Università
Ca'Foscari
Venezia

DEPT. OF ENVIRONMENTAL SCIENCES, INFORMATICS AND STATISTICS

Master's degree programme in Informatica - Computer Science

Second Cycle (D.M. 270/2004)

Final thesis

# Extending the pwntools library with known cryptographic attacks on RSA

**Supervisor:**

Chiar.mo Prof.

Riccardo Focardi

**Graduand:**

Loris Venuto

Matriculation Number 839749

**Co-Supervisor:**

Dott. Matúš Nemec

**Academic Year:**

2017 - 2018

# Abstract

Pwntools is a widely used framework and exploit development library that simplifies the way attacks are performed during a CTF competition. Pwntools offers functions for assembling and disassembling code, ELF analysis and manipulation, communication with external services, and more; unfortunately it lacks tools to perform cryptographic attacks, that are nowadays widespread in almost every small and big CTF. The objective of this thesis is to document the process of implementing some RSA related cryptographic attacks into a fork of the library to enrich its capabilities, with the possibility that the changes will be accepted and merged into the main branch.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

RSA (named after Rivest-Shamir-Adleman, its creators) is the most famous asymmetric-key cipher, and it is very widespread with many practical applications such as message encryption and digital signatures. The infeasibility of the factorization of the modulus back into the big primes $p$ and $q$ is what guarantees the security of this cipher. The bigger the modulus, the harder it should be to factorize. The NIST (National Institute of Standards and Technology) recommends 2048 bit keys for RSA, with 3072 bit recommended if security is needed after the year 2030. Many devices on the Internet, regardless of this recommendation, still use RSA key sizes such as 512 bit or 1024 bit, the latter being more widespread. RSA security can be compromised regardless of the key size, with side-channel attacks such as the padding oracle, first presented in 1998 [5], which exploits a vulnerability used in the padding scheme PKCS #1 v1.5. A substitute for that vulnerable padding scheme is Optimal asymmetric encryption padding (OAEP), although PKCS #1 v1.5 is still used in legacy software and devices. Regardless of the padding scheme used, some RSA keys are vulnerable to an attack presented recently, in 2017, [11]. That research discovered the possibility to factorise certain RSA 1024-bit keys in 90 days in the worst case. The keys have to be generated in a specific way by the *RSALib* library, which means that the principle where the primes $p$ and $q$ have to be chosen randomly is violated. To provide further insight on these issues, they will be described in depth in this thesis, and as a project they will be implemented in the form of an easy to use Python library. This will also provide an useful tool for CTF (capture the flag) competitions, and it will be integrated into the Pwntools framework which at the moment does not provide any cryptographic attacks.

The Project is divided in three main classes:

- `PyRoca` is the main class, which serves the purpose of performing the ROCA attack
- `ParameterFinder` is a class used to calculate the parameters *M'*, *mm*, *tt* that must be obtained to perform the attack. It implements the greedy *Algorithm 2* procedure described in [11] with a small bruteforce at the end to potentially improve results.
- `PaddingOracleAttack` is the class that can be used to perform a padding oracle attack.

The programs discussed in this library, `PyRoca`, `ParameterFinder`, `PaddingOracleAttack` are avaliable via these links:

- `https://github.com/lvenuto/roca-speed`
- `https://github.com/lvenuto/PaddingOracleAttack`

After the Thesis discussion, they will also be available in a fork of the *Pwntools* library [15].

The effort for the Project is represented by the following table:

| COCOMO BASIC (ORGANIC MODE) | | | |
|---|---|---|---|
| Project | Kloc | Effort | Schedule Time(months) |
| PaddingOracleAttack | 0.127 | 0.275 | 1.531 |
| ParameterFinder | 0.582 | 1.360 | 2.809 |
| PyRoca | 0.386 | 0.883 | 2.385 |

Table 1.1: Representation of the effort for the Project, in person-months, with COCOMO, where Kloc is one thousand lines of code

## 1.1 Structure of the thesis

In Chapter 2 the RSA cipher is introduced, showing its strengths and limitations, which are fundamental to the understanding of the attacks on the vulnerabilities described in this thesis. The RSA PKCS#1 v1.5 padding scheme is also introduced.
In Chapter 3 the existing research made on the topic of RSA security vulnerabilities is analyzed. More specifically, the work done on the research paper "The Return of

Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli" [11], and the research done on Padding Oracles as explained by the article "Practical Padding Oracle Attacks on RSA" [7]. Some of the functions presented in [11] are discussed more in depth and shown with examples on real executions of the program.
In Chapter 4 a speed improved attack for the ROCA vulnerability is presented, and the thought process used to reach the improved version is explored. The implementation is then included in a fork of the Pwntools repository on GitHub.
The Chapter 5 will serve the purpose of showing benchmarks of the new implementation. Performance graphs for keys in the range 512 bit to 4096 bit will be presented, including detailed bar graphs of the most used and practically factorizable RSA key length sizes, such as 512 bit, 1024 bit.
In Chapter 6 a practical, easy to use implementation of the attack in Python 2.7 is presented, and is then included in a fork of the Pwntools repository on GitHub. In Chapter 7 the topics of this thesis are summarized, and some possible future improvements for the Python projects are presented.

## 1.2 Contributions

This thesis makes the following contributions:

- give an in depth explanation on ROCA (return of Coppersmith's attack) and POA (padding oracle) attacks implementations, with practical examples
- provide an analysis on the performance of the algorithm for recovering the RSA primes as described in [11]
- provide and integrate into the CTF (capture the flag) Python library Pwntools [15] an easy to use, automated, and faster attack by about 20%, than the one described in the original paper
- provide benchmarks on this improved algorithm
- provide and integrate into Pwntools an easy to use attack for recovering plaintexts using padding oracle attacks as described in [7]

# Chapter 2

# Background information

This chapter is dedicated to provide the reader with basic concepts of the RSA cipher, and illustrate potential vulnerabilities that some of the implementations expose. This introduction will be by no means complete, considering the vastness of the topic; the efforts had to be focused towards explaining the most important parts that fall within the scope of this thesis.

**Chapter structure**

- In Section 2.1 the RSA cipher is introduced
- In Section 2.2 the RSA padding scheme PKCS#1 v1.5 is introduced

## 2.1 RSA cipher background

We let $n = p * q$ with $p$ and $q$ two big prime numbers. We also let a and b be such that $a * b \mod \Phi(n) = 1$. $\Phi(n)$ is the Euler function which returns the number of numbers less than or equal to n that are coprime to n. We have now our public and private keys, defined as:

- $(a, n)$ for the private key
- $(b, n)$ for the public key

With the keys we can perform respectively decryption, and encryption operations.

- $D_{SK}(y) = y^a \, mod \, n$
- $E_{PK}(x) = x^b \, mod \, n$

### 2.1.1 Square and multiply

To perform the exponentiation, which is used both for encryption and decryption, we can use the Square-and-Multiply algorithm. The operation of squaring, for example to the power of $b$ for encryption, cannot be performed as $b$ multiplications: it would be too inefficient. The algorithm is easily implemented by following these steps if we want to perform $x^e$:

- Convert the exponent e to binary.
- If we encounter a 0 in e, we square x.
- If we encounter a 1 in e, we square x, then multiply by x.

The complexity of this algorithm is $O(k^3)$ in the worst case. We can also perform a modulo operation every time we square or multiply, to prevent the number from exceeding n and also to improve the performance. We can perform that operation because it is true that:

$$x * y \bmod n = (x \bmod n)(y \bmod n) \bmod n \tag{2.1}$$

It follows that:

$$(x^m)^z \bmod n = (x^m \bmod n)^z \bmod n \tag{2.2}$$

it is important to note that, for the aforementioned reason, if we perform an exponentiation operation in Python and we need it modulo n, it is faster to write:

```
result = pow(x,e,n)
```

than:

```
exp = pow(x,e)
result = exp % n
```

### 2.1.2 The Euler function

The Euler function $\Phi(n)$ returns the number of numbers less than or equal to $n$ that are coprime to $n$. Now we can define the Euler Theorem:

**Theorem 1.** Let a and n be coprime, i.e. $gcd(a, n) = 1$, then $a^{\Phi(n)} \bmod n = 1$

This definition is important later to help us pick the public and private exponents.

### 2.1.3 Inverse modulo

An operation important for RSA is the inverse modulo. The exact algorithm that is shown here is included and used in the `PyRoca` library developed as a project for this Thesis, to calculate the inverse of the parameter $M$, used for the key generation and thus needed when trying to factorise it.

```
1  def EuclidExt(c,d):
2      d0 = d
3      e = 1
4      f = 0
5      while d != 0:
6          q = c/d   # integer division
7          tmp = c - q*d # this is c % d
8          c = d
9          d = tmp
10         tmp = e - q*f # new computation for the inverse
11         e = f
12         f = tmp
13     if c == 1:
14   return e % d0 # if gcd is 1 we have that e is the inverse
```

Listing 2.1: Algorithm avaliable in [20]

The operation of inverse modulo is often used to calculate the private exponent from the public exponent, which can be generated randomly. We have to make sure that $b$ is coprime to $\Phi(n)$, but as this happens with probability $\sim 0.6$, we can just iterate the exponent generation algorithm 2 or 3 times until this is true. Often though, 65537 is used as a fixed public exponent because using a low exponent improves the performance of encryption. If we do that we have to make sure that $\Phi(n)$ is not a multiple of 65537.

### 2.1.4 Generating RSA primes

The simplest way to generate the primes $p$ and $q$ is to generate two random numbers, each one the size of half the key. For example for a 512 bit key we need two primes of 256 bit. After the generation we can test the primality of the number with a few iterations of Miller-Rabin test: The algorithm is a NO-biased Montecarlo, and it is always correct when the number is not prime but can be wrong on deciding when the number is prime. The probability of being wrong however is less than $\frac{1}{4}$ and decreases exponentially with the number of tests. With 128 runs for example the probability of being wrong is so small to be negligible. The Miller-Rabin test is very much used in practical applications: in the `ParameterFinder` class, which is part of the `PyRoca` project, for the *gen_vuln_key* function the *is_prime* function call was used. This function makes use of the Miller-Rabin

test to verify if the supplied number is prime.

## 2.2 Reasons for RSA padding

Some of the reasons of the use of padding are that textbook RSA (RSA without any padding) gives place to a series of undesirable properties. In this example we demonstrate that we can make predictable changes to ciphertexts without knowing the private key. This is possible because of the multiplicative property of RSA, which is enunciated below.

$$E(m)E(s)\ mod\ n = E(m * s\ mod\ n) \tag{2.3}$$

*RSA multiplicative property*

In other words the multiplication (modulo n) of two ciphertexts is the same as the encryption of the multiplication of the two plaintexts. In practice we can:

- encrypt the number 3 with our public exponent $b$ and obtain a ciphertext $enc3 = (3^b)\ mod\ n$
- take a ciphertext of which we do not know the decrypted value, but let's assume it is 2. This ciphertext is called *enc2*.
- if we multiply the first ciphertext *enc3* by the second one, *enc2*, we obtain a ciphertext which decrypts to the number 6. In other words we made a change to the second ciphertext knowing that the result would be multiplied by 2, this without knowing the private exponent $a$.

Another undesirable property is that textbook RSA is deterministic: we can guess what is the ciphertext simply by encrypting plaintexts ourselves with the public key and compare them with the original ciphertext, if they match, we found our plaintext. This brute force attack can work just with simple plaintexts but it is a property that is absolutely not wanted nonetheless.

### 2.2.1 PKCS#1 v1.5 padding scheme

A PKCS #1 v1.5 [9] padded message is of the form:
Where the four component are:

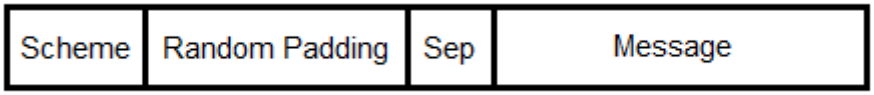


Figure 2.1: PKCS #1 v1.5 padding scheme

- Scheme is 2 bytes long, and is 0x0001 for signatures and 0x0002 for encryption. We will talk mostly about encryption scheme.
- Random Padding is at least 8 bytes long and is non-zero.
- a Separator is 0x00 and sits in between Random Padding and Message.
- the actual message we want to send.

One rule is that the whole padded message must fit the key size exactly. For example, for a 1024-bit key it must be 128 bytes. This is why random padding is of variable size to accommodate this specification. When a PKCS #1 v1.5 message is sent for decryption the following happens:

- the scheme is checked. If it does not match 0x0001 or 0x0002 an error is returned
- If the scheme is correct, the size of the random padding is checked in size, if it is less than 8 bytes an error is returned. if a 0x00 is not found before the end of the whole packet, an error is returned also.
- what lies after the separator is returned as the message

Depending on the error messages given, they can be exploited to reveal the plaintext for what is called a padding oracle attack, the oracle is the decryption server that gives us hints on the underlying plaintext.

# Chapter 3

# State of the art review

## 3.1 ROCA attack on weak RSA keys

When the basis of the RSA cipher in the Introduction were discussed, it was specified that it is important for the primes p and q to be randomly chosen. Essentially, what originated the ROCA vulnerability in *RSALib* is a poor selection of the primes that form the RSA key. As mentioned in [11, page 1], there are multiple reasons why one would not choose the primes in a pseudo-random way; Some are related to improve security (for example to adhere to NIST specifications), and some are to increase performance on low powered devices, such as security tokens. It would not be surprising if the *generator*, a number present in the generation of primes in *RSALib* as we will see later, is used because it was already in memory, as to improve performance. Unfortunately since no source code for the library is published, it cannot be said for sure. It is true that without the help of various advanced algorithms used by the ROCA attack, the keys would still be relatively safe from standard factorization techniques, even with this loss of entropy. An introduction to the basic functions and definitions of the ROCA attack will now be given.

### 3.1.1 Definitions

According to [11, page 3], all the RSA primes generated by *RSALib* are of the following form:

$$p = k * M + (65537^a \ mod \ M) \tag{3.1}$$

The integers $k$ and $a$ are the unknowns, while the integer $M$ is known and equal to some primorial (by definition the product of the first n primes):

```
while not is_prime(p):
    k = randint(min_k, max_k)
    a = randint(0, order-1)
    p = k*full_m + pow(65537, a, full_m)
```

Listing 3.1: Snippet of the key generation implemented in Python

$$\prod_{i=1}^{n} P_i = 2 * 3 * ... * P_n \tag{3.2}$$

*Definition of Primorial*

The value of $M$ is related to the key size, and is summarized in the following table:

| Size interval | M |
|---|---|
| $x < 512b$ | n/a |
| $512b \leq x < 992b$ | $P_{39}$ |
| $992b \leq x < 1984b$ | $P_{71}$ |
| $1984b \leq x < 3968b$ | $P_{126}$ |
| $3968b \leq x < 4097b$ | $P_{225}$ |

Table 3.1: M and its dependency from the key size

Now that we know that 3.1, and we know that $N = p * q$, we can write:

$$N = (k * M + 65537^a \bmod M) * (l * M + 65537^b \bmod M) \tag{3.3}$$

for $a, b, k, l \; \epsilon \; Z$

Since we are working modulo $M$:

$$\begin{aligned} N &= (65537^a \bmod M) * (65537^b \bmod M) \\ &= 65537^{a+b} \bmod M = 65537^c \bmod M \\ c &= log_{65537}(N) \bmod M = (a + b) \bmod M \end{aligned} \tag{3.4}$$

Pohlig-Hellman algorithm [14] guarantees the existence of the discrete logarithm in the group $G = [65537]$ subgroup of $Z_M^*$

The term *ord* is also defined as:

**Definition.** $ord = ord_M(65537)$

It represents the multiplicative order of 65537 in the group $Z_M^*$. The *ord* is a very important value because the cost in terms of time of the attack is given by the number of guesses (*ord*) *a*, multiplied by the time of execution of Coppersmith's algorithm. Since the number of attempts is very high even for smaller key sizes, an optimized value *M'* is necessary to make the attack practical.

### 3.1.2 Smaller parameter *M'*

From [11, page 4, section 2.3.3] we can see that by using a smaller parameter *M'* (divisor of *M*) with smaller corresponding *ord'*, the attack speed can be optimized. *ord'* is defined as:

**Definition.** $ord' = ord_{M'}(65537)$

We have to make sure that the primes are still of the form 3.1, with *M* replaced by *M'* and *a*, *k* replaced by *a'*, *k'*. The form of the modulus is still 3.3, but with new variables *a'*,*b'*,*c'*,*k'*,*l'*.

The smaller *M'* can be found with the greedy algorithm described in [11]. In the `ParameterFinder` project the greedy algorithm has been implemented, with a small brute force at the end. The strategy is to minimize $ord_{M'}(65537)$ and simultaneously maximize the size of *M'* (we must respect $\log_2(M') > \log_2(N)/4$ to use Coppersmith's algorithm). In each iteration, *ord'* is divided by some prime power divisor $p_j^{e_j}$ and the *M'* of maximal size is computed using the *Algorithm 2*, which is reported below for reference. The reward parameter used to determine the choice of the greedy algorithm is the reward-at-cost parameter, defined as:

$$\frac{\Delta size\ of\ ord_{M'}}{\Delta size\ of\ M'} = \frac{log_2(ord_{M'_{old}}) - log_2(ord_{M'_{new}})}{log_2(M'_{old}) - log_2(M'_{new})} \tag{3.5}$$

*Reward-at-cost formula, from [11]*

for $M'_{new}$ computed by *Algorithm 2* with $M'_{old}$, $ord' = ord_{M'_{old}}/p_j^{e_j}$ as input.

**Input :** primorial $M$, $ord'$ - divisor of $ord_M(65537)$

**Output:** $M'$ of maximal size with $ord_{M'}(65537)$ *which divides* $ord'$

$M' \leftarrow M$

**forall** *primes* $P_i|M$ **do**

  $ordp_i \leftarrow ord_{p_i}(65537)$;

  **if** $ord_{p_i}|ord'$ **then**

    $M' \leftarrow M'/p_i$;

  **end**

**end**

return $M'$

**Algorithm 2:** for the computation of the maximal divisor *M'* [11]

In the following table the execution of the greedy algorithm as implemented in the `ParameterFinder` class is illustrated for a starting $M$ which is used for an RSA-512 bit key. The starting M, referencing table 3.1.2 is set to $P_{39}$.

| Removed from ord | Reward-at-cost | Removed from M |
|---|---|---|
| $83^1$ | 0.86339 | [167] |
| $53^1$ | 0.84965 | [107] |
| $41^1$ | 0.84039 | [83] |
| $29^1$ | 0.82581 | [59] |
| $37^1$ | 0.72161 | [149] |
| $23^1$ | 0.35693 | [139,47] |
| $17^1$ | 0.29653 | [137,103] |
| $3^2$ | 0.25891 | [163,109,19] |

Table 3.2: Execution of the greedy algorithm as implemented in `ParameterFinder`

The resulting M' is calculated as

$$M' = \frac{M}{167 * 107 * 83 * 59 * 149 * 139...} \tag{3.6}$$

For the particular case of RSA-512, the greedy algorithm finds the optimum value of *M'* without the need of performing a tail brute force of the last *ord'* picked.

### 3.1.3 Searching guess *a'*

From *guessing strategy* [11, page 8]:

$$p = 65537^{a'} \; mod \; M' \tag{3.7}$$

The correct guess $a'$ is found by iterating through the order of the generator 65537 $ord'$ derived from $M'$. A nice optimization that can be done is searching only for p or q, and not both at the same time, effectively halving the search space and the search time. Finding one of the primes means we can easily obtain the other one: $p = N/q$ or $q = N/p$. To do this, we have to calculate our two starting guesses as:

$$\frac{c'}{2}, \; \frac{c' + ord'}{2} \tag{3.8}$$

We have no guarantees on which side of the interval will find p and which one q, we just know we are guaranteed to find one of the two primes.

### 3.1.4 Example: search of *a'*

We take as an example an RSA-512 bit key generated randomly. Let N = 0x96c5a4601c5f d4717d397e2da112c4fa3d25e65999560eae25601d5c8b82a068c4a34672c87e09a943844c153 1d8f229e205e6371b9d0298b67b90020042a533. We can now calculate c' = 648905, from the discrete logarithm of our $N$, and knowing that the generator is 65537. We now need the $ord'$ which is the smallest non-zero integer such that $65537^{ord'} \equiv 1 \; mod \; M'$ and for our $M'$ it is 1201200. We can now easily calculate the two intervals:

$$I_1 = \left[\frac{c'}{2}, \frac{c'}{2} + \frac{ord'}{2}\right]$$

$$I_2 = \left[\frac{c'}{2} + \frac{ord'}{2}, ord' - 1\right] \cup \left[0, \frac{c'}{2}\right]$$

$$I_1 = \left[\frac{648905}{2}, \frac{648905}{2} + \frac{600600}{2}\right] = [324452, 925052]$$

$$I_2 = \left[\frac{648905}{2} + \frac{1201200}{2}, 1201200 - 1\right] \cup \left[0, \frac{648905}{2}\right] = [925052, 1201199] \cup [0, 324452]$$

The correct guesses are:

$$a' = 794799 \in [324452, 925052]$$

$$b' = 1055306 \in [925052, 1201199] \cup [0, 324452]$$

Now that an example was shown, the claim that in each interval $I_1, I_2$ there is exactly one solution can be formally proved below. There are two cases:

CASE 1 Assuming that:

$$c' = (a' + b')\ mod\ ord'\ with\ 0 < c' < ord'$$
$$a' < b' < ord'\ ,\ a' + b' < ord'$$
$$\implies 0 < a' < \frac{c'}{2}\ and\ \frac{c'}{2} < b' < \frac{c'}{2} + \frac{ord'}{2}$$

Using the properties of addition of Natural Numbers:

$$a' < \frac{a' + b'}{2} < b' \implies a' < \frac{c'}{2}\ ,\ b' > \frac{c'}{2} \implies 0 < a' < \frac{c'}{2}$$
$$b' < a' + b',\ b' < ord' \implies 2 * b' < (a' + b') + ord'$$
$$\implies b' < \frac{a' + b'}{2} + \frac{ord'}{2} \implies \frac{c'}{2} < b' < \frac{c'}{2} + \frac{ord'}{2} \quad \square$$

We have proved that $a' \in I_2$ and $b' \in I_1$

CASE 2 Assuming that:

$$c' = (a' + b')\ mod\ ord'\ with\ 0 \leq c' < ord'$$
$$a' < b' < ord'\ ,\ a' + b' \geq ord'$$
$$\implies \frac{c'}{2} < a' < \frac{c'}{2} + \frac{ord'}{2}\ and\ \frac{c'}{2} + \frac{ord'}{2} < b' < ord'$$

Using the properties of addition of Modular Arithmetic:

$$(a' + b') \geq ord' \implies (a' + b') = c' + ord'$$
$$\implies c' = a' - (ord' - b') \implies a' > c' \implies a' > \frac{c'}{2}$$
$$\frac{a' + b'}{2} = \frac{c'}{2} + \frac{ord'}{2}\ ,\ a' < \frac{a' + b'}{2} < b'$$
$$\implies a' < \frac{c'}{2} + \frac{ord'}{2}\ ,\ b' > \frac{c'}{2} + \frac{ord'}{2}$$
$$\frac{c'}{2} < a' < \frac{c'}{2} + \frac{ord'}{2}\ and\ \frac{c'}{2} + \frac{ord'}{2} < b' < ord' \quad \square$$

We have proved that $a' \in I_1$ and $b' \in I_2$

### 3.1.5 Calculating *k'*

Considering $p$ of the form:

$$p = k' * M' + k_0 guess \tag{3.9}$$

where $k_0guess$ is:

$$k_0guess = 65537^{a'} \bmod M'$$

in the only unknown $k'$ as if it is a polynomial in the unknown $x$, we can use the *Howgrave-Graham* method to solve the equation $f(x) = 0 \mod p$. The polynomial $f(x) = x * M' + k_0guess$ has to be monic, so by diving by the quantity $M'$ we obtain:

$$f^*(x) = x + k_0guess * (M')^{-1}$$

By comparing with 3.9 we have that the solution (root) $x_0$ is equal to $k'$ which implies:

$$\frac{p}{M'} = x_0 + \frac{k_0guess}{M'}$$

Then:

$$k_0guess = p - x_0 * M'$$

## 3.2 Padding Oracle Attack

We first introduce the definition of a padding oracle from [7]:

**Definition.** *The padding oracle O(y) returns true if and only if RSA decryption of y is correctly padded according to PKCS#1 v1.5*

If we let $k$ be the size of the RSA modulus, and $B = 2^{k-16}$, we can write the interval of numbers which give origin to a plaintext, in hexadecimal, starting with $0x0002$; we recall that all the valid PKCS #1 v1.5 correctly padded messages have as the starting bytes $0x0002$ (at the moment we are not interested in signatures which start with $0x0001$). The said interval is [2B, 3B-1]. An example is shown to illustrate.
Let us take for the sake of the example a small size, 32 bit. If we follow our reasoning, $B = 2^{32-16}$. Now let us print out the interval with the help of Python's interactive shell.

```
1  >>> hex(2*B)
2  0x20000
3  >>> hex(3*B-1)
4  0x2ffff
```

Listing 3.2: Interval for correctly padded 32 bit messages

We can conclude that if the oracle returns True, then we know that m (the message) is in this interval.

### 3.2.1 Simulating the attack

For didactic purpose it can be useful to work directly on plaintext without passing through the encryption/decryption phase in the oracle, making this simulated attack very fast. We can change the calls to the real oracle from O(y) to sO(m). The call is modified but still checks for the compliance of m to the PKCS #1 v1.5 padding scheme. The equivalent of the operation $y' = yE(s) \mod n$, which is the multiplication of the ciphertext y for an integer s, is simulated by $m' = (m * s) \mod n$. We recall this operation is possible via the RSA multiplicative property (see 2.3).

### 3.2.2 Search intervals

We now need to compute a value $s_1$ which has the following property: $m * s_1 \mod n = m_1$ *correctly padded*. This can be done through multiple calls to the simulated oracle. Doing so gives us an $m_1$ which is also in the interval [2B, 3B-1]. We can now calculate new intervals for $m_0$.

$$m_1 = m_0 * s_1 - rn \; for \; some \; positive \; integer \; r$$

$$m_0 * s_1 = m_1 + rn$$

$$m_0 = \frac{(m_1 + rn)}{s_1}$$

Since $2B \leq m1 \leq 3B - 1$ we can write the new intervals for $m_1$ as:

$$\frac{2B + rn}{s_1} \leq m_0 \leq \frac{3B - 1 + rn}{s_1}$$

The value of r is an unknown, we can however limit its values since $r = (m_0 * s1 - m_1)/n$. The interval for r is then calculated as:

$$\frac{2B * s_1 - 3B + 1}{n} \leq r \leq \frac{(3B - 1)s_1 - 2B}{n}$$

Now we need to search for $s_1$, starting from $\frac{n}{3*B}$ because it is the smallest value of $s_1$ which gives us at least one interval.

### 3.2.3 Narrowing intervals

To narrow the interval for $m_0$, we iterate over the values of r and we intersect them with the interval of $m_0$, giving us a reduction of the search space. Only in case the intersection is non-empty, is the interval updated with the new one.

### 3.2.4 Modification for iterating the search

To iterate the search until we find one value for $m_0$ we need to modify the formula for calculating the intervals:

$$\frac{2B + rn}{s_i} \leq m_0 \leq \frac{3B - 1 + rn}{s_i} \tag{3.10}$$

*New intervals for $m_0$ at step $i$*

We need to modify slightly the formula for $r$ as well:

$$\frac{a * s_i - 3B + 1}{n} \leq r \leq \frac{b * s_i - 2B}{n}$$

*Values of r in the generic interval [a,b]*

### 3.2.5 Optimization for one interval

When we have one search interval, we can perform an optimized search. From the formula 3.10 we can derive the interval for $s_i$ as:

$$\frac{2B + rn}{m_0} \leq s_i \leq \frac{3B - 1 + rn}{m_0}$$

Since we have one search interval [a,b], we are sure that $m_0$ is in the interval [a,b] and we can write:

$$\frac{2B + rn}{b} \leq s_i \leq \frac{3B - 1 + rn}{a}$$

*The search space for $s_i$ is narrowed*

By limiting the interval of $s_i$, we make the attack converge in a number of steps that is linear in the number of bits. Thus as soon as we have one interval, we better switch to this type of search.

# Chapter 4

# Practical implementation and performance improvement of the ROCA attack

When trying to implement the ROCA attack in Python 2.7, some external libraries are required and they are reported here, in the order in which they appear in the function *coppersmith_howgrave_univariate* in `PyRoca`:

- a library to represent polynomials, needed to perform Coppersmith's attack
- a library to perform the lattice reduction
- a library for polynomial factorization

In this chapter we explore the alternatives, and try to decide which ones serve our purpose better. Am important thing to note when observing the benchmarks of this paragraph: the version of SageMath used, the 8.2, is already about 20% faster, due to updates to the library, than the version 7.5.2 that was used in the initial work by [11].

## 4.1 Representing polynomials in Python and factorization

In Coppersmith's algorithm polynomials are created two times, to "feed" the matrix used in LLL, and after the lattice reduction a polynomial is yet again generated and its roots are searched to calculate p or q. Since we are working with big numbers, and

Coppersmith's attack will be ran various times to find the prime p or q, it is imperative that the creation of the polynomial object be as fast as possible, to avoid a potential bottleneck even bigger than the LLL algorithm execution time.

### 4.1.1 Numpy package for scientific computing

When browsing the Web for scientific computation libraries, it is hard not to stumble across the Numpy library [13]. Numpy claims that "Polynomials in NumPy can be created, manipulated, and even fitted using the Convenience Classes of the `numpy.polynomial` package, introduced in NumPy 1.4". Unfortunately, we will be working with big numbers, even trying to factorize the smallest RSA key generated by *RSALib* and still used in practice, RSA 512-bit; the inability of Numpy to work with Big Integers automatically rules it out from the possible candidates.

### 4.1.2 Sympy library for symbolic mathematics

Another popular alternative is Sympy [18]. It boasts to be:

- Free: Licensed under BSD, SymPy is free both as in speech and as in beer.
- Python-based: SymPy is written entirely in Python and uses Python for its language.
- Lightweight: SymPy only depends on mpmath, a pure Python library for arbitrary floating point arithmetic, making it easy to use.
- A library: Beyond use as an interactive tool, SymPy can be embedded in other applications and extended with custom functions.

It also claims to be used by SageMath, which is the reference point of the implementation written and used in [11]. Moreover, it offers an easy to use polynomial representation and factorization that we seek. Its performance is now evaluated compared to SageMath polynomials implementation.

#### 4.1.2.1 Sympy performance compared to SageMath

To evaluate the performance, the following testing procedure was used.

- 1000 RSA-512 and RSA-1024 keys are randomly generated with the help of the `ParameterFinder` class method *gen_vuln_key()*, which is available in the final version of the Python ROCA project developed for this thesis . These two sizes are chosen because they are practically factorised even with a low expense in hardware, and are somewhat still used in practical applications (especially true for the 1024 bit size).
- for every key, a random *guess* is calculated and so is the polynomial deriving from that guess based on the formula: $x + (k_0 guess + inverse\ of\ M')\ mod\ n$
- The time to execute the various steps of Coppersmith's attack is measured and stored.

As it was mentioned in the beginning of this Chapter, it is interesting to benchmark the steps of Coppersmith's attack because that algorithm is ran multiple times, and makes use of polynomials. It was decided to divide the algorithm in 5 parts and measure time at each checkpoint. These checkpoints are: first creation of polynomials (used to populate the matrix), matrix population, execution of LLL, second creation of a polynomial (to be factored) and finally factorization.
All steps are now shown to illustrate. The code referenced is taken from the Python version called `PyRoca` developed as a project for this thesis, Coppersmith's algorithm is derived from the SageMath implementation available in [21]

```
1 ...
2         gg = []
3         for ii in range(mm):
4             for jj in range(dd):
5                 gg.append((fmpz_poly([0,XX])**jj) * fmpz_poly([n**(mm-ii)])
      * (expr(fmpz_poly([0,XX]))**ii) )
6         for ii in range(tt):
7             gg.append((fmpz_poly([0,XX])**ii) * (expr(fmpz_poly([0,XX]))**mm
      ) )
8 ...
```

Listing 4.1: First polynomials creation

```
1 ...
2     BB = IntegerMatrix(nn,nn)
3         for ii in range(nn):
4             for jj in range(ii+1):
5     BB[ii, jj] = int(gg[ii][jj])
6 ...
```

Listing 4.2: Matrix population

```
...
    W=LLL.Wrapper(BB, delta=0.5)
    W()
...
```

Listing 4.3: LLL Execution

```
...
    new_pol = fmpz_poly([0])
    for ii in range(nn):
        new_pol += (fmpz_poly([0,1]))**ii * fmpz(BB[0, ii] / XX**ii)
...
```

Listing 4.4: Second polynomial creation

```
...
    factorization = new_pol.factor()
...
```

Listing 4.5: Factorization step

Since the LLL section does not depend on polynomials (its execution time is dependant only on the speed of the LLL algorithm used), the results for it are not included in the testing results for polynomials, but they will be included in a summarizing table in the LLL section.

It should be noted that for completeness, the time of execution of Coppersmith's attack should be measured also for *correct guesses*, even if that situation happens only one time during the entire run of the attack; it will be also evident that the overall execution time of Coppersmith's algorithm is *lower* for incorrect guesses, than for correct guesses, possibly because the factorization step takes less time.

#### 4.1.2.2 Testing Hardware specifications

The hardware is summarized in the following table. Note that Coppersmith's attack runs on a single core of the processor (we will discuss later on how to scale it to multiple cores)

| Processor | Intel Core™ i3-4160 3.60 GHz |
|---|---|
| RAM | 8 GB |

#### 4.1.2.3 Testing Software specifications

The software is summarized in the following table. All the software is updated to the latest version as of the time of writing.

| | |
|---|---|
| Operating System | Ubuntu 18.04.1 LTS |
| SageMath version | 8.1 |
| Python version | 2.7.15 |
| Sympy version | 1.3 |

Table 4.1: Software used for testing the performance of Sympy against SageMath

#### 4.1.2.4 Testing results and comments

| **Execution step** | **SageMath** | **Sympy** | **Perf. Delta** |
|---|---|---|---|
| First creation of polynomials | 489 $\mu s$ | 11837 $\mu s$ | +2320.7 % |
| Matrix population time | 127 $\mu s$ | 538 $\mu s$ | +323.6 % |
| Second creation of polynomial | 259 $\mu s$ | 8006 $\mu s$ | +2991.1 % |
| Polynomial factorization time | 674 $\mu s$ | 11845 $\mu s$ | +1657.4 % |
| **Total** | 1549 $\mu s$ | 32226 $\mu s$ | +1980.4 % |

Table 4.2: Comparison between average polynomial creation time and factorization in SageMath and Sympy. Polynomials are those encountered during RSA-512 factorization

| **Execution step** | **SageMath** | **Sympy** | **Perf. Delta** |
|---|---|---|---|
| First creation of polynomials | 491 $\mu s$ | 8575 $\mu s$ | +1646.4 % |
| Matrix population time | 119 $\mu s$ | 302 $\mu s$ | +153.8 % |
| Second creation of polynomial | 283 $\mu s$ | 6589 $\mu s$ | +2228.3 % |
| Polynomial factorization time | 609 $\mu s$ | 9981 $\mu s$ | +1538.9 % |
| **Total** | 1502 $\mu s$ | 25447 $\mu s$ | +1594.2 % |

Table 4.3: Comparison between average polynomial creation time and factorization in SageMath and Sympy. Polynomials are those encountered during RSA-1024 factorization

From the benchmark results it is clear that, had the Sympy library been used in the project, it would make our program about 13.6 times slower for RSA-512 recovery, and 14.5 times for RSA-1024. We can also deduce from the *matrix population time* (it is a read operation from a Polynomial object and a write operation in the matrix), that Sympy polynomial objects are probably slow in creation, but once in memory they do not offer too much penalty over a polynomial object in SageMath, although one could

call a penalty of more than twice the time a pretty significant one. The factorization time is also a lot slower, considering that both implementations should be using the same Berlekamp-Zassenhaus factorization. At this point however, there is little to no point to continue to try and improve the results, in what appears to be a slowdown that is hard to improve without modifying the Sympy library because it is due to object creation time in the Python environment.

### 4.1.3 FLINT: Fast Library for Number Theory

The solution to the performance problem was found by analyzing what implementation SageMath uses to have this dramatic advantage on speed. It was discovered that for the operations on polynomials and factorization it uses the FLINT library, which is a library written in ANSI C. It was clear that to try to match SageMath's performance, a re-write of the ROCA attack would probably be needed. Fortunately, Python has the ability to use external C libraries with little overhead with the use of, for example, Cython [6]. This avenue was first explored because it did not require a complete re-write of the project, and because integrating Python code into the Pwntools [15] library would be trivial. A project that created bindings for some of the functions of the FLINT library was found on Github [8]. Not all the functions are implemented though. Namely, the polynomials modulo n are missing. When using Coppersmith's attack, it was verified experimentally that using polynomials without the modulo n did not impact the success rate at all. This is true only if the numbers are not negative, which in our case are not. In future work this road should be explored, implementing more functions from the FLINT C library, because as we will see later, it is a very fast, complete and easy to use library, which Sympy advertised but could not deliver. Now we test the performance difference compared to the winner of the previous test, SageMath.

#### 4.1.3.1 FLINT performance compared to SageMath

The testing procedure is the same as what already described in the sub subsection 4.1.2.1.

#### 4.1.3.2 Testing Hardware specifications

See specifications in the previous sub subsection 4.1.2.2.

#### 4.1.3.3 Testing Software specifications

The software is summarized in the following table. All the software is updated to the latest version as of the time of writing.

| | |
|---|---|
| Operating System | Ubuntu 18.04.1 LTS |
| SageMath version | 8.1 |
| Python version | 2.7.15 |
| FLINT version | 2.5.2 |
| Python FLINT version | 0.3.0 |

Table 4.4: Software used for testing the performance of SageMath against FLINT

#### 4.1.3.4 Testing results and comments

| **Execution step** | **SageMath** | **FLINT** | **Perf. Delta** |
|---|---|---|---|
| First creation of polynomials | 489 $\mu s$ | 80 $\mu s$ | -83.6 % |
| Matrix population time | 127 $\mu s$ | 160 $\mu s$ | +26 % |
| Second creation of polynomial | 259 $\mu s$ | 71 $\mu s$ | -72.6 % |
| Polynomial factorization time | 674 $\mu s$ | 452 $\mu s$ | -32.9 % |
| **Total** | 1549 $\mu s$ | 763 $\mu s$ | -50.7 % |

Table 4.5: Comparison between average polynomial creation time and factorization in SageMath and Python using FLINT. Polynomials are those encountered during RSA-512 factorization

| **Execution step** | **SageMath** | **FLINT** | **Perf. Delta** |
|---|---|---|---|
| First creation of polynomials | 491 $\mu s$ | 95 $\mu s$ | -80.7 % |
| Matrix population time | 119 $\mu s$ | 161 $\mu s$ | +35.3 % |
| Second creation of polynomial | 283 $\mu s$ | 80 $\mu s$ | -71.7 % |
| Polynomial factorization time | 609 $\mu s$ | 425 $\mu s$ | -30.2 % |
| **Total** | 1502 $\mu s$ | 761 $\mu s$ | -49.3 % |

Table 4.6: Comparison between average polynomial creation time and factorization in SageMath and Python using FLINT. Polynomials are those encountered during RSA-1024 factorization

As we can see from the benchmarks, the operations on polynomial on the Python FLINT version is about 2 times faster overall than the SageMath version for RSA-512 bit keys, and 1.97 times for RSA-1024 bit. it is interesting to see that there is a slight slowdown in the matrix population time, but in all other step the performance of Python FLINT cannot be matched. We can also see that out margin decreased when we increased the key size. This is a trend that we will see better7 later with a more complete benchmark on key sizes ranging from 512 to 4096 bit, but it is certainly a thing to keep an eye on.

#### 4.1.3.5 Testing with correct guesses

To see the complete picture we need to perform one more test. Previously it was mentioned that the benchmark was conducted generating a polynomial with random guesses that do not find p or q; Now we perform the test with polynomials that find the correct solution is calculated with the help of the `ParameterFinder` class. To calculate such polynomial we need $k_0guess$ that is obtained with the $calculate\_k_0guess(m, p, q)$ function, and since we have both primes $p$ and $q$ (remember that we have generated the keys ourselves), we have all the parameters needed; $m$ and its inverse are easily obtained. The test is restricted to only RSA-512.

| **Execution step** | **SageMath** | **FLINT** | **Perf. Delta** |
|---|---|---|---|
| First creation of polynomials | 481 $\mu s$ | 80 $\mu s$ | -83.4 % |
| Matrix population time | 127 $\mu s$ | 152 $\mu s$ | +19.7 % |
| Second creation of polynomial | 266 $\mu s$ | 71 $\mu s$ | -73.3 % |
| Polynomial factorization time | 1181 $\mu s$ | 659 $\mu s$ | -44.2 % |
| **Total** | 2055 $\mu s$ | 962 $\mu s$ | -53.2 % |

Table 4.7: Comparison between average polynomial creation time and factorization in SageMath and Python using FLINT. Polynomials are those encountered during RSA-512 factorization for a *correct* guess

We can observe an overall time increase, especially in the factorization time. This is probably because the factorization algorithm is quicker on deciding when there is no possible factorization for that polynomial. This case is not indicative of the overall performance when running the real attack for the reasons explained in 4.1.2.1.

## 4.2 LLL algorithm in Python with *fpylll*

*Fpylll* [1] is a Python wrapper for *fplll* [19], which contains implementations of several lattice routines, and the name derives from the algorithm on which it is based, LLL [10]. *Fplll* is the practical implementation of the theory described in [12] by Phong Nguyen and Damien Stehlé. It is currently the fastest LLL implementation available, having quadratic complexity.
In SageMath *fplll* is already built-in, while in Python we have to install first the wrapper and then *fplll* itself. It is interesting to see if there is any performance difference between the two, even if they use the same library version.

### 4.2.1 *fplll* performance compared, SageMath and Python

In the previous sub subsections 4.1.2.1 and 4.1.3.1, the execution time was shown for all but one step of Coppersmith's attack: the LLL execution time. The comparison will now be illustrated with a table.

#### 4.2.1.1 Testing Software specifications

| | |
|---|---|
| Operating System | Ubuntu 18.04.1 LTS |
| SageMath version | 8.1 |
| Python version | 2.7.15 |
| Python *fplll* version | 5.2.0 |

Table 4.8: Software used for testing *fplll* performance

Note: the SageMath version used includes *fplll* version 5.2.0.

#### 4.2.1.2 Benchmark results and comments

| **Execution step** | **SageMath** | **Python** | **Perf. Delta** |
|---|---|---|---|
| *fplll* execution time | 6331 $\mu s$ | 5290 $\mu s$ | -16.4 % |

Table 4.9: Comparison between average *fplll* execution time in SageMath and Python. Matrix encountered during RSA-512 factorization

| Execution step | SageMath | Python | Perf. Delta |
|---|---|---|---|
| *fplll* execution time | 6646 $\mu s$ | 5758 $\mu s$ | -13.4 % |

Table 4.10: Comparison between *fplll* execution time in SageMath and Python. Matrix encountered during RSA-1024 factorization

Surprisingly, from the benchmark it emerges that *fplll* is faster when executed in Python rather than in SageMath, even when using the same version of the procedure. Now we can summarize the overall execution time improvement we gained over the SageMath version.

## 4.3 Overall improvement from Python ROCA

Now that all the improvements have been explained and documented, we can summarize how much time improvement we can expect in total with the polynomial and *fplll* speedup at the same time. Coppersmith's attack execution time is the total sum of all the execution steps in Coppersmith's routine.

| Execution step | SageMath | Python | Perf. Delta |
|---|---|---|---|
| Coppersmith's attack total time | 7880 $\mu s$ | 6053 $\mu s$ | -23.2 % |

Table 4.11: Comparison between total execution time in SageMath and Python. Matrix and polynomials are encountered during RSA-512 factorization

| Execution step | SageMath | Python | Perf. Delta |
|---|---|---|---|
| Coppersmith's attack total time | 8148 $\mu s$ | 6519 $\mu s$ | -20 % |

Table 4.12: Comparison between total execution time in SageMath and Python. Matrix and polynomials are encountered during RSA-1024 factorization

From Python ROCA we can expect a 23.2% time saving for RSA-512, decreasing to 20 % for RSA-1024 bit keys.

## 4.4 Making the attack multi-process

In the current state of things, the *fpLLL* algorithm is not using more than 1 core for its operations. It was decided to use the "multiprocessing" package in Python instead

of using multithreading because of the GIL issue. The GIL is an interpreter-level lock which prevents execution of multiple threads at once in the Python interpreter. Each thread that wants to run must wait for the GIL to be released by the other thread, which means a multi-threaded Python application is actually single threaded. `PyRoca` runs automatically creates as many processes as there are core in the processor. If the number of cores cannot be determined, then it launches a single process.

### 4.4.1 The *batch_size*

The processes receive each a number of *guesses* as instructed by the *batch_size* variable. This variable, if not otherwise specified by the user, is set to 100, but it should be set accordingly to not bottleneck the system: if this number is too small, the processes will come back asking for more work frequently, and since the assignment of job is done in mutual-exclusion we will have serious slow-downs. The mutual exclusion is obtained with a lock object in Python and it is necessary because we do not want to have concurrency problems in our program; There is not really a drawback with having *batch_size* too big, with the exception of *guesses* that take very long to compute, in that case we would set it very small. With many number of cores (it was tested with upto 120 cores) it is normal to have 10-20 seconds of wind up time to assign jobs, but once the processes get their job, the interaction with the main thread is kept at a minimum by design.

### 4.4.2 The queue

To share the guesses with the processes a Python queue object is used. Every process runs through its queue and goes back to the main thread for more work when it is empty. A total of the attempts made is also updated to keep track of the statistics, so it is easier to calculate the number of attempts made per second, for example.

### 4.4.3 Termination

There are three instances in which the processes are stopped:

- we exhausted all the possible *guesses* (we ran through the entire ord')
- we found the solution
- termination (ctrl-c) was issued from the user

In the first case, we do not have to check for anything, the *join* in the main process will wait for the correct termination and tell us that no solution was found.
If a solution was found, we would like to terminate all the other processes because they cannot possibly find the solution in their batches of guesses. A multiprocessing *Event* is used for this purpose: at the end of every guess tried, all the processes check if this termination event has been triggered, and if it has then we gracefully exit. This event is checked after every guess because we do not want any delay in the termination. Unfortunately some guesses for high key sizes (>2048 bit) take up to 15 minutes each. We can expect in this case a very delayed termination. In future releases of the program a timeout timer could be added in the main process that forcefully terminates any processes that have not terminated in time to fix this issue. The problem has not been addressed in the initial release because keys that big are not practically factorizable with running times in the thousands of years.

### 4.4.4 Scaling on multiple processes

Intuitively, we would expect the performance to scale linearly with the number of cores in the processor, if the program is well optimized (no time is wasted in the mutual exclusion). To test this assumption, a benchmark on factorization of an RSA-1024 bit key on Amazon EC2 c5.9xlarge instance has been performed. More information on the hardware used are available in the Benchmark Paragraph.

#### 4.4.4.1 Test results

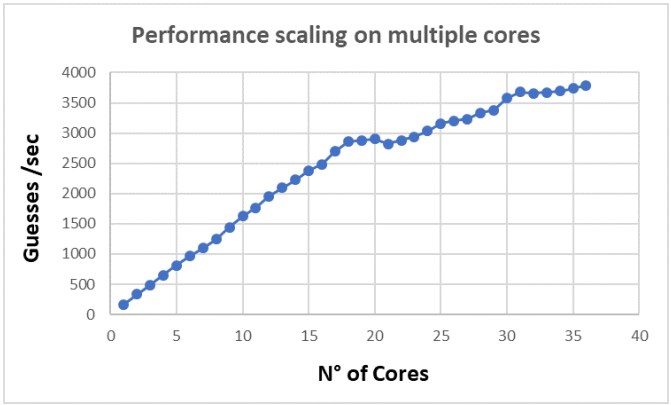


Figure 4.1: Performance scaling in `PyRoca` during factorization of RSA-1024 on 16 cores physical cores, 36 logical

The performance scales linearly up until about 18 cores. Coincidentally this is the number of physical cores in the Intel Xeon Platinum 8124M that we are testing. This processor uses Intel Hyper Threading technology to achieve the 36 core count. This means that of our 36 cores, half of them have to share the resources.

# Chapter 5

# PyRoca Benchmarks

In this chapter some benchmarks are presented on the full range of key sizes used by *RSALib*. Time for factoring RSA-512, RSA-1024 and RSA-2048 are given to compare with the speed of the old SageMath version. As mentioned in the beginning of Chapter 4, the comparison will be with the latest version, 8.2, which is already about 20% than the version used in the work by [11]. Additionally, since Amazon EC2 (Elastic Compute Cloud) instances were used for the benchmarks, and cost for cracking RSA-512, 1024, 2048 bit keys were included in the original paper, the cost estimate is updated with the new values from the improved version. Finally, a method of reducing the cost with Amazon EC2 Spot instances is discussed.

## 5.1 Amazon EC2

For the purpose of this test, it was decided to use the Amazon EC2 infrastructure, mainly because of the cost per performance advantage over the competitors, and also because of some techniques that can be put in place to save on the instances cost. This technique is using Amazon Spot instances and will be discussed in a later section. To perform the test C5 instances are selected: they are optimized for compute-intensive workloads. As `PyRoca` does not leverage the power of a video card for example, it would not make sense to rent an instance with that capability. For the test a c5.large and c5.9xlarge instance was used, respectively to test single core performance and scaling on multiple cores.

### 5.1.1 Virtual CPUs on the instances

An important thing to note is the concept of vCPUs on Amazon EC2: as most of the machines available are virtualized, we talk about virtual CPUs (vCPUs). For example, a c5.9xlarge instance has 18 CPU and 2 threads per core, or 36 vCPUs in total. A c5.large instance has 1 CPU and 2 threads per core, so 2 vCPUs in total. Regarding this last example, since we want to test single core (and single thread) performance, one of the virtual CPUs was disabled (although specifying to `PyRoca` that it should use only 1 core should be enough, as virtual cpu threads are not transparent to the OS, and show as real CPU cores).

### 5.1.2 Software

The software used in Amazon EC2 is summarized in the following tables.

| Software | |
|---|---|
| Operating System | Ubuntu 18.04.1 LTS |
| SageMath version | 8.1 |
| Python version | 2.7.15 |
| Python FLINT version | 0.3.0 |

Table 5.1: Software used for testing on Amazon EC2 instances

### 5.1.3 Hardware

All the hardware is virtualized in the Amazon EC2 cloud.

| | c5.large instance | c5.9xlarge instance |
|---|---|---|
| CPU | Intel Xeon Platinum 8158 3.0 GHz | |
| Number of vCPUs | 1 | 36 |
| RAM | 8 GB | 72 GB |
| Storage | 10 GB EBS volume | |
| Networking | Upto 10 Gbps | 10 Gbps |

Table 5.2: Hardware used on Amazon EC2 instances

### 5.1.4 Benchmark procedure

- random *RSALib* keys are generated with the *gen_vuln_key* function available in the `ParameterFinder` class and stored in a file. The keys should be per *RSALib* spec. The keys generated are in the range 512 to 4096 bit in step of 32 bit. The number of keys in the specified interval is 129.

- For each key in the aforementioned interval, the attack using a slightly modified `PyRoca` class is launched. The normal behaviour of the library had to be adjusted because we have to account that we could run into the *correct guess* and thus find the key or we could exhaust the possible candidates before the benchmarking time is up. The choice of benchmarking time for each key is explained in the next subsection.

### 5.1.5 Benchmark results

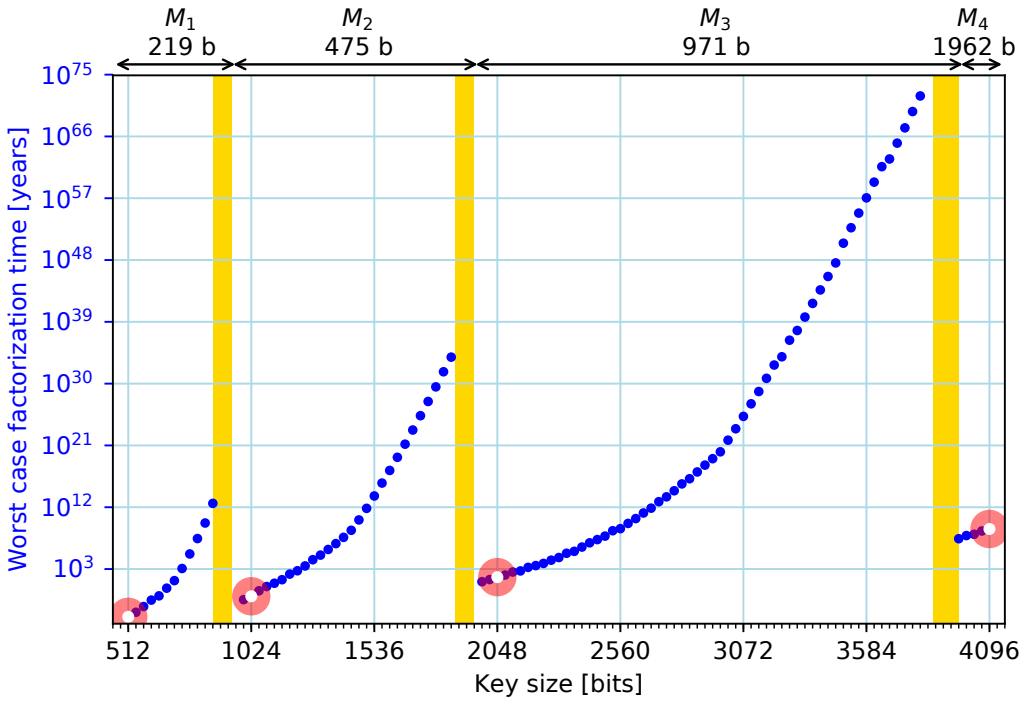


Figure 5.1: Performance of `PyRoca` with varying key size up to 4096 bit in 32 bit steps, graph generated with the help of an external script by [11]

#### 5.1.5.1 Comments

Unfortunately the granularity of the axes does not allow to notice the difference in performance with the graph from [11]. For this reason, more detailed graphs will be presented with regards to practically factorizable keys such as RSA-512 and RSA-1024 bit keys. Since the benchmark results purpose is to be compared with the results from [11], the same optimal parameters *mm*, *tt* and *M'* are used. Some of these parameter combination used for the benchmark do not give a 100% success rate in Coppersmith's algorithm (it does not always find p or q). In the current version of the `ParameterFinder` class, it was preferred to return only parameter combinations that have a 100% success rate, hence why it would be hard to reproduce all of the results exactly like they are presented here. Some of the popular key sizes such as RSA-512, 1024, 2048 are not affected by this issue.

#### 5.1.5.2 Time estimate (worst case)

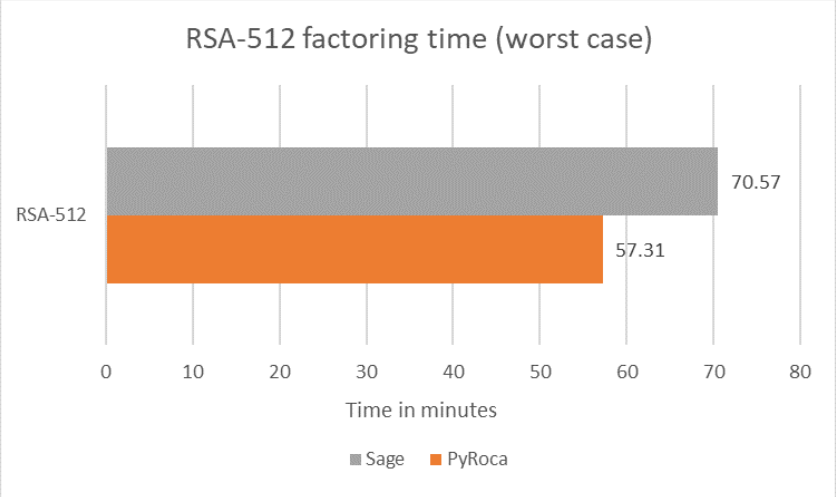


Figure 5.2: Performance of `PyRoca` compared against the SageMath version for RSA-512 bit keys

As we can observe from the data for the worst case on RSA-512, we have a decrease in execution time of 18.79%, which is less than what was measured in the previous chapter during polynomial and LLL testing. There are a couple of reasons that can explain this difference:

- In this test the sample size is only one. Only one key was tested during RSA-512

testing, but multiple different guesses were tested during the span of the test. This could have impacted the results. Testing multiple keys and multiple guesses was infeasible given the number of resources avaliable.

- the hardware used for the test, unlike the test on the polynomial and LLL speed, is done on a virtualized machine in the cloud. Amazon EC2 does not guarantee any minimum performance for the vCPUs, some variance in the data result is expected.

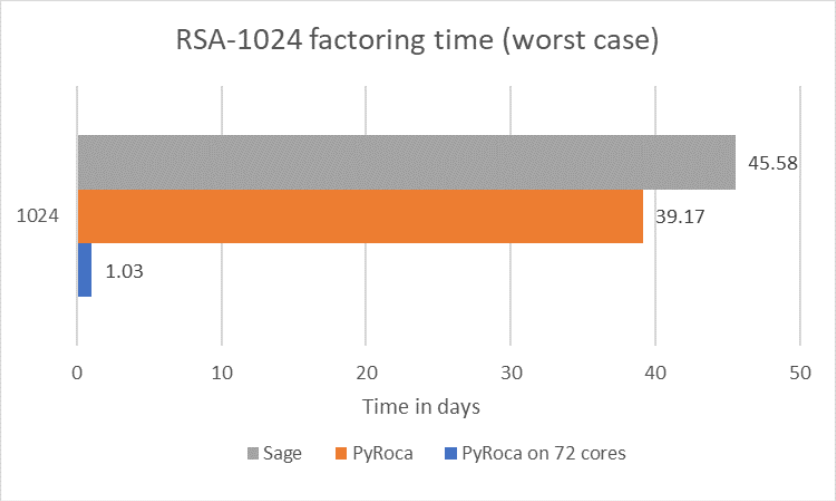


Figure 5.3: Performance of `PyRoca` compared against the SageMath version for RSA-1024 bit keys, additionally `PyRoca` is ran on all the cores avaliable on the machine

For RSA-1024 `PyRoca` was also allowed to use all the cores of the machine. We can deduce that with some relatively powerful hardware we can we can factor any ROCA weak RSA-1024 keys in a little over a day.

### 5.1.6 Cost per performance

In the following table we summarize the worst case cost to factor popular RSA-512, 1024, 2048 bit keys on EC2 c5.large and c5.9xlarge instances, at the current price per hour, which is respectively 0.085$ and 1.53$.

| c5.large instance | |
|---|---|
| Key size | Worst case cost |
| RSA-512 | 0.085$ |
| RSA-1024 | 69.28$ |
| RSA-2048 | 38053$ |

| c5.9xlarge instance | |
|---|---|
| Key size | Worst case cost |
| RSA-512 | 1.53$ |
| RSA-1024 | 61.83$ |

Table 5.3: Cost per factoring various key sizes. The data is extrapolated from previous benchmarks

There are a few things to note in these tables. First of all the cost of factoring RSA-512 is equal to the cost of the instance, because the least fraction of time Amazon EC2 rents for is 1 hour; it would not make sense to rent a big instance for an RSA-512 key, unless we want the result in a couple of seconds, but if we can wait an hour the first option is way more cost effective. Another thing to note is that since some of the benchmarks done in c5.large were working on single process (to compare to the result of SageMath), and the instance offers 1 core and 1 logical thread, it follows that the cost per performance for RSA-1024 and RSA-2048 would be not accurate, had we used the values from the previous benchmark; the instance can offer more performance than that. We can notice that is cheaper to run a big c5.9xlarge instance, instead of many c5.large. This can be not always true, in this case the performance per core of the big instance was higher, probably because the cpu wasn't shared amongst other EC2 customers. With the help of `PyRoca` the choice is made easier, because we do not have to split manually the workload on the c5.9xlarge instance, it is done automatically for us. All of the cost estimates are lower than what documented in [11], partly because of the faster algorithm, and partly because of the ever changing Amazon EC2 offering cost and performance.

### 5.1.7 Amazon Spot Instances

Finally we discuss a possible way to save on factorization cost. Spot instances are normal instances that have been unused; to avoid wasting this possible revenue, Amazon rents these instances for a much lower than standard price. The problem is that these instances can be terminated if we get out-bidded by another user of the platform; a bid is the maximum price we are willing to pay. The use of spot instances can become a possibility

in the future with an update to `PyRoca`: what is needed is a resume function, saving the last guess made, so even if our instance is terminated we could start again once the value of our bid becomes relevant again. With this technique, we can save up to 40% on the price of the instance, which for big keys can be very relevant.

# Chapter 6

# Practical implementation of a Padding Oracle Attack

A practical implementation based on the examples from [7] has been written in Python 2.7, ready to be integrated into the Pwntools repository fork. It only uses as an external library the *rsa* library (see [17]) for encryption and decryption of PKCS#1 padded ciphertexts.

## 6.1 Modification for the real implementation

A small modification of the examples from [7] was required to have a working implementation on real ciphertexts. In the introduction we shown examples on the messages, unfortunately now we do not have that luxury because the message we want is encrypted. The most important modification is the search for $s_1$.

```
1 def _search_s1(self, start_value, y0):
2         s1=start_value #starting value of s1
3         while True:
4             y1 = (self._encrypt_int(s1) * y0) % self.n
5             if(self.padding_oracle(y1)):
6                 break
7             s1 += 1 #try next value of s1
8         return s1
```

Listing 6.1: Search for $s_1$ with ciphertexts

The encryption functionality as it was said in the beginning of the chapter, is given from an external library. The call simply performs an encryption of an integer number. The encryption could have been implemented as a function inside the class, but since a PKCS#1 v1.5 padding implementation could be useful and is provided by the same library, it was decided to rely entirely on it, without adding duplicates of Python code.

## 6.2 Initiating the attack

The usage of the library has been made with modularity in mind. It was not made for a specific application, so the user needs to supply a function used to communicate with the oracle; the oracle can be a webserver, a local program etc. The function must be written to return a boolean value of True or False after it communicated with the oracle. The only input parameter of the function must be a string which will be generated with the Bleichenbacher attack [5] as implemented in this library. When the attack is initiated, the object must also be supplied with the following parameters:

- k: the size of the RSA key in bits used to encrypt $y_0$ (e.g. 1024 bit RSA key)
- n: the modulus n
- b: the public exponent of the RSA key
- $y_0$: the ciphertext to recover

## 6.3 Usage example

The usage is illustrated with a simple example.
We can start by generating an RSA keypair of 1024 bit using openssl:

```
1 openssl genrsa -out key 1024
```

Then we can import the generated public and private key into the RSA library.

```
1 #Reading the public key
2 f = open("key.pub", "rb")
3 pubkey = f.read()
4 f.close()
5 #Reading the private key
```

```
f = open("key", "rb")
prkey = f.read()
f.close()
#Importing them into the rsa library
prkey = rsa.PrivateKey.load_pkcs1(prkey, format='PEM')
pubkey = rsa.PublicKey.load_pkcs1_openssl_pem(pubkey)
#saving the modulus n
n = pubkey.n
```

Since we need to provide a function that talks to the Oracle, we write a simple oracle function that decodes the ciphertext, checks if the padding is correct, and returns a true or false boolean value. To speed up the attack, we make the fastest oracle possible: it returns true only if the message padding starts with 0x0002 and does not check additional parameters. This guarantees the minimum number of oracle calls, for the sake of this example.

```
def simulated_oracle(ciphertext):
    plaintext = decrypt_int(ciphertext) #decrypt the
    ciphertext with the rsa priv. key
    plaintext = long2hexstr(plaintext) #convert the
    long to hexstring
    plaintext = plaintext.decode("hex")
    if(plaintext[0] != '\x00' or plaintext[1] != '\x02'
    ):
        return False
    else:
        return True
```

Finally we generate a ciphertext and launch the attack:

```
y0 = encrypt_pkcs('testciphertext')
theattack = poa.PaddingOracleAttack(1024,n,pubkey.e,y0,
    simulated_oracle)
plaintext = theattack.run_attack()
```

When the attack terminates we can print the plaintext. With our example we get: 0002...007465737463697068657274657874. Naturally this is a PKCS #1 v1.5 formatted message. We recall that everything after the separator is the message. In fact for example

in Python 2.7 we can print the plaintext like so:

```
>>> ('74657374636970686572746574657874').decode('hex')
'testciphertext'
```

## 6.4 Speed of the attack

The speed of the attack depends mainly on how fast the $s_1$ that gives a correctly formatted ciphertext can be found. A benchmark was executed on randomly generated ciphertexts to evaluate performance. Since the speed of the attack depends on the number of oracle calls, the hardware used for the test is not relevant. Of course in a real scenario we have to keep in mind the response time of the oracle when we evaluate the performance, especially if we talk about a web server or a low power security token.

| Oracle Calls | | | |
|---|---|---|---|
| Mean | Median | Min | Max |
| 30531 | 20911 | 2669 | 188822 |

Table 6.1: Statistics on the number of oracle calls

From the benchmark results we can see that we get similar results as the “Good” Oracle from [7]. This is expected because in this test an unoptimized attack was used, and the oracle used fits the description of the “Good” oracle case of that table.

# Chapter 7

# Conclusions

In this thesis we have given the basis of the RSA cipher, and then discussed two practical attacks on security of RSA. For the ROCA attack, an improved version about 20% faster was shown; the speed increase was documented with benchmark to determine the function calls responsible for the speedup. The attack was also made multi process to leverage multi core processors, and the performance scaling was graphed. For the padding oracle attack, a simple and easy to use version was presented with usage examples on a real encrypted ciphertext. The performance of the attack was then evaluated in terms of oracle calls.

In future versions of this project, for what concerns the ROCA attack, a resume function of the attack is needed to allow the use of the much cheaper Amazon spot instances. It should be possible to further speed up the attack by implementing the ideas described in [4] and by using the very recent General Sieve Kernel [2] together with *fplll*. The padding oracle attack can be highly sped up by implementing the techiniques described in the paper [3]. Furthermore a recent paper [16] mentions the ability to parallelize the attack on multiple servers sharing the same public key.

# Bibliography


[1] Martin Albrecht. *A Python interface for fplll*. URL: https://github.com/fplll/fpylll.

[2] Martin Albrecht. *The General Sieve Kernel (G6K)*. URL: https://github.com/fplll/g6k.

[3] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J. Tsay. "Efficient Padding Oracle Attacks on Cryptographic Hardware." In: *CRYPTO 2012* 7417 (2012), pp. 608–625.

[4] Daniel J. Bernstein and Tanja Lange. *The cr.yp.to blog*. URL: https://blog.cr.yp.to/20171105-infineon.html.

[5] Daniel Bleichenbacher. "Chosen ciphertext attacks against protocols based on the RSA encryption standard". In: *Advances in Cryptology: Proceedings of CRYPTO'98* 1462 (1998), pp. 1–12.

[6] *Cython C Extensions for Python*. URL: https://cython.org/.

[7] Riccardo Focardi. *Practical Padding Oracle Attacks on RSA*. 2011. URL: http://secgroup.dais.unive.it/wp-content/uploads/2012/11/Practical-Padding-Oracle-Attack-on-RSA.html.

[8] Fredrik Johansson. *Python extension module wrapping FLINT*. URL: https://github.com/fredrik-johansson/python-flint.

[9] B. Kaliski. "PKCS #1: RSA Encryption Version 1.5". 1998.

[10] A.K. Lenstra, H.W. Lenstra, and Lovász. "Factoring polynomials with rational coefficients". In: *L. Math. Ann.* 261 (2009), pp. 515–534. DOI: https://doi.org/10.1007/BF01457454.

[11] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. "The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli". In: *24th ACM Conference on Computer and Communications Security (CCS'2017)*. ACM, 2017, pp. 1631–1648. ISBN: 978-1-4503-4946-8/17/10.

[12] Phong Q. Nguyen and Damien Stehlé. "An LLL algorithm with quadratic complexity". In: *Siam J. Comput.* 39 (2009), pp. 874–903. DOI: 10.1137/070705702.

[13] *Numpy package for scientific computing with Python*. URL: http://www.numpy.org/.

[14] S. C. Pohlig and M. E. Hellman. "An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance". In: *IEEE Trans. on Info. Theory* IT-24 (1978), pp. 106–110.

[15] *Pwntools Github repository*. URL: https://github.com/Gallopsled/pwntools.

[16] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. *The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations*. Cryptology ePrint Archive, Report 2018/1173. https://eprint.iacr.org/2018/1173. 2018.

[17] Sybren Stüvel. *pure-Python RSA implementation*. URL: https://stuvel.eu/rsa.

[18] *Sympy Python library for symbolic mathematics*. URL: https://www.sympy.org/.

[19] The FPLLL development team. "fplll, a lattice reduction library". Available at https://github.com/fplll/fplll. 2016. URL: https://github.com/fplll/fplll.

[20] *The RSA cipher*. URL: https://secgroup.dais.unive.it/teaching/cryptography/the-rsa-cipher.

[21] David Wong. *Lattice based attacks on RSA*. URL: https://github.com/mimoo/RSA-and-LLL-attacks.