Stéganographie et RSA

Yann Trou, Matthieu Joulain

Mai 2022

Table des matières

Chapitre 1

Introduction

Ce projet a pour objectif de camoufler un message crytpé dans une image et de pouvoir le récupérer. Pour cela on utilise la stéganographie, processus qui remplace les bits de poids faible des pixels de l'image par ceux d'un message crypté, altérant peu l'image finale par rapport à l'originale. Pour encrypter un message, on utilisera le chiffrement asymmétrique RSA avec des paires de clés stockées dans des fichiers.

Chapitre 2

RSA

L'algorithme RSA est apparu à une époque où l'on cherchait un moyen de communiquer sans qu'il y ait besoin de s'échanger des clés secrètes, qui était la principale faille des systèmes cryptographiques. L'algorithme RSA repose sur le principe des fonctions à sens unique avec trappe : étant donné un x, il est facile de calculer *f(x)* mais étant donné *f(x)*, il est très difficile de retrouver x, sauf si l'on connait une trappe t. Ainsi, RSA nous donne un couple de clés publiques (e, n) permettant de chiffrer un message et un couple de clés secrètes (d, n) servant de trappe pour l'algorithme.

2.1 Algorithme général [2]

- On choisit deux nombres p et q premiers de grande taille (500 chiffres ou alors 1024 bits) distincts.
- On calcule le module de déchiffrement $n = p * q$
- On calcule la valeur de l'indicatrice d'Euleur $phi(n) = (p-1) * (q-1)$
- On calcule l'exposant de chiffrement e copremier avec phi et strictement inférieur à phi
- On calcule l'exposant de déchiffrement d, inverse de $e \mod phi$, strictement inférieur à phi.

On obtient le couple (e, n), clé publique et (d, n) clé privée.

Pour crypter un message, il faut que le message soit un entier naturel strictement inférieur à n. On obtient alors $C = m^e \mod n$ z Pour décrypter le message, on effectue $m = C^d \mod n$

2.2 Test probabiliste de primalité : Miller-Rabin [8]

Le test de primalité de Miller-Rabin est un test probabiliste qui, pour un nombre, indique s'il est probablemenent premier ou non. Le test est basé sur le petit théorème de fermat indiquant que pour tout p premier et a entier non divisible par p alors $a^{p-1} \equiv 1 \mod p$. On propose $p > 2$ premier, s et d deux entiers avec s non nul et d impair tels que $p-1 = 2^s * d$, alors pour tout entier a non divisible par p on a $a^d \equiv 1$. Si au contraire $a^d \neq 1(\mod p)$ alors p n'est pas premier, il est composé et on considère a comme un témoin de miller. Après, même si la proposition est validée, p n'est pas forcément premier, il est fortement probablement premier. Il faut donc répéter le test avec plusieurs entiers a indépendants pour que la probabilité que p ne soit pas premier devienne très faible. Le test de miller-rabin s'appuie sur le fait qu'un nombre composé possède toujours un témoin de miller, contrairement à un nombre premier.

```
let miller_rabin n a =
  (*n-1 = 2^s * d, retourne le couple (d, s)*)
  let rec miller_rabin_sd nm1 s =
    let nm1_shift = bsr nm1 s in
    let last_bit = band nm1_shift (i2b 1) in
    if (eq_big_int last_bit (i2b 0)) = true then miller_rabin_sd nm1 (s + 1)
    else ((b2n nm1_shift), s)
```

```
  in

  let (d, s) = miller_rabin_sd  (n2b (n -/ one)) 0 in

  let x = mod_pow a d n in

  if x =/ one || x =/ (n -/ one) then false
  else
    let rec miller_rabin_aux i =
      if i = (s - 1) then true
      else
        let y = mod_num (x */ x) n in
        if y =/ (n -/ one) then false
        else miller_rabin_aux (i + 1)
    in miller_rabin_aux 0;;
;;


let is_prime n k =
  let rec is_prime_aux i =
    if i =/ k then true
    else
      let a = 2 + Random.full_int (n2i (n -/ two)) in
      (*Printf.printf "a: %d\n" a; *)
      let a_num = i2n a in
      if miller_rabin n a_num = true then false
      else is_prime_aux (i +/ one)
  in is_prime_aux zero;;
;;
```

2.3 PGCD Binaire [1]

Deux entiers sont premiers entre eux si leur plus grand diviseur commun est 1. Pour trouver *e* copremier avec *phi*, on va tester tous les chiffres de 2 à *phi* et calculer leur PGCD. Afin de ne pas saturer la pile et d'optimiser les calculs, on va utiliser l'algorithme du PGCD binaire. C'est un algorithme récursif qui utilise les règles suivantes pour deux entiers a et b, avec a supérieur à b :

- a = 0 : retourne b
- a pair, b pair : 2 * pgcd(a/2, b/2)
- a impair, b pair : pgcd(a, b/2)
- a pair, b impair : pgcd(a/2, b)
- a impair, b impair : pgcd((a-b)/2, b)

```
let rec bin_pgcd a b =
  let find_sup a b = if a >/ b then a else b in let sup = find_sup a b in
  let find_sub a b = if a >/ b then b else a in let sub = find_sub a b in
  if a =/ zero then b
  else if (mod_num sup two =/ zero) = true && (mod_num sub two =/ zero ) = true then two */
  ↪  (bin_pgcd (sup // two) (sub // two))
  else if (mod_num sup two =/ zero) = false && (mod_num sub two =/ zero) = true then bin_pgcd sup
  ↪  (sub // two)
  else if (mod_num sup two =/ zero) = true && (mod_num sub two =/ zero) = false then bin_pgcd (sup
  ↪  // two) sub
  else bin_pgcd ((sup -/ b) // two) sub
;;

let rec find_coprime i phi=s
  if i >=/ phi then i2n (-1)
  else if bin_pgcd phi i =/ one then i
  else find_coprime (i +/ one) phi
;;
```

2.4 Algorithme d'euclide étendu [3]

Pour calculer d inverse de $e \mod n$ on utilise l'algorithme d'euclide étendu. Cet algorithme est une extension du PGCD dans le sens ou il retourne le PGCD mais aussi un couple de coefficients de Bézouts tels que $ns + at = PGCD(a, s)$. Dans le cas où a et s sont premiers entre eux, t est alors l'inverse de $a \mod n$. On peut donc simplifier l'équation par $at \equiv 1 \mod n$

```
let extended_euclid a n =
  let rec extended_euclid_aux t r newr newt =
    if (newr =/ zero) = false then
      let quo = quo_num r newr in
      let _t = newt in
      let _newt = t -/ (quo */ newt) in
      let _r = newr in
      let _newr = r -/ (quo */ newr) in
      extended_euclid_aux _t _r _newr _newt
    else
      (r, t)
    in

  let (r, t) = extended_euclid_aux zero n a one in
  if r >/ one then (i2n (-1))
  else if t </ zero then (t +/ n)
  else t
;;
```

2.5 Exponentiation modulaire [6]

Pour le cryptage / décryptage de messages, on utilise un algorithme d'exponentiation modulaire. Avec de grands nombres premiers, c'est cette fonction qui nous sert de fonction à sens unique (avec trappe). La méthode naive effectue la puissance puis le modulo. Ce n'est pas optimal car le calcul de la puissance va demander énormément de ressources. A la place, on peut exploiter le fait que si l'exposant est pair, alors $a^e = a^{e/2} * a^{e/2}$. Cela permet de diviser par deux le nombre de calculs mais la puissance de a reste trop large. On peut introduire le modulo lors des étapes intermédiaires pour réduire le volume de calcul de la puissance afin d'accélérer la méthode.

```
let rec mod_pow x y n =
  let base = mod_num x n in
  if y =/ zero then i2n 1
  else if y =/ one then x
  else
    if (mod_num y two) =/ zero then mod_pow (base */ ( mod_num base n)) (y // two) n
    else mod_num (base */ (mod_pow base (y -/ one) n)) n
  ;;
;;
```

2.6 Limites

La principale limite de notre implémentation de l'algorithme RSA provient du générateur de nombres aléatoires. OCaml, via son module Random, propose des entiers aléatoires de 32 ou 64 bits au maximum ($2^{32} - 1$ ou $2^{64} - 1$). Ce problème peut être contourné en tirant deux nombres aléatoires et en mettant l'un en exposant de l'autre. La vraie problématique est liée au test de miller-rabin. Nous n'avons pas trouvé comment générer de nombres aléatoires dans l'intervalle [2, n-1] avec le contournenment ci-dessus. L'autre limite tient du fait que nous n'avons pas implémenté de padding pour les messages qui seraient trop petits pour l'image. La solution aurait été d'utiliser l'Optimal Asymmetric Encryption Padding [4], méthode de remplissage utilisée dans la plupart des cas avec RSA.

Chapitre 3

Stéganographie

Pour ouvrir et traiter une image on utilise Graphics et CamlImages. On récupère un message codé en dur, que l'on va transformer en tableau de int, chaque int représentant un bit d'un octet d'un caractère ASCII du message.

```
let string_to_bits s =
  String.fold_right (fun car out -> let ascii = Char.code car in
  let rec ascii_to_bits ascii (out,acc) =
    if acc < 8 then
      if ascii/2 = 0 then ascii_to_bits (ascii/2) (((ascii mod 2)::(out)),acc+1)
      else ascii_to_bits (ascii/2) (((ascii mod 2)::(out)),acc+1)
    else
      out
  in
  ascii_to_bits ascii (out,0)) s []
;;
```

On va ensuite transformer cette suite de bits en grand nombre (num) en parcourant le tableau de droite à gauche et en les accumulant au format décimal.

```
let bits_to_num bits =
  let (number,index) = List.fold_right (fun bit (acc,n) -> if bit = 0 then (acc,n+1)
  else (acc +/ ( i2n (bit) */ (two **/ (i2n n))),n+1)) bits (zero,0) in
  number;;
```

On va pouvoir le crypter via l'algoritme RSA précisé ci-dessus, avec des clés générées aléatoirement, pour obtenir le message crypté. On transforme le message crypté (de type num) en un tableau de int représentant les bits du nombre

```
let num_to_bits num =
  let rec num_to_bits_aux num acc =
    (*Printf.printf "number : %s" (n2s num);*)
    if num =/ zero then (n2i (mod_num num two))::acc
    else num_to_bits_aux (quo_num num two) ((n2i (mod_num num two))::acc) in
    let list = num_to_bits_aux num [] in
  Array.of_list list;;
```

On va charger l'image [5] pour obtenir un tableau contenant des tableaux contenant des objets Color (de la librairie Graphics) représentant des triplets de int pour les pixels de l'image.

```
let load_rgb_matrix name =
  let img = Images.load name [] in
  let gimg = Graphic_image.array_of_image img in
  let rgb color =
```

```
    let quot n = n mod 256, n / 256 in
    let b, rg = quot color in
    let g, r = quot rg in
    r, g, b
  in
  Array.map (Array.map rgb) gimg;;
```

On récupère les métadonnées de l'image pour plus tard. On va parcourir l'image pour récupérer chaque pixel en rgb afin de remplacer leurs bits de poids faible par les bits du message crypté.

```
let black = {r = 0; g=0; b=0; } in
  let acc = ref 0 in
  let img = Rgb24.make w h black in
    for i = 0 to w-1 do
      for j = 0 to h-1 do
          let (_r,_g,_b) = image.(j).(i) in
          let colorencrypt = _r land 254 in
          if !acc < Array.length bitscrypted then
            let rfinal = (colorencrypt) lor (bitscrypted.(!acc)) in
            let couleur = {r = rfinal; g = _g ; b = _b} in
            acc := !acc+1;
            Rgb24.set img i j couleur;
            (*Printf.printf "rfinal : %d / g : %d / b : %d \n" rfinal _g _b;*)
          else
            let rfinal = colorencrypt lor (Random.int 1) in
            let couleur = {r = rfinal; g = _g ; b = _b}in
            Rgb24.set img i j couleur;
            (*Printf.printf "rfinal : %d / g : %d / b : %d \n" rfinal _g _b;*)
      done;
    done ;

(* sauvegarde de l'image *)
Bmp.save "mario_res.bmp" [] (Images.Rgb24 img);;
```

On sauvegarde l'image obtenue [7]. Pour décrypter l'image, on l'ouvre puis on récupère les bits de poids faible des pixels de l'image. On va les transformer en num pour pouvoir les décrypter et obtenir le message original.

3.1 Limitations

Comme on n'a pas de remplissage implémenté pour le cryptage RSA, on est obligé de préciser la taille du message à déchiffrer afin d'éviter les valeurs de l'image.

Bibliographie

[1] Algorithme binaire de calcul du pgcd. *Wikipedia*. https://fr.wikipedia.org/wiki/Algorithme_binaire_de_calcul_du_PGCD.

[2] Chiffrement rsa. *Wikipedia*. https://fr.wikipedia.org/wiki/Chiffrement_RSA.

[3] Extended euclidean algorithm. *Wikipedia*. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.

[4] Optiml asymmetric encryption padding. *Wikipedia*. https://fr.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding.

[5] Developez.com. Ouvrir / enregistrer une image [ocaml]. https://www.developpez.net/forums/d999058/autres-langages/langages-fonctionnels/caml/ocaml-ouvrir-enregistrer-image/.

[6] d.marino. Fast modular exponentiation. http://www.cs.ucf.edu/~dmarino/progcontests/modules/matexpo/RecursionFastExp.pdf.

[7] StackOverflow. How to read a bitmap in ocaml ? https://stackoverflow.com/questions/612886/how-to-read-a-bitmap-in-ocaml.

[8] Wikipedia. Test de primalité de miller-rabin.