# Certificats x509 - RSA

LSE - ÉQUIPE SÉCURITÉ

Mars 2021


Thomas Berlioz

Sébastien Delsart

Alexandre Fresnais

Martin Grenouilloux

Antoine Jouan


# Table des matières

# 1 Origine

## 1.1 Sur les certificats x509

Les certificats d'authentification interviennent lors d'une connexion sécurisée (HTTPS) à un site internet et assurent l'intégrité et la confidentialité des informations échangées avec ce site web. On peut le voir comme l'identité numérique publique des sites internet.

La norme x509 est une spécification qui décrit à quoi ressemble un certificat numérique. On y retrouve plusieurs informations :

— une clé publique liée à une identité, c'est-à-dire un ensemble de données qui permettent d'identifier quelqu'un (nom, prénom, organisation, localisation...)
— une période de validité
— le type de chiffrement utilisé
— la signature de l'autorité de certification qui a délivré le certificat

Ainsi, à chaque fois que Alice envoie une requête à un site sécurisé, elle est certaine qu'elle communique bien avec ce site et que ce site communique bien avec elle.

## 1.2 Problème et motivation

Ce projet est né à partir du papier *Ron was Wrong, Whit is right*[1] publié en 2012. Il met en évidence un problème dans la génération des clés publiques RSA. En effet, si on se base sur un échantillon de plusieurs millions de clés publiques, on constate un ratio anormalement élevé (2 pour 1000) de clés avec une faille de sécurité.

Il serait alors intéressant de faire la même expérience 10 ans après en trouvant des facteurs communs dans des clés publiques RSA de certificats x509. Pour ça, il faut tout d'abord construire une base de données conséquente de certificats afin d'obtenir des statistiques sur les résultats obtenus. Le projet se découpe ainsi naturellement en 2 phases : le *crawler*, chargé de récupérer des certificats sur le net, et le *cracker*, qui doit casser des paires de certificats en trouvant des facteurs communs dans les clés publiques. Il est important de souligner que ce projet s'est étalé sur seulement 3 semaines à côté des études de chacun des membres du groupe.

# 2 Crawler

Au début du projet, l'objectif était fixé à 100 millions de certificats x509 en 1 semaine. Pour cela, plusieurs pistes ont été envisagées avant d'aboutir à un résultat.

1. `https://eprint.iacr.org/2012/064.pdf`

## 2.1 Premières idées

### 2.1.1 Certstream

La première piste a été la solution proposée par *Certstream*[2] en utilisant *certstream-python*[3]. Ce réseau permet de récupérer en temps réel les nouveaux certificats depuis le *Certificate Transparency Log Network*[4]. Cependant, ce choix a vite été écarté pour plusieurs raisons :

— il est impossible de récupérer des certificats anciens, obligatoires pour avoir des données réparties dans le temps afin de corréler les résultats avec les périodes de génération des clés
— pour atteindre 100 millions de certificats en une semaine, il faut en moyenne 165 certificats par seconde, ce qu'il est impossible d'atteindre de cette manière

### 2.1.2 Scan IPv4

C'est en réalité la première option qui a été envisagée. Il s'agit d'un basique scan de toutes les IPv4 de 0.0.0.0 à 255.255.255.255 en récupérant des certificats x509 avec des clés RSA sur les ports 443 dès que c'est possible.

Cependant, il est impossible de récupérer l'ensemble des noms de domaines associés à une adresse IPv4 et celle-ci n'est évidement pas suffisante pour accéder au certificat dans la très grande majorité des cas, le serveur DNS ne retournant que les enregistrements PTR, c'est à dire le fournisseur la plupart du temps, et non les enregistrements de type A associés. Impossible donc de récupérer une grande quantité de certificats de cette manière.

### 2.1.3 Certificate Search

*Certificate Search*[5] est une plateforme de recherche de certificats qui permet notamment d'obtenir un certificat à partir du nom de domaine auquel il est associé. En manipulant l'outil pour le tester, on réalise rapidement qu'un ID unique est associé à tous les certificats. Par exemple, en cherchant, `lse.epita.fr` on trouve le tout premier certificat généré pour ce domaine en 2013 avec son ID dans l'URL : `https://crt.sh/?id=5148604`. En partant de l'ID 1, il est donc possible de remonter à un certificat généré en 2000 et de monter de manière plus ou moins linéaire dans le temps jusqu'à nos jours.

Ce site est simple d'utilisation et répond à l'ensemble des critères. Pour récupérer un certificat au format PEM, il suffit de placer le paramètre `d` dans la requête avec l'ID du certificat. Pour récupérer 100 millions de certificats, il suffit de faire 100 millions de requêtes avec des ID différents choisis pour avoir une répartition dans le temps intéressante. De plus,

2. `https://certstream.calidog.io`
3. `https://github.com/CaliDog/certstream-python`
4. `https://certificate.transparency.dev`
5. `https://crt.sh`

on peut facilement cibler une année ou un mois en particulier s'il faut mettre en relation une mise à jour d'un outil de génération de facteurs premiers et des collisions dans les certificats.

## 2.2 Implémentation

Plusieurs points étaient nécessaires pour une implémentation qui respecte l'objectif :

— avoir 165 requêtes par seconde en moyenne pendant une semaine
— espérer que l'infrastructure du site supporte le débit bien supérieur à d'habitude
— stockage intelligent des certificats pour qu'ils soient rapidement accessibles
— capacité de stockage suffisante

### 2.2.1 Configuration réseau

Il est évident que le premier problème à résoudre est de contourner la limite de temps imposée entre chaque requête par le site. Il a fallu d'abord trouver que la vérfication se fait par l'IP à la source de requête. En conséquence, une *range* de $2^{60}$ addresses IPv6 est utilisée, donc assez grande pour que non seulement chaque *thread* ait sa propre IP de source pour les requêtes, mais aussi pour qu'il puisse changer d'IP tous les 10 certificats afin d'être sûr de ne jamais ban l'IP utilisée.

Une fois le point de relais configuré, un tunnel est créé avec `wireguard` et le sous-réseau associé au processus. Ainsi, il suffit de fournir à chaque machine une partie des adresses disponibles et le *crawler* se charge tout seul de distribuer et de changer correctement l'IP utilisée à chaque requête.

### 2.2.2 Stockage des certificats

Afin de rendre le stockage des certificats pratique et rapidement accessible, une base de données était néssaire. `sqlite3` étant simple d'installation et d'usage avec son module en Python a été choisi. De plus, nous avions déjà de l'expérience avec cet outil. Voici le schéma utilisé pour stocker les certificats :

```
CREATE TABLE ‘certificates‘ (
  ‘id‘ INTEGER NOT NULL,
  ‘common_name‘ TEXT NOT NULL,
  ‘not_before‘ NUMERIC NOT NULL,
  ‘not_after‘ NUMERIC NOT NULL,
  ‘modulus‘ TEXT NOT NULL,
  ‘certificate‘ TEXT NOT NULL
);
```

Le stockage d'un certificat nécessite environ 6500 octets, voici le détail du calcul :

| Column | Size (bytes) |
|---|---|
| id | 4 |
| common_name | 255 max |
| not_before | 10 |
| not_after | 10 |
| modulus | 4096 |
| certificate | 2048 |

Il est intéressant de noter qu'il est possible de réduire cette taille en compressant le module des clés en hexadécimal ou en base 64. Pour 100 millions de certificats, il faudra donc au maximum 650 Go en supposant que tous les certificats ont les valeurs maximales pour chaque colonne. De plus prendre une moyenne de taille de clés à 4096 est une sécurité car la plupart des clés feront 2048 bits.

Il a fallu également réfléchir au problème d'accès à la base de donnée par les différents processus en parallèle. Afin de ne pas avoir à gérer les droits d'écriture en bloquant certains *threads* et ne pas complexifier le code, chacun possède sa propre base de données qu'il suffit de fusionner une fois les certificats récupérés.

### 2.2.3 Monitoring des résultats

Avec une trentaine de machine qui font tourner simultanément plusieurs centaines de *threads*, il était impossible d'avancer à l'aveugle. Impossible également de vérifier manuellement l'état des machines et de lire chaque erreur à l'origine de la mort d'un processus. Un script d'analyse des *logs* a donc été fait, qui permet d'envoyer toutes les 60 secondes l'avancement vers un hôte qui centralise les informations sous la forme d'un tableau de bord qui recense pour chaque machine le nombre de certificats enregistrés, le nombre d'erreurs rencontrées avec le nombre de *threads* qui tournent et la taille actuelle de la base de données associée.

## 2.3 Résultats

Après 5 jours, environ 170 millions de certificats ont été récupérés. Une trentaine de machines sur plusieurs serveurs ont été impliquées, que ce soit en parallèle ou à la suite. Le pic de requêtes a été d'environ 1000 requêtes par seconde, pour une base de données finale de 450 Go. Des clés de toutes tailles ont été trouvées, de 512 à 30720 bits, avec évidement la grande majorité des clés de 2048 bits.

# 3 Cracker

La quantité énorme de certificats récupérés nous a obligé à repenser de nombreuses fois l'algorithme afin d'obtenir des temps de calculs raisonnables.

## 3.1 Algorithme naïf

Une première approche pour trouver des facteurs communs est d'établir une liste de modulos $n$ et de les comparer deux à deux en testant si leur PGCD [6] est différent de 1. Si c'est le cas, alors il est égal au facteur commun des deux clés et on peut ainsi facilement retrouver les deux clés privées respectives.

Le problème de cet algorithme est qu'il possède une complexité en $\theta(n^2)$. Il n'est donc pas envisageable pour des listes de plusieurs dizaines de millions de clés publiques comme nous le souhaitons

## 3.2 Batch-gcd

Il existe cependant un algorithme quasi-linéaire bien plus efficace, utilisé dans l'article *Ron was wrong, Whit is right* [7]. L'algorithme *Batch-gcd* [8] calcule efficacement le PGCD de chaque élément d'une liste $x$ avec le produit de tous les autres. En d'autres termes, il calcule la liste $S$ de la manière suivante :

$$S[0] = gcd(x[0], x[1]x[2]x[3]...x[n])$$
$$S[1] = gcd(x[1], x[0]x[2]x[3]...x[n])$$
$$...$$
$$S[i] = gcd(x[i], x[0]...x[i-1]x[i+1]...x[n])$$

Si le PGCD obtenu est différent de 1, alors l'élément *x[i]* possède un facteur premier commun avec un autre élément de la liste.

Son fonctionnement repose sur le principe d'un *product-tree* pour calculer efficacement le produit $Z = X[0]X[1]X[2]...$, représenté par un arbre dont les feuilles correspondent aux facteurs, et la racine au produit. Mais aussi d'un *remainder tree* qui utilise la même structure de donnée, toujours pour remonter efficacement les résultats des PGCD.

### 3.2.1 Premier essai en Python

La première implémentation de l'algorithme a été faite en Python pour qu'il soit facile et rapide de s'habituer au mécanimse et à ses subtilités, mais surtout pour pouvoir utiliser la bibliothèque SageMath. Une fois codé, il a fallu tester l'efficacité et la rapidité du programme sur des échantillons réduits :

6. Plus Grand Commun Diviseur
7. `https://eprint.iacr.org/2012/064.pdf`
8. `http://facthacks.cr.yp.to/batchgcd.html`

| Taille de l'échantillon | Temps de calcul en secondes |
|---|---|
| 1 000 | 14 |
| 2 000 | 50 |
| 10 000 | 1320 |

Ces résultats sont obtenus sans *multithreading* ni optimisation. Pour 30 000 certificats, l'execution du programme a été arrêtée manuellement apres 1h30.

Une première idée pour accélerer l'algorithme a été de paralléliser les calculs. Cependant, que ce soit en lançant plus de machines ou en utilisant la même avec plusieurs *threads*, c'est non seulement une complexification du code pour faire voyager les données entre les processus mais également des résultats peu convaincants à cause de l'interpréteur Python. Pour gérer des nombres immenses, il fallait passer à un autre langage, conclusion réalisée tardivement car il a fallu attendre les premières grosses bases de données du *crawler* pour les tests à grande échelle.

### 3.2.2 Implémentation en Rust

En dressant un état de l'art sur les implémentations de l'algorithme *Batch-gcd*, une implémentation en Rust de l'algorithme [9] par Fedor Indutny [10] s'est montrée extrêmement efficace sur un premier jeu de certificats qui surpassait déjà tous nos tests en Python.

9. `https://github.com/indutny/bulk-gcd`
10. urlhttps ://github.com/indutny

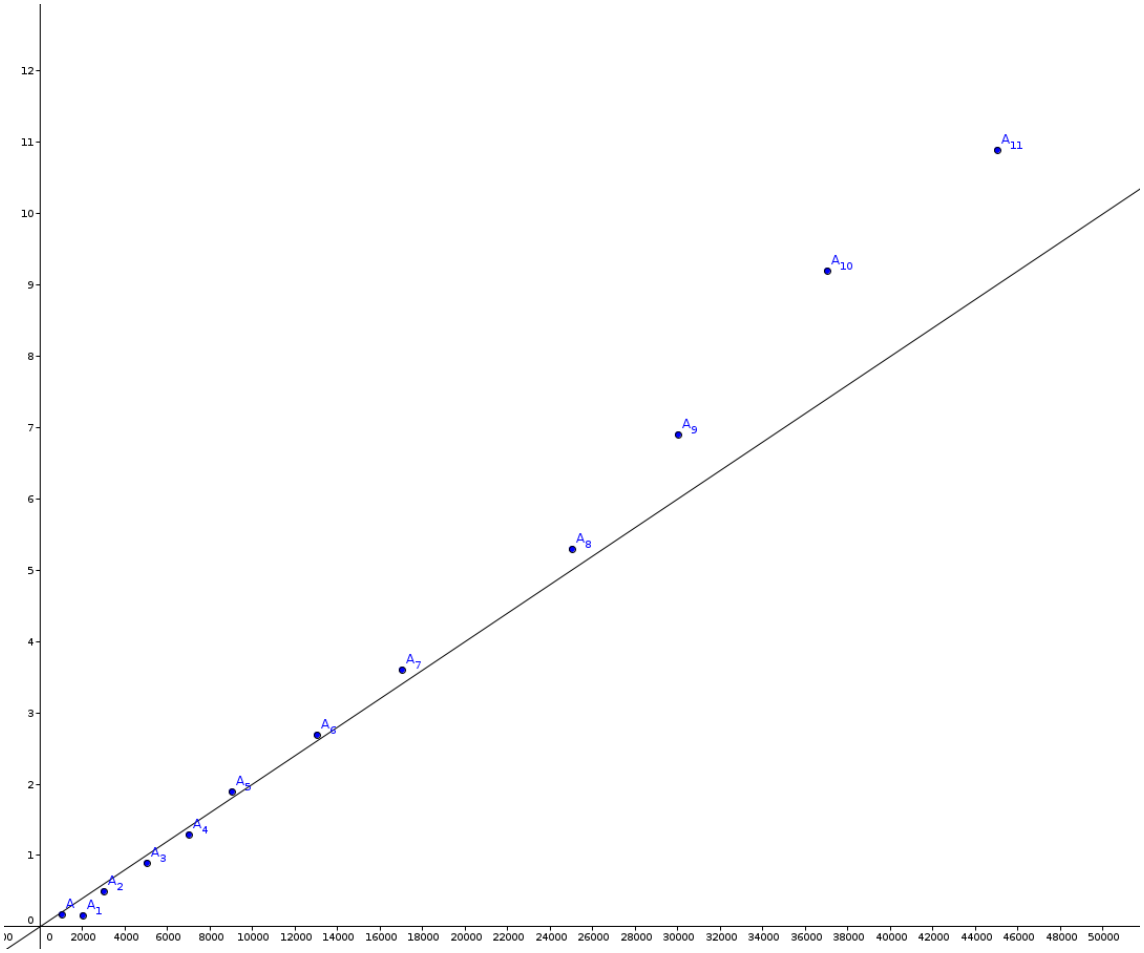


FIGURE 1 – Évolution du temps de calcul en fonction du nombre de certificats

Le graphe ci-dessus montre, avec pour comparaison la droite d'équation $y = x$, que les temps de calculs sont linéaires. Il est alors possible de dresser les estimations de temps de calculs suivantes :

| Taille de l'échantillon | Temps de calcul |
|---|---|
| 50 000 000 | 2h56 - 3h11 |
| 100 000 000 | 5h53 - 6h23 |
| 170 000 000 | 10h01 - 10h51 |
| 500 000 000 | 29h26 - 31h56 |
| 1 000 000 000 | 59h - 64h |

### 3.2.3 Implémentation en C++

Si cette première approche en Rust était effectivement un moyen efficace d'avoir des résultats exploitables à petite échelle, elle souffrait tout de même d'un problème de gestion de la RAM si on chargeait une base de donnée trop importante. Ainsi nous avons fini par recoder l'algorithme entièrement en C++ pour ne plus avoir à corriger et optimiser un code en Rust qu'on ne comprenait pas toujours.

Cette version finale possède le même problème de chargement des listes de facteurs de l'arbre en RAM qui fait planter le programme, mais il a permis d'essayer différentes optimisations, que ce soit en repassant par des BDD intermédiaires ou en découpant les calculs en plusieurs étapes.

Enfin, il était inimaginable de présenter des résultats sortis d'un code pris ailleurs et utilisé sans être compris, quitte à prendre le temps de le recoder entièrement pour au final avoir les mêmes soucis sans avoir le temps de trouver la bonne optimisation pour soulager la RAM.

### 3.2.4 Premiers résultats

Sur l'échantillon des 10 000 premiers certificats, environ 20 certificats ont la même clé publique. Parfois cela correspond simplement à une réutilisation du certificat pour un sous-domaine, même si le choix est difficle à comprendre. Cependant il y a aussi des sites sans lien évident, dont le seul point commun apparent est par exemple leur thématique comme le sport, les produits pour animaux voire leur proximité géographique. Par exemple, *mp4.sk*[11] et *wall.cz*[12] possédaient la même clé publique en 2017, et se situent respectivement en Slovaquie et en République Tchèque, deux pays voisins. En vérifiant manuellement les certificats sur le site, il ne s'agit pas d'une erreur dans la récupération des certificats mais bien d'une anomalie, comme mis en évidence ici :

| | common_name | modulus | nb |
|---|---|---|---|
| 1 | www.juliemustard.com | 0x856cca5f5932408c0f0bdbf4577630657... | 2 |
| 2 | englische-antiquitaeten-evans.de | 0xae5c2ecf9f0e86d19533a4db282120939... | 2 |
| 3 | beautyhere-allday.com | 0xb58aaedb03538603320ffc808701c3b8e... | 10 |
| 4 | texxas.de | 0xcb737d6ae200a2d0e38f91500dbc9e06... | 2 |
| 5 | monkeyphones.com | 0xcd140f32faceaf4e2ca07fff8b793f10ce1... | 2 |
| 6 | devfest.co | 0xd5da3c2ae84c4ec989ac92d439aed845... | 10 |

On constate également la présence de clés publiques dupliquées pour le même domaine mais sur plusieurs certificats avec des périodes de validité différentes. En revanche, beaucoup de ces sites ne sont plus disponibles.

## 4 Conclusion et suite

Ce projet était intéressant et instructif dans l'ensemble. C'était une première expérience néssaire du travail en groupe au sein d'une équipe nouvelle, et elle aurait pu l'être encore

11. Certificat : `https://crt.sh/?id=83166981`
12. Certificat : `https://crt.sh/?id=82853860`

avec le temps néssaire et l'investissement que le sujet aurait mérité. Toute l'équipe sécurité du LSE de la génération 2023 espère pouvoir améliorer le cassage des clés afin d'enfin publier des résultats concret sur la quantité immense de certificats récoltés. Voici également donc quelques points qu'il serait intéressants de pousser si le projet venait à être continué ou repris.

1. Améliorer le *cracker* afin d'éviter d'être limité par la RAM
2. Faire une analyse poussée des résultats afin d'avoir des statistiques sur un si grand nombre de certificats
3. Présenter une conférence sur le sujet
4. Faire des recherches approfondies sur le code de `Let's Encrypt` [13]
5. Générer des millions de clefs RSA sur une distribution vierge

De plus, et même si la frustration de ne pas pouvoir publier un analyse détaillée de notre grosse base de donnée est bien présente, nous sommes fiers du code produit, que ce soit le *crawler* fonctionnel ou le *cracker* dont la structure permet de facilement le modifier afin d'avancer dans la recherche de la bonne optimisation.

# 5 Pour aller plus loin


1. `https://facthacks.cr.yp.to/batchgcd.html`
2. 2012 : "Ron was wrong, Whit is right", *Arjen K. Lenstra and James P. Hughes and Maxime Augier and Joppe W. Bos and Thorsten Kleinjung and Christophe Wachter*, `https://eprint.iacr.org/2012/064`
3. 2012 : "Mining your Ps and Qs : detection of widespread weak keys in network devices", *Nadia Anne Heninger, Zakir Durumeric, Eric Wustrow, J Alex Halderman*, `https://dl.acm.org/doi/10.5555/2362793.2362828`
4. 2013 : "Factoring RSA keys from certified smart cards : Coppersmith in the wild", *Bernstein et. al*
5. `https://crypto.stackexchange.com` : The GDC strikes back to rsa in 2019 good randomness is the only solution (see `https://cutt.ly/PxUPbWc`).


13. `https://letsencrypt.org/`