Rapport TP Attaque CPA sur RSA

Marc NGUYEN - Mathieu POIGNANT

Introduction

L'objectif de ce TP est de réaliser une attaque par CPA (Correlation Power Analysis) sur une implémentation RSA non protégée.

On dispose de courbes de consommation de courant et des plaintexts associés afin de retrouver la clé de chiffrement utilisée.

Nous avions le numéro d'étudiant **9**.

Principe de l'attaque

On part du principe que le bit de poids fort de la clé vaut 1 car la courbe d'acquisition de la consommation de l'exponentiation modulaire commence au premier bit non nul de la clé. On initialise donc la variable contenant la clé trouvée avec un 1 et on regarde les valeurs de consommation en partant de t = 1.

Ensuite, pour chaque bit de la clé, on réalise les étapes suivantes :

- On initialise à zéro une matrice hypothèse de taille 1000 * 2 car il y a 1000 plaintexts et 2 hypothèses par bit.
- Puis pour chaque plaintext et chaque hypothèse de bit, on ajoute le bit hypothèse à la sous-clé déjà trouvée, on réalise l'exponentiation modulaire avec la clé hypothèse et on calcule le poids de Hamming.
  Si le bit hypothèse est 0, on réalise un square supplémentaire avant de calculer le poids de Hamming car on regarde la consommation sur t et t+1.
  On peut donc remplir une case de la matrice hypothèse.
- On récupère les valeurs de consommation sur t et t+1 des 1000 traces (on repère s'il y a un square suivi d'un multiply ou deux square consécutifs). On filtre les -1000 qui apparaissent sur les traces car ils ne correspondent pas à l'exécution du chiffrement.
- On crée la matrice de corrélation à partir des deux matrices précédentes. On obtient alors une matrice contenant le coefficient de corrélation pour les deux hypothèses de bit.
- On prend le bit d'hypothèse qui a la plus forte corrélation avec la consommation mesurée et on l'insère en tête de notre hypothèse de clé qui est construite en little-endian.

- Ensuite on incrémente t :
    - de 2 si le bit vaut 1 car le bit correspond à la réalisation d'un square suivi d'un multiply
    - de 1 si le bit vaut 0 car le bit correspond à la réalisation d'un seul square

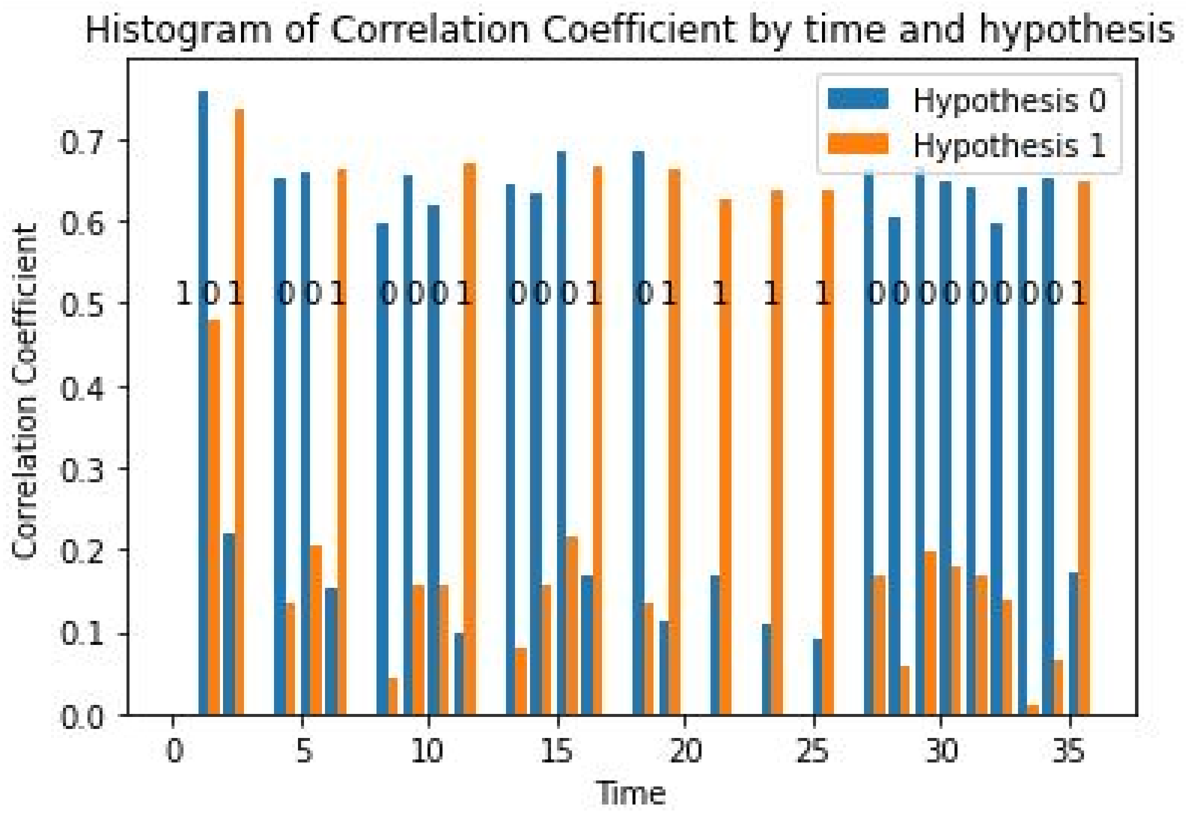
La clé calculée est inversée pour être remise en big-endian avant d'être retournée.

On réalise ensuite une attaque par factorisation sur N afin de trouver p et q pour remonter à la clé privée d et on compare les deux résultats.

Résultats

On arrive à trouver la clé par CPA avec notre code et celle-ci correspond à celle trouvée par factorisation de N. La clé retrouvée est : **101001000100010111100000001**.
Sortie du programme :

```
Factoring : 101001000100010111100000001
CPA : 101001000100010111100000001
Equality : True
```



Notre code est disponible en annexe et au lien suivant :
https://github.com/Darkness4/CPA-RSA

Annexe : code python

```python
"""
Main Program.

Decrypt RSA private key using Correlation Power Analysis.
"""

import matplotlib.pyplot as plt
import numpy as np
from typing import Iterable, Tuple
from functools import reduce
import os

# Project Parameters
ASSETS_PATH: str = os.path.join(os.path.dirname(__file__), "assets")
MODULO_PATH: str = os.path.join(ASSETS_PATH, "N.txt")
CURVE_FILE_NAMING_PATTERN: str = "curve_{:d}.txt"
PLAINTEXT_FILE_NAMING_PATTERN: str = "msg_{:d}.txt"
PUBLIC_KEY: int = 2 ** 16 + 1

with open(MODULO_PATH, "r") as file:
    N = int(file.readline())


class NumberUtils:
    """Utility class that handles various operations on numbers."""

    @staticmethod
    def hamming_weight(number: int) -> int:
        """Calculate the Hamming distance from 0."""
        return bin(number).count("1")

    @staticmethod
    def prime_factors(n) -> Iterable[int]:
        """Compute the prime factors of the given number."""
        i = 2
        while i * i <= n:
            if n % i:
                i += 1
            else:
                n //= i
                yield i
        if n > 1:
            yield n

    @staticmethod
    def mod_inverse(a: int, b: int) -> int:
        """Apply the extended euclidean algorithm to find the gcd between a and b."""
        if b <= 1:
            raise ZeroDivisionError
        old_r, r = a, b
        old_t, t = 1, 0

        while old_r > 1:
            if r == 0:
                raise ZeroDivisionError("not coprime")

            q = old_r // r
            old_r, r = r, old_r - q * r
            old_t, t = t, old_t - q * t

        return old_t if (old_t >= 0) else b + old_t

    @staticmethod
    def hamming_weight_for_rsa(
        data: int, exponent_bin: Iterable[int], n: int
    ) -> int:
        """Calculate key with Square-And-Multiply, and compute Hamming Weight."""
        result = 1
        for bit in reversed(exponent_bin):
            result = (result * result) % n  # Square
            if bit == 1:
                result = (result * data) % n  # Multiply

        if exponent_bin[0] == 0:  # Assume squaring if hypothesis is 0.
            result = (result * result) % n
        return NumberUtils.hamming_weight(result)

    @staticmethod
    def corr(A: np.ndarray, B: np.ndarray) -> np.ndarray:
        """Compute the correlation matrix using Pearson coefficient."""
        A = (A - A.mean(axis=0)) / A.std(axis=0)
        B = (B - B.mean(axis=0)) / B.std(axis=0)
        return ((B.T).dot(A) / B.shape[0])[0]

    @staticmethod
    def bit_list_to_int(bits: Iterable[int]) -> int:
        """Convert a list of bit to an integer."""
        return reduce(lambda acc, value: acc * 2 + value, bits)


def load_power_consumptions(
    number_of_trace: int, number_of_points_per_trace: int
) -> np.ndarray:
    """Load the traces stored in files."""
    power_consumptions = np.zeros(
        (number_of_trace, number_of_points_per_trace)
    )
    for i in range(number_of_trace):
        filename = CURVE_FILE_NAMING_PATTERN.format(i)
        pathfile = os.path.join(ASSETS_PATH, filename)
        with open(pathfile, "r") as file:
            # Parse a line of float and ignore -1000.0
            data_iter = filter(
                lambda x: x != -1000.0,
                map(float, file.readline().rstrip().split()),
            )
            data = np.fromiter(data_iter, float)
            if number_of_points_per_trace != np.size(data):
                print(
                    f"WARNING: {number_of_points_per_trace} points used on {np.size(data)}
total data\n"
                    f"You may want to increase [number_of_points_per_trace] by
{np.size(data)-number_of_points_per_trace}"
                )
            power_consumptions[i][:number_of_points_per_trace] = data[
                :number_of_points_per_trace
            ]
    return power_consumptions


def fetch_plaintexts(number_of_plaintext: int) -> Iterable[int]:
    """Load the plaintexts."""
    for i in range(number_of_plaintext):
        filename = PLAINTEXT_FILE_NAMING_PATTERN.format(i)
        pathfile = os.path.join(ASSETS_PATH, filename)

        with open(pathfile, "r") as file:
            yield int(file.readline().rstrip())


def compute_hypothesis_matrix(
    plaintexts: Iterable[int], old_key: Tuple[int]
) -> np.ndarray:
    """Return the hypothesis matrix.

    Since we are "building" the bits of the key, we need the old_key.
    Based on the old_key, the following hypothetical bit is either 0 or 1.
    Therefore, the hypothesis matrix is of shape (len(plaintexts), 2).
    """
    hypothesis_matrix = np.zeros((len(plaintexts), 2))
    for (plaintext_idx, plaintext) in enumerate(plaintexts):
        for hypothesis_bit in (0, 1):
            hypothesis_key = (hypothesis_bit,) + old_key
            hypothesis_matrix[
                plaintext_idx, hypothesis_bit
            ] = NumberUtils.hamming_weight_for_rsa(
                plaintext, hypothesis_key, N
            )
    return hypothesis_matrix
```

```python
def compute_key_by_factoring() -> int:
    """Calculate the private key using prime number factoring."""
    p, q = NumberUtils.prime_factors(N)
    if p * q != N:
        raise ValueError("p * q != N")
    totient = (p - 1) * (q - 1)

    return NumberUtils.mod_inverse(PUBLIC_KEY, totient)


def compute_key_by_cpa(
    number_of_plaintext,
    number_of_points_per_trace,
    enable_plot=False,
) -> int:
    """Calculate the private key using CPA.

    The aim of this function is to build the private key gradually, bit by bit.
    At each iteration, a hypothesis bit will be added to the key and a correlation
    matrix will follow.

    Based on the correlation matrix, we select the best hypothesis.
    """
    plaintexts = tuple(
        fetch_plaintexts(number_of_plaintext=number_of_plaintext)
    )
    power_consumptions = load_power_consumptions(
        number_of_trace=number_of_plaintext,
        number_of_points_per_trace=number_of_points_per_trace,
    )  # shape (1000, 36)

    # Plot Variables
    x: np.ndarray = np.arange(number_of_points_per_trace)
    corr1: np.ndarray = np.zeros(number_of_points_per_trace)
    corr0: np.ndarray = np.zeros(number_of_points_per_trace)

    # Key start with 1 for obvious reasons. Key is little-endian.
    key_builder = [1]
    time = 1
    while time < number_of_points_per_trace:
        hypothesis_matrix = compute_hypothesis_matrix(
            plaintexts, tuple(key_builder)
        )  # shape (1000, 2)
        power_consumption_of_bit = power_consumptions[
            :, time : time + 1
        ]  # shape (1000, 1)
        correlation_matrix = NumberUtils.corr(
            hypothesis_matrix, power_consumption_of_bit
        )  # shape (2,)
        corr0[time] = correlation_matrix[0]
        corr1[time] = correlation_matrix[1]
        print(f"acc = {max(correlation_matrix[0], correlation_matrix[1])}")
        if abs(correlation_matrix[0] - correlation_matrix[1]) < 0.1:
            print("WARNING: Uncertain bit !")
        best_hypothesis = np.argmax(correlation_matrix)

        key_builder.insert(0, best_hypothesis)  # .prepend
        if best_hypothesis == 1:  # If new bit is Square-Multiply.
            time += 1  # Skip to avoid counting the same operation.
        time += 1

    # Plot
    if enable_plot:
        bar0 = plt.bar(
            x,
            corr0,
            width=0.4,
            align="edge",
            label="Hypothesis 0",
        )
        plt.bar(
            x + 0.4,
            corr1,
            width=0.4,
            align="edge",
            label="Hypothesis 1",
        )

        bar0_filtered = tuple(filter(lambda x: x.get_height() > 0.01, bar0))
        plt.text(0.0, 0.5, str(key_builder[-1]))
        for rect, bit in zip(bar0_filtered, reversed(key_builder[:-1])):
            plt.text(rect.get_x(), 0.5, str(bit))
        plt.legend()
        plt.xlabel("Time")
        plt.ylabel("Correlation Coefficient")
        plt.title(
            "Histogram of Correlation Coefficient by time and hypothesis"
        )

        plt.show()

    return NumberUtils.bit_list_to_int(reversed(key_builder))


if __name__ == "__main__":
    number_of_plaintext = 1000
    number_of_points_per_trace = 36
    enable_plot = False
    expected = compute_key_by_factoring()
    result = compute_key_by_cpa(
        number_of_plaintext=number_of_plaintext,
        number_of_points_per_trace=number_of_points_per_trace,
        enable_plot=enable_plot,
    )
    print(f"Factoring : {expected:b}")
    print(f"CPA : {result:b}")
    print(f"Equality : {result == expected}")
```