = lastx - quotient * x, x
      y, lasty = lasty - quotient * y, y
      return a, lastx * (-1 if a < 0 else 1), lasty * (-1 if b < 0 else 1)
```

```
def invmod(a, m):
	"""
	Compute the modular mutliplicative inverse
	of the given number mod the second number
	:param a: Number you want to know the mod inverse (integer)
	:param m: mod (integer) print("d (bin) =", bin(d))
	:return: the modular inverse (integer)
	"""
	g, x, y = extended_gcd(a, m)
	if g != 1:
	raise ValueError
	return x % m

def getModulo(file_path):
	"""
	Retrieve the n parameter of the RSA algorithm from
	the specified text file
	:param file_path: Path of the mod file (string)
	:return: The modulus N (integer)
	"""
	f = open(file_path, 'r')
	N = f.read()
	return int(N)

def read_entries(type, number):
	"""
	Reads all the msg and curves file to put their data
	into lists of int or float according to their type
	:param type: Type of data (msg or curve) (string)
	:param number: Number of file (integer)
	:return: list of int (if type == MSG_TITLE) or
	list of list of float (if type == TRACE_TITLE)
	"""
	entries_t = []
	for k in range(number):
	file = open(PATH + type + str(k) + FILE_FORMAT, "r")
	for line in file: # read rest of lines
		if type == TRACE_TITLE:
			entries_t.append([float(x) for x in line.split()])
		else:
			entries_t.append(int(line.split()[0]))
	return entries_t

def hamming_weight(x):
	"""
```

```
        Compute the Hamming Weight of the given number
        :param x: Number you want to have the hamming weight of (integer)
        :return: athe Hamming weight (integer)
        """
        return bin(x).count("1")

def convertBinListToInt(bin_list):
        """
        Convert a binary list ([1, 0, 1, 0]) to an integer
        :param bin_list: Binary list
        :return: Integer representation of the binary list (integer)
        """
        dec = int("".join(map(str, bin_list)),2)
        return dec

def plot_correlation_histogram(corr_coeff_y0, corr_coeff_y1):
        """
        Plot an histogram of correlation coefficients
        for each bit for each hypothesis
        :param corr_coef_y0: list of correlation coefficients for bit = 0
hypothesis
        :param corr_coef_y1: list of correlation coefficients for bit = 1
hypothesis
        """
        bar_width = 0.1
        index = np.arange(len(corr_coeff_y0))
        plt.figure()
        plt.bar(index, corr_coeff_y0, bar_width, label="Hypothèse bit = 0")
        plt.bar(index+bar_width, corr_coeff_y1, bar_width, label="Hypothèse
bit = 1")
        plt.ylabel("Coefficients de corrélation")
        plt.xlabel("Bit de la clé")
        plt.title("Coefficients de corrélation pour chaque bit et selon
l'hypothèse de bit")
```