Middle East Technical University
Department of Computer Engineering

CENG 489

Spring 2019-2020

Homework 1 - Smashing RSA for fun and profit(?)!

Deadline: 10.03.2020, Tuesday, 23:59

Overview

In this assignment we will break the RSA encryption system. RSA, is a public-key cryptosystem which is commonly used in secure communication. Some examples of where you can find RSA implementations are Linux package managers(*e.g. apt,pacman*), SSL and digital signatures. In this homework we will explain and implement two different attacks on RSA.

First attack is one of the simplest attacks that can be performed on RSA and is extremely rare to happen in the real world. But it may be useful if you are interested in cryptography (and/or cybersecurity competitions). It is called "common modulus attack" and with this attack we can get the plaintext from two different ciphertexts (of the same plaintext).

Second attack is a side channel attack (SCA). In SCAs we don't try to break the cryptosystem directly, but we try exploit the implementation. We can use the extra information from time measurements, power consumption levels or even acoustic measurements along with many other side channels to find out the plaintext or cipher keys. In this assignment we will perform a power analysis side channel attack on RSA.

Attack 1: Common Modulus Attack

Consider the following scenairo:

The sysadmins of CENG issue each person in staff roster with an individual RSA key. To simplify the public key infrastructure (PKI) maintenance for the department, admins decide to generate a single modulus N and each person will be issued a personalized *public* encryption exponent e, and corresponding *private* decryption exponent d. Moreover, all key generations are performed by admins. This way the admins will also have a copy of each generated RSA key-set. They swear that they will not spy on us and propose the following reasons:

- When someone loses a key, admins can reissue the key. Possibly preventing data loss.
- In case of an internal security leak, admins can easily investigate it with all the keys available to them.

Further assume that to speed up the process of generating e, admins decide to select it from a set of pregenerated prime numbers. Therefore the greatest common divisor (gcd) of each generated e pair is 1 [$gcd(e_i, e_j) = 1$].

Now suppose that a professor wants to send the exam questions, m, to two assistants of the course. He encrpyts the questions with each asistant's respective public exponents, e_1 and e_2, to get $c_1 = m^{e_1} \pmod n$ and $c_2 = m^{e_2} \pmod n$. Later he sends c_1 and c_2 to the assistants.

In the above situation, an eavesdropper can recover the exam questions from c_1 and c_2 using the *common modulus* attack.

Recall that RSA performs encryption as follows:

$$C = M^e \pmod{N}$$

We also need to remember *Bézout's identity.*

Bézout's identity. *Let a and b be integers with greatest common divisor d. Then, there exist integers x and y such that $ax + by = d$. More generally, the integers of the form $ax + by$ are exactly the multiples of d.*

Since we know that e_1 and e_2 are chosen from a set of primes, they are co-prime, that is $gcd(e_1, e_2) = 1$. Then following the Bézout's identity, we know that there exists x and y, which satisfies $e_1x + e_2y = 1$. And we can find x and y with the Extended Euclidean Algorithm. To sum up everything we know, we have the following:

$$\begin{aligned} c_1 &= m^{e_1} \pmod{n} \text{(m encrpyted with } e_1\text{)} & gcd(e_1, e_2) &= 1 \\ c_2 &= m^{e_2} \pmod{n} \text{(m encrpyted with } e_2\text{)} & e_1x + e_2y &= 1 \end{aligned}$$

Now onto the attack. Main idea is to take the xth and yth powers of c_1 and c_2 to cancel out the public exponents e_1 and e_2.

$$\begin{aligned} (c_1)^x * (c_2)^y &= (m^{e_1})^x * (m^{e_2})^y \pmod{n} \\ &= m^{e_1x} * m^{e_2y} \pmod{n} \\ &= m^{e_1x+e_2y} \pmod{n} \\ &= m^1 = m \pmod{n} \end{aligned}$$

That's it. We broke the RSA. We also now have the exam questions!
Your task is to write a program which finds out the plaintext from the given two ciphertexts, public exponents and the common modulus.

Program Specifications

- You can write your programs in any language you choose as long as they can be run on inek machines. Python is highly recommended because it makes dealing with huge numbers easy.
- Since you can use any programming language you will have to provide a `Makefile` containing commands to compile (if necessary) and run your programs. `make run` command will be run in the directory containing your Makefile during evaluation.
- Your program should read a csv file called `crackme.csv`, which will be placed to the same path with your program, for the values of c_1, c_2, e_1, e_2 and n. First field will be the name of the parameter (i.e. one of the following: c1, c2, e1, e2 or n). Second field will be the value of the parameter in hexadecimal format.
- You should output the plaintext of the message to the `stdout` and you should output nothing else.
- Submit your programs as a zip file named `eXXXXXXXp1.zip`, where `XXXXXXX` is your student id.

Note: A sample input/output will be provided.

Attack 2: Power Analysis SCA

In this side channel attack we will monitor the power consumption of the chip performing the RSA encryption or decryption to find out the key. Instead of attacking the RSA algorithm itself, we are trying to exploit the implementation of the algorithm. We will focus on one specific implementation error. There also may be other implementation errors which allow different side channels to be exploited.

RSA utilizes modular exponentiation with big numbers. Since we deal with huge numbers exponentiation takes time. So we need an efficient algorithm for this. Luckly we have such an algorithm called exponentiation by squaring (or square and multiply). Psuedo code looks like this:

function ModPower(b, exp)
  $r \leftarrow 1$
  for bit in exp **do**
    $r \leftarrow r * r \mod n$ (Squaring step)
    if $bit == 1$ **then**
      $r = b * r \mod n$ (Multiplication step)
    end if
  end for
  return r
end function

Important thing to take away from this function is that we go over the exponent bit by bit. And remember, we are using keys as exponents to encrypt or decrypt the messages in RSA. You see where this is going?

If you were given a list of squaring and multiplication operations during the execution of this function you can find out the exponent (in our context, the key). For example if the performed operations are `[Square, Multiply, Square, Multiply, Square, Square, Square, Multiply]` we can easily say that the exponent is `11001`:

Square	Multiply	Square	Multiply	Square	Square	Square	Multiply
1		1		0	0	1	

Now think about the power consumption of the processor during this process. For simplicity assume that the processor only consumes power during squaring and multiplication.

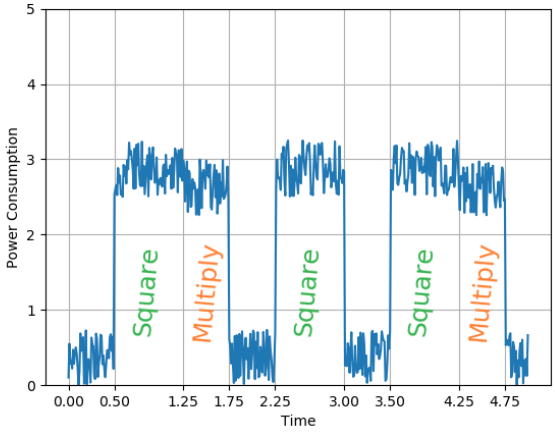

Figure 1: A Sample power trace with performed operations annotated

In the power trace if the power consumption is increased for a short duration bit is 0, and if it is increased for a longer duration bit is 1.

As you can see it is quite straightforward to get the key from a power trace. The program you will write will take in a power trace and a ciphertext and will output the plaintext.

Program Specifications

- You can write your prorams in any language of your choosing as long as they can be run on inek machines. Python is highy recommended because it makes dealing with huge numbers easy.
- Since you can use any programming language you will have to provide a `Makefile` containing commands to compile (if necessary) and run your programs. `make run` command will be run in the directory containing your Makefile during evaluation.
- Your program should read the power trace from a file called `ptrace.trc`, which will be placed to the same path with your program. This file will contain the power measurements of a chip during decryption. Each line in the file will be a single power measurement value between 0 and 15 sampled in regular intervals.
- Your program will read a ciphertext encrypted with a public key and the value of n from `stdin` as a hex string separated by a newline character and will output the corresponding plaintext to `stdout`. It shouldn't output anything other than the plaintext.
- Submit your programs as a zip file named `eXXXXXXXp2.zip`, where `XXXXXXX` is your student id.

Note: A sample input/output will be provided.