Analyse des biais de RNG pour les mécanismes cryptographiques et applications industrielles

Mohamed Traore

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Mathématiques

Arrêté ministériel : 25 mai 2016

Présentée par

Mohamed TRAORE

Thèse dirigée par **Philippe ELBAZ-VINCENT**, Université Grenoble Alpes

préparée au sein du **Laboratoire Institut Fourier**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Analyse des biais de RNG pour les mécanismes cryptographiques et applications industrielles

Analysis for Anomalies in Cryptographic RNG and Industrial Applications

Thèse soutenue publiquement le **23 mai 2022**,
devant le jury composé de :

Monsieur Lilian BOSSUET
Professeur des Universités, Université Jean Monnet Saint-Étienne, Examinateur

Madame Cécile DUMAS
Ingénieur docteur, CEA Centre de Grenoble, Examinatrice

Monsieur Jean-Guillaume DUMAS
Professeur des Universités, Université Grenoble Alpes, Examinateur, Président

Monsieur Philippe ELBAZ-VINCENT
Professeur des Universités, Université Grenoble Alpes, Directeur de thèse

Monsieur Pascal LAFOURCADE
Maître de conférences HDR, Université Clermont Auvergne, Rapporteur

Monsieur Cédric LAURADOUX
Chargé de recherche, INRIA Centre Grenoble-Rhône-Alpes, Examinateur

Madame Marine MINIER
Professeur des Universités, Université de Lorraine, Examinatrice

Monsieur Guénaël RENAULT
Docteur en sciences HDR, ANSSI, Rapporteur

– Résumé –

Dans ces travaux, on analyse des certificats SSL/TLS X.509 utilisant le chiffrement RSA issus de centaines de millions de matériels connectés, à la recherche d'anomalies et on étend les travaux de Hastings, Fried et Heninger (2016) notamment. Notre étude a été réalisée sur trois bases de données provenant de l'EFF (2010-2011), de l'ANSSI (2011-2017) et de Rapid7 (2017-2021). Plusieurs vulnérabilités affectant des matériels de fabricants connus furent détectées : modules de petites tailles, modules redondants, certificats invalides mais toujours en usage, modules vulnérables à l'attaque ROCA ainsi que des modules dits «PGCD-vulnérables» (i.e. des modules ayant des facteurs communs).

On a identifié 1,550,382 certificats dont les modules sont PGCD-vulnérables, permettant de factoriser 14,765 modules de 2048 bits ce qui, à notre connaissance, n'a jamais été fait. En analysant certains modules PGCD-vulnérables, nous avons pu rétro-concevoir en partie le générateur de modules (de 512 bits) utilisé par certaines familles de pare-feux, ce qui a permis la factorisation instantanée de 42 modules de 512 bits, correspondant aux certificats provenant de 8,817 adresses IPv4. Dans la dernière partie de nos travaux, nous analysons les codes sources et les méthodes en charge du processus de génération de clefs RSA des versions d'OpenSSL (couvrant la période 2005 à 2021) et avons pu remonter aux causes de plusieurs de ces vulnérabilités.
Mots clés : Génération de paramètres cryptographiques, Certificats SSL/TLS, Déviation statistique dans les RNG, Cryptologie, OpenSSL, Matériels connectés.

– Abstract –

In this work, we analyze SSL/TLS X.509 certificates using RSA encryption from hundreds of millions of connected devices, looking for anomalies and we extend the work of Hastings, Fried and Heninger (2016) in particular. Our study was carried out on three databases from EFF (2010-2011), ANSSI (2011-2017) and Rapid7 (2017-2021). Several vulnerabilities affecting devices from well-known manufacturers were detected : small moduli, redundant moduli, invalid certificates but still in use, moduli vulnerable to the ROCA attack as well as so-called "GCD-vulnerable" moduli (i.e. moduli with common factors).

We identified 1,550,382 certificates whose moduli are GCD-vulnerable, allowing to factorize 14,765 moduli of 2048 bits which, to our knowledge, has never been done. By analyzing some GCD-vulnerable moduli, we were able to partially reverse-engineer the 512-bit modulus generator used by some families of firewalls, which allowed the instantaneous factorization of 42 512-bit moduli, corresponding to the certificates from 8,817 IPv4 addresses. In the last part of our work, we analyze the source codes and the methods in charge of the RSA key generation process of the versions of OpenSSL (covering the period 2005 to 2021) and were able to trace the causes of several of these vulnerabilities.
Keywords : Generation of cryptographic parameters, SSL/TLS Certificates, Statistical biases of RNG, Cryptology, OpenSSL, Network devices.

Remerciements

Bismi-LLâhir-Rahmânir-Raheem
Lis, au nom de ton Seigneur qui a créé,
qui a créé l'homme d'une adhérence.
Lis! Ton Seigneur est le Très Noble,
qui a enseigné par la plume,
a enseigné à l'homme ce qu'il ne savait pas.

Je ne saurais écrire ces quelques lignes sans remercier mon directeur de thèse Philippe Elbaz-Vincent qui m'a pris sous son aile dès ma première année de master (M1). Merci pour tes conseils, ton soutien et ta patience à mon égard. Je te remercie également pour les repas d'équipe, les déplacements pour des conférences en Bretagne et à Bordeaux, et d'avoir mis à ma disposition tous les outils dont j'avais besoin pour le bon déroulement de cette thèse. Tes serveurs de calcul «ifcrypt» me manqueront.

Je tiens aussi à remercier tous les membres de mon jury de soutenance, en particulier, mes deux rapporteurs Pascal Lafourcade et Guénaël Renault qui ont donné de leur temps pour le suivi de ma thèse et pour la lecture et l'analyse de mon manuscrit.

Mes remerciements vont également à tout le personnel du laboratoire Institut Fourier, en particulier, Didier Depoisier (le responsable du service informatique) pour sa disponibilité et mes collègues Cyril Hugounenq et Étienne Marcatel pour leur compagnie et leur aide.

Revenons un peu en arrière! Merci à mes parents Modibo Traoré et Makane Touré pour les valeurs qu'ils m'ont inculquées. Étant le dernier de la fratrie, je dis merci à tous mes frères et sœurs qui m'auront servi de support. Merci également à tous les enseignants de ma petite ville de Kolokani, notamment ceux du premier cycle «A», du second cycle «B» et du Lycée Famolo Coulibaly de Kolokani (LFCK). Je remercie spécialement mon mentor Brama Zala actuellement conseiller pédagogique au Centre d'Animation Pédagogique (CAP) de Kolokani ainsi que Moussa N'diaye ancien directeur du CAP de Kolokani.

Un grand merci à l'un de mes grands frères Koké Traoré et à deux de mes beaux-frères Nan Siriman Soumano (dit Elvis) et Sékou Amadou Traoré (dit Kabila) sans lesquels je ne serais peut-être pas venu en France et n'aurais probablement pas fait cette thèse. Vous vous réveilliez tôt pour accompagner ce petit bachelier à faire ses démarches administratives, car celui-ci ne connaissait rien dans les affaires de cette grande capitale qu'est Bamako.

Que mon cousin Gaoussou Touré et mes anciennes camarades de classe Ksenia et Michi, rencontrées à Grenoble, soient aussi remerciés. Sans oublier ma petite Kiné dont le soutien moral a été très important; tu vois, ton *bonheur* comme tu m'appelles, ne t'a pas oubliée et te mentionne dans ce petit paragraphe.

Ayant bénéficié du programme «300 jeunes cadres pour le Mali» après le baccalauréat, je dois remercier mon pays de m'avoir donné l'opportunité de faire une formation de qualité durant ces 8 dernières années. Je profite de l'occasion pour dire merci à tous mes camarades boursiers d'excellence, en particulier, Mama Dembélé, Zié Drissa Diarra, Mamadou Konaté, Mahamadou Samassa, Mahamadou Sidibé, Salif Sogoba et Amadou Yoro Thiam pour leur proximité et leurs aides de diverses natures.

Enfin, je tourne le regard vers les membres de mon petit groupe de football de l'UFRAPS de Grenoble : Adel et son frère Lounès, Alex, Edème, François, Gaëlle et ses enfants, Ilian, Ilyes, Lionel, Mathias, Mike, Ninou, Régis et son Icalu, Rémi et ses enfants, Simon, Taka, Toni et sa fille Ornella. Les gars, je vous dis merci pour ces moments de détente passés ensemble durant les mercredi et vendredi midi que ce soit à l'ombre d'un arbre, sous la pluie, le vent, la neige ou encore la canicule.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Chapitre 1

Introduction

Sommaire

1.1 Contexte

En début 2017, un rapport de Gartner [120] sur les objets connectés à l'Internet prévoyait 8.4 milliards de matériels connectés (serveurs, routeurs, pare-feux, etc.), soit une hausse de 31% par rapport à 2016. Dans le même rapport, on estimait que cette quantité atteindrait 20.4 milliards en 2020. Parallèlement à cela, des recherches de Cisco [1] indiquaient qu'il y aurait 27.1 milliards d'appareils connectés en 2021. Mais un rapport plus récent (septembre 2021) de «IoT Analytics» [2] affirme qu'il n'y en aurait finalement que 12.3 milliards. Bien qu'il y ait une grande différence entre ce résultat et les estimations faites en 2017, on peut néanmoins déduire des trois rapports qu'il y a une augmentation considérable des échanges de données. Cela implique une importante sollicitation des protocoles cryptographiques pour sécuriser la transmission des données qui sont sensibles.

L'usage de ces protocoles nécessitera ce que l'on appelle des *clefs*. Elles constituent l'une des briques de base de la cryptographie moderne et seront abordées dans la section 2.1 du chapitre 2. Le concept de clefs impose que la sécurité soit basée sur les clefs et non sur la confidentialité des algorithmes cryptographiques. De ce fait, les clefs doivent être choisies de façon *aléatoire* afin d'empêcher leur prédiction par d'autres entités. Cette tâche est difficile, car elle fait appel aux *générateurs de nombres aléatoires* qui sont des objets connus comme étant difficiles à mettre en place. La plupart des familles de matériels connectés assurent la génération de leurs clefs en utilisant des bibliothèques cryptographiques connues (souvent à code source ouvert) comme OpenSSL [116]. Cependant, il existe quelques rares familles qui se servent de leurs propres bibliothèques (c'est-à-dire développées en interne).

Que ce soit une bibliothèque à code source ouvert ou une bibliothèque développée en interne, il peut y avoir des anomalies lorsque certaines conditions sont réunies telles qu'un manque d'entropie du générateur de nombres aléatoires (comme ce fut, par exemple, le cas dans les études énumérées ci-dessous).

— En 2012, Lenstra *et al.* [73] ont analysé 11.7 millions de certificats SSL/TLS et ont pu factoriser 4.3% des modules RSA distincts (soit 0.27 million de modules).

— Toujours en 2012 et indépendamment de l'étude précédente, Heninger *et al.* [52] ont factorisé 0.5% des modules RSA distincts issus de près de 6 millions de certificats TLS.

— En 2013, Bernstein *et al.* [17] sont parvenus à factoriser 184 modules RSA distincts de 1024 bits provenant de 2 millions de certificats électroniques extraits de la base de données nationale taïwanaise de certificats «Citizen Digital Certificates».

Ces études se sont focalisées principalement sur le chiffrement RSA qui est actuellement le chiffrement asymétrique le plus utilisé dans les services web. On donnera plus d'éléments, dans la section 5.1 du chapitre 5, sur ces études et sur d'autres études plus récentes. Notons qu'il existe également des situations où le problème vient de l'algorithme de génération des nombres premiers. Par exemple, en 2017, Nemec *et al.* [85] ont découvert une vulnérabilité (appelée *vulnérabilité ROCA* où ROCA signifie «Return Of the Coppersmith Attack») contre les

1. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf.
2. https://iot-analytics.com/number-connected-iot-devices/.

clefs RSA générées par la bibliothèque RSALib d'Infineon[3]. Ils ont pu l'exploiter pour compromettre plusieurs clefs RSA en se servant d'un résultat de Coppersmith [28] basé sur les réseaux mathématiques. Plus de détails seront donnés sur cette vulnérabilité dans la sous-section 4.5.2 du chapitre 4.

Similairement aux études précédentes, mais de manière plus approfondie, les résultats qui seront présentés dans ce document découlent d'une analyse focalisée sur le chiffrement RSA et effectuée sur plusieurs centaines de millions de certificats SSL/TLS provenant de divers types de matériels connectés à l'Internet.

1.2 Plan et contributions

L'ensemble de ce document est composé de sept chapitres; le chapitre 2, le chapitre 3 et le chapitre 4 étant des chapitres de rappel. Le chapitre 5, le chapitre 6 et le chapitre 7 contiennent les travaux ayant été effectués durant cette thèse.

— Dans le chapitre 2, on fera des rappels sur les différents types de chiffrements cryptographiques (par flot, par bloc, symétriques, asymétriques et hybrides classiques), les codes d'authentification de message (ou MAC pour «Message Authentication Codes»), les signatures électroniques ainsi que les fonctions de hachage cryptographiques.

— Le chapitre 3 sera consacré à la présentation des certificats électroniques qui sont des outils permettant d'associer une identité de façon *fiable* à une clef publique lors de l'utilisation d'un chiffrement asymétrique. Cette présentation inclura les modèles de confiance, la définition d'une infrastructure à clefs publiques (ou PKI pour «Public Key Infrastructure») et les deux normes de PKI les plus répandues actuellement : PKIX [18] et OpenPGP [44]. On parlera également du protocole TLS qui utilise principalement des certificats X.509 et qui est répandu sur l'Internet pour sécuriser la transmission des données. On clora le chapitre par un exposé de certains types de vulnérabilité que l'on peut rencontrer avec les certificats électroniques.

— Le chapitre 4 aura pour but de présenter la composition d'une clef RSA [101] et les primitives de chiffrement et de déchiffrement de ce cryposystème. On y présentera aussi les principaux tests de primalité (tels que celui de Miller-Rabin dans la section 4.2) et le théorème des nombres premiers qui donne une estimation de la quantité de nombres premiers d'un intervalle donné (voir section 4.3). Le chapitre 4 contiendra également les principales méthodes de factorisation d'entiers (dans la section 4.4) et un théorème de Coppersmith (dans la section 4.5) qui permet d'exploiter une exposition partielle de paramètres privés d'une clef RSA. Enfin, il y aura une présentation des méthodes de génération de paramètres RSA qui sont recommandées par des normes comme la norme IEEE-P1363 [3], la norme FIPS 186-4 [59] du NIST ainsi que les guides de BSI [23] et de l'ANSSI dans la section 4.6.

— On effectuera, dans le chapitre 5, une analyse de trois bases de données de certificats SSL/TLS X.509 provenant de «EFF SSL Observatory», de l'ANSSI et de «Project Sonar»

3. https://www.infineon.com/.

de Rapid7. Ces certificats ont été collectés sur des matériels connectés (à l'Internet) de plusieurs types : routeurs, pare-feux, téléphones connectés, caméras connectées, etc. Lors de cette analyse, on suivra une méthodologie similaire à celle de Hastings, Fried et Heninger [51]. Cela nous permettra de détecter en particulier une vulnérabilité que l'on appellera *PGCD-vulnérabilité* et qui consiste à factoriser des modules RSA en faisant des calculs de plus grands communs diviseurs (ou PGCD). Les modules ayant cette vulnérabilité seront appelés *modules PGCD-vulnérables* et des analyses complémentaires montreront que la plupart d'entre eux ont été générés avec OpenSSL [116]. Notons aussi que l'on signalera toute autre anomalie que l'on constatera même si elle n'appartient pas à cette méthodologie.

— Dans le chapitre 6, l'objectif sera d'étudier les processus de génération de clefs RSA de plusieurs versions d'OpenSSL afin de déterminer les conditions dans lesquelles la PGCD-vulnérabilité peut être observée.

— Quant au chapitre 7, il contiendra les codes sources des implémentations ayant été faites dans les chapitres précédents avec des outils de programmation dont Python et PARI/GP [117]. On y détaillera également la façon dont les données ont été traitées et les ressources (espace de stockage, mémoire RAM) nécessaires à leur manipulation.

À travers les analyses effectuées dans le chapitre 5, on a pu observer plusieurs types de vulnérabilité : des modules de petites tailles (voir sous-section 5.3.2), des modules redondants ainsi que des certificats expirés mais toujours en usage (voir sous-section 5.3.3). On a également détecté des anomalies que l'on qualifie d'*inattendues* et que l'on présente dans la sous-section 5.3.5. D'autre part, des modules vulnérables à l'attaque ROCA et plusieurs modules PGCD-vulnérables ont été recensés. Après avoir constaté que les modules PGCD-vulnérables provenant des pare-feux «ZX_ZW2P» avaient une forme particulière, on a pu rétro-concevoir partiellement le générateur de modules de ces matériels. Cette rétro-conception partielle conduisit à une factorisation *instantanée* d'autres modules. L'ensemble des modules factorisés correspond à plusieurs milliers de certificats. Les détails sont donnés dans la sous-section 5.3.4 et la sous-section 5.3.7.

Ayant remarqué que la plupart des modules PGCD-vulnérables trouvés ont été générés par OpenSSL, on s'est tourné vers l'étude du processus de génération des clefs RSA de cette bibliothèque depuis sa version 0.9.8 (sortie en 2005). Cela nous a permis de remonter (dans le chapitre 6) aux causes de la PGCD-vulnérabilité.

Chapitre 2

Rappels sur des principes de la cryptographie

Sommaire

Dans ce chapitre, on va faire quelques rappels sur les principes de la cryptographie pour introduire certaines notions et terminologies que l'on utilisera dans les chapitres suivants.

On suppose qu'un client veut envoyer un message m confidentiel (tel qu'un numéro de carte de crédit) à un serveur. Le processus consistant à cacher le sens de m en le transformant en un message c, est le *chiffrement*. Le processus inverse est appelé *déchiffrement*. Les deux processus se servent de deux fonctions mathématiques : la fonction de chiffrement (que l'on note E) et la fonction de déchiffrement (notée D). Notons que les messages m et c sont parfois appelés *texte clair* et *texte chiffré* respectivement.

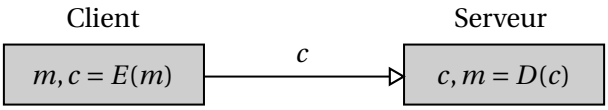


Figure 2.1 – ***Chiffrement et déchiffrement.***

2.1 Algorithmes cryptographiques

Définition 2.1.1 ([103, p.2])**.** Un *algorithme cryptographique* (ou simplement *chiffrement*) est un algorithme qui calcule la valeur des fonctions mathématiques E et D.

Il y eut des chiffrements dont la sécurité était basée uniquement sur la confidentialité de l'algorithme utilisé, mais ceux-ci ne sont plus adaptés aux besoins actuels. Une alternative est apparue avec la cryptographie moderne en apportant le concept de *clefs* selon lequel la sécurité d'un chiffrement doit reposer uniquement sur ses clefs et non sur la confidentialité de l'algorithme. Généralement, deux clefs (que l'on note K_e et K_d) sont utilisées, l'une dans la fonction de chiffrement et l'autre dans la fonction de déchiffrement.

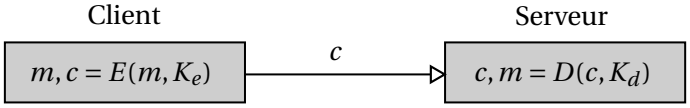


Figure 2.2 – ***Chiffrement et déchiffrement avec des clefs.***

2.1.1 Chiffrements symétriques et chiffrements asymétriques

Selon les types de clefs utilisées, on peut classer les chiffrements en deux familles : celle des *chiffrements symétriques* et celle des *chiffrements asymétriques*.

Définition 2.1.2 ([78, def.1.24])**.** Un chiffrement est dit *symétrique* si pour tout couple $(K_e; K_d)$ de clefs associées, il est *facile*[1] de déterminer K_e et K_d, l'une à partir de l'autre.

1. Par exemple, en temps polynomial.

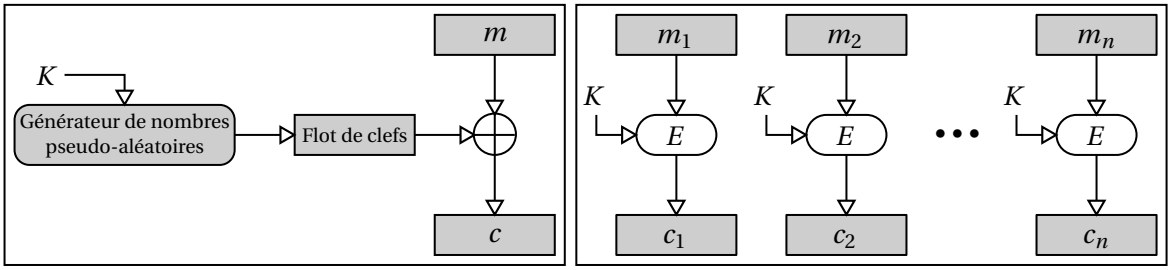
Dans la plupart des chiffrements symétriques, $K_e = K_d = K$ (on parle de *clef sécrète*, car le client et le serveur uniquement doivent la connaître). Ils doivent se mettre d'accord sur sa valeur (de façon confidentielle) avant de débuter leur communication. Cette tâche est ardue puisque les deux entités sont en général distantes. Cette difficulté, connue sous le nom du *problème de distribution de clefs*, peut être solutionnée avec les chiffrements asymétriques.

Définition 2.1.3 ([78, def.1.50])**.** Un *chiffrement asymétrique* est fait de sorte que pour toutes clefs K_e et K_d associées, il soit *facile* d'avoir K_e à partir de K_d, mais que l'inverse soit *difficile*.

Dans les chiffrements asymétriques, K_e est appelée *clef publique* et peut être connue de tous. Par contre, K_d (appelée *clef privée*) ne doit être connue que par le destinataire du texte chiffré. Chaque communicant doit donc détenir deux clefs contrairement aux chiffrements symétriques où les communicants partagent les mêmes clefs. Dans la suite de ce chapitre, les couples de clefs (publique ; privée) du client et du serveur seront notés $(P_A; S_A)$ et $(P_B; S_B)$ respectivement.

2.1.2 Chiffrements par flot et chiffrements par bloc

Les algorithmes cryptographiques sont souvent répartis entre les *chiffrements par flot* et les *chiffrements par bloc*. Dans un chiffrement par flot, le chiffrement se fait bit par bit (ou parfois octet par octet) en utilisant un *flot de clefs*[2] et une fonction de substitution bit-à-bit (rapide) comme le XOR[3] [78, def.1.38]. La valeur du flot de clefs est généralement fournie par un générateur de nombres pseudo-aléatoires initialisé avec une clef K (voir Figure 2.3a).



(a) ***Chiffrement par flot en utilisant le XOR.*** (b) ***Chiffrement par bloc en mode ECB.***

Figure 2.3 – ***Exemples de chiffrement par flot et de chiffrement par bloc.***

Par contre, dans un chiffrement par bloc, le texte clair est découpé en n blocs de même taille (64 bits ou 128 bits par exemple) puis il est chiffré bloc par bloc tout en utilisant ce que l'on appelle *un mode d'opération* [78, def.1.26]. Sur la Figure 2.3b, on donne un exemple en utilisant le mode ECB («Electronic Code Book») où chaque bloc est chiffré séparément. C'est le mode d'opération le plus simple et n'est normalement jamais utilisé en cryptographie à cause de l'absence de sécurité. Il existe des modes d'opération cryptographiquement sûrs tels que le mode CTR que l'on peut utiliser (voir [37, 39], [43, chap.4] et [78, chap.7]).

2. Le flot de clefs est une suite de bits faisant la même taille que le texte clair [78, def.1.37].
3. XOR (eXclusive OR) est la fonction *OU exclusif*. Elle est aussi notée par le symbole $\oplus$.

Notons que tous les chiffrements asymétriques sont des chiffrements par bloc alors qu'un chiffrement symétrique est soit un chiffrement par flot, soit un chiffrement par bloc. Pour plus de détails sur les chiffrements par flot et les chiffrements par bloc, on renvoie le lecteur aux chapitres 6 et 7 de «Handbook of Applied Cryptography» [78], aux chapitres 12 à 17 de «Applied Cryptography : Protocols, Algorithms, and Source Code in C» [103] et le chapitre 3 de «Cryptography Engineering : Design Principles and Practical Applications» [43].

2.1.3 Cryptographie hybride classique

L'un des inconvénients des chiffrements asymétriques est leur lenteur par rapport aux chiffrements symétriques. En conséquence, on utilise les chiffrements asymétriques pour chiffrer principalement des petites quantités de données (un numéro de carte de crédit par exemple). Ils peuvent toutefois être combinés avec les chiffrements symétriques pour tirer le meilleur de chacun des deux. Cette technique porte le nom de *cryptographie hybride* [66] et son principe de fonctionnement est brièvement donné par le Protocole 2.1. On rajoute le qualificatif *classique* pour marquer la différence avec une combinaison de chiffrements classique et quantique (qui est souvent qualifiée d'hybride aussi).

Protocole 2.1 Cryptographie hybride classique

1: Le serveur envoie sa clef publique P_B au client.
2: Le client génère une clef aléatoire K puis envoie $E(K, P_B)$ au serveur.
3: Le serveur retrouve K en calculant $D(E(K, P_B), S_B)$.
4: Les deux communiquent ensuite en utilisant un chiffrement symétrique de clef K.

2.2 Intégrité du message

Le chiffrement d'un message le protège contre des menaces dites *passives*, c'est-à-dire qu'il permet d'assurer sa confidentialité. Cependant, il ne le protège pas contre des menaces *actives* où celui qui intercepte le texte chiffré peut le modifier (même s'il lui est impossible d'en lire le contenu) avant qu'il n'arrive à destination. Les outils cryptographiques qui nous protègent contre ce type de menaces peuvent être construits à l'aide des chiffrements symétriques (avec, par exemple, les *codes d'authentification de message* ou MAC pour «Message Authentication Codes») et des chiffrements asymétriques (avec les *schémas de signature*).

2.2.1 Codes d'authentification de message ou MAC

Définition 2.2.1 ([43, p.89])**.** Une fonction de MAC est une fonction qui prend deux arguments (une clef K et un message m de taille quelconque) et produit une sortie de taille fixe.

La sortie produite est notée MAC(m, K) et est appelée *MAC* ou *étiquette*. Le Protocole 2.2 donne un aperçu de la façon dont un MAC peut être utilisé. On pourra trouver plus de détails sur cet outil dans [84, 58, 38], [43, chap.6] et [78, chap.9].

Protocole 2.2 Contrôle d'intégrité par un MAC

1: Le client et le serveur se mettent d'accord sur une clef K.
2: Le client calcule $m' = \text{MAC}(m, K)$ et envoie $(m \parallel m')$ au serveur. ▷ *Le symbole* $\parallel$ *désigne la concaténation de deux chaînes de caractères.*
3: Le serveur calcule le MAC du message reçu et le compare au MAC reçu.
4: Si les deux MAC correspondent, le serveur considère que le message n'a pas été altéré.

Pour qu'une fonction MAC soit considérée comme sûre, il doit être impossible de façon calculatoire pour une autre entité d'obtenir un MAC qui n'a encore été calculé par aucun des détenteurs de la clef sécrète. En effet, toute entité ignorant K et capable de calculer un MAC valide pour un message, pourra se faire passer pour l'un des communicants.

À noter que si la confidentialité du message est requise, le client envoie le texte chiffré et son MAC. Le serveur vérifiera l'exactitude du MAC avant de procéder au déchiffrement.

2.2.2 Schémas de signature numérique

Un schéma de signature numérique apporte une assurance similaire à celle d'un MAC. Étant basé sur du chiffrement asymétrique, il est composé d'un *algorithme de signature* (qui utilise la clef privée) et d'un *algorithme de vérification* (utilisant la clef publique).

Protocole 2.3 Contrôle d'intégrité par une signature

1: Le client calcule $s = E(m, S_A)$ et envoie $(m \parallel s)$ au serveur.
2: Le serveur calcule $v = D(s, P_A)$ et le compare à m.
3: Si les deux correspondent, alors le serveur considère que m n'a pas été altéré.

L'algorithme de signature produit une sortie (appelée *signature*) à partir du message qui doit être signé et de la clef privée de celui qui signe. Puis la signature est attachée au message. L'algorithme de vérification prend en charge un message et une signature, et répond si oui ou non la signature est valide.

Un schéma de signature présente des exigences de sécurité similaires à celles d'un MAC, c'est-à-dire qu'il ne doit pas être possible pour une autre entité de produire une signature valide sur un message n'ayant pas encore été signé par celui qui détient la clef privée.

Notons qu'on peut combiner un schéma de signature et un chiffrement asymétrique pour assurer à la fois la confidentialité et l'intégrité. Dans ce cas, la technique la plus utilisée est la suivante : le client utilise son algorithme de signature pour signer le texte clair, puis il chiffre le texte clair et la signature en utilisant la clef publique du serveur. À la réception, le serveur déchiffre en utilisant sa clef privée, puis il vérifie la signature en se servant de l'algorithme de vérification du client.

2.2.3 Fonctions de hachage cryptographiques

Puisque les chiffrements asymétriques sont plus lents que les chiffrements symétriques, il s'ensuit que les schémas de signature sont moins efficaces que les codes d'authentification

de message. Donc, au lieu de signer un long message, on signe généralement une sorte de *message résumé* ou *condensat* fourni par une *fonction de hachage cryptographique*.

Une fonction de hachage cryptographique est utilisée pour compresser un message de taille quelconque en un petit message *aléatoire* de taille fixe. Les fonctions de hachage sont des fonctions publiques, donc supposées connues de tous. Leur utilisation ne requiert pas de clef et elles ont d'autres propriétés qui les rendent intéressantes comme la difficulté de trouver des collisions [4] ou la difficulté de les inverser (c'est-à-dire de trouver un message original à partir d'un message résumé).

Protocole 2.4 Contrôle d'intégrité par une signature et une fonction de hachage

1: Le client calcule $s = E(h(m), S_A)$ et envoie $(m \parallel s)$ au serveur.
2: Le serveur calcule $v = D(s, P_A)$ et le compare à $h(m)$.
3: Si les deux correspondent, alors le serveur considère que m n'a pas été altéré.

2.2.4 Non-répudiation

Il n'y a pas d'équivalence entre les codes d'authentification de message et les schémas de signature. Dans un schéma de signature, l'algorithme de vérification est public, donc la signature peut être vérifiée par tous. Lorsque le serveur reçoit un message signé par le client, il peut démontrer à tous, que c'est bien ce client qui l'a signé. En conséquence, le client ne peut pas nier de l'avoir fait : c'est la *non-répudiation*.

Par contre, on ne peut pas faire cela avec les codes d'authentification de message, car la clef est sécrète et ne doit être connue que par les communicants. De plus, puisque chacun des communicants connaît la clef sécrète, il sera difficile d'identifier lequel d'entre eux est réellement à l'origine du MAC : c'est la *déniabilité* (ou la possibilité de nier). Cette propriété est parfois souhaitable si l'on ne veut pas laisser de traces par exemple.

2.3 Conclusion

Dans ce chapitre, on a fait un bref rappel des bases de la cryptographie moderne. D'une part, dans la sous-section 2.1.1, on a introduit les chiffrements asymétriques comme des alternatifs aux chiffrements symétriques pouvant résoudre le problème de distribution des clefs. D'autre part, dans la sous-section 2.2.4, on présenta les signatures numériques comme étant des outils permettant d'assurer la non-répudiation. On doit cependant signaler que ces deux affirmations ne sont vraies que si l'on est capable d'associer de façon fiable une identité à une clef publique. Cela fera l'objet du chapitre suivant où l'on parlera d'infrastructures à clefs publiques et de certificats électroniques.

4. Une fonction h a une collision si l'on connaît deux messages $x \neq y$ tels que $h(x) = h(y)$.

Chapitre 3

Rappels sur les PKI, les certificats électroniques et le protocole SSL/TLS

Sommaire

Dans le chapitre 2, on a affirmé que le problème de distribution de la clef sécrète pour les chiffrements symétriques pouvait être résolu en utilisant les chiffrements asymétriques. Mais en réalité, on doit combiner les chiffrements asymétriques avec ce que l'on appelle des *Infrastructures à clefs publiques* (ou PKI pour «Public Key Infrastructure»). En effet, avec les chiffrements asymétriques uniquement, on sera confronté à d'autres types de problèmes tels que ceux qui sont listés ci-dessous.

Authentification d'une clef publique. Avant de chiffrer un message donné en utilisant une clef publique, on doit s'assurer que cette clef est bien celle de l'entité que l'on veut réellement contacter (cela permettra d'éviter des attaques de type MitM [1]). Autrement dit, il faut que l'on soit capable d'associer une identité à une clef publique.

Révocation d'une clef publique. À la suite du vol d'une clef privée par exemple, l'entité propriétaire aura besoin de faire savoir à ses contacts que sa clef publique n'est plus digne de confiance. Cette action est connue sous le nom de *révocation*.

Assurance de la non-répudiation. Le but de la signature numérique est d'assurer la non-répudiation, c'est-à-dire qu'elle permet d'éviter qu'une entité ne puisse nier d'être à l'origine d'une action. Cependant, cela ne peut bien sûr être garanti que lorsqu'on peut légalement associer une identité à une clef publique.

Application d'une politique. Il doit y avoir un ensemble de règles et d'engagements pour assurer la bonne gestion des clefs. Cela permettra de contrôler entre autres la durée de vie des clefs et leurs tailles.

Pour répondre à ces exigences, les outils utilisés, à savoir les PKI, seront présentés dans ce chapitre. Deux normes de PKI largement employées (PKIX [18] et OpenPGP [44]) seront particulièrement traitées. Aussi, on présentera brièvement le fonctionnement du protocole TLS [97] qui est utilisé pour sécuriser des communications sur un réseau informatique et dont l'implémentation repose essentiellement sur une PKI. Le lecteur pourra trouver plus d'informations sur ces notions dans les ouvrages suivants :

— «Architectures de sécurité pour internet» [35] de J-G. Dumas, P. Lafourcade et P. Redon ;

— «Cryptography and Public Key Infrastructure on the Internet» [102] de K. Schmeh ;

— «Public Key Infrastructure Implementation and Design» [26] de S. Choudhury, K. Bhatnagar et W. Haque ;

— «Cryptography and Network Security - Principles and Practice» [125] de W. Stallings ;

— «SSL and TLS Essentials : Securing the Web» [118] de S. Thomas ;

— «SSL and TLS : Theory and Practice» [86] de R. Oppliger.

3.1 Modèles de confiance

L'architecture d'une PKI repose sur ce que l'on appelle un *modèle de confiance*. Dans cette section, on va présenter les modèles de confiance les plus populaires actuellement.

1. Man-in-the-middle attack.

3.1.1 Confiance directe

Ce modèle qui est le plus simple, nécessite que l'une des deux entités souhaitant communiquer transmette sa clef publique à l'autre entité de façon directe. Par exemple, dans le cas de deux personnes, cela peut se faire par courrier électronique. Afin d'éviter une attaque de type MitM, elle peut également lui envoyer le condensat par une fonction de hachage de cette clef à travers un autre moyen de communication tel qu'un SMS.

En plus de l'authentification directe, ce modèle peut résoudre le problème de révocation. En revanche, il sera difficile, d'une part, de garantir la non-répudiation (car une entité peut nier d'être la propriétaire d'une clef publique donnée), et d'autre part, de mettre en place une politique locale. Le modèle présente également d'autres inconvénients tels que le fait qu'il nécessite que les deux entités soient en contact au préalable et la difficulté d'envoyer des données chiffrées à plusieurs entités distinctes.

3.1.2 Toile de confiance (ou WoT pour «Web of Trust»)

Le modèle précédent peut être amélioré lorsque l'on donne la possibilité aux différentes entités de se faire confiance les unes à travers les autres. Par exemple, admettons que l'on a trois entités E_1, E_2 et E_3 telles que E_1 a confiance en E_2 qui a aussi confiance en E_3 ; ces deux confiances ayant été établies avec le modèle de confiance directe. Dans une telle situation, E_2 peut transmettre la clef publique de E_3 à E_1 de manière sécurisée (chiffrement avec la clef publique de E_1 et signature avec la clef privée de E_2). Une fois que E_1 aura vérifié la signature et déchiffré le message, elle aura accès à la clef publique de E_3. Ce modèle permet d'établir la confiance de cette façon de proche en proche.

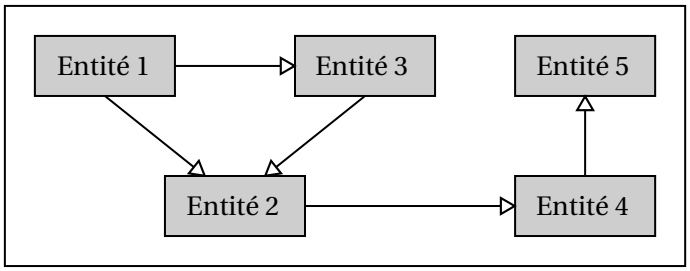


Figure 3.1 – ***Modèle WoT.***

Notons cependant que pour éviter toute confusion entre les clefs publiques, l'identité de E_3 (signée par E_2) devra aussi être associée à sa clef avant l'envoi à E_1. Une structure de données adaptée à cette dernière tâche porte le nom de *certificat électronique* ou simplement *certificat*. En plus de l'identité, on verra, dans la sous-section 3.4.2 et la sous-section 3.5.1, qu'un certificat contient d'autres informations telles qu'un numéro de série ou encore une date de validité.

La révocation avec ce modèle est beaucoup plus difficile qu'avec le modèle de confiance directe. En effet, dès qu'une clef n'est plus digne de confiance, l'entité propriétaire doit le faire savoir à toutes les autres entités qui l'ont (ce qui n'est pas évident). La mise en place d'une

politique est également difficile. En revanche, la non-répudiation pourrait se faire facilement, car chaque relation de confiance donnerait une garantie de l'association d'une entité à une clef. Mais d'un point de vue juridique, une telle preuve serait difficilement recevable.

Le modèle WoT est plus efficace que le modèle de confiance directe. Mais en pratique, il devrait être utilisé dans un environnement local comme une entreprise. Il ne marche pas aussi bien sur des réseaux informatiques de plusieurs milliers d'utilisateurs comme l'Internet (qui regroupe des centaines de millions d'utilisateurs). On présentera, dans la section 3.5, la norme OpenPGP qui repose sur ce modèle de confiance.

3.1.3 Confiance hiérarchique

Pour combler les lacunes des deux modèles précédents, une possibilité serait de mettre en place une autorité indépendante qui se chargera de la signature des certificats. Une telle autorité porte le nom d'*autorité de certification* (ou CA pour «Certificate Authority»).

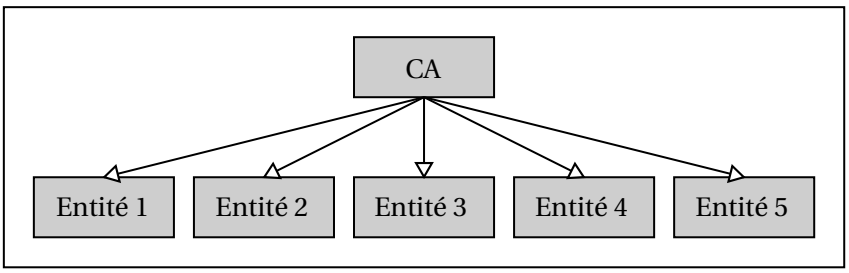


Figure 3.2 – ***Modèle de confiance hiérarchique.***

La réalisation de l'authentification est immédiate : chaque entité pourra effectuer cette tâche sur un certificat en utilisant la clef publique de la CA l'ayant signé. Mais concernant les autres demandes (révocation, non-répudiation, mise en place d'une politique), il faudra des éléments et conditions supplémentaires que l'on détaillera dans le cadre de la norme PKIX (voir section 3.4) puisque celle-ci repose sur le modèle de confiance hiérarchique.

La révocation sera gérée par la CA (voir sous-section 3.4.6) et la non-répudiation est possible puisque chaque entité donne une preuve de son identité lors de son enregistrement auprès d'une CA. Aussi, l'application d'une politique est plus facile : changement de clefs à intervalles réguliers, choix de la taille des clefs, etc.

3.2 Définition d'une PKI

Selon le RFC 4949 [107] («Internet Security Glossary, Version 2»), une PKI est un ensemble de matériels, logiciels, politiques et fonctions qui sont nécessaires pour créer, gérer, stocker, distribuer et révoquer des certificats basés sur la cryptographie asymétrique. Une PKI permet donc, en particulier, d'associer de façon *à priori* fiable une identité à une clef publique à travers un certificat (ensemble de données authentifiées par une signature). Il existe actuellement plusieurs normes de PKI dont les composantes et les formats de certificats des unes peuvent différer de ceux des autres.

Il est à noter qu'avant d'aboutir à l'idée des certificats, plusieurs techniques avaient déjà été proposées sur la façon de distribuer des clefs publiques, parmi lesquelles, on a :

Annonce publique. L'idée est de laisser chacun publier sa clef publique. Cette approche est comparable au fait de laisser chacun faire sa propre carte d'identité. Elle a donc un inconvénient évident qui est l'usurpation d'identité. Cependant, elle est utilisée par certains utilisateurs de PGP[2] qui ont pris l'habitude d'attacher leurs clefs publiques aux courriers électroniques qu'ils envoient.

Répertoire accessible au public. On associe à chaque utilisateur un dossier portant son nom et contenant sa clef publique. Ces dossiers sont mis dans un répertoire dynamique dont la gestion est confiée à une autorité de confiance. Les inscriptions se font en personne ou à travers un moyen de communication sûr et chaque utilisateur peut remplacer sa clef publique à tout moment. Lorsqu'un utilisateur a besoin de la clef publique d'un autre utilisateur, il doit passer par l'autorité de confiance pour l'avoir. Cette approche est proposée dans l'article original de W. Diffie et M. Hellman [34] qui décrit la cryptographie à clef publique.

Bien que cette proposition soit plus sûre que la première, c'est tout le système qui sera compromis si quelqu'un parvient à obtenir la clef privée de l'autorité de confiance. De plus, passer par l'autorité de confiance chaque fois que l'on a besoin de la clef publique d'un autre utilisateur peut devenir fastidieux. Cette approche n'a donc pas été largement implémentée, mais elle inspira certaines normes de PKI actuelles.

L'idée des certificats fut suggérée pour la première fois en 1978 par L. Kohnfelder [66]. Ils sont un élément essentiel des PKI actuelles dans la résolution des problèmes signalés dans les propositions précédentes entre autres. Par exemple, on n'est plus obligé de passer par une autorité de confiance lorsqu'on a besoin du certificat d'une entité donnée. Quant à leur format, il diffère selon la norme de PKI utilisée. Comme mentionné au début de cette section, il existe plusieurs normes de PKI, donc plusieurs formats de certificats. Mais la norme X.509, promulguée par l'ITU[3], est la plus utilisée sur l'Internet notamment dans le protocole SSL/TLS (voir section 3.7) pour sécuriser des connexions et le protocole S/MIME pour chiffrer et signer des courriers électroniques. Elle comporte un modèle particulier du fonctionnement de la certification que l'on détaillera dans la section 3.4. Les autres normes dont OpenPGP définissent des modèles différents et seront présentées dans la section 3.5 et la section 3.6.

Bien que les PKI soient les outils les mieux adaptés à la distribution des clefs publiques, leur sécurité fut mise en question en 2011 par plusieurs attaques[4, 5]. Mais depuis lors, des évolutions intéressantes ont été effectuées, faisant actuellement des PKI les principales méthodes d'authentification de l'Internet (à travers le SSL/TLS et le S/MIME notamment).

2. Pretty Good Privacy [44]. Une implémentation est disponible à : `https://www.gnupg.org/`.

3. ITU ou «International Telecommunication Union» est l'agence des Nations unies pour le développement spécialisé dans les technologies de l'information et de la communication, `https://www.itu.int`.

4. An Attack Sheds Light on Internet Security Holes, `https://www.nytimes.com/2011/04/07/technology/07hack.html`.

5. Fake DigiNotar web certificate risk to Iranians, `https://www.bbc.com/news/technology-14789763`.

3.3 Variantes de PKI hiérarchiques

Le modèle de confiance hiérarchique permet plusieurs variantes dont la hiérarchie à deux niveaux, la variante toilée, la certification croisée et les hiérarchies de CA. Il peut donc être implémenté de différentes manières.

3.3.1 Hiérarchie à deux niveaux

Il s'agit de la plus simple des variantes de PKI hiérarchiques. Le premier niveau (ou le niveau inférieur) est composé d'utilisateurs et le second niveau (ou le niveau supérieur) ne comprend qu'une CA. Chaque utilisateur aura uniquement besoin du certificat de cette CA dont il pourra ensuite extraire la clef publique pour vérifier la signature d'un autre certificat que cette CA aurait émis. La hiérarchie à deux niveaux est illustrée par la Figure 3.2.

3.3.2 Variante toilée

Similaire à la variante précédente à la différence que le certificat d'un utilisateur peut être signé par plusieurs CA. Dans ce cas, la vérification de la signature d'un certificat donné nécessitera les certificats de chacune des CA ayant signé. Un autre inconvénient de cette variante est la quasi-impossibilité de la mise en place d'une politique locale compte tenu du nombre de CA. La Figure 3.3 donne une illustration de la variante.

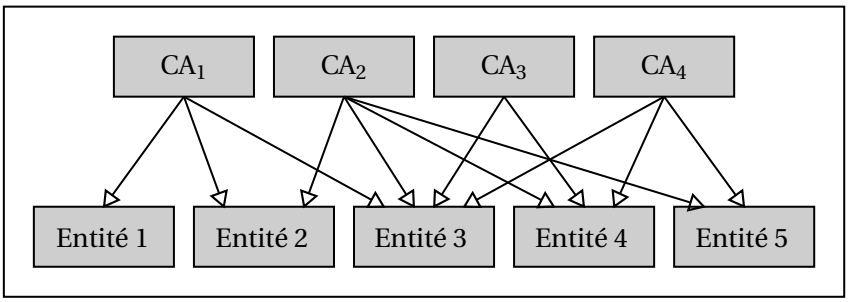


Figure 3.3 – ***Variante toilée.***

3.3.3 Certification croisée

Une variante dans laquelle le certificat d'un utilisateur est signé par une seule CA, et des CA peuvent signer mutuellement leurs certificats (voir Figure 3.4). Cela permet d'établir une confiance entre les utilisateurs de deux CA. Cependant, un inconvénient de cette variante est qu'un certificat doit toujours être accompagné du certificat de sa CA.

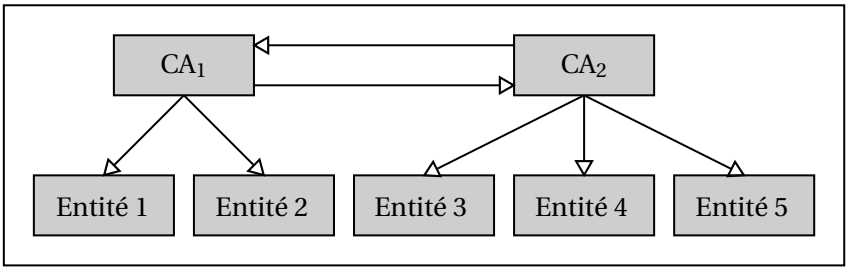


Figure 3.4 – ***Certification croisée.***

3.3.4 Hiérarchies de CA

Dans le cas de plus de deux CA, le modèle de certification croisée serait difficile à mettre en place. En revanche, on pourrait fixer une CA de niveau supérieur qui émet des certificats à d'autres CA qui pourront ensuite signer des certificats d'utilisateurs ou éventuellement des certificats d'autres CA subordonnées (voir Figure 3.5). La CA en haut de la hiérarchie porte le nom de *CA racine* et son certificat (appelé *certificat racine*) est *auto-signé* (ou «self-signed» en anglais), c'est-à-dire qu'il signe son propre certificat. Dans la section suivante (section 3.4), on présentera la norme PKIX qui repose sur cette variante de PKI hiérarchique. On verra, lors de la validation d'un certificat dans la sous-section 3.4.5, que tous les certificats intermédiaires sont nécessaires.

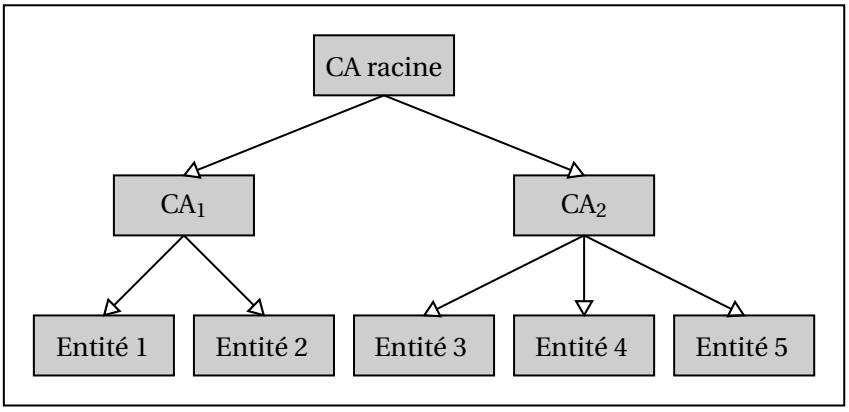


Figure 3.5 – ***Hiérarchies de CA.***

3.4 Norme PKIX

La norme X.509 fut créée par l'ITU en 1988 dans le cadre de la norme X.500 [53] qui définit un service d'annuaire (un serveur ou un ensemble distribué de serveurs qui maintient une base de données d'informations sur les utilisateurs). Les informations sont composées d'une association du nom d'utilisateur à l'adresse réseau, ainsi que d'autres attributs et informations sur les utilisateurs. La norme comprend également un système de nomenclature (appelé «Distinguished Naming») qui décrit des entités par leurs positions dans une certaine hiérarchie. Un exemple de cette nomenclature ressemble à ce qui suit :

```
CN=Mohamed Traoré; OU=Institut Fourier; O=Université Grenoble Alpes; C=FR
```

Ce nom décrit une personne avec un «Common Name» (CN) de "Mohamed Traoré" qui travaille dans une «Organisational Unit» (OU) nommée "Institut Fourier" d'une «Organisation» (O) appelée "Université Grenoble Alpes" et située en "France" («Country» (C)).

La norme X.509 sert également de base à l'une des PKI les plus utilisées actuellement : la PKIX pour «Public Key Infrastructure X.509». Cette PKI est l'œuvre de l'IETF [6] et standardise tout ce qui est important pour la mise en place d'une PKI sur l'Internet. Son groupe de travail fut formé en 1995, elle repose sur le modèle de la confiance hiérarchique, et elle utilise les formats de certificats et de listes de révocation de certificats de la norme X.509.

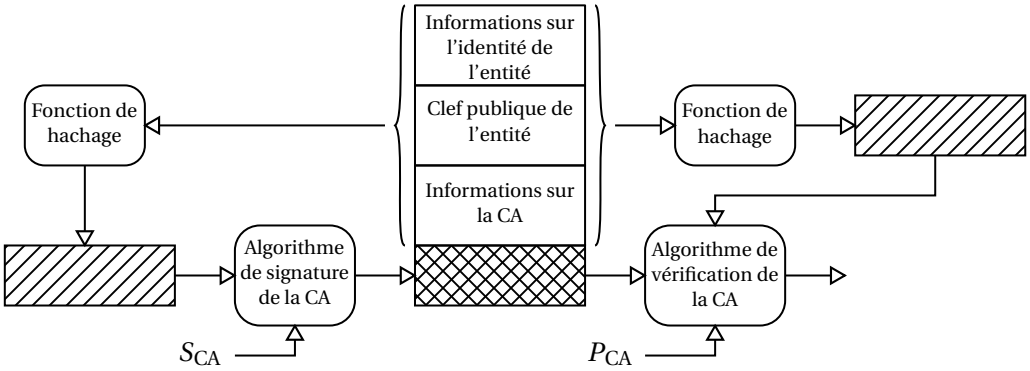


Figure 3.6 – ***Création d'un certificat X.509 et vérification de sa signature.***

Avec la norme PKIX, une CA est une autorité de confiance telle qu'une agence gouvernementale ayant la confiance d'une communauté d'utilisateurs. Le certificat fourni par une CA contiendra entre autres le nom de l'entité, le nom de la CA, la clef publique de l'entité, une période de validité et un numéro de série de certificat (voir sous-section 3.4.2). L'ensemble de ces données est signé avec la clef privée de la CA. Quant à la validation d'un certificat, on vérifie sa date de validité, on utilise la clef publique de la CA pour certifier la signature qu'il contient (voir Figure 3.6) ainsi que son statut de révocation. Ce processus de validation sera plus détaillé dans la sous-section 3.4.5.

Notons que la clef publique d'une CA est typiquement certifiée par une autre CA (voir Figure 3.7). Ce mécanisme de certification en chaîne entre des CA peut être utilisé autant de fois qu'on le souhaite. Le point où la chaîne s'arrête correspondra à une CA qui doit signer son propre certificat. Ce certificat est dit *auto-signé* (ou «Self-signed Certificate») et un tel certificat qui fait foi est généralement appelé *certificat racine* (ou «Root Certificate»). Les certificats racines sont essentielles au processus de validation des clefs publiques. Ils doivent être intrinsèquement reconnus par les applications, car aucun autre certificat ne les signe. Lors de la validation d'un certificat donné, la signature de chaque certificat de la chaîne de certification devra être vérifiée.

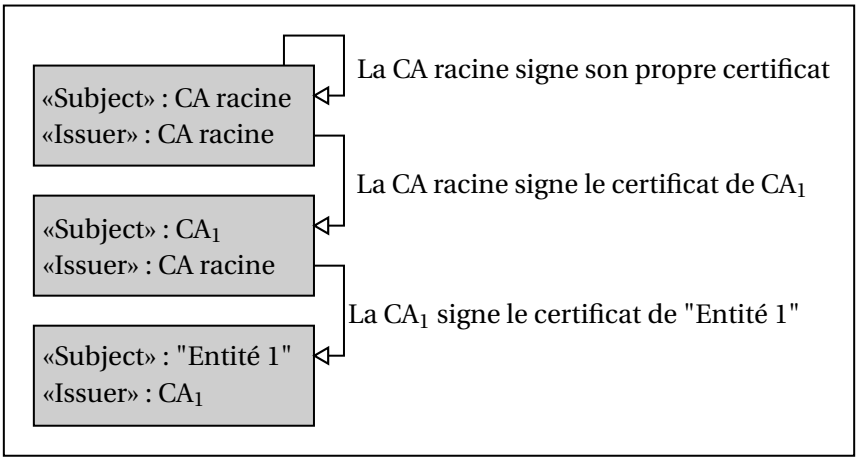


Figure 3.7 – ***Chaîne de certificats X.509.***

6. Internet Engineering Task Force, https://www.ietf.org/.

Des navigateurs web comme Mozilla Firefox utilisent des certificats X.509 pour valider des clefs afin d'établir des connexions SSL/TLS. Ce navigateur dispose d'un grand nombre de certificats de CA installés (par défaut) que l'on peut consulter en ouvrant le menu :

```
Paramètres > Vie privée et sécurité > Afficher les certificats
```

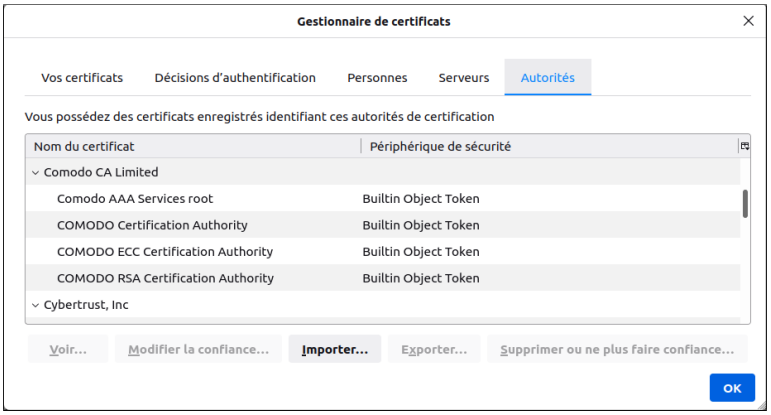


Figure 3.8 – ***Certificats de CA dans Mozilla Firefox.***

Enfin, comme toute PKI actuelle, la norme PKIX dispose d'un mécanisme de révocation de certificats, c'est-à-dire qu'une CA a la possibilité d'annuler une action de certification. Cette fonctionnalité est en effet intéressante dans certaines situations comme lorsqu'une CA fournit un certificat par erreur, ou lorsqu'un utilisateur perd ou divulgue accidentellement sa clef privée. Ces certificats ne peuvent évidemment pas être remplacés par la CA puisqu'ils ont déjà été émis. En plus, ils peuvent paraître légaux puisqu'ils ont des signatures valides et il est possible qu'ils aient des périodes de validité non dépassées. Toutefois, la norme PKIX dispose d'outils permettant de gérer ces situations. Ces outils seront présentés dans la sous-section 3.4.6 et leur utilisation constituera la dernière étape du processus de validation d'un certificat X.509.

3.4.1 Éléments d'une PKIX

L'exemple que l'on donnera ici sera celui d'une PKI composée d'une seule CA. L'ensemble des composantes internes de cette PKI porte le nom de *centre de confiance* (ou «Trust Centre»). Les principaux éléments de ce centre de confiance sont listés ci-dessous.

Certification Authority. La CA est l'élément le plus important d'un centre de confiance. Elle a pour rôle de générer des certificats X.509 qu'elle doit ensuite délivrer. Elle peut aussi fournir les listes de révocation de certificats (ou CRL pour «Certificate Revocation List») lors de la validation d'un certificat.

Registration Authority. La RA est un élément optionnel qui peut effectuer des tâches administratives à la place de la CA. Lors de la demande d'un certificat, on peut s'adresser à une RA qui vérifiera les données puis les transmettra à la CA.

CRL Issuer. Cet élément optionnel peut être utilisé pour publier des CRL à la place de la CA.

Repository. Lorsqu'un certificat est généré par la CA, il doit être disponible pour d'autres utilisateurs. À cet effet, un serveur (appelé «Repository» ou DIR pour «Directory») est mis en place pour contenir tous les certificats créés par la CA. Il peut aussi contenir des CRL et les garde jusqu'à ce qu'ils soient retirés.

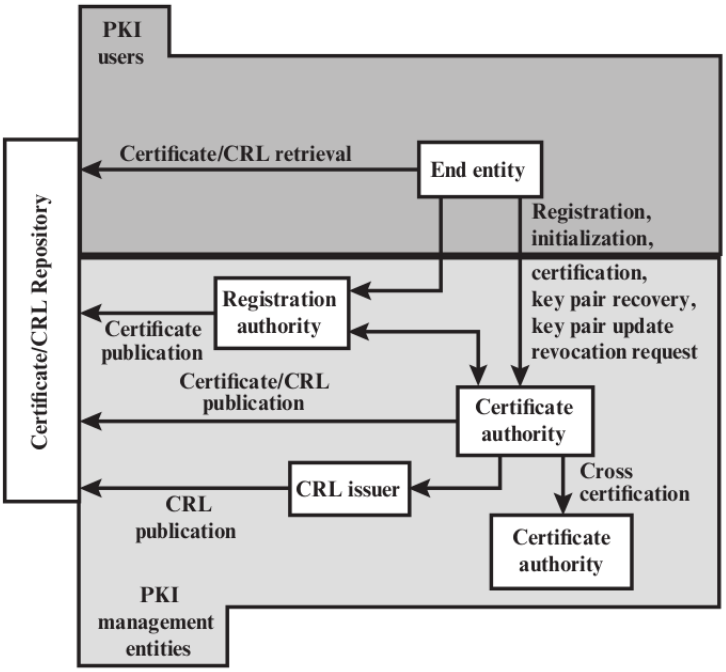


Figure 3.9 – ***Relations entre les éléments d'une PKIX.*** *(Source [125])*

Les composantes (comme la RA) qui peuvent décharger la CA de certaines tâches ont été introduites pour limiter les fonctions du serveur qui détient la clef privée de la CA, et donc de réduire les risques. La RA est alors considérée comme un serveur délégué. Des extensions récentes de la norme X.509 ont aussi ajouté un second rôle délégué, l'*autorité de validation* (ou VA pour «Validation Authority»), qui détient la réponse aux demandes concernant la validité d'un certificat après sa création. Avec un usage judicieux des RA et VA, il est possible de construire des PKI où le serveur de la CA est seulement accessible par un nombre limité d'autres serveurs, réduisant donc les menaces en provenance de l'extérieur du réseau.

En plus de ces composantes que l'on de vient de présenter ci-dessus, il existe aussi des composantes décentralisées.

End Entity. Une entité finale est typiquement un poste de travail avec un logiciel cryptographique, un serveur web ou tout autre dispositif pouvant traiter des certificats. Pour obtenir un certificat auprès d'une CA, elle peut soit faire une *demande de signature de certificat* (ou CSR pour «Certificate Signing Request»), soit laisser la CA se charger des deux tâches (génération des clefs et signature du certificat). Dans le second cas, une po-

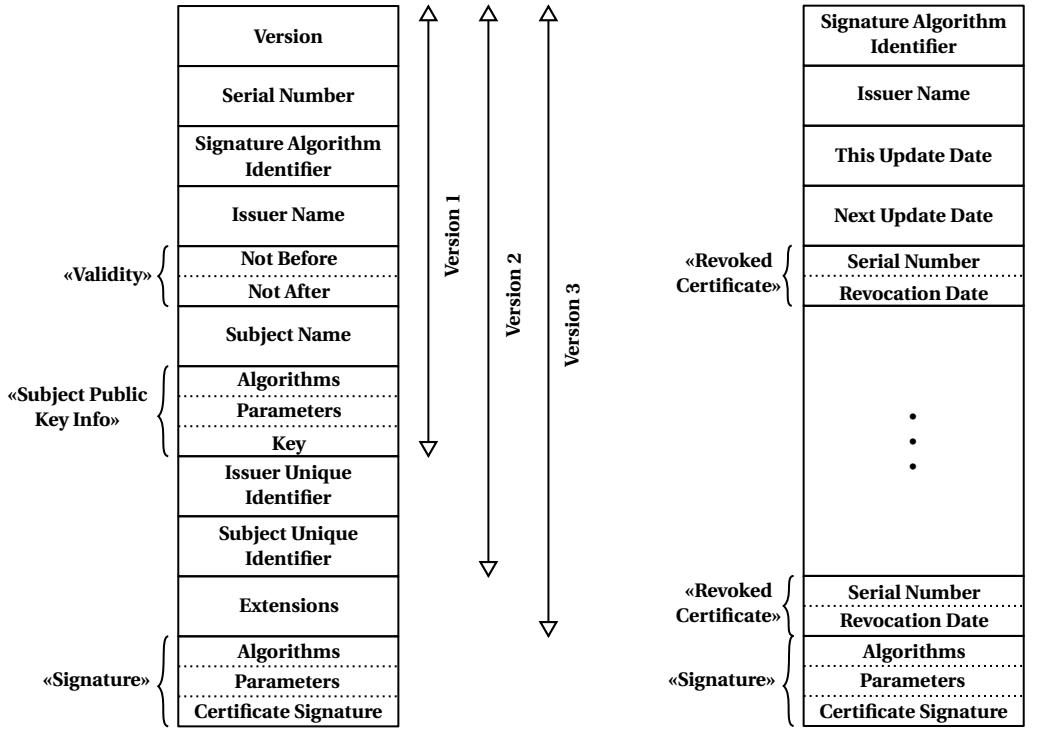
litique de *sauvegarde de clef privée* pourrait être mise en place dans le but de pouvoir répondre à une éventuelle perte de clef privée par l'entité finale.

Revocation Authority. La REV est l'endroit où une entité doit se rendre lorsqu'elle veut que son certificat soit révoqué.

Personal Security Environment. Le PSE est l'endroit où la clef privée de l'entité est stockée. C'est généralement une carte à puce ou un fichier chiffré sur un disque dur.

3.4.2 Contenu d'un certificat X.509

Les certificats conformes à la dernière version de la norme X.509 sont des structures de données contenant jusqu'à onze champs différents. Ils sont décrits en utilisant une notation spéciale connue sous le nom de «Abstract Syntax Notation One» ou ASN.1 [70, 71] (dont on donne plus de détails dans la sous-section 7.2.1 du chapitre 7 lors du traitement des certificats qui nous ont été fournis par l'ANSSI). Leur ordre dans le certificat correspond à ce qui est illustré par la Figure 3.10a.



(a) ***Champs d'un certificat X.509.*** (b) ***Liste de révocation de certificats.***

Figure 3.10 – ***Formats de la norme X.509.***

Version. Le champ «Version» identifie la version particulière de la norme X.509 à laquelle le certificat est conforme. Sa valeur (qui est 1 par défaut) change en fonction du nombre de champs du certificat comme on peut le constater sur la Figure 3.10a. Bien que les

spécifications actuelles de la norme X.509 soient à la version 7, aucun changement n'a été apporté au nombre de champs depuis la version 3. La valeur du champ «Version» est donc 1 ou 2 ou 3.

Serial Number. La valeur de ce champ est un entier naturel non nul (généralement sur 20 octets) attribué à un certificat par une CA. Cette dernière qui a le contrôle total sur ce champ doit s'assurer que la valeur est unique pour chaque certificat qu'elle émet afin de pouvoir identifier les certificats qu'elle délivre.

Signature Algorithm Identifier. Dans les spécifications de la norme X.509, ce champ porte le nom «Signature» bien que ce choix soit inapproprié puisque le champ ne contient aucune signature. Par contre, comme son nom l'indique ici, il identifie l'algorithme cryptographique utilisé par la CA (pour signer les certificats) ainsi que les paramètres de cet algorithme. Notons que ces informations sont répétées dans le dernier champ du certificat dont la plupart des implémentations préfèrent utiliser le contenu plutôt que celui du champ «Signature Algorithm Identifier».

Issuer Name. Le champ «Issuer Name» identifie la CA qui a créé et émis le certificat. Il prend la forme d'un «Distinguished Name» [61] (ou nom en format X.500) qui commence généralement par le nom d'un pays puis celui d'une ville, celui d'une organisation, etc.

Validity. Ce champ, composé de deux dates «Not Before» et «Not After», définit la période durant laquelle le certificat est valide. Au delà, le certificat est considéré comme invalide et ne devrait plus être utilisé. On verra, dans la sous-section 3.4.5, que ce champ ne suffit pas à lui seul pour décider de la validité d'un certificat.

Subject Name. Le champ «Subject Name» identifie l'entité finale qui détient la clef privée ayant été certifiée. Comme le champ «Issuer Name», ce champ prend la forme d'un «Distinguished Name». Son contenu peut s'étendre aux utilisateurs particuliers et son élément le plus important est le «Common Name» qui correspond typiquement au nom réel du sujet certifié.

Subject Public Key Info. Ce champ contient la clef publique de l'entité finale et est toute la raison du certificat. Il identifie aussi l'algorithme de chiffrement utilisé par l'entité ainsi que tout autre paramètre associé. Par exemple, si l'algorithme à clef publique est le chiffrement RSA, alors ce champ contiendra le module et l'exposant public.

Issuer Unique Identifier. Ce champ optionnel, ayant été introduit à partir de la version 2 de la norme X.509, permet d'identifier deux CA différentes ayant le même «Issuer Name».

Subject Unique Identifier. Comme le cas précédent, ce champ fut introduit à partir de la version 2 de la norme X.509 et permet à deux sujets d'avoir le même «Subject Name». Par exemple, les certificats de deux personnes de même nom travaillant dans la même organisation peuvent être identifiés à l'aide des valeurs de ce champ.

Extensions. Le champ «Extensions» fut introduit dans la version 3 de la norme X.509 qui est actuellement la dernière version. Il donne la possibilité aux autorités de certification d'ajouter leurs propres informations privées aux certificats.

Signature. Ce champ est le dernier élément d'un certificat X.509. Dans les spécifications, il est nommé «Encrypted» et contient l'identifiant de l'algorithme de signature, un résumé (ou condensat) de tous les champs du certificat, et une signature de ce résumé.

3.4.3 Enregistrement d'un certificat X.509

Quand un utilisateur fait une demande de certificat auprès d'une CA, l'enregistrement est géré par la RA qui opère comme une interface entre l'utilisateur et la CA. Les demandes sont faites à travers un formulaire d'enregistrement fourni par la RA. Les principales fonctions de cette dernière sont la vérification de l'identité de l'utilisateur, la gestion de l'enregistrement et la transmission des informations à la CA pour compléter le processus d'enregistrement. Les étapes suivantes décrivent globalement ce processus d'enregistrement.

1. L'utilisateur envoie une demande de formulaire d'enregistrement à la RA, car le format du formulaire est prédéfini par cette dernière. Notons que la demande du formulaire peut se faire en ligne selon la politique qui régit les fonctions de la RA.
2. À la réception de la demande, la RA envoie le formulaire d'enregistrement à l'utilisateur.
3. L'utilisateur soumet le formulaire complété à la RA.
4. À partir des informations figurant sur le formulaire, la RA vérifie l'identité de l'utilisateur et envoie la demande d'enregistrement à la CA. La vérification faite à cette étape dépend de la politique qui régit les fonctions de la RA, mais aussi de l'usage futur du certificat. Par exemple, si le but est d'utiliser le certificat pour une opération financière, le processus de vérification pourrait être plus strict.
5. Après la vérification de la demande, la CA envoie sa réponse à la RA. La réponse peut être négative si l'utilisateur ne remplit pas une condition primordiale.
6. Si la réponse de la CA est positive, la RA enregistre l'utilisateur et lui transmet les informations d'enregistrement.

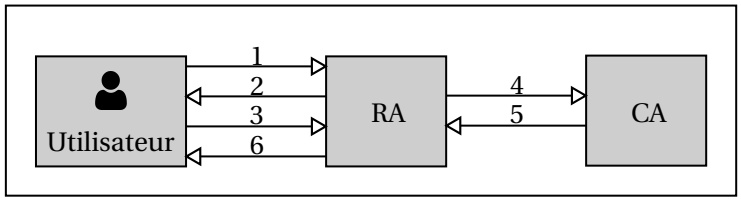


Figure 3.11 – ***Enregistrement d'un certificat X.509.***

Dès que le processus d'enregistrement est complet, une paire de clefs doit être générée dont la composante publique est mise dans le certificat délivré à l'utilisateur. Cette paire de clefs devrait être générée par l'utilisateur. Cependant, suivant la disponibilité des ressources nécessaires à la génération de la paire de clefs et l'usage prévu du certificat, la CA pourrait aussi s'en charger. Par exemple, si la clef privée doit servir pour authentifier l'utilisateur, la CA pourrait s'occuper de la génération de la paire de clefs. En revanche, si c'est pour garantir la non-répudiation, il est conseillé que l'utilisateur le fasse. En effet, si la CA le fait, elle devra transmettre la clef privée à l'utilisateur (ce qui présente des risques) et devrait avoir un moyen de sauvegarder cette clef pour débloquer la situation d'une perte de la clef originale par l'utilisateur. En plus de ces risques, cet utilisateur pourrait même nier ultérieurement d'être l'auteur de plusieurs opérations du processus d'enregistrement.

3.4.4 Mise à jour d'un certificat X.509

Un certificat est valide pour une durée limitée dont le décompte commence le jour où le certificat est délivré. Lorsque le certificat arrive à terme, on peut le mettre à jour. Ce processus de mise à jour devrait être effectué automatiquement dès que les 70% ou 80% de la vie d'un certificat sont écoulés. Cela offre aux autres utilisateurs une période raisonnable de transition vers le nouveau certificat. À cet effet, la CA délivre une paire de certificats de renouvellement de clefs.

— Le premier certificat contient l'ancienne clef publique et est signé avec la nouvelle clef privée. Cela permet aux utilisateurs ayant des certificats signés avec la nouvelle clef privée de construire une chaîne de certification valide vers les certificats signés avec l'ancienne clef privée.

— Le second certificat contient la nouvelle clef publique et est signé avec l'ancienne clef privée. Cela permet aux utilisateurs ayant des certificats signés avec l'ancienne clef privée de construire une chaîne de certification valide vers les certificats signés avec la nouvelle clef privée.

De cette façon, les utilisateurs dont les certificats sont signés avec l'ancienne clef privée et les utilisateurs dont les certificats sont signés avec la nouvelle clef privée peuvent valider leurs certificats mutuellement. Cet exemple correspond au cas où la paire de clefs est remplacée. Il est cependant possible de conserver la même paire de clefs lors d'une mise à jour de certificat.

3.4.5 Validation d'un certificat X.509

L'utilisation d'un certificat est précédée d'une vérification de sa validité. Cela comprend généralement les étapes suivantes : vérifier la date de validité pour s'assurer que le certificat n'est pas expiré ; vérifier que la signature est valide ; vérifier que le certificat a été délivré par une CA fiable ; s'assurer que le certificat peut être utilisé aux fins prévues ; vérifier que le certificat n'a pas été révoqué.

Ce processus de validation est assez complexe et n'est pas implémenté par tous. Les rares applications cryptographiques (dont OpenSSL et Microsoft CryptoAPI) qui l'implémentent, utilisent souvent des techniques différentes en fonction de leurs politiques. Cependant, ces applications offrent plusieurs options, donc leur usage nécessite la compréhension du processus de validation. Les étapes énumérées ci-dessus sont détaillées ci-dessous.

Construction de la chaîne de certification et validation des signatures. On commence par localiser le certificat de la CA ayant signé le certificat cible en cherchant parmi les certificats intermédiaires un certificat dont le «Subject» correspond au «Issuer» du certificat cible. Si plusieurs certificats sont trouvés, on ne garde que celui dont l'extension «Subject Key Identifier» convient. Si, encore une fois, plusieurs certificats correspondent, on garde uniquement le plus récent.

Notons qu'en raison de la révocation de certains certificats intermédiaires, on peut trouver plusieurs chaînes de signature et chaque élément de chacune de ces chaînes doit être vérifié pour obtenir la chaîne valide. Une fois que la CA est trouvée, on vérifie la signature du certificat cible avec la clef publique de cette CA. Si la vérification

échoue, le processus de validation s'arrête et le certificat cible est jugé invalide. Si la signature convient et que la CA est reconnue par l'application, la vérification passe à l'étape suivante. Si cette CA n'est pas reconnue, alors son certificat est traité à son tour comme un certificat cible et doit passer les vérifications précédentes.

La construction d'une chaîne complète vers un certificat reconnu requiert que l'application soit en possession de tous les certificats de cette chaîne. Cela nécessite qu'elle ait une base de données de certificats intermédiaires ou que le protocole utilisant le certificat fournisse les certificats intermédiaires. Le protocole SCVP [7] fournit un mécanisme pour demander à un serveur une chaîne de certification, permettant ainsi d'éviter une boucle au début de cette étape.

Vérification de la date de validité et de l'usage prévu du certificat. Après la construction de la chaîne de signature, il s'agira de vérifier la validité temporelle de cette chaîne. On serait dans l'une des situations suivantes.

— La période de validité d'aucun certificat de la chaîne n'est dépassée.

— Le certificat cible a été signé pendant que l'un au moins des certificats de la chaîne était au-delà de sa période de validité.

— La chaîne est actuellement au-delà de sa période de validité, mais le certificat cible a été signé dans une période où chaque certificat de la chaîne était valide.

Dans le premier cas, la vérification passe à l'étape suivante. Mais dans les deux autres cas, l'utilisateur devrait au moins recevoir un avertissement et la suite de la décision dépendra de la politique de l'application utilisée. Par exemple, certaines applications cryptographiques requièrent que la période de validité d'un certificat soit incluse dans la période de validité du certificat de sa CA.

Une fois que la chaîne est construite, on doit également vérifier que certains champs d'extension X.509 sont également valides.

— **Basic Constraints.** Il est nécessaire pour les CA et limite la profondeur de la chaîne sous un certificat de CA particulier.

— **Name Constraints.** Il permet de limiter l'espace des noms d'identité certifiés sur le certificat d'une CA donnée.

— **Key Usage; Extended Key Usage.** Ils limitent les objectifs pour lesquels une clef de certificat peut être utilisée. Par exemple, le «Key Usage» d'un certificat de CA doit avoir la valeur «Allow» pour autoriser la signature d'autres certificats.

Consultation d'autorités de révocation. Après la vérification des signatures et des périodes de validité de la chaîne de certification, on peut vouloir consulter les CA mentionnées sur chaque certificat de la chaîne afin de vérifier qu'ils sont actuellement valides. Pour cela, les certificats peuvent contenir des extensions qui indiquent des emplacements d'une liste de révocation de certificats ou qui dirigent vers des répondeurs OSCP. Ces méthodes permettent de vérifier qu'un certificat n'a pas été révoqué par sa CA.

7. Server-based Certificate Validation Protocol.

3.4.6 Révocation d'un certificat X.509

La création d'un certificat par une CA permet d'associer une identité à une clef publique. Cette association peut être invalidée par plusieurs événements dont ceux listés ci-dessous.

— Le dépassement de la période de validité du certificat.

— La compromission de la clef privée du certificat (cas d'un vol par exemple).

— La compromission de la clef privée de la CA (auquel cas tous les certificats émis par cette CA devront être révoqués).

— L'altération du contenu du certificat comme un changement dans l'identité de l'entité propriétaire (qui pourra ensuite demander un nouveau certificat avec ses nouvelles informations et révoquer l'ancien certificat).

Pour gérer de telles situations, on doit pouvoir révoquer un certificat à travers une sorte de notification publique pour faire savoir à d'autres entités connectées qu'il n'est plus digne de confiance. En général, deux mécanismes sont utilisés pour effectuer cette tâche.

Publication périodique. Elle repose sur les *listes de révocation de certificats* (ou CRL pour «Certificate Revocation Lists»). Ce mécanisme est implémenté par la version originale de la norme X.509. Une CRL est un document électronique signé par une CA, qui est régulièrement mis à jour et qui contient une liste de certificats révoqués par cette CA. Pour les certificats de CA révoqués par d'autres CA, les numéros de série sont placés dans des *listes de révocation d'autorités* (ou ARL pour «Authority Revocation Lists») qui sont formatées de la même façon que les CRL. La première version de la norme X.509 définissait le format des CRL par une structure de données élémentaire. Mais cette version présentait certains problèmes liés notamment à la sécurité (des informations d'une CRL pouvaient être modifiées à l'insu de la CA et des utilisateurs) et la taille (pas de mécanisme pour contrôler ou limiter la taille d'une CRL). L'apparition de la deuxième version de la norme X.509 corrigea ces problèmes avec un nouveau format qui est d'ailleurs celui utilisé dans la troisième et l'actuelle version de la norme X.509. Ce format est illustré par la Figure 3.10b et contient les champs suivants entre autres.

— **Version.** Ce champ indique la version de la CRL. Sa valeur est actuellement 2, car le champ fut introduit à la version 2 des CRL.

— **Signature Algorithm Identifier.** Ce champ contient l'identifiant de l'algorithme avec lequel la CA a signé la CRL.

— **Issuer Name.** Il contient le «Distinguished Name» de la CA ayant délivré la CRL.

— **This Update Date.** Il contient la date à laquelle la CRL a été publiée.

— **Next Update Date.** Sa valeur est la date à laquelle la prochaine CRL sera publiée (ou l'actuelle CRL expirera).

— **List of Revoked Certificate.** C'est une liste des certificats révoqués. Chaque certificat révoqué est représenté par des informations telles que son numéro de série, sa date de révocation, des extensions.

— **CRL Extensions.** Ce champ, introduit dans la version 2, contient des informations supplémentaires comme, par exemple, la raison de la révocation (compromission de clef, changement d'affiliation, compromission de CA, etc.).

Dès qu'un certificat est révoqué, l'information de révocation est publiée dans une CRL usuellement fournie par la CA du certificat révoqué. Il peut y avoir un décalage entre le moment où le certificat est révoqué et le moment où l'information est publiée dans une CRL. Ce délai doit être court et dépend de la politique de la CA (par exemple, toutes les heures ou tous les jours selon les CA).

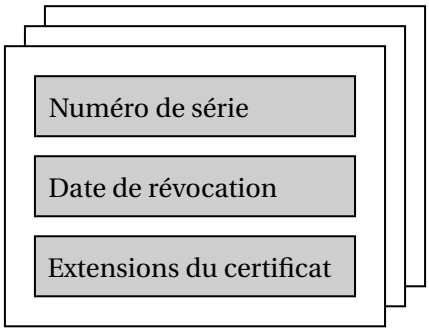


Figure 3.12 – ***Contenu d'un certificat X.509 révoqué.***

Dans la sous-section 3.4.5, on a signalé qu'une étape de la validation d'un certificat peut passer par la consultation de CRL. Cela se fait globalement de la manière suivante.

— Vérifier le champ de la date de validité du certificat.

— Si le certificat n'est pas expiré, l'utilisateur procède à une demande de la dernière CRL auprès de la CA.

— À la réception de cette CRL, l'utilisateur valide la signature numérique de la CA qui délivre la CRL.

— Enfin, l'utilisateur vérifie si le numéro de série du certificat est présent sur la CRL. S'il y est, cela signifie que le certificat a été révoqué.

L'utilisateur a besoin de suivre le processus ci-dessus chaque fois qu'il veut vérifier la validité d'un certificat. Cependant, ce processus ne peut pas être effectué tout le temps, notamment pour des raisons de connexion internet. En conséquence, la dernière CRL ne peut pas toujours être obtenue pour une vérification. Pour débloquer cette situation, on utilise des *CRL mises en cache* qui sont des CRL stockées localement par les utilisateurs sur leurs matériels. Cela augmente considérablement la vitesse du processus de vérification et ne nécessite pas de demande de CRL auprès des CA. Mais les CRL mises en cache pourraient contenir des informations obsolètes. Donc les utilisateurs doivent régulièrement les mettre à jour en téléchargeant leurs dernières copies.

Notons aussi que le coût de maintenance et de transmission d'une CRL a été identifié comme un facteur important du coût d'exécution d'une PKIX. Ce coût peut être réduit en utilisant une *CRL delta* (ou «Delta CRL» en anglais) [18] qui contient uniquement des certificats ayant été révoqués depuis la publication de la dernière CRL.

Requête en ligne. Elle se fait en utilisant principalement le *protocole de vérification de certificat en ligne* (ou OCSP pour «Online Certificate Status Protocol»). Cet outil fut conçu avec l'objectif de réduire les coûts de transmission des CRL et éliminer le délai entre la révocation d'un certificat et la publication de l'information de révocation dans une CRL. Un certificat peut contenir une référence vers un *serveur* ou *répondeur OCSP*. Ce répondeur est une entité de confiance qui, en recevant une demande relative à la révocation d'un certificat, répond au demandeur avec les informations sur le statut de ce certificat. Ce processus peut se résumer aux étapes suivantes.

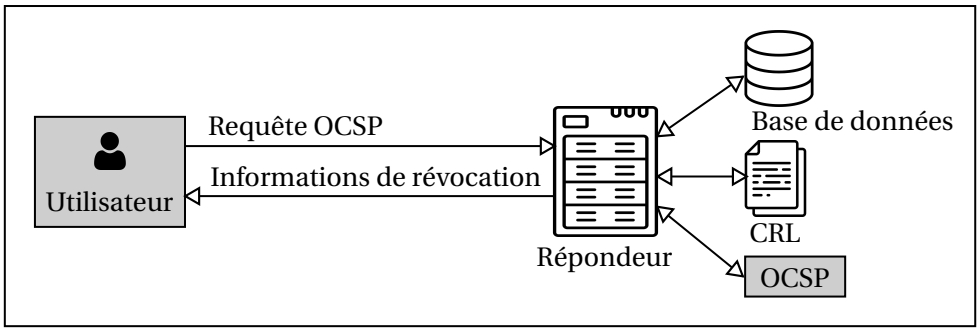


Figure 3.13 – ***Protocole OCSP.***

1. Un utilisateur envoie au répondeur OCSP une demande composée entre autres d'un condensat de «Issuer Name», d'un condensat de «Subject Name», du numéro de série du certificat et du numéro de la version du protocole.
2. Le répondeur traite la demande et répond avec les informations sur le statut du certificat. Ces informations, signées numériquement par le répondeur, ont une période de validité et peuvent prendre trois valeurs : *bon* (ou «Good»), *révoqué* (ou «Revoked») et *inconnu* (ou «Unknown»).

   La première valeur indique que le certificat est valide (donc non révoqué) ; la deuxième valeur signifie que le certificat a été révoqué et n'est donc plus valide ; la troisième valeur indique que les informations concernant le statut de ce certificat n'ont pas pu être récupérées.
3. Suivant la réponse du serveur OCSP, l'utilisateur prend la décision qui convient.

L'utilisation du protocole OCSP évite de télécharger une liste entière de certificats révoqués. Cependant, en plus d'avoir une connexion internet permanente, le protocole présente d'autres inconvénients.

— Il pourrait y avoir un décalage entre une révocation et la connaissance de cette information par le répondeur bien que toute l'opération se fasse en ligne.

— Le protocole OCSP ne définit pas la manière dont l'information est extraite d'un répertoire de CRL par le répondeur.

— Puisque les réponses sont systématiquement signées, cela peut donc nuire à la performance du processus.

En pratique, les informations de révocation sont obtenues à travers les CRL. Mais le protocole OCSP devient de plus en plus populaire malgré les limitations que l'on vient d'énumérer. Il permet à des organisations de valider des certificats efficacement (en temps et en mémoire) et certaines organisations amélioreraient même le fonctionnement du protocole en mettant en relation plusieurs répondeurs OCSP de sorte que si l'un d'entre eux ne dispose pas d'une information, qu'il puisse se connecter à un autre répondeur pour l'obtenir.

3.5 Norme OpenPGP

La norme OpenPGP [44], proposée par l'IETF en 1998, vient à l'origine du logiciel cryptographique PGP[8] conçu par P. Zimmermann en 1991. Ce logiciel fournit des services comme l'authentification et la confidentialité. Il est actuellement populaire et est typiquement utilisé pour sécuriser des courriers électroniques à travers l'Internet. Parmi les fonctionnalités qui ont fait de ce logiciel un outil populaire et puissant, il y a sa flexibilité d'usage sur différentes plate-formes, sa vitesse de chiffrement et sa disponibilité gratuite sur l'Internet.

À la différence de la PKIX, la norme OpenPGP repose sur le modèle *toile de confiance* (ou WoT pour «Web of Trust») qui est aussi une invention de P. Zimmermann. Ce modèle remplace la liaison identité-clef à travers une CA (dans le cas d'une PKIX) par une liaison à travers plusieurs chaînes semi-fiables. Dans un système WoT, l'utilisateur final maintient une base de données de clefs et identités associées dont chaque association reçoit deux cotes de confiance. La première cote indique à quel point le lien entre la clef et l'identité est fiable, et la seconde cote indique le degré de confiance d'une identité particulière pour importer de nouvelles relations. Les utilisateurs peuvent créer et signer un certificat ainsi qu'importer des certificats créés par d'autres utilisateurs. Cependant, il y a des règles à suivre ; elles seront détaillées dans la sous-section 3.5.1 et la sous-section 3.5.2 à travers une présentation des certificats supportés par PGP et des porte-clefs PGP.

Quant à la révocation, elle est gérée différemment de celle de la norme X.509, car PGP n'utilise pas de CA. Un utilisateur peut révoquer sa clef en postant sur un serveur public un message de révocation signé. Tout autre utilisateur, en connaissance de ce message, doit supprimer cette clef de sa base de données.

3.5.1 Certificats utilisés dans PGP

Pour la distribution des certificats, PGP utilise un *serveur de certificats PGP*. Ce serveur agit comme un répertoire où une CA publie les certificats qu'elle délivre dans le cas d'une PKIX. Un utilisateur peut mettre son certificat à la disposition d'autres entités en le publiant sur le serveur et peut également y extraire le certificat d'un autre utilisateur. Ces certificats doivent suivre un certain format qui, à la fois, définit leur structure et doit être reconnu par d'autres implémentations de PKI dans le but de permettre l'interopérabilité. À cet effet, le PGP supporte deux formats de certificats que l'on présente ci-dessous.

8. Pretty Good Privacy [44]. Une implémentation est disponible à : `https://www.gnupg.org/`.

— Le premier format est le format X.509 déjà présenté dans la section 3.4. L'introduction du champ «Extensions» dans l'actuelle et la troisième version de ce format rend les certificats X.509 suffisamment flexibles pour contenir des informations relatives aux exigences d'une organisation.

— Le second format est propre à PGP et son unique particularité est qu'un certificat peut être signé par plusieurs entités y compris une CA et le propriétaire lui-même. La signature par le propriétaire permet d'assurer la non-répudiation. Les informations contenues dans un tel format de certificat sont les suivantes.

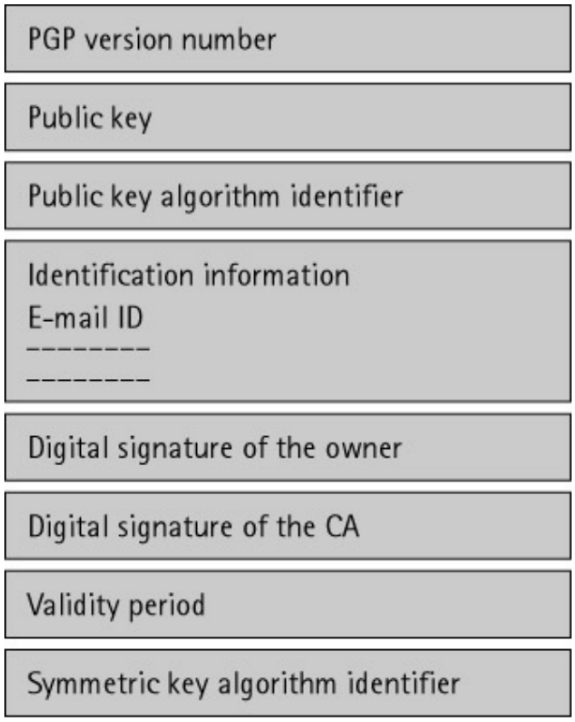


Figure 3.14 – ***Format de certificat OpenPGP.*** *(Source : [26])*

PGP Version Number. La version de PGP utilisée pour créer la paire de clefs.

Public Key. La clef publique du propriétaire se trouve dans ce champ.

Public Algorithm Identifier. L'identifiant de l'algorithme utilisé avec la clef publique pour assurer le chiffrement. Par exemple, le chiffrement RSA.

Identity Information. Ce champ contient des informations comme le nom du propriétaire, son adresse électronique ou encore sa photo.

Digital Signature of the Owner. Le résultat de la signature par le propriétaire est placé dans ce champ.

Digital Information of the CA. Ce champ contient la signature numérique de la CA qui aurait fourni le certificat.

Validity Period. Une paire de dates indiquant l'intervalle de temps pendant lequel le certificat peut être utilisé.

Symmetric Key Algorithm Identifier. Contient l'identifiant de l'algorithme utilisé avec la clef secrète pour le chiffrement. Par exemple, le chiffrement AES.

Le logiciel PGP utilise donc une clef symétrique et une paire de clefs (dont l'une est privée et l'autre est publique) pour fournir des services cryptographiques. La clef symétrique est valide uniquement pour une session et ne peut être réutilisée. La clef publique est dans le certificat qui se trouve sur un serveur de certificats PGP. Tout utilisateur peut se procurer de ce certificat et utiliser sa clef publique.

Signalons que depuis 2010, Symantec détient PGP et n'offre de version gratuite du logiciel qu'en cas de révision du code source par les pairs. Il existe toutefois d'autres implémentations de la norme OpenPGP telles que GPG [9] qui sont accessibles à tous.

3.5.2 Porte-clefs PGP

En PGP, on utilise les clefs pour assurer l'authenticité et la confidentialité des données. L'authentification passe par la signature d'un condensat du message. La confidentialité est assurée par le chiffrement du message par une clef de session (symétrique) et le chiffrement de cette clef de session par la clef publique du destinataire. Les clefs asymétriques sont stockées chez un utilisateur dans ce que l'on nomme des *porte-clefs*. Chaque utilisateur de PGP dispose de deux types de porte-clefs : l'un public et l'autre privé.

— Le porte-clefs privé contient les paires de clefs asymétriques de l'utilisateur lui-même. Les clefs privées ne devraient y être stockées que sous forme chiffrée (par exemple à l'aide d'un chiffrement symétrique dont la clef secrète pourrait être le condensat par une fonction de hachage d'une phrase secrète).

— Le porte-clefs public est composé des clefs publiques d'autres utilisateurs de PGP. Ces clefs sont soit envoyées par courrier électronique par les utilisateurs eux-mêmes, soit téléchargées sur un serveur de certificats PGP. Chaque clef ajoutée à un porte-clefs doit être signée par le propriétaire de ce porte-clefs.

Le porte-clefs public de chaque utilisateur comprend également des informations qui indiquent la validité d'une clef particulière ainsi que le niveau de la confiance qui peut être placée en cette clef de sorte que son propriétaire puisse valider d'autres clefs. À partir de ces informations, on peut assigner trois niveaux de confiance à une clef PGP.

«Complete Trust». Affecter un tel niveau à un utilisateur signifie qu'on lui fait *totalement* confiance pour signer et valider les clefs publiques d'autres utilisateurs.

«Marginal Trust». L'assignation de ce niveau de confiance à un utilisateur indique qu'on ne lui fait que *partiellement* confiance pour valider d'autres clefs.

«No Trust». Cette valeur signifie qu'on ne lui fait *aucune* confiance, ne serait-ce que pour valider les clefs publiques d'autres utilisateurs.

9. GNU Privacy Guard, https://gnupg.org.

En fonction du niveau de confiance que l'on accorde à une clef, on distingue trois types de validité : *«Valid»* si au moins une clef de niveau «Complete Trust» ou deux clefs de niveau «Marginal Trust» l'ont signée ; *«Marginally valid»* si elle fut signée par au moins deux clefs de niveau «Marginal Trust» ; *«Invalid»* sinon (et les clefs signées par cette dernière sont ignorées).

3.6 D'autres normes de PKI

Dans la section 3.4 et la section 3.5, on a présenté les normes PKIX et OpenPGP qui sont actuellement les plus répandues. Mais puisqu'elles ne convenaient pas à tous, certaines organisations mirent en place d'autres PKI dont on énumère quelques unes ci-dessous.

1. ISIS est une architecture de PKI mise en place en Allemagne, car la PKIX ne répondait pas aux exigences de la loi allemande sur les signatures.
2. La PKIX étant une norme complexe du point de vue de certains experts, en 1996, R. Rivest et L. Lampson ont proposé une architecture PKI nommée SDSI [10](ou «Simple Distributed Security Infrastructure») dont le but était d'éviter un système hiérarchique de nomenclature et fonctionner plutôt avec des noms locaux (et non globaux comme c'est le cas avec la PKIX). Parallèlement à celle-ci, un groupe de travail fut formé par l'IETF pour concevoir une norme en concurrence avec la PKIX et qui pourrait corriger quelques complexités inhérentes à cette dernière. Elle fut nommée SPKI (ou «Simple Public Key Infrastructure») [127] et son format était compatible avec PKIX mais pas avec OpenPGP. En 1996, les groupes de travail de SPKI et SDSI ont fusionné en incorporant les deux idées pour donner naissance au SDSI/SPKI 2.0 [11].
3. Le groupe de travail du W3C a publié une suite de normes de chiffrement et de signature de documents XML qui ont une spécification de PKI compagnon appelée XKMS («XML Key Management Specification») [12]. Cette dernière peut être utilisée pour enregistrer, localiser et valider des clefs pouvant être certifiées par une CA externe d'une PKIX, un signataire de clef PGP, de clef SPKI ou de l'infrastructure XKMS elle-même.

3.7 Fonctionnement du protocole TLS

L'apparition du «World Wide Web» au début des années 1990 ouvra la possibilité à des achats de manière électronique. Cependant, certains émettaient des réserves concernant la transmission d'informations sensibles comme des numéros de cartes de crédit. Pour remédier à cela, la solution était d'utiliser des techniques cryptographiques. Mais, il n'y avait aucun consensus ni sur les techniques à utiliser ni sur les couches du modèle TCP/IP auxquelles les appliquer. En général, il y a plusieurs possibilités d'utiliser des techniques cryptographiques sur diverses couches du TCP/IP. Mais en pratique, l'une semblait appropriée et fut exploitée

10. https://people.csail.mit.edu/rivest/sdsi10.html.
11. https://people.csail.mit.edu/rivest/pubs/RL96.slides-maryland.pdf.
12. https://www.w3.org/TR/xkms2/.

par «Netscape Communications». Ils l'ont nommée «Secure Sockets Layer» ou SSL et elle correspondait à une sorte de couche intermédiaire entre la couche transport et la couche application du modèle TCP/IP. Son rôle était d'établir des connexions sécurisées et de transmettre des données à travers ces connexions.

Puisque son fonctionnement est très lié à la couche transport, le protocole SSL fut techniquement ajouté à la classe des protocoles de sécurité de cette couche. Sa première version (SSL 1.0) vit le jour en 1994, mais elle était utilisée uniquement à l'interne puisqu'elle avait plusieurs failles de sécurité dont, par exemple, l'incapacité d'assurer l'intégrité des données. En 1995, le navigateur web «Netscape Navigator» est apparu avec une implémentation de la nouvelle version SSL 2.0 [41]. En 1996, il y eut quelques améliorations (notamment suggérées par PCT [13]) aboutissant à SSL 3.0 [45] par une équipe sous la supervision de Taher Elgamal. Notons cependant que l'IETF déprécia SSL 3.0 ainsi que tout le protocole SSL en 2015 [16].

Après la publication de SSL 3.0 et de PCT, il y eut beaucoup de confusion. Netscape et une large partie de l'Internet utilisaient SSL 3.0, alors que Microsoft utilisait PCT. En plus, d'une part, Microsoft devait supporter le protocole SSL pour des raisons d'interopérabilité. D'autre part, l'entreprise avait une autre proposition STLP («Secure Transport Layer Protocol») qui était une sorte de variante de SSL 3.0 fournissant des fonctionnalités supplémentaires qu'elle jugeait essentielles telles que la prise en charge du protocole UDP.

Afin de résoudre ces problèmes et normaliser un protocole TLS unifié, l'IETF forma un groupe de travail qui publia une première version TLS 1.0 [6] en 1999. Il s'en est ensuivi trois autres versions : TLS 1.1 [33] en 2006, TLS 1.2 [98] en 2008 et TLS 1.3 [97] en 2018. Cependant, notons qu'en mars 2021, le TLS 1.0 et le TLS 1.1 furent dépréciés [81].

En plus de la sécurisation d'achats de manière électronique (se faisant sur des sites internet via le protocole HTTPS [96]), le protocole TLS permet de façon générale de sécuriser la transmission de tout type de données électroniques. Par exemple, il sécurise l'envoi et la réception des courriers électroniques dans le protocole SMTPS [77]. Dans le cas du HTTPS, l'établissement d'une connexion chiffrée entre un client et un serveur avec le protocole TLS suit un processus de prise de contact standard que l'on peut résumer aux étapes suivantes.

Étape 1. Le client, à travers son navigateur web, contacte un serveur web qui est supposé être hébergé à une adresse sécurisée.

Étape 2. Le serveur répond à cette demande en envoyant son certificat électronique au navigateur web du client.

Étape 3. Le client vérifie que le certificat reçu est valide. On rappelle que les certificats sont délivrés par des autorités de certification connues telles que Verisign [14].

Étape 4. Après la validation du certificat du serveur, le client génère une clef de session aléatoire qui sera utilisée pour chiffrer toutes les communications avec le serveur. Notons que cette clef (sécrète) ne doit être utilisée qu'une fois.

Étape 5. Le client chiffre la clef de session avec la clef publique du serveur et l'algorithme asymétrique qui lui est associé. Ce message chiffré et le certificat électronique du client

13. Le «Private Communications Technology» était un protocole développé par Microsoft dans les années 1990 pour corriger les failles de sécurité du SSL 2.0 et pour forcer Netscape à confier le contrôle du protocole SSL alors propriétaire à un organisme de normalisation ouvert [108].

14. VeriSign, Inc. https://www.verisign.com/.

seront transmis au serveur web. Le chiffrement par la clef publique du serveur évite qu'une autre entité ne puisse intercepter la clef de session.

Entre les étapes **3** et **4**, le serveur pourrait optionnellement choisir de confirmer l'identité d'un utilisateur dans des situations plus délicates comme lors d'un envoi d'informations financières confidentielles d'une banque à un client. Cela empêche, par exemple, des attaques de type MitM [24].

3.8 Exemples de défaillances liées aux certificats

3.8.1 Cas de Comodo

En 2011, le groupe Comodo [15], une entreprise de cybersécurité, fut attaquée par un pirate informatique. Selon des experts de sécurité, ce serait une conséquence de la prolifération des CA. Des concepteurs de navigateurs tels que Microsoft, Mozilla, Google et Apple autorisent un grand nombre d'entités à travers le monde (sociétés privées et gouvernementales) de créer des certificats et donc à être des CA. Plusieurs CA privées, à leur tour, travaillent avec des revendeurs et délèguent d'autres sociétés inconnues pour fournir des certificats. Les chaînes de confiance qui en résultent incluent plusieurs centaines d'acteurs dont n'importe lequel peut en fait être un maillon faible. À l'époque, l'EFF [16], un groupe de protection de libertés civiles, en explorant l'Internet recensa près de 700 CA. D'autres experts disaient que leur analyse en avait certainement raté plusieurs et donc qu'il y en aurait beaucoup plus.

Lors de l'attaque [17], le pirate prétendait être un activiste solitaire et patriote iranien, et disait venger l'attaque du ver informatique StuxNet qui visait les installations nucléaires iraniennes un an auparavant. Il avait infiltré plusieurs partenaires de Comodo et les avait utilisés pour menacer la sécurité de plusieurs sites web de renom. Plus précisément, le pirate a infiltré l'ordinateur d'un revendeur italien et a utilisé son accès aux systèmes de Comodo pour créer automatiquement des certificats pour des sites web exploités par Google, Yahoo, Microsoft, Skype et Mozilla. Avec les certificats, le pirate pouvait configurer des serveurs qui semblent fonctionner pour ces sites et essayer de voir les courriers électroniques déchiffrés de plusieurs millions de personnes. Puisqu'une CA peut techniquement créer un certificat au nom d'un site web sans l'autorisation de ce dernier, le pirate pouvait également créer des certificats à volonté au nom de n'importe quel site web.

La révocation qui était l'outil essentiel de Comodo et les concepteurs de navigateurs fut inefficace. En effet, après la découverte de l'incident, Google, Mozilla et Microsoft se précipitèrent de produire des correctifs afin que leurs navigateurs puissent détecter et rejeter les faux certificats. Mais cette solution nécessitait que plusieurs millions d'utilisateurs mettent à jour leurs navigateurs, ce qui avait peu de chance de fonctionner puisque beaucoup font rarement des mises à jour. Dans ce cas, les concepteurs de navigateurs ne pouvaient pas faire plus que de bannir Comodo. Mais faire cela à un grand acteur comme Comodo aurait pu

15. Comodo Group, https://comodosslstore.com.
16. Electronic Frontier Foundation, https://www.eff.org/.
17. An Attack Sheds Light on Internet Security Holes, https://www.nytimes.com/2011/04/07/technology/07hack.html.

avoir des conséquences considérables sur l'Internet. En effet, Comodo contrôlait au moins 95 100 certificats à l'époque, donc des millions d'internautes auraient rencontré des avertissements inquiétants lors de la visite de certains sites web, provoquant ainsi une cascade de problèmes pour les utilisateurs et les propriétaires de ces sites. Pour éviter cela, la plupart des concepteurs de navigateurs autorisaient les utilisateurs d'accéder à des sites web alternatifs ; cette faiblesse pouvait évidemment être exploitée par d'autres pirates également.

À la suite de cette attaque, Mozilla, Microsoft et Google dirent qu'ils allaient travailler ensemble et avec les CA et la communauté de sécurité sur des améliorations du système. Une approche proposée par des ingénieurs de Comodo et Google permettraient aux sites web de spécifier les CA qui délivrent des certificats pour leurs sites.

3.8.2 Cas de DigiNotar

Encore en 2011, la CA néerlandaise de DigiNotar (appartenant à «Vasco Data Security» [18]) détecta une intrusion [19] dans ses systèmes. La société a immédiatement révoqué un certain nombre de faux certificats qui avaient été créés juste après l'intrusion. L'attaque consistait à faire croire aux utilisateurs qu'ils accédaient aux sites web tels que Google, alors qu'en réalité, quelqu'un d'autre pouvait avoir surveillé les communications. On pense que des centaines de faux certificats avaient été générés lors de ce piratage (au moins 500 selon certains). Parmi les domaines touchés apparaissaient Google, Facebook, Twitter et Skype.

Au moment de l'attaque, une partie importante des certificats de la société néerlandaise (18.7%) allait mystérieusement à des utilisateurs en Iran, pendant que 76.5% étaient aux Pays-Bas et 4.8% ailleurs. Cependant, après la révocation des certificats créés à la suite de l'intrusion, l'activité iranienne a chuté à 3.6%.

3.8.3 Cas de Windows IIS

En septembre 2021, des experts de sécurité ont découvert une attaque généralisée [20] ciblant des utilisateurs Windows. L'attaque utilisait des logiciels malveillants pour lancer une fausse alerte de certificats expirés sur des sites web invitant les utilisateurs à télécharger une mise à jour qui contenait en fait un logiciel malveillant (TVRAT) et qui permettait aux pirates d'accéder à distance aux ordinateurs infectés. L'attaque visait le serveur «Windows Internet Information Services» qui est le serveur web de Microsoft installé sur toutes les versions de Windows depuis Windows 2000, Windows XP et Windows Server 2003.

Cette attaque met en lumière l'importance de la visibilité sur la gestion du cycle de vie des certificats pour mieux comprendre les risques potentiels comme celui-ci. Elle nous apprend davantage à ne pas sous-estimer l'importance de la gestion des PKI. Concernant cette attaque, un système d'analyse et de détection aurait pu déterminer si un certificat a été installé sur un serveur et prévenir donc l'attaque. Un autre point est de considérer les utilisateurs comme une partie de la solution ; cela passe par une éducation sur une variété d'importantes

18. Elle fut renommée OneSpan, https://www.onespan.com/.
19. Fake DigiNotar web certificate risk to Iranians, https://www.bbc.com/news/technology-14789763.
20. https://www.keyfactor.com/blog/how-malware-was-used-to-create-fake-expired-certificate-alerts/.

mesures de sécurité y compris le fait qu'ils ne doivent pas accepter et installer un certificat qu'ils ne connaissent pas.

3.8.4 Cas de certificats expirés

Les certificats expirés provoquent des pannes imprévues du système et peuvent même ouvrir des portes à des pirates informatiques pour effectuer des attaques.

— En 2013, Microsoft Azure, le principal fournisseur de cloud de l'époque, a connu une panne mondiale pendant plusieurs heures en raison d'un certificat expiré [21].

— En 2014, à cause d'un certificat expiré, des dizaines de milliers de terminaux utilisés pour des paiements par carte de crédit aux États-Unis ont cessé de fonctionner [22].

— En début de novembre 2021, un certificat expiré le 31 octobre dans le Windows 11 provoqua des dysfonctionnements de certains logiciels. Les utilisateurs n'arrivaient pas à ouvrir l'outil de capture, le clavier tactile ou encore le panneau émoji. Ils ont été mis au courant des problèmes en leur proposant des outils alternatifs et une mise à jour a été effectuée quelques jours plus tard [23].

Une session SSL/TLS qui utilise un certificat expiré ne devrait être considérée comme fiable. Accepter un tel certificat rend un utilisateur vulnérable à des attaques de type MitM. Pour remédier à cela, tous les certificats doivent être identifiés et supprimés des serveurs.

3.9 Conclusion

Dans ce chapitre, on a donné des éléments de rappel sur la façon dont on peut associer une identité à une clef publique de manière fiable. Cela incluait évidemment les PKI, et donc les certificats électroniques. On a présenté les deux principales normes de PKI (c'est-à-dire PKIX et OpenPGP) ainsi que les formats de certificats qu'elles proposent.

On a également parlé du protocole TLS qui est basé sur les certificats X.509 et largement utilisé sur l'Internet actuellement pour sécuriser la transmission des données. Ces rappels nous permettront de bien appréhender le chapitre 5 dont l'objectif principal consistera à analyser plusieurs millions de certificats SSL/TLS X.509.

21. https://azure.microsoft.com/en-us/blog/windows-azure-service-disruption-from-expired-certificate/.
22. https://www.welivesecurity.com/2014/12/15/hypercom-payment-terminals-bricked-malware/.
23. https://www.zdnet.com/article/microsoft-just-fixed-the-windows-11-problem-caused-by-an-expired-digital-certificate/.

Chapitre 4

Le chiffrement RSA : génération des paramètres et recommandations

Sommaire

L'idée de la cryptographie asymétrique est souvent attribuée à W. Diffie et M. Hellman qui ont présenté pour la première fois un algorithme asymétrique en 1976 [34]. Cet algorithme, appelé *algorithme d'échanges de clefs Diffie-Hellman*, permettait à deux entités de convenir sur la valeur d'une clef sécrète. Mais il ne pouvait être utilisé ni pour chiffrer ni pour déchiffrer. Deux ans plus tard, R. **R**ivest, A. **S**hamir et L. **A**dleman inventèrent le chiffrement *RSA* [101] dont la sécurité reposerait sur la difficulté de factoriser de grands nombres entiers. Jusqu'à présent, ce chiffrement est le chiffrement asymétrique le plus utilisé dans les services web bien que d'autres chiffrements asymétriques aient vu le jour depuis 1978. Parmi ceux-ci, on a le chiffrement ElGamal [42] et variantes comme la cryptographie basée sur les courbes elliptiques (ou ECC pour «Elliptic Curve Cryptography») [64, 80] dont la sécurité repose sur le problème du logarithme discret.

Le but de ce chapitre sera de donner quelques éléments de rappel sur le chiffrement RSA. On discutera notamment les points qui sont en rapport avec les analyses que l'on fera dans le chapitre 5 et le chapitre 6, à savoir principalement la vérification de la primalité d'un entier, les principales méthodes de factorisation d'entiers ainsi que les méthodes de génération des paramètres d'une clef RSA qui sont recommandées par certaines normes connues.

Les informations contenues dans la section 4.2 et la section 4.4 s'inspirent de l'ouvrage «A Course in Computational Algebraic Number Theory» [27] de Henri Cohen. Cependant, les résultats sur la distribution des nombres premiers dans la section 4.3 proviennent de «The Distribution of Prime Numbers» [67] de Dimitris Koukoulopoulos. Concernant les nombres premiers, le lecteur pourra également trouver plus de détails dans les ouvrages suivants :

— «Opera de Cribro» [46] de J. Friedlander et H. Iwaniec ;

— «The Prime Numbers and Their Distribution» [114] de G. Tenenbaum et M. Mendès France ;

— «The Great Prime Number Race» [88] de R. Plymen.

4.1 Paramètres du chiffrement RSA

Le chiffrement RSA étant asymétrique, il n'est pas inconditionnellement sûr. Cependant, cela ne constitue pas un vrai problème de sécurité, car sa conception repose sur une *fonction à sens unique avec trappe* qui est une fonction pour laquelle le calcul d'une image directe est facile pendant que celui d'une image réciproque est difficile sous réserve de détenir une information sécrète (appelée *trappe*).

Par exemple, soit n le produit de deux nombres premiers p et q distincts relativement grands, et soit e un entier naturel étranger à $\phi(n)$ où ϕ est l'indicatrice d'Euler [1]. La fonction E de $\mathbb{Z}/n\mathbb{Z}$ dans $\mathbb{Z}/n\mathbb{Z}$ définie par : $E(x) = x^e$ est à sens unique avec trappe. En effet,

— le calcul d'une image directe par E se fait efficacement par exponentiation modulaire ;

— si l'on n'a que n et e, on ne pourra pas calculer n'importe quelle image réciproque en temps polynomial par E ;

— mais si l'on connaît p ou q, on peut trouver un entier d en temps polynomial tel que la fonction D de $\mathbb{Z}/n\mathbb{Z}$ dans $\mathbb{Z}/n\mathbb{Z}$ définie par : $D(x) = x^d$ est la réciproque de E.

1. $\phi(n)$ est le nombre d'entiers compris entre 1 et n (inclus) et étrangers à n.

En réalité, E est *supposée* être à sens unique, car il n'existe pas actuellement de démarche mathématique permettant de montrer qu'une fonction est vraiment à sens unique. On verra ci-dessous que E correspond à la fonction de chiffrement du cryptosystème RSA, et que sa trappe n'est rien d'autre que la clef privée du destinateur.

Le chiffrement RSA est utilisé pour assurer la confidentialité et garantir l'authenticité des données numériques telles que des courriers électroniques. Il est également utilisé dans la télégestion des sessions, dans les serveurs et navigateurs web pour sécuriser la transmission des données ainsi que dans des systèmes commerciaux comme des cartes de crédit. La représentation des clefs publique et privée utilisées dans ce chiffrement peut changer en fonction des normes et chaque norme peut proposer plusieurs façons de les représenter. Dans ce document, on prendra la représentation qui est commune aux normes PKCS#1 [82] et IEEE-P1363 [3]. La clef publique est le couple d'entiers positifs :

$$(n; e)$$

où n, appelé *module*, est le produit de deux nombres premiers p et q distincts, relativement grands et généralement de même taille en bits. L'entier e, appelé *exposant public*, doit être étranger à $\lambda(n)$, c'est-à-dire $\text{PGCD}(e, \lambda(n)) = 1$, où $\lambda(n) = \text{PPCM}(p-1, q-1)$. Quant à la clef privée, elle est représentée par le couple d'entiers positifs :

$$(n; d)$$

où d, appelé *exposant privé*, est l'inverse de e modulo $\lambda(n)$, c'est-à-dire : $e \times d \equiv 1 \bmod \lambda(n)$.

Notons également que selon la norme PKCS#1, il est possible d'avoir un module RSA qui est le produit de plus de deux nombres premiers impairs distincts [82, sec.3.1] ; on parlera dans ce cas de *module RSA multi-premiers*. Il est possible de générer de tels modules avec des versions récentes de bibliothèques cryptographiques connues telles que OpenSSL [116] comme on le verra dans la section 6.3 du chapitre 6.

Les primitives de chiffrement et de déchiffrement opèrent de la façon suivante. Lorsqu'on veut chiffrer une donnée représentée par un entier m (où $0 \leqslant m < n$), on calcule l'entier :

$$c = m^e \bmod n$$

et lorsqu'on veut déchiffrer ce dernier, on détermine l'entier :

$$c^d \bmod n$$

On peut démontrer que ces deux opérations sont bien inverses l'une de l'autre [101]. En plus, on remarque que $c = E(m)$ et que $c^d \bmod n = D(c)$ où E et D sont les fonctions données en exemple au début de cette section.

Déchiffrer c sans connaître d peut parfois se ramener au calcul de la racine e-ième de c modulo n, ce qui est un problème aussi difficile que factoriser n [19, 22]. Par contre, la factorisation de n permet de retrouver d, donc de déchiffrer c. De façon réciproque, si l'on connaît d, on peut factoriser n en temps polynomial [30, 101, 94].

L'article «Twenty Years of Attacks on the RSA Cryptosystem» [19] publié en 1999 par Dan Boneh donne un récapitulatif des attaques découvertes contre le chiffrement RSA dans les

vingt ans ayant suivi son invention. Dans cet article, on peut aussi trouver d'autres techniques reposant, par exemple, sur l'exploitation des temps d'exécution («Timing attacks») ou la consommation d'énergie («Power analysis»). Ces attaques permettent, dans le cas de mauvaises implémentations, de déchiffrer c sans nécessairement factoriser n.

4.2 Tests de non-primalité et tests de primalité

La principale tâche difficile dans la génération des paramètres du chiffrement RSA est le choix efficace de nombres premiers relativement grands. D'une part, ce choix doit se faire de façon aléatoire dans le sens où la probabilité qu'un nombre premier particulier soit choisi doit être la même que celle d'un autre nombre premier de même taille. D'autre part, ces nombres devraient également satisfaire à des contraintes supplémentaires afin d'empêcher des attaques spécialisées. On verra quelques unes de ces contraintes lors de la présentation des recommandations de certaines normes dans la section 4.6.

Dans cette section, n désignera un entier positif arbitraire (et non un module RSA). En pratique, un nombre premier utilisé pour le chiffrement RSA est généralement choisi de la manière suivante :

1. générer aléatoirement un entier positif impair n d'une certaine taille ;
2. tester la primalité de n ;
3. si n est composé, alors retourner à la première étape.

Une légère modification de ce processus pourrait le rendre encore plus efficace. Il s'agit de faire un autre choix à la dernière étape : au lieu de retourner systématiquement à la première étape lorsque n est composé, on pourrait chercher un nombre premier dans la suite $n+2$, $n+4$, $n+6$, $n+8, \cdots$. Mais cette technique, connue sous le nom de «next prime» (ou *premier suivant*), pourrait créer un biais *à priori* indésirable, car les nombres premiers n'auront pas la même probabilité d'être choisis.

À la deuxième étape du processus, la primalité est testée soit par un test qui prouve que n est un nombre premier (on parle de *test de primalité*), soit par un test qui établit que n est un nombre probablement premier (on parle de *test de primalité probabiliste*). Dans le premier cas, n est dit *nombre premier prouvable*, et dans le second cas, n est appelé *nombre premier probable*. Les tests de primalité probabilistes sont exacts lorsqu'ils déclarent que n est composé, mais ne fournissent aucune preuve mathématique que n est premier lorsqu'ils le déclare probablement premier. C'est pour cette raison qu'ils sont parfois appelés *tests de non-primalité* plutôt que tests de primalité probabilistes. Notons qu'ils sont également plus utilisés que les tests de primalité bien que ceux-ci concluent avec certitude qu'un nombre est premier : en effet, ces derniers requièrent généralement des ressources considérables. Toutefois, une combinaison des deux types de test peut être utilisée et dans laquelle le test de non-primalité précède. Notons aussi qu'un test de chacun des deux types peut faire usage de nombres aléatoires dans son fonctionnement interne ; si c'est le cas, le test est dit *randomisé*, sinon il est qualifié de *déterministe* (car le résultat est reproductible).

Enfin, on doit signaler qu'il existe des techniques qui construisent n spécialement de sorte qu'il puisse être établi par un raisonnement mathématique qu'il est réellement premier.

Ces techniques sont appelées *techniques de génération constructives de nombres premiers*. L'une d'entre elles, connue sous le nom d'*algorithme de Shawe-Taylor*, est recommandée par la norme X9.31 [8, app.B.3] de l'ANSI et par la norme FIPS 186-4 [59, app.C.10] du NIST.

Dans les sous-section 4.2.2 et sous-section 4.2.3, on présentera les tests de non-primalité les plus utilisés actuellement, c'est-à-dire le test de Solovay-Strassen et celui de Miller-Rabin. Pour des raisons historiques, on donnera aussi un aperçu du test de Fermat dans la sous-section 4.2.1. Il y aura également dans la sous-section 4.2.4, une brève présentation des tests de primalité les plus répandus actuellement.

4.2.1 Test de Fermat

Avant d'employer des algorithmes plus efficaces lors de la vérification de la primalité d'un entier positif n, on commence en général par la détection d'éventuels petits facteurs premiers en utilisant, par exemple, une méthode naïve comme celle des *divisions successives*. Cette méthode consiste à diviser n successivement par des nombres premiers inférieurs à une certaine borne pour vérifier qu'aucun d'eux n'est l'un de ses facteurs. La bibliothèque cryptographique OpenSSL [116] (dont on étudiera la génération de clefs RSA dans le chapitre 6) effectue des divisions successives par les 2048 premiers nombres premiers.

Certains utilisent aussi le *test de Fermat* qui découle du petit théorème de Fermat. Ce théorème établit que : si p est un nombre premier et si a est un entier non divisible par p, alors : $a^{p-1} \equiv 1 \bmod p$. Donc, pour un entier positif n dont on veut vérifier la primalité, trouver un entier a (avec $1 \leqslant a \leqslant n-1$) tel que $a^{n-1} \not\equiv 1 \bmod n$ suffira à montrer que n est composé. Par contre, si a vérifie $a^{n-1} \equiv 1 \bmod n$, cela ne ferait qu'apparaître n comme un nombre premier potentiel dans le sens où il satisfait au petit théorème de Fermat en base a. Cela se traduit par l'Algorithme 4.1.

Algorithme 4.1 : Test de primalité de Fermat

Entrées : un entier impair $n \geqslant 3$ et un nombre d'itérations $t \geqslant 1$.
Sortie : une réponse «n est premier» ou «n est composé».

```
1 début
2     pour i ← 1 à t faire
3         Choisir un entier aléatoire a tel que 2 ⩽ a ⩽ n − 2.
4         r ← a^{n−1} mod n.
5         si r ≠ 1 alors retourner «n est composé».
6     fin
7     retourner «n est premier».
8 fin
```

Si l'Algorithme 4.1 renvoie «n est composé», alors n est certainement composé. Mais s'il renvoie «n est premier», aucune preuve de primalité de n n'est fournie. Contrairement aux deux autres tests (que l'on présentera ci-dessous), cet algorithme n'est pas suffisant, ne serait-ce que pour donner une réponse probable. Cependant, il est utilisé pour les raisons suivantes.

— La quantité $a^{n-1} \bmod n$ peut être calculée efficacement en utilisant l'exponentiation modulaire [103, p.244].

— Bien qu'il ne donne qu'une condition nécessaire de primalité, les exceptions sont rares.

Mais ces exceptions sont infinies. En effet, il existe des entiers composés n appelés *nombres de Carmichael* (dont le plus petit est : $n = 561 = 3 \times 11 \times 17$) pour lesquels $a^{n-1} \equiv 1 \bmod n$ pour tout a étranger à n. En 1994, Alford, Granville et Pomerance [5] ont prouvé qu'il y a une infinité de nombres de Carmichael et même qu'à partir d'un entier positif x, leur nombre est au moins x^c pour un c proche de 0.1. Dans ces conditions, si tous les facteurs premiers de n sont relativement grands, il est fort probable que le test de Fermat déclare que n est premier même si le nombre d'itérations t est élevé. Cette déficience dans le test de Fermat est corrigée dans le test de Solovay-Strassen et le test de Miller-Rabin qui reposent tous deux sur des critères plus forts que le petit théorème de Fermat.

4.2.2 Test de Solovay-Strassen

Dès lors que n passe un test élémentaire comme celui des divisions successives ou celui de Fermat, la vérification de sa primalité se poursuit avec un test plus sophistiqué. Un résultat découlant du petit théorème de Fermat conduisit au test de Solovay-Strassen [109, 110]. Il s'agit du *critère d'Euler* qui énonce que si p est un nombre premier impair, $a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \bmod p$ pour tout entier a non divisible par p; où $\left(\frac{a}{p}\right)$ est le symbole de Legendre. Ce critère est plus fort que le petit théorème de Fermat et permet d'écarter l'entier 561 par exemple. Mais il n'élimine pas tous les contre-exemples, car le troisième nombre de Carmichael $n = 1729$ vérifie $a^{\frac{n-1}{2}} \equiv 1 \bmod n$ pour tout entier a étranger à n. Néanmoins, le critère d'Euler est utilisé comme le socle du test de Solovay-Strassen pour la raison suivante : si n est composé, alors au plus $\frac{\phi(n)}{2}$ de tous les entiers a (où $1 \leqslant a \leqslant n-1$) vérifient $a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \bmod n$.

Algorithme 4.2 : Test de primalité probabiliste de Solovay-Strassen

Entrées : un entier impair $n \geqslant 3$ et un nombre d'itérations $t \geqslant 1$.
Sortie : une réponse «n est premier» ou «n est composé».

```
1 début
2     pour i ← 1 à t faire
3         Choisir un entier aléatoire a tel que 2 ⩽ a ⩽ n − 2.
4         r ← a^((n−1)/2) mod n.
5         si r ≠ 1 et r ≠ n − 1 alors retourner «n est composé».
6         si r ≢ (a/n) mod n alors retourner «n est composé».
7     fin
8     retourner «n est premier».
9 fin
```

Si $\text{PGCD}(a, n) = d$, alors d est un diviseur de $r = a^{\frac{n-1}{2}} \bmod n$. Donc tester que $r \neq 1$ à la ligne 5 de l'Algorithme 4.2 élimine la nécessité de tester que $\text{PGCD}(a, n) \neq 1$. Si l'algorithme renvoie «n est composé», alors n est certainement composé, car les nombres premiers ne violent pas le critère d'Euler. Réciproquement, si n est réellement premier, l'Algorithme 4.2 le

déclarera toujours premier. D'autre part, si n est un entier composé, la probabilité que le test de Solovay-Strassen le déclare premier est inférieure à $\left(\frac{1}{2}\right)^t$. Cette quantité est connue sous le nom de *probabilité d'erreur*. Notons enfin que le test de Solovay-Strassen fut le premier test de son type et fut popularisé par l'avènement de la cryptographie asymétrique notamment le chiffrement RSA. Son utilisation est de plus en plus rare, car il existe une alternative (le test de Miller-Rabin) qui est encore plus efficace.

4.2.3 Test de Miller-Rabin

Il s'agit du test de primalité probabiliste le plus usité actuellement. Soit p un nombre premier impair et $p-1=2^s r$ où r est un entier impair ; pour tout entier a étranger à p, on a : soit $a^r \equiv 1 \bmod p$ ou $a^{2^j r} \equiv -1 \bmod p$ pour un entier j tel que $0 \leqslant j \leqslant s-1$. Ceci sert de base au test de Miller-Rabin [79, 94], car si n est composé, alors on peut montrer qu'au plus $\frac{1}{4}$ des nombres a (où $1 \leqslant a \leqslant n-1$) vérifient le résultat énoncé. D'où l'Algorithme 4.3.

Algorithme 4.3 : Test de primalité probabiliste de Miller-Rabin

Entrées : un entier impair $n \geqslant 3$ et un nombre d'itérations $t \geqslant 1$.
Sortie : une réponse «n est premier» ou «n est composé».

```
1  début
2      Écrire n − 1 = 2^s r avec r impair.
3      pour i ← 1 à t faire
4          Choisir un entier aléatoire a tel que 2 ⩽ a ⩽ n − 2.
5          y ← a^r mod n.
6          si y ≠ 1 et y ≠ n − 1 alors
7              j ← 1.
8              tant que j ⩽ s − 1 et y ≠ n − 1 faire
9                  y ← y^2 mod n.
10                 si y = 1 alors retourner «n est composé».
11                 j ← j + 1.
12             fin
13             si y ≠ n − 1 alors retourner «n est composé».
14         fin
15     fin
16     retourner «n est premier».
17 fin
```

La complexité de l'Algorithme 4.3 est la même que celle de l'exponentiation modulaire, à savoir $\mathcal{O}\left((\log n)^3\right)$. Notons cependant qu'il y a certains types d'algorithmes d'exponentiation modulaire [27, p.9–10] pour lesquels la complexité passera en $\mathcal{O}\left((\log n)^2\right)$. L'Algorithme 4.3 est du même type que l'Algorithme 4.2 dans le sens où si n est déclaré composé, alors il est sûrement composé, et s'il est réellement premier, il sera toujours déclaré premier. Si n est un entier composé impair, la probabilité que le test de Miller-Rabin le déclare premier est

inférieure à $\left(\frac{1}{4}\right)^t$. Notons qu'avec cette version de l'Algorithme 4.3, il sera difficile de prouver la primalité d'un entier donné. En effet, pour tout ensemble fini de valeurs de a, il est possible de construire des nombres composés qui passent le test de Miller-Rabin [7]. Il existe toutefois une variante de ce test due à Miller [79, 10] et qui, en supposant l'hypothèse de Riemann généralisée [67, chap.8], fournit un test de non-primalité et de primalité à la fois.

Le test de Miller-Rabin est recommandé par plusieurs normes dont la norme X9.31 [8, p.23] de l'ANSI, la norme IEEE-P1363 [3, app.A.15.1] et la norme FIPS 186-4 [59, app.C.3.1] du NIST. En conséquence, il est utilisé par la plupart des bibliothèques cryptographiques qui existent actuellement. On donne ci-dessous les principales raisons pour lesquelles il prit la place du test de Solovay-Strassen.

— Le test de Solovay-Strassen est plus coûteux en calculs et plus difficile à mettre en place à cause des symboles de Jacobi.

— La probabilité d'erreur du test de Solovay-Strassen est majorée par $\left(\frac{1}{2}\right)^t$, alors que celle du test de Miller-Rabin est majorée par $\left(\frac{1}{4}\right)^t$.

— Toute valeur de a pour laquelle le test de Miller-Rabin déclare que n est composé est aussi une valeur pour laquelle le test de Solovay-Strassen déclarera que n est composé.

4.2.4 Test de Lucas

Il existe d'autres tests de non-primalité moins utilisés que celui de Miller-Rabin. Il y a par exemple celui de Lucas qui est recommandé notamment par la norme X9.31 [8, app.B.3] et la norme FIPS 186-4 [59, app.C.3.3]. Il existe plusieurs versions de ce test, mais celle proposée dans la norme FIPS 186-4 est due à R. Baillie et S. S. Wagstaff [11]. Dans leur article, les deux auteurs ont aussi conçu un nouveau test qui combine ceux de Lucas, de Fermat et de Miller-Rabin. Il a été amélioré au cours de la même année par C. Pomerance, J. L. Selfridge et S. S. Wagstaff [91] et il fut nommé *test BPSW*. Sa complexité est en $\mathcal{O}\left((\log n)^3\right)$ et on l'utilise, par exemple, dans la fonction `ispseudoprime` de PARI/GP 2.12.0 [117].

La combinaison des tests de Miller-Rabin et de Lucas est également recommandée par la norme FIPS 186-4. En fait, cette norme propose deux scenarii pour tester la primalité : soit utiliser plusieurs itérations du test de Miller-Rabin, soit utiliser un nombre raisonnable d'itérations du test de Miller-Rabin suivi d'une itération du test de Lucas. La norme justifie l'utilisation du second scenario par le fait qu'on ne connaît actuellement aucun entier composé qui passe la combinaison des deux tests [59, app.F.3]. Le nombre d'itérations du test de Miller-Rabin dépend de plusieurs facteurs dont l'algorithme de chiffrement pour lequel le nombre premier sera utilisé, la probabilité d'erreur souhaitée et la taille de l'entier à tester. Sur la Figure 4.1, on donne, en fonction de la taille (en bits) de l'entier à tester, le nombre minimal d'itérations du test de Miller-Rabin pour une probabilité d'erreur inférieure à 2^{-100}. Notons que dans les normes qui seront présentées dans la section 4.6, la probabilité d'erreur est choisie parmi 2^{-128}, 2^{-112} et 2^{-80} bien qu'aucune preuve ne soit donnée pour justifier le choix de ces probabilités. Cette absence de preuve pourrait aussi expliquer pourquoi la

norme FIPS 186-4 recommande d'effectuer une itération du test de Lucas après des itérations du test de Miller-Rabin. Sur la Figure 4.1, on doit aussi signaler que la pente que l'on observe est due au choix des paramètres proposés notamment dans «Handbook of Applied Cryptography» [78, p.147–148].

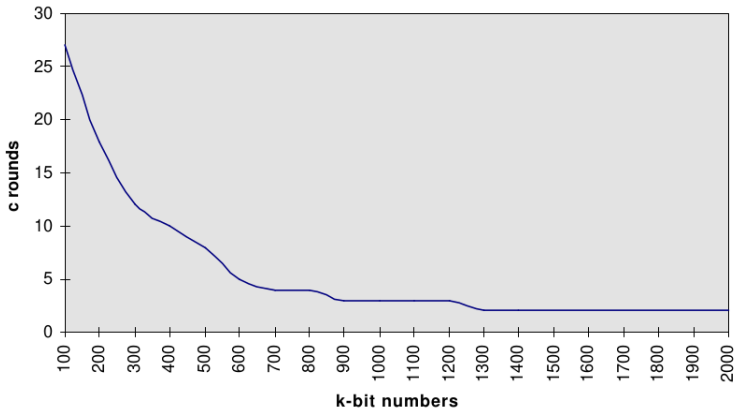


Figure 4.1 – ***Nombre d'itérations du test de Miller-Rabin.*** *(Source : [8])*

4.2.5 Tests de primalité

Maintenant, supposons que n n'a pas été déclaré composé par un test de non-primalité comme celui de Miller-Rabin. Alors on est presque certain qu'il est premier. Si l'on veut être sûr qu'il est premier, on utilise un test de primalité. Il existe plusieurs tests de ce type, mais on présentera de façon brève uniquement ceux qui sont actuellement les plus utilisés.

— Le test «$n-1$» de Pocklington-Lehmer [27, p.16–17] repose sur une sorte de réciproque du petit théorème de Fermat. Il devient impraticable lorsque la taille de n dépasse 100 chiffres décimaux.

— Le test APR a été mis en place en 1980 par Adleman, Pomerance et Rumely. L'algorithme fut simplifié et amélioré en 1981 par H. W. Lenstra et H. Cohen, et est devenu APRCL. Avec cet algorithme, il est possible de tester la primalité des nombres ayant moins de 1000 chiffres décimaux en temps raisonnable. La complexité est en $\mathcal{O}\left((\log n)^{c\log\log\log n}\right)$ avec c une constante. Cette complexité est presque polynomiale puisqu'à toutes fins utiles, la quantité $\log\log\log n$ se comporte comme une constante [27, chap.9].

— En 1986, Goldwasser et Kilian ont inventé un algorithme de test de primalité basé sur les courbes elliptiques. Cet algorithme a été considérablement modifié par Atkin qui l'implémenta ensuite avec Morain et l'ont appelé ECPP pour «Elliptic Curve Primality Proving» [9]. Il est utilisable pour les nombres ayant plus de 1000 chiffres décimaux et la complexité est en $\mathcal{O}\left((\log n)^{6+\varepsilon}\right)$ [69, p.711] avec $0<\varepsilon<1$. Cette complexité peut être ramenée à $\mathcal{O}\left((\log n)^{5+\varepsilon}\right)$ lorsqu'une technique de multiplication rapide est utilisée.

— En 2002, Agrawal, Kayal et Saxena [4] ont prouvé l'existence d'un algorithme de test de primalité déterministe de temps polynomial en $\mathcal{O}\left((\log n)^{12}\right)$. Il y a des variantes et

raffinements de cet algorithme qui en affectent la complexité. Par exemple, Pomerance et H. W. Lenstra [75] en ont démontré une qui a une complexité en $\mathcal{O}\left((\log n)^6\right)$.

4.2.6 Récapitulatif

Pour des raisons d'efficacité, la primalité de n est testée en pratique principalement avec des algorithmes probabilistes tels que celui de Solovay-Strassen ou plus souvent celui de Miller-Rabin (recommandations des normes connues présentées dans la section 4.6).

On donne dans le tableau suivant, un récapitulatif des tests de non-primalité et des tests de primalité présentés ci-dessus avec leurs complexités respectives. Pour les tests de Miller-Rabin et de Solovay-Strassen, l'entier t correspond au nombre d'itérations.

Tests de primalité/non-primalité	**Complexités**
Test BPSW	$\mathcal{O}\left((\log n)^3\right)$
Test AKS	$\mathcal{O}\left((\log n)^{12}\right)$
Test APRCL	$\mathcal{O}\left((\log n)^{c \log\log\log n}\right)$
Test d'Atkin et Morain (ou ECPP)	$\mathcal{O}\left((\log n)^{6+\varepsilon}\right)$
Test probabiliste de Miller-Rabin	$\mathcal{O}\left(t \times (\log n)^3\right)$
Test de primalité de Miller-Rabin	$\mathcal{O}\left((\log n)^4\right)$
Test probabiliste de Solovay-Strassen	$\mathcal{O}\left(t \times (\log n)^3\right)$
Test de primalité de Solovay-Strassen	$\mathcal{O}\left((\log n)^4\right)$

Tableau 4.1 – ***Tests de primalité et tests de non-primalité avec leurs complexités.***

4.3 Distribution des nombres premiers

Un autre point qui mérite d'être analysé concerne le nombre de nombres aléatoires (de la même taille) qu'on devra tester en moyenne pour trouver un nombre premier. Contrairement à certains ensembles spéciaux qui ont une structure régulière tels que l'ensemble des carrés parfaits, les nombres premiers semblent ne suivre aucun style apparent. Pour un entier positif x relativement grand (près de 80 bits actuellement), donner la quantité de nombres premiers paraît difficile. Mais il existe des résultats permettant d'en obtenir des approximations.

Pour un entier $x > 2$, la fonction donnant la quantité de nombres premiers inférieurs ou égaux à x est notée π. En analysant des tables de grands nombres premiers en 1792, Gauss, alors qu'il n'avait que 15 ans, a observé que :

$$\pi(x) \sim \frac{x}{\log x}.$$ [2]

2. Si f et g sont deux fonctions, alors $f \sim g$ signifie que $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$.

Quelques années plus tard, il observa de nouveau que la densité des nombres premiers autour de x est d'environ $\frac{1}{\log x}$, et conjectura une nouvelle formulation en affirmant qu'une meilleure approximation de $\pi(x)$ est donnée par le *logarithme intégral* défini par :

$$\mathrm{li}(x) = \int_2^x \frac{\mathrm{d}t}{\log t}.$$

La démonstration de la conjecture de Gauss sur $\pi(x)$ aura pris plus d'un siècle. En 1859, Riemann [100] a donné le début d'une démonstration où il explique comment $\pi(x)$ est très lié aux propriétés analytiques de la fonction définie par :

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s},$$

connue actuellement sous le nom de *fonction zêta de Riemann* [67]. Il a aussi proposé une démarche dont l'achèvement conduirait à une meilleure compréhension de la distribution des nombres premiers. En particulier, elle établirait l'existence d'une constante $c > 0$ tel que :

$$|\pi(x) - \mathrm{li}(x)| \leqslant c\sqrt{x} \times \log x \quad \text{(pour tout entier } x \geqslant 2)$$

qui est une forme forte de confirmation de la supposition de Gauss. En 1895, Hadamard [48] et von Mangoldt [121] ont démontré rigoureusement toutes les étapes de la démarche de Riemann sauf une qui reste jusqu'à présent insaisissable et qui est connue sous le nom de l'*hypothèse de Riemann*. Cependant, en 1896, Hadamard [49] et de La Vallée Poussin [31] ont démontré une forme faible de l'hypothèse de Riemann, mais qui était suffisamment forte pour aboutir à une démonstration de la conjecture de Gauss que l'on appelle aujourd'hui *théorème des premiers nombres*.

Théorème 4.3.1 (Théorème des nombres premiers [67, chap.8])**.** *On a :* $\pi(x) \sim \mathrm{li}(x)$.

Si un entier est choisi au hasard entre deux entiers naturels a et b (où $a < b$), alors la probabilité pour qu'il soit premier sera :

$$\frac{\pi(b) - \pi(a)}{b - a}.$$

Par exemple, un entier naturel de 1024 bits généré aléatoirement sera premier avec une probabilité d'environ :

$$\frac{\pi\left(2^{1024}\right) - \pi\left(2^{1023}\right)}{2^{1024} - 2^{1023}} \approx \frac{\mathrm{li}\left(2^{1024}\right) - \mathrm{li}\left(2^{1023}\right)}{2^{1024} - 2^{1023}} = \frac{1}{2^{1024} - 2^{1023}} \int_{2^{1023}}^{2^{1024}} \frac{\mathrm{d}t}{\log t} \approx \frac{1}{710}.$$

Autrement dit, sur 710 nombres aléatoires de 1024 bits, en moyenne l'un sera premier. Si on se restreint aux nombres impairs, la probabilité double. De façon générale, des analyses plus poussées avancent une hypothèse selon laquelle les nombres premiers se comportent comme s'ils étaient aléatoires. Plusieurs résultats sont en accord avec cette hypothèse. Par exemple, la progression arithmétique $ak + b$ contient une infinité de nombres premiers tant que a et b sont deux entiers naturels étrangers, et les nombres premiers restent répartis également entre ces progressions arithmétiques [67, chap.9, chap.12].

4.4 Méthodes de factorisation d'entiers

La plupart des discussions sur la cryptanalyse du chiffrement RSA se sont focalisées sur la factorisation du module n en deux nombres premiers, car cette tâche est utilisée comme une sorte de référence pour évaluer la sécurité du chiffrement RSA. Par exemple, déterminer $\phi(n)$ en n'ayant que n est équivalent à factoriser n [99]. Aussi, avec les algorithmes qui existent actuellement, calculer d en n'ayant que n et e semble au moins prendre le même temps que la factorisation de n [57].

Cependant, il existe d'autres méthodes permettant d'attaquer le chiffrement RSA telles que les attaques temporelles et les attaques par faute dont les principes de fonctionnement sont différents de celui de la factorisation.

Attaque temporelle. Elle exploite les temps d'exécution de l'algorithme de déchiffrement. Depuis Paul Kocher [65], on sait que l'on peut déterminer une clef privée en gardant une trace du temps qu'un ordinateur prend pour déchiffrer les messages [56]. Dans le cas du chiffrement RSA, on peut l'expliquer en utilisant, par exemple, la fonction d'exponentiation modulaire, mais l'attaque peut être adaptée à d'autres chiffrements asymétriques en utilisant toute fonction n'ayant pas un temps d'exécution constant. En pratique, les variations dans les temps d'exécution d'exponentiation modulaire ne sont pas extrêmes. Néanmoins, il y a suffisamment de variation dans certaines implémentations pour mettre cette attaque en pratique [65]. Il existe toutefois des contre-mesures assez simples qui peuvent être mises en place comme :

- s'assurer que les exponentiations prennent le même temps avant de donner le résultat (cela est facile mais joue sur la performance) ;
- ajouter un temps d'attente aléatoire dans l'algorithme d'exponentiation afin de brouiller le temps d'exécution ;
- multiplier le texte chiffré par un nombre aléatoire avant d'effectuer l'exponentiation afin d'empêcher une analyse bit par bit.

Attaque par faute. Cette approche, présentée dans «On the Importance of Checking Cryptographic Protocols for Faults» [20] et plus tard dans «Fault-Based Attack of RSA Authentication» [87], est une attaque sur le processeur qui génère la signature numérique RSA. Elle induit des fautes dans le calcul de la signature en réduisant la puissance du processeur. Les fautes font que l'algorithme produit une signature invalide qui pourra ensuite être analysée par l'attaquant pour trouver la clef privée. Dans ce dernier article, ils ont montré qu'une telle analyse pouvait être faite et l'ont démontré en extrayant une clef privée RSA de 1024 bits en presque 100 heures avec un microprocesseur disponible dans le commerce. L'attaque requiert que l'attaquant ait un accès physique à l'ordinateur de la cible et qu'il soit capable de contrôler le courant d'entrée du processeur. Elle est donc moins sérieuse, néanmoins, elle mérite considération.

Dans cette section, on présentera les principales méthodes de factorisation d'entiers (des plus élémentaires aux plus sophistiquées). On donnera des exemples sur des modules RSA qu'on construit pour être rapidement factorisables dans le cas des méthodes non génériques.

4.4.1 Méthode «$p-1$» de Pollard

La méthode «$p-1$» de Pollard [89] est une méthode de factorisation qui permet de trouver de façon efficace un facteur premier p d'un entier composé n si $(p-1)$ est B*-lisse*, c'est-à-dire si tous les facteurs premiers de $(p-1)$ sont inférieurs à un entier B donné.

D'après le petit théorème de Fermat, $a^m = 1 \bmod p$ pour tout multiple m de $(p-1)$ et pour tout entier a non multiple de p. Donc si g est le PGCD de $(a^m - 1)$ et n, alors p divise g. Il est toutefois possible que g soit égal à n, auquel cas on n'obtient pas de facteur non trivial. L'algorithme est donné ci-dessous, et avant de le dérouler, il faut dans un premier temps essayer de deviner la borne B pour ensuite calculer m dont on peut donner la valeur suivante : PPCM$(1,2,\ldots,B)$.

Algorithme 4.4 : Méthode de factorisation «$p-1$» de Pollard

Entrées : un entier composé $n > 1$.
Sorties : un facteur non trivial de n.

```
tant que vraie faire
    a ← nombre aléatoire dans [1; n]
    g ← PGCD(a, n)
    si 1 < g < n alors
        retourner g
    fin
    x ← a^m mod n
    g ← PGCD(x − 1, n)
fin
```

Si g est un facteur trivial de n et que l'on désire chercher un autre facteur de n, on peut dérouler à nouveau l'algorithme avec une autre valeur de B (donc une autre valeur de m). Cet algorithme a une complexité en $\mathcal{O}(B \log B (\log n)^2)$ [89] ; il est donc intéressant lorsque B est relativement petit.

Pour éviter cette attaque, les facteurs premiers p et q d'un module RSA n devraient être choisis de sorte que $(p-1)$ et $(q-1)$ contiennent tous les deux au moins un facteur premier plus grand que 10^{20} (soit environ 2^{66}) [123].

Exemple 4.1. Soit le module RSA de 512 bits suivant.

$$n = \texttt{0xce}\texttt{ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff68}$$
$$\texttt{2cffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff43}$$

On a pu factoriser ce module sur un i7-4770 CPU@3.40GHz×8 en 10 secondes en utilisant une implémentation de l'Algorithme 4.4 en PARI/GP 2.12.0 [117]. Ses facteurs premiers sont :

$$p = \texttt{0xcf}\texttt{0000000000000000000000000000000000000000000000000000000000000001}$$
$$q = \texttt{0x}\texttt{ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff43}$$

La factorisation de n fut rapide avec la méthode «$p-1$» de Pollard, car l'un de ses facteurs premiers, à savoir p, s'écrit $p = 23 \times 3^2 \times 2^{248} + 1$ (donc p est 24-lisse). Rappelons que l'on a construit ce module spécialement pour qu'il soit facilement factorisable avec cette attaque.

4.4.2 Méthode ρ de Pollard

L'idée qui est derrière cette méthode est la suivante. On note f une fonction polynôme à coefficients entiers. On définit une suite (x_i) comme suit :

$$\begin{cases} x_0 \in [1;n] \\ x_{i+1} = f(x_i) \bmod n \quad \text{(pour tout entier } i \geqslant 0\text{)}. \end{cases}$$

Si p est un diviseur premier (inconnu) de n, alors la suite $y_i = x_i \bmod p$ vérifiera la même récurrence. Si f est *convenablement* choisie, il n'est pas excessif de supposer que cette suite se comportera comme la suite d'itérations d'une application aléatoire de $\mathbb{Z}/p\mathbb{Z}$ dans $\mathbb{Z}/p\mathbb{Z}$. Une telle suite doit être ultimement périodique et une analyse mathématique montre qu'il est raisonnable d'espérer que la période ou la pré-période aura une longueur en $\mathcal{O}\left(\sqrt{p}\right)$.

Si $y_{i+t} = y_i$, cela signifiera que $x_{i+t} \equiv x_i \bmod p$, donc $\mathrm{PGCD}(x_{i+t} - x_i, n) > 1$. Ce plus grand commun diviseur sera rarement égal à n et donnera donc un facteur non trivial de n (mais pas nécessairement p). Le nombre d'étapes nécessaires sera en $\mathcal{O}\left(\sqrt{p}\right) = \mathcal{O}\left(n^{\frac{1}{4}}\right)$ et le temps total des opérations sera en $\mathcal{O}\left(n^{\frac{1}{4}}(\log n)^2\right)$. Il est évident que cette méthode sera efficace puisque les opérations sont simples et son temps d'exécution dépend de la taille du plus petit facteur premier de n plutôt que celle de n lui-même. Donc elle peut remplacer, par exemple, la méthode des divisions successives pour détecter des petits facteurs premiers.

Définition 4.4.1. Soit une suite d'éléments d'un ensemble fini E définie par $y_{i+1} = f(y_i)$. Une telle suite est ultimement périodique, c'est-à-dire qu'il existe deux entiers M et $T > 0$ tels que pour tout $i \geqslant M$, $y_{i+T} = y_i$, mais $y_{M-1+T} \neq y_{M-1}$. L'entier M est appelé la *pré-période* et T (choisi aussi petit que possible) est appelé la *période*.

Algorithme 4.5 : Méthode ρ de Pollard

Entrées : une fonction polynôme f et un entier composé $n > 1$.
Sorties : un facteur non trivial p de n, ou échec.

```
x ← x_0
y ← f(x) mod n
p ← PGCD(x − y, n)
tant que p = 1 faire
    x ← f(x) mod n
    y ← f(y) mod n
    y ← f(y) mod n
    p ← PGCD(x − y, n)
fin
si p = n alors
    Échec
sinon
    retourner p
fin
```

Pour trouver deux entiers positifs i et $t > 0$ tels que $y_{i+t} = y_i$ efficacement (sans calculer M et T), Pollard et Floyd ont suggéré une méthode consistant à calculer simultanément avec la suite (y_i), la suite (z_i) définie par :

$$\begin{cases} z_0 & = & y_0 \\ z_{i+1} & = & f(f(z_i)) \bmod n \quad \text{(pour tout entier } i \geqslant 0\text{)}. \end{cases}$$

Clairement, $z_i = y_{2i}$. Si i est un multiple de T plus grand que M, on doit avoir $z_i = y_{2i} = y_i$, donc le problème est résolu. Cela conduit à une méthode simple et efficace. Cependant, elle demande trois évaluations de fonction à chaque étape, et cela peut paraître beaucoup. Une version améliorée a été proposée par Brent, mais les deux méthodes restent comparables ; le nombre d'évaluations de fonction a été diminué et compensé par des comparaisons nécessaires pour trouver une collision. Une autre modification de la version de Brent donne généralement de résultats meilleurs que les deux versions précédentes. Plus de détails sont donnés sur ces versions dans l'ouvrage «A Course in Computational Algebraic Number Theory» [27, sec.8.5].

Quant au choix de la fonction f, il est fait de sorte à minimiser le nombre d'opérations. De ce fait, on voudra prendre un polynôme de petit degré. Mais on devra éviter les polynômes linéaires, car ils ne seront pas aléatoires (et donneront donc de mauvais résultats). Les polynômes du second degré semblent en pratique bien fonctionner tant qu'on évite quelques cas spéciaux. Les plus faciles à calculer sont ceux de la forme :

$$f(x) = x^2 + c.$$

Des choix possibles de c sont $c = \pm 1$, mais $c = 0$ devrait être évité. On doit aussi éviter $c = -2$ puisque $x_{i+1} = x_i^2 - 2$ devient trivial si l'on pose $x_i = u_i + \dfrac{1}{u_i}$.

Notons que l'algorithme peut échouer en indiquant que la période modulo les différents facteurs premiers de n est essentiellement la même. Dans ce cas, on ne déroule pas l'algorithme avec une autre valeur de x_0, mais avec un autre polynôme, par exemple $x^2 - 1$ ou $x^2 + 3$. Notons également que bien que l'on croie que les polynômes mentionnés ci-dessus se comportent comme des applications aléatoires, il n'est pas prouvé que c'est vraiment le cas.

4.4.3 Méthode de Fermat

On désigne par n un entier composé. Si deux entiers a et b sont tels que $a^2 \equiv b^2 \bmod n$ et $a \not\equiv \pm b \bmod n$, alors n divise l'entier : $a^2 - b^2 = (a-b)(a+b)$. Puisque n ne divise ni $(a-b)$ ni $(a+b)$, alors $\mathrm{PGCD}(a-b, n)$ et $\mathrm{PGCD}(a+b, n)$ sont des facteurs de n. Pour un entier b donné, la méthode de Fermat [72] essaie de trouver un entier a tel que $a^2 - n = b^2$.

Algorithme 4.6 : Méthode de factorisation de Fermat

Entrées : un entier composé impair $n > 1$.
Sorties : un facteur non trivial de n.

```
a ← ⌈n^{1/2}⌉
x ← a^2 − n
tant que x n'est pas un carré parfait faire
    a ← a + 1
    x ← a^2 − n
fin
retourner (a − x^{1/2}) et (a + x^{1/2})
```

Habituellement, l'algorithme démarre avec $a = \lceil n^{1/2} \rceil$, car cette valeur fonctionne pour tout entier composé n. Cependant, les grands facteurs premiers de n doivent être proches de $a = \lceil n^{1/2} \rceil$ pour qu'on puisse les trouver en temps raisonnable.

Si $n = p \times q$ est un module RSA, de Weger [32] a montré que l'efficacité de la technique de Fermat est régie par le rapport $\mathcal{O}\left(\frac{\Delta^2}{4\sqrt{n}}\right)$ avec $\Delta = p - q$. Il a également montré que si $|\Delta| < c \times n^{\frac{1}{4}}$ pour une constante c, cette méthode factorise n efficacement.

Exemple 4.2. Soit le module RSA de 512 bits suivant.

$$n = \texttt{0x4000000000000000000000000000080000000000000000000000000000000c9}$$
$$\texttt{0000000000000000000000000005f000000000000000000000000000000071ed}$$

Le module n a été factorisé sur un i7-4770 CPU@3.40GHz×8 en 5 minutes en utilisant une implémentation de l'Algorithme 4.6 en PARI/GP 2.12.0 [117]. Ses facteurs premiers sont :

$$p = \texttt{0x800000000000000000000000000000000000000000000000000000005f}$$
$$q = \texttt{0x800000000000000000000000000010000000000000000000000000000133}$$

On rappelle encore que ce module a été construit spécialement pour qu'il soit facilement factorisable par la méthode de Fermat.

4.4.4 Factorisation de Lenstra par les courbes elliptiques

Cette méthode repose sur des idées semblables à celle de la méthode «$p-1$» de Pollard, mais elle utilise le groupe des points d'une courbe elliptique sur $\mathbb{Z}/p\mathbb{Z}$ au lieu du groupe $(\mathbb{Z}/p\mathbb{Z})^\times$. La méthode est due à H. W. Lenstra et est l'une des trois principales méthodes qui sont actuellement répandues. Les deux autres, à savoir le crible quadratique (ou QS pour «Quadratic Sieve») et le crible du corps de nombres (ou NFS pour «Number Field Sieve»), seront présentées dans la sous-section 4.4.6 et la sous-section 4.4.7 respectivement.

De plus, cette méthode possède d'autres propriétés qui la rendent intéressante même si elle est utilisée en conjonction avec d'autres algorithmes. Elle requiert peu d'espace et a une complexité temporelle en :

$$\mathcal{O}\left(e^{(\sqrt{2}+o(1))\sqrt{\log p \log\log p}}\right).$$

Elle est sensible à la taille des diviseurs premiers, c'est-à-dire que sa complexité dépend de la taille du plus petit diviseur p de n plutôt que celle de n lui-même. Donc elle peut, par exemple, être utilisée pour détecter de petits facteurs premiers après avoir utilisé entre autres les divisions successives et la méthode ρ de Pollard. Elle peut trouver sans difficulté majeure des nombres premiers ayant une taille allant de 10 à 20 chiffres décimaux. Mais lorsque n a plus de 70 chiffres décimaux par exemple et est le produit de deux nombres premiers n'ayant pas de forme spéciale, la méthode des courbes elliptiques ne sera pas capable de le factoriser. Dans ce dernier cas, on pourra utiliser le QS ou le NFS.

4.4.5 Factorisation par fraction continue

Cette section présente la méthode de factorisation par fraction continue (ou CFRAC pour «Continued Fraction Factorization Method»). Bien qu'elle ait été remplacée par des méthodes encore meilleures, elle est importante pour deux raisons. Premièrement, elle est historiquement la première méthode dont la complexité est asymptotiquement sous-exponentielle, et entre 1960 et 1970, elle était la principale méthode de factorisation utilisée. Deuxièmement, elle partage un certain nombre de propriétés avec les méthodes QS et NFS. L'idée principale de CFRAC, aussi bien que le QS et le NFS, est de trouver des entiers x et y tels que :

$$x^2 \equiv y^2 \bmod n \quad \text{avec } x \not\equiv \pm y \bmod n.$$

Puisque $x^2 - y^2 = (x-y)(x+y)$, il s'ensuit que PGCD$(x-y,n)$ ou PGCD$(x+y,n)$ sera un facteur non trivial de n. La recherche de tels entiers x et y est une tâche difficile, mais l'astuce commune aux trois méthodes citées ci-dessus est de trouver des congruences de la forme :

$$x_i^2 \equiv (-1)^{e_{0i}} \times {p_1}^{e_{1i}} \times \cdots \times {p_m}^{e_{mi}} \bmod n$$

où les p_j sont des petits nombres premiers. Avec suffisamment de congruences et par élimination de Gauss sur $\mathbb{Z}/2\mathbb{Z}$, on peut espérer trouver une relation de la forme :

$$\sum \varepsilon_i (e_{0i}, \cdots, e_{mi}) \equiv (0, \cdots, 0) \bmod n$$

où $\varepsilon = 0$ ou 1. Puis, si

$$x = \prod x_i^{\varepsilon_i} \quad \text{et} \quad y = (-1)^{v_0} \times {p_1}^{v_1} \times \cdots \times {p_m}^{v_m}$$

où $\sum \varepsilon_i (e_{0i}, \cdots, e_{mi}) = 2(v_0, \cdots, v_m)$, on aura $x^2 \equiv y^2 \bmod n$. Si de plus, $x \not\equiv \pm y \bmod n$, cela permettra d'avoir un facteur de n. L'ensemble des nombres premiers p_i $(1 \leqslant i \leqslant m)$ que l'on a choisi pour trouver les congruences est appelé *base de facteurs*. Notons que les trois méthodes diffèrent essentiellement dans la façon de générer les congruences.

La méthode CFRAC est issue d'idées de Legendre, Kraitchik, Lehmer et Powers. Elle fut développée afin de factoriser le septième nombre de Fermat F_7 par Brillhart et Morrison [83]. Elle consiste à chercher de petites valeurs de t telles que $x^2 \equiv t \bmod n$ a une solution. Une fois que l'on a t, il y a une grande chance qu'il soit le produit de certains nombres premiers de notre base de facteurs. Si c'est le cas, on aura l'une des congruences recherchées.

Si t est petit et $x^2 \equiv t \bmod n$, on peut écrire $x^2 = t + kb^2 n$ où k et b sont deux entiers, donc $\left(\frac{x}{b}\right)^2 - kn = \frac{t}{b^2}$ sera aussi petit. Autrement dit, $\frac{x}{b}$ est une bonne approximation de $\sqrt{kn}$. Il est connu que le développement en fraction continuée d'un nombre réel donne une bonne approximation rationnelle. On calcule le développement en fraction continuée de kn pour un certain nombre de valeurs de k. Cela donne de bonnes approximations rationnelles $\frac{U}{V}$, puis on essaie de factoriser l'entier correspondant $t = U^2 - V^2 kn$ sur notre base de facteurs. Si on y parvient, on aura une nouvelle congruence.

Avec au moins $m+2$ congruences et une élimination de Gauss sur $\mathbb{Z}/2\mathbb{Z}$, on peut obtenir une congruence $x^2 \equiv y^2 \bmod n$ et trouver ainsi un facteur non trivial de n.

4.4.6 Crible quadratique

Le crible quadratique (QS pour «Quadratic Sieve») formait avec la méthode des courbes elliptiques les plus puissantes méthodes de factorisation utilisées dans les années 1990. Il est dû à Pomerance bien que Kraitchik avait aussi des idées similaires. Il a le même principe de fonctionnement que CFRAC ; la grande différence étant dans la façon dont les congruences sont générées. On veut que $x^2 \bmod n$ ne soit pas trop grand, mais on accepte des restes plus grand que $\sqrt{n}$. Le moyen le plus simple de faire cela est de considérer le polynôme :

$$Q(a) = \left(\lfloor \sqrt{n} \rfloor + a\right)^2 - n.$$

Pour $x = \lfloor \sqrt{n} \rfloor + a$, on a : $Q(a) \equiv x^2 \bmod n$ et tant que $a = \mathcal{O}(n^\varepsilon)$, on aura $Q(a)$ en $\mathcal{O}\left(n^{\frac{1}{2}+\varepsilon}\right)$. Le point crucial et à partir duquel le nom de la méthode découle, est que contrairement à CFRAC, on n'a pas besoin de factoriser tous les $x^2 \bmod n$ sur la base de facteurs. En effet, puisque Q est un polynôme avec des coefficients entiers, on utilise un *crible*.

Le choix du polynôme Q est intéressant, mais puisqu'il est seul, les valeurs croissent plus rapidement qu'on ne le veuille. Il existe une amélioration particulière du QS appelée MPQS («Multiple Polynomial Quadratic Sieve») dont l'idée est d'utiliser plusieurs polynômes Q tels que la taille de $Q(a)$ peut être gardée aussi petite que possible. Par exemple, Montgomery suggéra des polynômes de la forme $Q(x) = Ax^2 + 2Bx + C$ avec $A > 0, B^2 - AC > 0$ et tels que $n \mid (B^2 - AC)$. Cela donne des congruences aussi intéressantes que dans le cas précédent :

$$AQ(x) = (Ax + B)^2 - (B^2 - AC) \equiv (Ax + B)^2 \bmod n.$$

4.4.7 Crible du corps de nombres

Le crible du corps de nombres (ou NFS pour «Number Field Sieve») est la méthode de factorisation la plus récente et la plus puissante que l'on connaît. L'idée est la même que pour le QS, c'est-à-dire de trouver des congruences modulo n en utilisant un crible sur une base de facteurs, puis faire une élimination de Gauss sur $\mathbb{Z}/2\mathbb{Z}$ pour obtenir une congruence de carrés, et donc avec un peu de chance factoriser n.

Avant l'apparition du NFS, toutes les méthodes modernes de factorisation avait une complexité temporelle prévue d'au mieux en :

$$\mathcal{O}\left(e^{\sqrt{\log n \log\log n}+(1+o(1))}\right).$$

Avec le théorème de Canfield, Erdos et Pomerance [25] certains pensaient que cela ne pouvait être améliorée, sauf peut-être pour le $(1+o(1))$. L'invention du NFS par Pollard a changé cette croyance, puisque la complexité temporelle de ce dernier est en :

$$\mathcal{O}\left(e^{(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}+(C+o(1))}\right)$$

pour une petite constante C. Asymptotiquement, il est meilleur que ce qui existait avant.

Pour les nombres n'ayant pas de forme spéciale, il semble que le point de rupture du NFS et le MPQS se situe au niveau des nombres d'une taille d'environ 130 chiffres décimaux.

D'autre part, pour les nombres ayant une forme spéciale tels que ceux de Mersenne $2^p - 1$ ou ceux de Fermat $2^{2^k} + 1$, le NFS peut être considérablement simplifié et reste praticable pour des valeurs de n au delà de 120 chiffres décimaux. Par exemple, avec cette technique, en 1990, A. K. Lenstra et Manasse ont pu factoriser le neuvième nombre de Fermat $F_9 = 2^{512} + 1$ qui fait 155 chiffres décimaux. Les trois facteurs ont 7, 49 et 99 chiffres décimaux, et le plus petit des trois était évidemment déjà connu.

4.5 Expositions partielles de paramètres privés

Les résultats énoncés dans cette section s'inspirent des travaux de Coppersmith sur la recherche de petites solutions d'équations modulaires polynomiales à une variable [28, 29] en utilisant des réseaux mathématiques. Ces résultats ont servi lors de l'attaque ROCA [85] (présentée brièvement dans la sous-section 4.5.2) qui a permis de factoriser des modules RSA générés par la bibliothèque cryptographique RSALib utilisée par des matériels d'Infineon. Ils ont également été utilisés par Bernstein *et al.* [17] pour factoriser des modules RSA des clefs provenant de cartes d'identité taïwanaises. Dans la sous-section 5.3.7 du chapitre 5, les résultats de Coppersmith auront aussi permis de factoriser les modules de plusieurs dizaines de clefs RSA des pare-feux «ZX_ZW2P».

Le lecteur pourra trouver plus d'informations sur les attaques contre le chiffrement RSA basées sur les réseaux mathématiques dans la thèse «Cryptanalysis of RSA Using Algebraic and Lattice Methods» [3] de G. Durfee.

4.5.1 Résultat de Coppersmith

Supposons qu'on a un polynôme f et un entier M. On voudrait trouver un entier x_0 tel que $f(x_0) \equiv 0 \bmod M$. Le principal outil que l'on va utiliser est attribué à plusieurs auteurs. Sa première utilisation serait due à Håstad [50]. Il a ensuite été utilisé implicitement par Coppersmith [28], et énoncé sous la forme suivante par Howgrave-Graham [54].

Théorème 4.5.1 (Howgrave-Graham [54])**.** *Soit h un polynôme à coefficients réels de degré δ, et soit X un réel donné. Supposons qu'il y a un certain x_0 tel que $|x_0| < X$ et que les deux conditions suivantes sont satisfaites.*

1. $h(x_0) \in \mathbb{Z}$
2. $\|h(xX)\| < \dfrac{1}{\sqrt{\delta}}$

Alors $h(x_0) = 0$.

Ce résultat suggère de chercher un polynôme h dont la norme est petite et qui admet x_0 pour solution. L'idée est de trouver un ensemble de polynômes qui admettent tous x_0 pour solution modulo M. À partir de ces polynômes, on construit un réseau, puis on utilise l'algorithme LLL pour avoir un vecteur court de ce réseau. Le premier vecteur de la base obtenue par cet algorithme représente un polynôme de petite norme $h(xX)$ qui satisfera $h(x_0) \in \mathbb{Z}$.

3. https://theory.stanford.edu/~gdurf/durfee-thesis-phd.pdf.

S'il satisfait la deuxième condition du Théorème 4.5.1, on résout l'équation $h(x) = 0$ avec une méthode classique de résolution d'équations telle que celle de Newton-Raphson [92] pour trouver x_0. Les polynômes que Howgrave-Graham recommande sont ceux de la forme :

$$g_{i,j}(x) = x^j n^{m-i} f^i(x) \text{ avec } 0 \leqslant i \leqslant m-1 \text{ et } 0 \leqslant j \leqslant \delta - 1$$
$$h_i(x) = x^i f^m(x) \text{ avec } 0 \leqslant i \leqslant t-1$$

Algorithme 4.7 : Méthode de Coppersmith

Entrées : une fonction f et un entier composé $n > 1$.
Sorties : un facteur premier p de n.

pour $i \leftarrow 0$ *à* $m-1$ **faire**
  $g_i(x) = n^{m-i} f^i(x)$
fin
pour $i \leftarrow 0$ *à* $t-1$ **faire**
  $h_i(x) = x^i f^m(x)$
fin
$B \leftarrow$ base dont les vecteurs sont les vecteurs coefficients de $g_i(xX)$ et $h_i(xX)$.
$B' \leftarrow \text{LLL}(B)$.
$V \leftarrow$ le premier vecteur de B'.
Construire le polynôme g à partir de V.
$R \leftarrow$ l'ensemble des solutions entières de g.
pour $x_0 \in R$ **faire**
  $p \leftarrow \text{PGCD}(n, f(x_0))$
  si $1 < p < n$ **alors**
    retourner p
  fin
fin
retourner *Changer m ou t ou les deux.*

Le théorème suivant de Coppersmith correspond à une application du Théorème 4.5.1 à un module RSA si une quantité suffisante de bits d'un de ses facteurs premiers est connue.

Théorème 4.5.2 (Coppersmith [28])**.** *Soit $n = p \times q$ un module RSA de taille k bits tel que p a une taille d'environ $\frac{k}{2}$ bits. Connaissant au moins les $\frac{k}{4}$ bits de poids faible de p ou les $\frac{k}{4}$ bits de poids fort de p, on peut factoriser n en temps polynomial.*

Soit $n = p \times q$ un module RSA avec p et q ayant la même taille. En supposant que $p = p_0 + x_0$ où p_0 est un entier que l'on connaît, le polynôme f sera défini par : $f(x) = p_0 + x$. Si la taille de p_0 fait au moins le quart de celle de n, alors d'après le Théorème 4.5.2, on peut factoriser n en temps polynomial.

Il existe d'autres méthodes semblables à celle de Coppersmith, c'est-à-dire des méthodes efficaces si un grand nombre de bits d'un des paramètres privés sont connus. Par exemple, dans l'article «An Attack on RSA Given a Small Fraction of the Private Key Bits» [21], Boneh, Durfee et Yankel ont montré que si l'exposant public e d'une clef RSA (dont le module n est

sur k bits) est suffisamment petit et que si l'on connaît au moins les $\frac{k}{4}$ bits de poids faible de l'exposant privé d, alors on peut retrouver d entièrement en temps polynomial en $\mathcal{O}(e \log e)$.

4.5.2 Un exemple : l'attaque ROCA

Comme signalé au début de la section 4.2, la génération des clefs RSA nécessite de choisir aléatoirement deux grands nombres premiers satisfaisant à certaines conditions, ce qui peut être un processus lent notamment dans des petits dispositifs électroniques tels que des cartes à puce. En voulant résoudre ce problème de vitesse, les développeurs de RSALib ont décidé de choisir des modules dont les facteurs premiers sont uniquement de la forme :

$$k \times M + 65537^a \bmod M$$

où M est le produit des l premiers nombres premiers $(2,3,5,7,\cdots)$ et l est une constante qui dépend uniquement de la taille du module. La sécurité repose sur les entiers naturels k et a. La valeur $l = 39$ (c'est-à-dire $M = 2\times3\times\cdots\times167$) est utilisée pour générer les facteurs premiers d'un module RSA dont la taille appartient à l'intervalle $[512;960]$. Les valeurs $l = 71,126,225$ sont utilisées pour générer les facteurs premiers d'un module RSA dont la taille appartient à $[992;1952]$, $[1984;3936]$, $[3968;4096]$ respectivement. Soit n un module RSA de RSALib.

$$n = p \times q \text{ où } p = k \times M + 65537^a \bmod M \text{ et } q = h \times M + 65537^b \bmod M$$

Dans l'article ROCA [85], leur but était de calculer a et k pour factoriser n. Si l'on connaît la valeur de a et que $65537^a \bmod M$ est suffisamment grand, on peut retrouver k en temps polynomial selon le Théorème 4.5.2.

Une approche naïve aurait consisté à chercher k avec l'algorithme de Coppersmith pour chacune des valeurs possibles de a. Malheureusement, le nombre de valeurs possibles de a est trop grand pour pouvoir les tester toutes en temps raisonnable. Mais dans l'article, ils ont trouvé une représentation alternative de p et q en utilisant un diviseur M' de M. Cela a ainsi conduit à la recherche d'une inconnue a' (remplaçante de a) qui appartient à un intervalle dont le parcours se fait en temps raisonnable. En fait, ils ont cherché M' de sorte que :

— p et q gardent leur forme initiale, c'est-à-dire qu'il existe a', b', k' et h' tels que :

$$p = k' \times M' + 65537^{a'} \bmod M' \text{ et } q = h' \times M' + 65537^{b'} \bmod M'$$

— l'algorithme de Coppersmith trouve k' pour une valeur correcte de a' : suffisamment de bits doivent être connus ($\log_2 M' > (\log_2 n)/4$) ;

— le temps global de la factorisation soit minimal : le nombre d'essais ($\mathrm{ord}_{M'}(65537)$) et le temps de chaque essai (temps d'exécution de l'algorithme de Coppersmith) doivent conduire à un temps minimal.

Exemple 4.3. En utilisant les bibliothèques C/C++ de PARI/GP 2.12.0 et de OpenMPI-v4.0 [1], on a implémenté une version parallélisée (d'ordonnancement statique) de l'algorithme de Coppersmith sur un i7-4770 CPU@3.40GHz×8. En utilisant toutes les ressources de cette machine, on peut factoriser, par exemple, n'importe quel module RSA de 512 bits généré par RSALib en moins de 3 heures. Signalons qu'il faudra une semaine sur la même machine pour factoriser un tel module avec CADO-NFS (une implémentation du NFS) [115].

4.6 Méthodes recommandées de génération des clefs RSA

Certaines méthodes de factorisation que l'on a présentées dans la section 4.4 permettent de factoriser des modules RSA lorsque ceux-ci ont certaines formes spéciales ou certaines tailles. Ceci indique qu'un module RSA devrait être généré en respectant certains critères pour que sa factorisation soit difficile avec les méthodes que l'on connaît actuellement. Dans cette section, on va donner les recommandations de quelques normes sur les méthodes sûres de génération des paramètres d'une clef RSA, en particulier le module.

4.6.1 Recommandations de IEEE-P1363

Dans la norme IEEE-P1363 [3] de l'IEEE [4], on propose un algorithme qui permet de générer un module RSA dont les facteurs premiers sont sur la même taille et appartiennent à deux intervalles de sorte que leur produit soit sur la taille voulue. Chacun de ces facteurs premiers est généré avec l'Algorithme 4.8 suivant.

Algorithme 4.8 : Génération des nombres premiers RSA par la norme IEEE-P1363

Entrées : un intervalle $I = [p_{\min}; p_{\max}]$ et un entier naturel impair e.
Sorties : un nombre premier aléatoire p de I tel que $\text{PGCD}(p-1, e) = 1$.

1 $k_{\min} \leftarrow \lceil (p_{\min}-1)/2 \rceil$ et $k_{\max} \leftarrow \lfloor (p_{\max}-1)/2 \rfloor$
2 Choisir un entier aléatoire k dans l'intervalle $[k_{\min}; k_{\max}]$.
3 $p \leftarrow 2k+1$
4 **si** *PGCD*$(p-1, e) = 1$ *et* p *est premier* **alors**
5 | **retourner** p
6 **sinon**
7 | **aller à** 2
8 **fin**

Dans l'Algorithme 4.8 ci-dessus, la primalité est vérifiée par un test de Miller-Rabin [94] avec une probabilité d'erreur de 2^{-100}. À noter que l'entier e représente l'exposant public de la clef RSA et que cette norme recommande de choisir sa valeur parmi les 5 premiers nombres premiers de Fermat, c'est-à-dire $3, 5, 17, 257$ et 65537. Quant aux bornes des intervalles, on prend : $p_{\min} = 2^{M-1}$, $p_{\max} = 2^M - 1$, $q_{\min} = \lfloor 2^{n-1}/p + 1 \rfloor$, $p_{\max} = \lfloor 2^n/p \rfloor$ où $M = \lfloor (n+1)/2 \rfloor$.

La norme IEEE-P1363 propose aussi deux autres algorithmes basés sur l'Algorithme 4.8 et qui sont destinés à générer des nombres premiers avec des conditions congruentielles (c'est-à-dire des nombres premiers congrus à a modulo r où r est un nombre premier impair et a un entier non multiple de r) et des nombres premiers dits *forts*.

Définition 4.6.1. Un nombre premier fort est un grand nombre premier p tel que $(p-1)$ a un grand facteur premier (noté r), et $(p+1)$ et $(r-1)$ ont chacun un grand facteur premier [112].

4. Institute of Electrical and Electronics Engineers, https://www.ieee.org/.

4.6.2 Recommandations du NIST

Dans la norme FIPS 186-4 [59, app.B.3], le NIST [5] propose des algorithmes permettant de générer des modules RSA dont les facteurs p et q sont dans l'une des situations suivantes.

1. p et q sont des *nombres premiers probables*, c'est-à-dire des nombres premiers dont la primalité a été vérifiée à l'aide d'un test probabiliste.
2. p et q sont des *nombres premiers prouvables*, c'est-à-dire des nombres premiers pour lesquels on peut donner une preuve de primalité.
3. p et q sont des nombres premiers qui satisfont les conditions suivantes : $(p-1)$ a un facteur premier p_1 ; $(p+1)$ a un facteur premier p_2 ; $(q-1)$ a un facteur premier q_1 et $(q+1)$ a un facteur premier q_2. Dans ce cas et dans cette norme, p et q sont appelés *nombres premiers avec des conditions*, et p_1, p_2, q_1 et q_2 sont appelés *nombres premiers auxiliaires* de p et q.

Dans chacune de ces situations, les nombres p et q peuvent être générés lorsqu'ils sont tous deux sur 1024 bits ou sur 1536 bits, mais c'est uniquement dans la dernière situation qu'on peut les générer lorsqu'ils sont sur 512 bits.

L	**Nombres premiers**		
	probables	**prouvables**	**avec des conditions**
1024	Non	Non	Oui
2048	Oui	Oui	Oui
3072	Oui	Oui	Oui

Tableau 4.2 – ***Méthodes autorisées de génération des nombres premiers.***

Lorsqu'on veut générer des nombres premiers avec des conditions, on doit faire le choix parmi les différentes propositions suivantes.

— p_1, p_2, q_1, q_2, p et q doivent être des nombres premiers probables.

— p_1, p_2, q_1, q_2, p et q doivent être des nombres premiers prouvables.

— p_1, p_2, q_1 et q_2 doivent être des nombres premiers prouvables, et p et q doivent être des nombres premiers probables.

Les tailles minimales et les tailles maximales de p_1, p_2, q_1 et q_2 dépendent de la taille L du module n. Elles peuvent être fixes ou choisies aléatoirement sous réserve des restrictions données dans le tableau suivant.

L	**Tailles min de p_1, p_2, q_1 et q_2**	**Valeurs max de $\lvert p_1\rvert+\lvert p_2\rvert$ et $\lvert q_1\rvert+\lvert q_2\rvert$**	
		p et q probables	**p et q prouvables**
1024	100 bits	496 bits	239 bits
2048	140 bits	1007 bits	494 bits
3072	170 bits	1518 bits	750 bits

Tableau 4.3 – ***Tailles minimales et maximales de p_1, p_2, q_1 et q_2.***

5. National Institute of Standards and Technology, https://www.nist.gov/.

Dans cette norme, lors de la génération d'une clef RSA, l'exposant public e de la clef doit être impair et strictement compris entre 2^{16} et 2^{256}. Les deux facteurs premiers p et q du module doivent appartenir à l'intervalle $[2^{(L-1)/2}; 2^{L/2}-1]$ et doivent être distants de $2^{L/2-100}$ au moins (cette dernière contrainte empêche la factorisation du module par des méthodes similaires à celle de Fermat en temps raisonnable). On donne ci-dessous l'algorithme qui permet de générer p et q lorsqu'on veut qu'ils soient des nombres premiers probables. Les algorithmes traitant les autres situations en découlent.

Algorithme 4.9 : Génération des nombres premiers probables

Entrées : un entier naturel L et un entier naturel impair e.
Sorties : un nombre premier aléatoire p de $L/2$ bits tel que PGCD$(p-1, e) = 1$.

```
1  i ← 0
2  tant que i < 5L/2 faire
3  |  Choisir aléatoirement un entier p dans l'intervalle [2^(L/2-1); 2^(L/2) − 1].
4  |  p ← p ⊕ 1                                   ▷ rendre p impair
5  |  si p < 2^((L-1)/2) alors
6  |  |  aller à 3
7  |  fin
8  |  si PGCD(p − 1, e) = 1 et p est premier alors
9  |  |  retourner p
10 |  fin
11 |  i ← i + 1
12 fin
```

La primalité des nombres p et q doit être vérifiée à l'aide des tests de Miller-Rabin avec des probabilités d'erreur de 2^{-112} (si n est sur $L = 2048$ bits) et 2^{-128} (si n est sur $L = 3072$ bits). La ligne **3** est en réalité l'appel à un générateur de nombres aléatoires (ou RNG pour «Random Number Generator») approuvé par le NIST [14, 119, 15] et qui peut prendre en charge le *niveau de sécurité* («security strength»). Les correspondances entre les niveaux de sécurité d'un RNG et la taille du nombre généré sont données dans la norme SP 800-57 [13].

Pour générer un module RSA dont on veut que les facteurs premiers soient des nombres premiers prouvables, on utilise l'algorithme précédent, mais au lieu de générer un nombre aléatoire puis vérifier sa primalité avec un test probabiliste, on doit utiliser la méthode de construction de nombres premiers de Shawe-Taylor présente dans la norme FIPS 186-4 [59, app.C.10] et la norme X9.31 [8, app.B.5].

L'algorithme permettant de générer des nombres premiers avec des conditions peut être aussi utilisé pour générer des nombres premiers forts à condition de choisir convenablement les tailles de leurs nombres premiers auxiliaires.

4.6.3 Recommandations du BSI

Dans le guide technique [23] du BSI[6] se trouvent trois algorithmes de génération de nombres premiers : génération uniforme par échantillonnage de rejet, génération uniforme

6. Bundesamt für Sicherheit in der Informationstechnik, https://www.bsi.bund.de.

par un échantillonnage de rejet plus efficace, et génération par recherche incrémentale.

Dans chacun de ces algorithmes, les facteurs premiers p et q du module n d'une clef RSA doivent être choisis dans l'intervalle $I = [p_{\min}; p_{\max}]$ où $p_{\min} = 2^{(L-1)/2}$, $p_{\max} = 2^{L/2}$ et L est la taille de n. La valeur minimale de L doit être de 2000 bits, et ce, jusqu'à la fin de l'année 2022 au plus tard. Après cette période, il faudra passer aux modules de 3000 bits au moins. Lors de la génération de p et q, on doit également s'assurer que $(p-1)$ et $(q-1)$ sont étrangers à l'exposant public e de la clef RSA. Quand $L = 2000$ bits, cette norme recommande de choisir la valeur de e parmi les entiers compris largement entre $(2^{16}+1)$ et $(2^{1824}-1)$.

L'algorithme de génération uniforme par échantillonnage de rejet est simple et consiste à tirer aléatoirement un nombre impair dans I, puis le retourner si c'est un nombre premier.

Algorithme 4.10 : Génération uniforme par échantillonnage de rejet

Entrées : l'intervalle I.
Sorties : un nombre premier aléatoire p de I.

$p \leftarrow 0$
tant que *p n'est pas premier* **faire**
  $p \leftarrow$ nombre aléatoire de I
  $p \leftarrow p \oplus 1$
fin
retourner p

Dans les deux autres algorithmes, on utilisera $B\#$ qui désigne la *primorielle* de l'entier naturel B, c'est-à-dire le produit de tous les nombres premiers inférieurs ou égaux à B.

L'algorithme de génération uniforme par un échantillonnage de rejet plus efficace a pour but de choisir de façon aléatoire un nombre premier congru à r modulo $B\#$ où r est un entier naturel inférieur à $B\#$ et étranger à $B\#$. L'entier B doit, au préalable, être choisi de sorte que sa primorielle soit beaucoup plus petite que l'amplitude de I ; on note cela $B\# \ll (p_{\max} - p_{\min})$.

Algorithme 4.11 : Génération uniforme par un échantillonnage de rejet plus efficace

Entrées : l'intervalle $I = [p_{\min}; p_{\max}]$ et un entier naturel B tel que $B\# \ll (p_{\max} - p_{\min})$.
Sorties : un nombre premier aléatoire p de I.

$r \leftarrow 0$ et $p \leftarrow 0$
tant que $PGCD(B\#, r) \neq 1$ **faire**
  Choisir aléatoirement un entier r dans l'intervalle $[1; B\#]$.
fin
tant que *p n'est pas premier* **faire**
  Choisir aléatoirement un entier k dans l'intervalle $[(p_{\min}-r)/(B\#); (p_{\max}-r)/(B\#)]$.
  $p \leftarrow k \times (B\#) + r$
fin
retourner p

L'Algorithme 4.11 permet en fait de générer des nombres premiers de la forme $ak+b$ où a et b sont deux entiers naturels étrangers. Si k est choisi au hasard, alors la distribution des nombres premiers générés ne peut pratiquement pas être distinguée d'un tirage uniforme de nombres premiers. Cela résulte de la version quantitative du théorème de Dirichlet sur les progressions arithmétiques [105, 111].

L'algorithme de génération par recherche incrémentale est très similaire à l'algorithme précédent, mais au lieu de remonter à l'étape d'avant lorsque l'entier généré n'est un nombre premier, on augmente cet entier de $B\#$ puis on vérifie sa primalité de nouveau du moment qu'il est toujours dans l'intervalle I.

Dans ces algorithmes, pour des raisons de rapidité, la primalité des nombres générés doit être testée par des tests probabilistes de Miller-Rabin, et la probabilité qu'un nombre déclaré premier soit composé doit être de 2^{-100} au plus. La norme recommande plus l'utilisation des deux premiers algorithmes, car le troisième présente des biais statistiques généralement indésirables dans la distribution des nombres premiers générés. Cependant, il reste utilisé (voir par exemple le tableau 1 de [122]) et selon la norme rien n'indique actuellement que ces biais peuvent être utilisés pour une attaque.

4.6.4 Recommandations de l'ANSSI

Les méthodes recommandées par l'ANSSI dans son «Guide de sélection d'algorithmes cryptographiques» [7] sont une combinaison des recommandations des normes précédentes. Pour la génération d'un module RSA, les conditions sur les deux nombres premiers sont celles provenant du NIST. Cependant, pour la génération des nombres premiers, la norme recommande les deux premiers algorithmes proposés par le BSI, car ceux-ci fourniraient des nombres premiers permettant d'éviter des attaques de type ROCA. Les générateurs aléatoires que la norme recommande sont ceux du NIST, à savoir le HMAC-DRBG, le Hash-DRBG et le CTR-DRBG. La primalité est testée par des tests de Miller-Rabin avec une probabilité d'erreur ne devant pas dépasser 2^{-128}. Les modules doivent avoir au moins une taille de 3072 bits et l'exposant public doit valoir au moins 2^{16}.

4.7 Conclusion

Dans ce chapitre, on a défini les paramètres du chiffrement RSA (module, exposant public et exposant privé). Les modules étant composés de nombres premiers, il était nécessaire de connaître des tests de non-primalité et des tests de primalité notamment ceux qui sont actuellement les plus utilisés par les bibliothèques cryptographiques et les plus recommandés par les normes connues ; le test de Miller-Rabin ayant reçu un traitement particulier, car il est préféré aux autres tests dans les normes que l'on a présentées. On donne dans le tableau suivant, un récapitulatif de la section 4.6.

Normes	Tailles des modules	Exposants publics
IEEE-P1363	$\geqslant 1024$	3, 5, 17, 257, 65537
NIST	$\geqslant 1024$	$\left[2^{16}+1;2^{256}\right]$
BSI	$\geqslant 2000$	$\left[2^{16}+1;2^{184}-1\right]$
ANSSI	$\geqslant 3072$	$\geqslant 2^{16}+1$

Tableau 4.4 – ***Récapitulatif des recommandations des normes connues.***

7. https://www.ssi.gouv.fr/uploads/2021/03/anssi-guide-selection_crypto-1.0.pdf.

Pour préparer le chapitre suivant, on a également présenté des attaques contre le chiffrement RSA (dont les principales méthodes de factorisation d'entiers). La connaissance de ces attaques permet notamment d'éviter des paramètres vulnérables ; tel est d'ailleurs le but des normes. Cependant, on verra (à travers des cas réels) lors de l'analyse de certificats électroniques dans le chapitre 5 que ces recommandations ne sont pas toujours suivies, ce qui rend donc vulnérables les clefs RSA de plusieurs matériels connectés.

Chapitre 5

Analyse de bases de données de certificats SSL/TLS X.509

Sommaire

Note : Pour diverses raisons, certaines informations de ce chapitre sont partiellement anonymisées.

Dans les études effectuées par Lenstra *et al.* [73], par Heninger *et al.* [52], par Bernstein *et al.* [17], et par Hastings, Fried et Heninger [51], il ressort que la conception et l'évaluation d'un générateur aléatoire pour des besoins cryptographiques sont difficiles. Dans ces études (détaillées dans la section 5.1), le problème venait entre autres de la source d'aléa, de l'algorithme de génération des clefs et de l'utilisation de bibliothèques cryptographiques connues dans des conditions *non optimales*. Ces conditions seront étudiées dans le chapitre 6 lors de l'analyse de quelques unes de ces bibliothèques. Un autre exemple plus récent porte le nom de vulnérabilité ROCA [85] (voir sous-section 4.5.2 du chapitre 4) qui affectait les clefs RSA générées par la bibliothèque RSALib d'Infineon [1], car les développeurs de cette dernière avaient décidé de générer des clefs RSA ayant une certaine forme pour que la bibliothèque puisse gagner en vitesse.

Similairement aux études [73, 52] et [51], le chapitre que l'on aborde a pour but d'étudier les vulnérabilités que l'on peut rencontrer dans les clefs cryptographiques présentes dans les certificats SSL/TLS utilisés sur l'Internet. L'étude se focalisera uniquement sur les certificats SSL/TLS utilisant le chiffrement asymétrique le plus répandu dans les services web, à savoir le chiffrement RSA. On dispose de tels certificats en grande quantité provenant de différentes sources et regroupés dans trois bases de données. La première date de 2010 et provient du projet «EFF SSL Observatory» [40]. La deuxième est composée de certificats SSL/TLS distincts collectés entre 2011 et 2017 par l'ANSSI [2]. Quant à la dernière, elle contient des certificats SSL/TLS collectés entre janvier 2017 et janvier 2021 dans le projet «Project Sonar» [95], mais aussi des certificats SSL/TLS qu'on a collectés entre mars et juillet 2019 à des adresses IPv4 provenant du même projet. Malgré le fait que la base de données de l'EFF soit très ancienne, sa comparaison avec les deux autres nous permettra d'avoir une idée précise de l'évolution des vulnérabilités des clefs cryptographiques par matériel électronique du passé au présent comme aucune autre étude ne l'a fait. Comme on le verra dans la sous-section 5.3.7, son analyse aura également permis de détecter une vulnérabilité toujours présente dans les deux autres bases de données et qui ne pouvait être constatée uniquement avec ces dernières.

Bases de données	Périodes de collecte	Tous les certificats	Certificats RSA	Certificats RSA distincts	Modules RSA distincts
SSL Observatory	2010-2011	4,021,766	4,019,595	4,019,595	3,933,365
ANSSI	2011-2017	197,186,889	183,804,874	183,804,874	133,521,678
Project Sonar	2017-2021	603,667,152	577,697,620	161,133,631	134,127,181

Tableau 5.1 – ***Contenus des bases de données.***

5.1 État de l'art des analyses de clefs cryptographiques de l'Internet

On va présenter dans cette section quelques études ayant déjà été faites sur les vulnérabilités pouvant être rencontrées avec les clefs cryptographiques utilisées sur l'Internet.

1. https://www.infineon.com/.
2. Agence Nationale de la Sécurité des Systèmes d'Information, https://www.ssi.gouv.fr/.

5.1.1 Étude de Lenstra *et al.* (2012)

En voulant vérifier l'hypothèse selon laquelle des choix différents seraient faits à chaque fois que des clefs cryptographiques étaient générées, en 2012, Lenstra *et al.* [73] ont analysé 11.7 millions de clefs publiques. L'ensemble était composé de 5.5 millions de clefs PGP[3] et un peu moins de 0.1 million de certificats X.509 qu'ils avaient eux-mêmes collectés, ainsi que de 6.2 millions de certificats X.509 dont la majorité provenait de «EFF SSL Observatory» [40]. Ces clefs leur ont fournis 6.4 millions de modules RSA distincts et le reste était réparti presque également entre les clefs ElGamal [42] et les clefs DSA [59], plus une clef ECDSA [59].

Durant les analyses, ils ont constaté la présence de quelques doublons parmi les clefs ElGamal et DSA avec des propriétaires non apparentés. De toute évidence, cela posait un problème, car ces propriétaires avaient connaissance des clefs privées des uns des autres (sans nécessairement le savoir). Toutefois, cela restait insignifiant face au cas RSA. En effet, parmi 6.6 millions de certificats X.509 et clefs PGP utilisant le chiffrement RSA, 0.27 million (soit 4.3%) partageaient leurs modules et appartenaient rarement à la même entité.

En plus de cela, ils ont trouvé 30,097 modules (soit 0.48%) blacklistés comme vulnérables au «Debian OpenSSL vulnerability» [126] et ont pu factoriser les modules de 21,419 certificats X.509 et clefs PGP à cause de la présence de facteurs communs entre eux. Ils ont également pu factoriser une poignée de modules RSA ayant de petits facteurs en utilisant notamment une implémentation de la méthode ECM («Elliptic Curve Method») de H. W. Lenstra [74] (que l'on a déjà présentée dans la sous-section 4.4.4 du chapitre 4).

Enfin, ils conclurent que ces vulnérabilités pourraient indiquer que l'ensemencement correct des générateurs de nombres aléatoires restait toujours un problème.

5.1.2 Étude de Heninger *et al.* (2012)

En 2012, en analysant 5.8 millions de certificats TLS distincts (provenant de 12.8 millions d'hôtes) et de 6.2 millions de certificats SSH distincts (provenant de 10.2 millions d'hôtes), Heninger *et al.* [52] ont constaté que 5.57% des hôtes TLS et 9.6% des hôtes SSH utilisaient les mêmes clefs. Pour les hôtes TLS, au moins 5.23% d'entre eux utilisaient des certificats mis par défaut dans les dispositifs (et qui n'avaient pas été remplacés par les utilisateurs) et 0.34% semblaient générer les mêmes clefs à cause d'un dysfonctionnement du générateur de nombres aléatoires. Plus inquiétant encore, ils ont pu retrouver :

— d'une part, les clefs privées RSA de 0.5% des hôtes TLS et 0.03% des hôtes SSH à cause de la présence de facteurs communs entre différents modules RSA avec un algorithme appelé «Batch GCD» (voir section 7.4 du chapitre 7) pouvant calculer efficacement le plus grand commun diviseur (PGCD) de chaque couple d'éléments distincts d'un ensemble d'entiers relativement grands ;

— et d'autre part, les clefs privées DSA de 1.03% d'hôtes SSH en raison d'un caractère aléatoire insuffisant de la signature numérique.

La grande majorité des dispositifs concernés semblaient être des dispositifs sans tête ou embarqués. Ils ont montré que ces types de dispositifs utilisaient majoritairement un noyau

3. Pretty Good Privacy [44]. Une implémentation est disponible à : `https://www.gnupg.org/`.

Linux et que certaines sources d'entropie étaient absentes lors du premier démarrage. Cela créait alors une fenêtre de vulnérabilité pendant laquelle le fichier `\dev\urandom` pourrait être prédictible, d'où la répétition des clefs, car la plupart des bibliothèques cryptographiques telles qu'OpenSSL [116] utilisent par défaut ce fichier comme source d'entropie. À noter que depuis lors, ce problème a été corrigé au niveau du noyau Linux et que plusieurs dizaines de fabricants identifiés (dont la plupart sont des grandes marques) en ont été informés.

5.1.3 Étude de Bernstein *et al.* (2013)

En 2013, Bernstein *et al.* [17] ont factorisé 184 modules RSA distincts sur plus de 2 millions de modules RSA de taille 1024 bits provenant de la base de données nationale taïwanaise de certificats «Citizen Digital Certificates». Ces certificats étaient générés par des cartes à puce permettant à chacun des citoyens de s'authentifier numériquement auprès de services gouvernementaux tels que pendant la déclaration de revenus ou lors de la modification des immatriculations de voitures en ligne. Sur certaines de ces cartes, le générateur de nombres aléatoires utilisé était défectueux et rendait donc des certificats vulnérables.

Parmi les 184 modules, 103 ont été factorisés à cause de la présence de facteurs communs en utilisant l'algorithme «Batch GCD» présenté par Heninger *et al.* [52]. Ces découvertes ont été transmises aux autorités taïwanaises qui, au lieu de revoir les méthodes employées et de remplacer toutes les cartes, ont décidé de révoquer les certificats concernés et de fournir de nouvelles cartes seulement aux personnes concernées. Mais cela ne régla pas le problème. En effet, les 81 modules restants ont été trouvés à l'aide de la technique de Coppersmith [28] en raison de l'existence de motifs (patterns) dans les écritures hexadécimales des 103 modules précédemment factorisés. Ils montrèrent que le générateur de nombres aléatoires avait un déficit d'entropie, mais également qu'il y avait un problème avec l'algorithme de génération de nombres premiers et l'ont partiellement rétro-conçu.

5.1.4 Étude de Hastings, Fried et Heninger (2016)

En 2016, Hastings, Fried et Heninger [51] ont pu analyser 81.2 millions de modules RSA distincts extraits de 131 millions de certificats HTTPS distincts collectés entre juillet 2010 et mai 2016. Ces certificats étaient composés de certificats qu'ils avaient eux-mêmes collectés (dont ils appelaient l'ensemble P&Q), de ceux de «EFF SSL Observatory» [40], de «HTTPS Ecosystem» [36], de «Project Sonar» [95] ainsi que de Censys [4]. En calculant le plus grand commun diviseur de chaque couple de modules distincts, ils en ont pu factoriser 313,330. Puis, après avoir identifié les matériels dont ces modules provenaient, ils se sont rendus compte que certains fabricants n'avaient jamais produit de correctifs malgré le fait qu'ils aient été avertis après l'étude de Heninger *et al.* [52].

Ils ont également remarqué que le nombre d'hôtes vulnérables avait augmenté et que de nouvelles implémentations vulnérables avaient vu le jour depuis 2012.

4. https://censys.io/.

5.1.5 Étude de N. Amiet et Y. Romailler (2018)

En 2018, lors de la convention de hacker DEF CON 26[5], Nils Amiet et Yolan Romailler[6], deux chercheurs de «Kudelski Security»[7], ont exposé une étude qu'ils ont faite sur des clefs publiques téléchargées sur l'Internet. L'ensemble était composé de :

— 240 millions de clefs extraites de certificats X.509 dont plus de 200 millions étaient utilisées dans le protocole HTTPS, et plus d'un million dans chacun des protocoles SMTPS, POP3S et IMAPS ;

— 94 millions de clefs SSH comprenant à la fois des clefs d'hôtes et des clefs d'utilisateurs, et provenant de la base de données de CRoCS[8] (71 millions), de scans SSH (17 millions), de github.com (4.7 millions) et de gitlab.com (1.2 million) ;

— 10 millions de clefs PGP issues principalement de serveurs de clefs, notamment des serveurs de clefs SKS (9.5 millions), de keybase.io (220,000) et de github.com (8,000).

Dans cette base de données, le chiffrement le plus utilisé fut le chiffrement RSA. Le reste était réparti entre d'autres chiffrements tels que ECC (4.05%), DSA (0.75%) ou encore ElGamal (0.72%) et GOST R 34.10-2001. Ils en ont déduit que les certificats X.509 et les clefs PGP utilisant la cryptographie basée sur les courbes elliptiques devenaient de plus en plus fréquents ; la courbe secp256r1 étant la courbe la plus utilisée (97.68%).

Avec une implémentation personnalisée de l'algorithme «Batch GCD» écrite en langage Chapel[9], ils ont pu factoriser près de 210,000 clefs RSA composées de 207,000 clefs extraites de certificats X.509, de 3,100 clefs SSH et de 295 clefs PGP. À partir de ce résultat, ils ont découvert qu'un modèle de cartes à puce (dont les clefs provenaient de la base de données de CRoCS) avait un mauvais générateur de nombres aléatoires, et que cela causait l'apparition de facteurs communs entre les modules des clefs RSA de ces cartes.

En plus de cela, ils ont trouvé environ 4,000 clefs RSA (où 33% étaient sur 2048 bits) vulnérables à l'attaque ROCA et dont la plupart (97%) étaient des clefs PGP.

Pour conclure, ils ont recommandé la cryptographie basée sur les courbes elliptiques, car elle serait plus facile en mettre en œuvre, plus résistante aux défaillances et aurait des meilleures performances que le chiffrement RSA. Cependant, à ceux qui voudraient continuer d'utiliser le chiffrement RSA, ils ont conseillé de passer aux clefs de taille 4096 bits. De plus, ils ont également évoqué le cas des clefs RSA ayant des modules de plus de 2 facteurs premiers, c'est-à-dire les modules RSA mutli-premiers ; ils ne les ont pas exactement recommandées, mais ils ont jugé qu'il serait intéressant de les prendre en compte, car ces dernières auraient de meilleures performances et leurs modules seraient plus difficiles à factoriser.

5. https://defcon.org/html/defcon-26/dc-26-index.html.
6. https://research.kudelskisecurity.com/2018/10/16/reaping-and-breaking-keys-at-scale-when-crypto-meets-big-data/.
7. https://kudelskisecurity.com.
8. Centre for Research on Cryptography and Security, https://crocs.fi.muni.cz/.
9. https://chapel-lang.org/.

5.1.6 Étude de J. Kilgallin et R. Vasko (2019)

En 2019, Jonathan Kilgallin et Ross Vasko [60], deux employés de chez Keyfactor [10], ont analysé plus de 75 millions de certificats actifs qui utilisent le chiffrement RSA. Ces certificats ont été collectés entre 2015 et 2019 en parcourant l'Internet, et ont fourni 58 millions de modules RSA distincts. Ils ont découvert qu'en moyenne le module d'un certificat sur 172 était factorisable. Plus précisément, le calcul des facteurs communs a permis de factoriser 250,000 modules, et en raison de la redondance de certains modules, cela a correspondu à près de 435,000 certificats.

Par contre, en faisant la même analyse sur 100 millions de certificats distincts provenant de CT logs, ils n'ont pu factoriser que 5 modules seulement. Ils ont expliqué la différence entre les deux taux de vulnérabilité par le fait que la plupart des certificats qu'ils avaient eux-mêmes collectés venaient de matériels IoT, et que ces derniers pourraient être soumis à des contraintes de conception et à une entropie limitée. Parmi ces matériels, 50% auraient déjà été prévenus lors des études de Heninger *et al.* [52] et Hastings, Fried et Heninger [51].

Les ressources de développement et de calculs dépensées pour cette étude, hors acquisition des données, auraient totalisé moins de 3000 $. Ils en ont conclu qu'ils venaient donc de montrer, à travers cette étude, que même avec peu de ressources, on pouvait parvenir à obtenir une centaine de millions de clefs RSA et d'en compromettre des centaines de milliers.

5.2 Présentation des bases de données

Dans cette section, on va présenter les certificats RSA contenus dans les trois bases de données. Cela se fera à travers une classification en fonction de la taille de leurs modules.

5.2.1 Présentation de la base de données provenant de l'EFF

«EFF SSL Observatory» [40] est un projet de l'EFF [11] dont le but principal est d'analyser les certificats électroniques de tous les sites web utilisant le protocole HTTPS. Dans ce projet, ils collectent tous les certificats SSL accessibles au public et disponibles aux adresses IPv4 dans le but de rechercher des vulnérabilités, de documenter les pratiques des autorités de certification et d'aider les chercheurs intéressés par l'infrastructure de chiffrement du Web. Leur collecte de certificats se fait en utilisant l'outil d'analyse de réseaux Nmap [76] dans des programmes Python [93] dont les codes sources sont disponibles dans leur dépôt [12] Git.

La base de données de certificats SSL/TLS que l'on a pu télécharger sur la page web [13] du projet a été créée entre 2010 et 2011. Elle est composée de 4,021,766 certificats SSL/TLS dont 4,019,595 utilisant le chiffrement RSA. Le Tableau 5.2 donne une classification de ces certificats en fonction de la taille de leurs modules RSA.

10. https://www.keyfactor.com.
11. Electronic Frontier Foundation, https://www.eff.org/.
12. https://github.com/EFForg/observatory.
13. https://www.eff.org/fr/observatory.

Tailles	Certificats	Certificats distincts	Modules distincts
512	71,400	71,400 [100%]	56,318 [78.88%]
768	53,970	53,970 [100%]	29,127 [53.97%]
1024	2,793,370	2,793,370 [100%]	2,752,325 [98.53%]
1536	759	759 [100%]	755 [99.47%]
2048	1,064,478	1,064,478 [100%]	1,059,442 [99.53%]
3072	260	260 [100%]	256 [98.46%]
4096	34,407	34,407 [100%]	34,193 [99.38%]
8192	231	231 [100%]	231 [100.0%]
16384	9	9 [100%]	9 [100.0%]
autres	711	711 [100%]	709 [99.72%]
Totaux	4,019,595	4,019,595 [100%]	3,933,365 [97.85%]

Tableau 5.2 – ***Certificats RSA provenant de l'EFF*** – *Les pourcentages entre crochets sont les proportions des nombres de la colonne concernée par rapport à ceux de la colonne précédente.*

5.2.2 Présentation de la base de données provenant de l'ANSSI

Créée entre 2011 et 2017, cette base de données est composée de 183,804,874 certificats SSL/TLS distincts dont 72.64% utilisent le chiffrement RSA. Ces derniers sont classifiés dans le Tableau 5.3. À la différence des deux autres bases de données, pour préserver l'anonymat des utilisateurs, celle-ci ne fournit pas les adresses auxquelles les certificats ont été collectés.

Tailles	Certificats	Certificats distincts	Modules distincts
512	93,827	93,827 [100%]	70,014 [74.62%]
768	270,861	270,861 [100%]	31,258 [11.54%]
1024	3,641,242	3,641,242 [100%]	3,478,657 [95.53%]
1536	952	952 [100%]	931 [97.79%]
2048	162,312,987	162,312,987 [100%]	114,396,897 [70.48%]
2096	8,708	8,708 [100%]	8,678 [99.66%]
2432	2,191	2,191 [100%]	2,028 [92.56%]
3072	118,012	118,012 [100%]	94,073 [79.71%]
4096	17,345,968	17,345,968 [100%]	15,430,692 [88.96%]
8192	4,612	4,612 [100%]	3,788 [82.13%]
16384	174	174 [100%]	150 [86.21%]
autres	5,340	5,340 [100%]	4,512 [84.49%]
Totaux	183,804,874	183,804,874 [100%]	133,521,678 [72.64%]

Tableau 5.3 – ***Certificats RSA provenant de l'ANSSI*** – *Les pourcentages entre crochets sont les proportions des nombres de la colonne concernée par rapport à ceux de la colonne précédente.*

5.2.3 Présentation de la base de données provenant de Rapid7

Cette base de données est composée, d'une part, de certificats SSL/TLS (collectés entre janvier 2017 et janvier 2021) provenant du projet «Sonar Project» de Rapid7 [14], et d'autre part, de certificats SSL/TLS que l'on a collectés entre mars et juillet 2019 en utilisant les codes

14. https://www.rapid7.com/.

Python de «EFF SSL Observatory» sur des adresses IPv4 téléchargées de «Sonar Project». Dans ces codes, il est possible de modifier le nombre d'adresses à consulter en parallèle (8192 par défaut) et le temps maximal d'attente de la réponse de chaque requête (500 millisecondes par défaut). Ces deux paramètres devraient avoir des valeurs raisonnables, sinon l'activité de la machine effectuant les requêtes peut être considérée comme anormale et cette machine pourrait donc se faire blacklister. Il faudrait aussi penser à bien mélanger les adresses IPv4 pour éviter de scanner en même temps celles qui appartiennent à une même organisation. Une organisation pourrait en effet considérer comme activité malveillante le fait de scanner plusieurs adresses lui appartenant, surtout si les requêtes viennent de la même machine.

Le nombre des certificats provenant de «Sonar Project» étant plus important que celui des certificats qu'on a collectés, la base de données portera le nom de Rapid7. Le Tableau 5.4 donne une classification de ses certificats RSA en fonction de la taille de leurs modules.

Tailles	Certificats	Certificats distincts	Modules distincts
512	1,340,387	161,775 [12.07%]	146,500 [90.56%]
768	617,891	561,456 [90.87%]	47,083 [08.39%]
1024	71,672,669	16,180,187 [22.58%]	14,933,841 [92.30%]
1040	1,302,419	264,685 [20.32%]	242,007 [91.43%]
1232	13,977	1,358 [09.72%]	1,358 [100.0%]
1248	5,527	1,174 [21.24%]	1,155 [98.38%]
1536	41,070	11,287 [27.48%]	11,252 [99.69%]
2048	480,048,723	134,145,937 [27.94%]	110,029,246 [82.02%]
2432	17,025	5,185 [30.46%]	5,097 [98.30%]
3072	1,079,198	647,907 [60.04%]	633,230 [97.73%]
4096	21,470,526	9,140,387 [42.57%]	8,064,667 [88.23%]
8192	66,202	5,059 [07.64%]	4,838 [95.63%]
16384	433	204 [47.11%]	202 [99.02%]
autres	21,573	7,030 [32.59%]	6,705 [95.38%]
Totaux	577,697,620	161,133,631 [27.89%]	134,127,181 [83.24%]

Tableau 5.4 – ***Certificats RSA provenant de Rapid7*** *– Les pourcentages entre crochets sont les proportions des nombres de la colonne concernée par rapport à ceux de la colonne précédente.*

5.3 Recherche de vulnérabilités

5.3.1 Notre méthodologie

Dans le but d'analyser nos trois bases de données, nous allons suivre un certain nombre d'étapes similaires à celles de l'étude faite en 2016 par Hastings, Fried et Heninger [51].

1. On commencera par identifier les modules de tailles inférieures ou égales à 768 bits. De tels modules sont à priori factorisables depuis 2010 bien que la difficulté pourrait augmenter au voisinage des 768 bits [62]. On identifiera aussi les modules redondants, c'est-à-dire les modules qui sont simultanément utilisés dans des certificats SSL/TLS appartenant ou non à différentes entités.

2. Pour chacune des bases de données, on déterminera, s'il y en existe, les modules ayant des facteurs communs. Ces modules sont dits *PGCD-vulnérables* et seront déterminés en utilisant une implémentation parallélisée en PARI/GP [117] de l'algorithme «Batch GCD» donné dans l'étude de Heninger *et al.* [52] et la section 7.4 du chapitre 7. Cette étape s'achèvera avec l'utilisation de méthodes élémentaires de factorisation d'entiers telles que celle de «$p-1$» de Pollard, celle de Fermat, ou encore celle de Pollard-Rho [90].
3. Les facteurs premiers ainsi obtenus sont classés par famille de matériels. Pour chaque famille, on étudiera la distribution des nombres premiers lui appartenant à la recherche d'éventuels biais exploitables. On vérifiera aussi s'il y a des motifs communs entre certains nombres premiers d'une même famille comme ce fut le cas dans les travaux de Bernstein *et al.* [17]. Dans une telle situation, la méthode de Coppersmith [28] sera utilisée pour essayer de factoriser d'autres modules provenant de cette famille.
4. À l'aide des étapes précédentes, on essayera, d'une part, d'identifier les bibliothèques cryptographiques connues qui auraient été utilisées par les matériels vulnérables (cela nous aidera à remonter aux origines des anomalies notamment à l'aide de leurs codes sources), et d'autre part, de rétro-concevoir les méthodes de génération de paramètres cryptographiques qui ne seraient pas connues du public.
5. Enfin, on ne manquera pas de signaler toute autre anomalie (n'ayant nécessairement pas de lien avec le module) que l'on constatera dans les bases de données.

5.3.2 Modules de petites tailles

Dans le Tableau 5.2, le Tableau 5.3 et le Tableau 5.4, la présence des modules de tailles inférieures ou égales à 768 bits est due en grande partie à l'utilisation d'anciens matériels électroniques et/ou d'anciennes versions de bibliothèques cryptographiques dans lesquelles les tailles des clefs sont petites par défaut. La Figure 5.1 donne les emplacements des certificats RSA provenant de Rapid7 et ayant cette vulnérabilité.

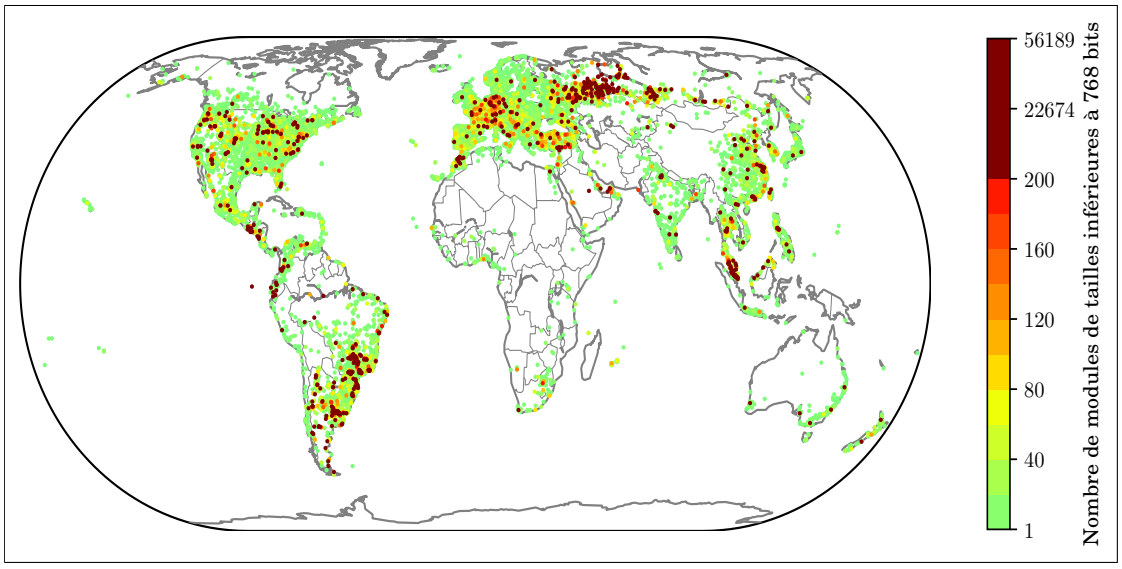


Figure 5.1 – ***Emplacements des certificats provenant de Rapid7 et contenant des modules de tailles inférieures ou égales à 768 bits.***

Elle montre qu'ils sont presque partout dans le monde, et les principaux pays et villes concernés sont cités dans le Tableau 5.5.

Pays	Modules d'au plus 768 bits	Villes où il y en a le plus
RUS	396,211	Perm (46,924) ; Tcheliabinsk (36,823) ; Iekaterinbourg (34,939)
BRA	163,454	Londrina (56,189) ; Uberlândia (16,053) ; Uberaba (8,360)
USA	117,010	Denver (5,015) ; Seattle (3,639) ; Phoenix (3,004)
ECU	96,050	Manta (35,401) ; Cuenca (22,607) ; Guayaquil (10,832)
ARG	95,144	Buenos Aires (7,919) ; Mar del Plata (3,993) ; Mendoza (2,819)
GTM	48,972	Guatemala (29,516) ; Santa Cruz del Quiché (1,580)
CHN	46,434	Guangzhou (2,841) ; Shanghai (2,809) ; Pékin (2,509)
MYS	44,150	Kuala Lumpur (5,104) ; Shah Alam (3,174) ; Petaling Jaya (2,377)
FRA	38,541	Paris (1,437) ; Rennes (259) ; Toulouse (253)
DEU	17,278	Francfort-sur-le-Main (587) ; Hambourg (526) ; Berlin (401)

Tableau 5.5 – ***Villes ayant les plus grands nombres de certificats provenant de Rapid7 et dont les tailles des modules sont inférieures ou égales à 768 bits.***

D'une part, certains chiffres de ce tableau peuvent paraître étonnants au premier abord. Par exemple, Phoenix (USA) avait une population estimée à 1,680,992 habitants [15] en 2019 alors qu'en cette même année, Seattle (USA) avait une population de 753,675 habitants. On pourrait donc s'attendre à ce qu'il y ait plus de matériels électroniques vulnérables à Phoenix qu'à Seattle. Mais cette dernière est une ville de l'État de Washington qui est un État ayant plus de matériels connectés que l'État de l'Arizona auquel Phoenix appartient [16]. D'ailleurs, le fait que la Figure 5.1 soit semblable à la Figure 5.2 pourrait montrer que les zones les plus connectées sont plus vulnérables; la Figure 5.2 étant une carte provenant de Shodan [17] et donnant les positions d'environ 4 milliards de matériels connectés dans le monde en 2014.

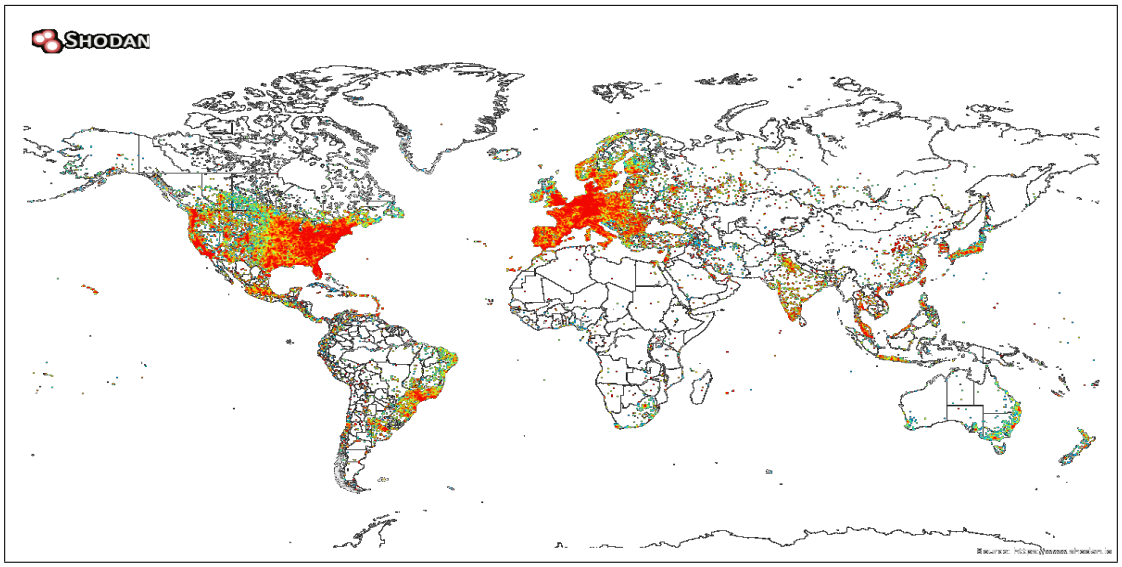


Figure 5.2 – ***Positions d'environ 4 milliards de matériels connectés en 2014*** – *L'élaboration de cette carte par John Matherly (créateur du moteur de recherche Shodan) aurait pris 17 heures : 5 heures pour collecter les données et 12 heures pour placer les points.*

15. https://www.census.gov/programs-surveys/popest/data/tables.2019.html.
16. https://www.usnews.com/news/best-states/rankings/infrastructure/internet-access.
17. https://www.shodan.io.

D'autre part, le Tableau 5.5 montre également que les pays les plus connectés ne sont pas nécessairement ceux qui ont le plus de matériels vulnérables. Dans les endroits où la population a des faibles revenus, on pourrait s'attendre à un usage généralisé de matériels anciens ou bas de gamme. Les taux de vulnérabilité de ces zones peuvent donc être plus élevés même s'il y a moins de matériels connectés. Les villes de Buenos Aires (ARG) et de Paris (FRA) pourraient être un exemple de cette situation.

5.3.3 Certificats non valides et modules redondants

En plus des modules de petites tailles, nos bases de données contiennent également des certificats expirés et toujours en activité (c'est-à-dire des *certificats non valides et actifs*), mais aussi des certificats dont les modules sont utilisés dans d'autres certificats (c'est-à-dire des *modules redondants*). La Figure 5.3 et la Figure 5.4 donnent respectivement les proportions des certificats invalides et actifs, et les modules redondants.

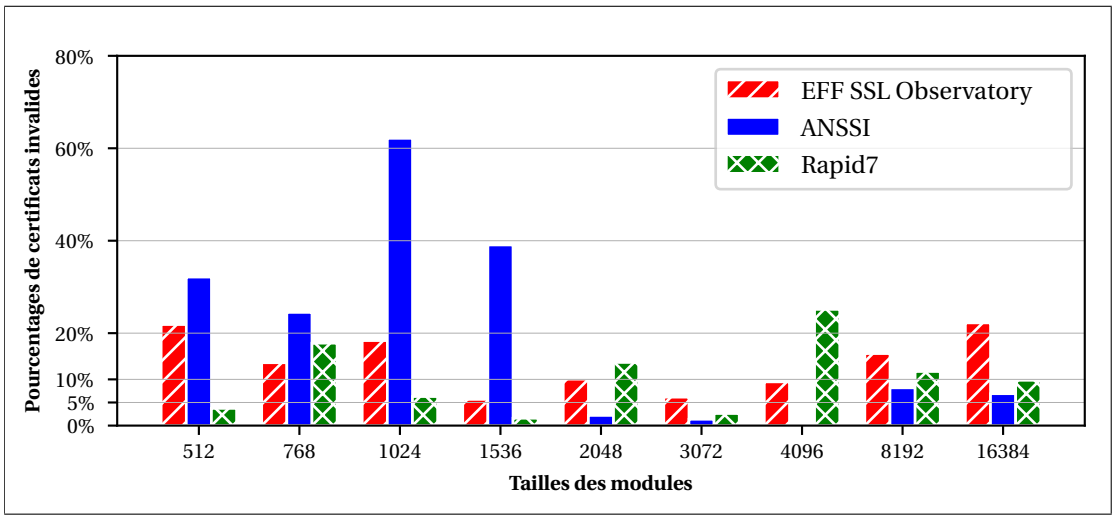


Figure 5.3 – ***Proportions des certificats invalides.***

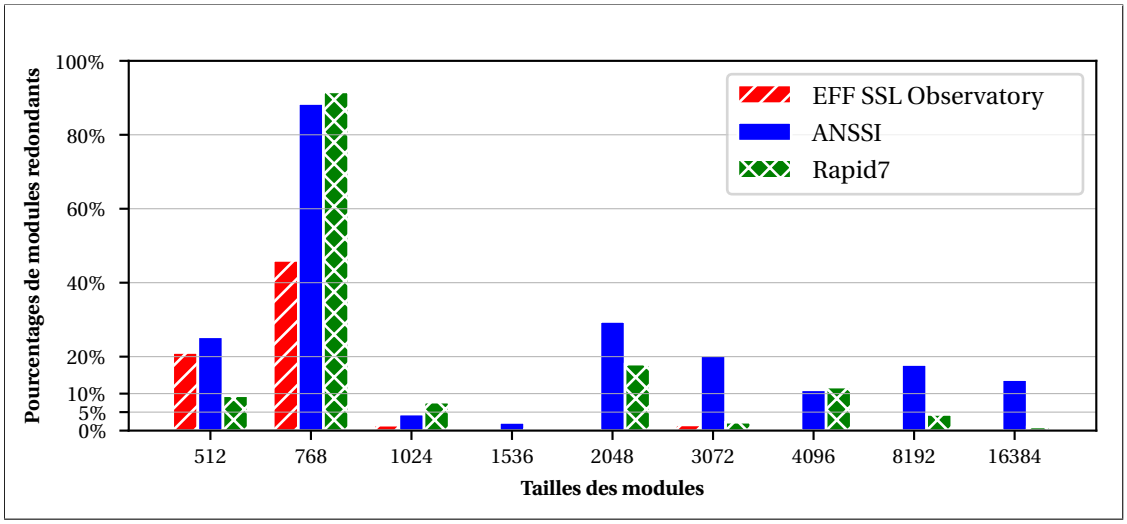


Figure 5.4 – ***Proportions des modules redondants.***

Le non-renouvellement des certificats et la non mise à jour des matériels connectés sont les causes principales de la présence des certificats non valides. Leur utilisation peut, par exemple, entraîner des problèmes dans la chaîne de vérification des certificats lors d'une tentative de connexion sécurisée.

Quant aux modules redondants, leur présence est due entre autres à l'usage des certificats par défaut dans les matériels connectés, à l'utilisation d'un mauvais générateur aléatoire de clefs et à l'usage d'un même module dans différents certificats appartenant à la même entité. Il faut noter que dans les deux premiers cas, le même module redondant se retrouve utilisé dans différents certificats appartenant à différentes entités, ce qui entraîne une perte de sécurité, car chaque entité aura connaissance des paramètres privés des autres entités. Concernant le premier cas, il est en effet courant que des fabricants de matériels connectés mettent un même certificat (par défaut) dans leurs matériels de même version. Dans une telle situation, certains mentionnent dans leurs guides de l'utilisateur que ce certificat devrait être remplacé une fois après avoir démarré le matériel. Malheureusement, la plupart des utilisateurs ne le font pas. On verra un exemple de cette situation dans la section suivante.

5.3.4 Modules PGCD-vulnérables

L'analyse des trois bases de données a aussi montré qu'elles contenaient des modules PGCD-vulnérables, c'est-à-dire des modules ayant des facteurs communs (voir Tableau 5.6). On les a déterminés avec une implémentation de l'algorithme «Batch GCD» de Heninger *et al.* [52] en PARI/GP [117]. Le code source est disponible à la section 7.4 du chapitre 7.

Bases de données	Modules distincts PGCD-vulnérables	Certificats distincts correspondants	Adresses IP correspondantes
EFF SSL Observatory	6,508	12,614	12,614
ANSSI	12,698	18,884	N/A
Rapid7	89,719	123,898	257,742

Tableau 5.6 – ***Certificats avec des modules PGCD-vulnérables.***

La majeur partie de ces modules provient des certificats contenus dans des matériels réseaux de fabricants connus. Dans le Tableau 5.7, le Tableau 5.8 et le Tableau 5.9, on procède à une classification de ces modules suivant les matériels dont ils proviennent.

Tailles des modules	Fabricants	Matériels	Certificats distincts	Modules distincts	Modules distincts PGCD-vulnérables
512	CL	?	1,285	1,285	32 [2.49%]
	ZX	?	5,474	396	15 [70.31%]
	CR	?	22	20	5 [22.73%]
1024	JP	?	30,848	29,776	5,481 [19.62%]
	ST	?	1,798	1,727	201 [12.85%]
	2W	?	3,132	2,989	89 [6.48%]
	KR	?	1,414	1,250	78 [8.56%]
	XR	?	3,740	3,463	75 [4.95%]
	DE	?	3,875	3,848	59 [1.65%]
	TL	?	1,107	116	49 [75.25%]
	TH	?	47,223	47,167	40 [0.08%]

Tableau 5.7 – ***Modules PGCD-vulnérables de la base de données provenant de l'EFF*** – *Les pourcentages entre crochets sont les proportions des modules distincts PGCD-vulnérables par rapport aux certificats distincts.* ***Les matériels et leurs fabricants ont été anonymisés.***

Tailles des modules	Fabricants ou provenances	Matériels	Certificats distincts	Modules distincts	Modules distincts PGCD-vulnérables
512	CL	?	3,809	3,804	182 [4.78%]
	ZX	?	8,143	581	22 [1.51%]
	CR	?	32	30	10 [37.50%]
	DL	?	42	42	3 [7.14%]
	NT	?	483	59	2 [0.41%]
1024	JP	?	43,415	41,652	10,071 [26.91%]
	CS	?	5,221	4,849	418 [12.83%]
	KR	?	1,974	1,531	238 [22.44%]
	ST	?	1,599	1,531	199 [12.45%]
	2W	?	6,234	3,892	184 [8.21%]
	XR	?	3,448	3,113	137 [8.15%]
	TL	?	1,345	124	73 [90.63%]
	DE	?	1,232	1,213	67 [6.41%]
	TH	?	17,868	17,858	32 [0.18%]
	DL	?	176	173	25 [15.34%]
2048	RU	?	26	25	4 [15.38%]
	CC	?	130	95	2 [3.08%]

Tableau 5.8 – ***Modules PGCD-vulnérables de la base de données provenant de l'ANSSI*** – *Les pourcentages entre crochets sont les proportions des modules distincts PGCD-vulnérables par rapport aux certificats distincts.* ***Les matériels et leurs fabricants ont été anonymisés.***

La base de données de Rapid7 étant la plus récente, on donnera dans le Tableau 5.9 plus de détails sur les matériels dont proviennent les modules PGCD-vulnérables qu'elle contient.

Tableau 5.9 – ***Modules PGCD-vulnérables de la base de données provenant de Rapid7*** – *Les pourcentages entre crochets sont les proportions des modules distincts PGCD-vulnérables par rapport aux IPs scannées. Par exemple, pour la marque AD, les 14 modules distincts de 2048 bits PGCD-vulnérables se trouvent dans 0.199% des adresses scannées utilisant des matériels «AD_m1», soit donc 28 adresses environ.* ***Les matériels et leurs fabricants ont été anonymisés.***

Tailles des modules	Modèles des dispositifs	Types des dispositifs	IPs scannées	Certificats distincts	Dates de validité	Modules distincts	Modules distincts PGCD-vulnérables
2W							
1024	?	Passerelle	2,275,051	2,271,386	07-2017/08-2037	1,939,408	2,335 [0.590%]
AD							
2048	AD_m1	Routeur	14,039	9,502	06-2015/06-2023	9,484	14 [0.199%]
AB							
2048	?	Routeur	45,429	5,798	00-0000/00-0000	5,798	11 [0.097%]
AL							
2048	?	?	4,218	3,075	01-2014/01-2024	3,054	126 [4.576%]
AS							
1024	AS_m1	Logiciel	1,274,061	1,102,443	12-2009/01-2020	1,099,770	8,104 [0.763%]
2048	AS_m1	Logiciel	9,725,530	8,905,658	01-2018/01-2028	8,894,917	2,686 [0.029%]
AC							
1024	AC_m1	Passerelle	2,403	1,823	01-2010/12-2029	1,687	86 [7.657%]
2048	AC_m2	Téléphone IP	54	37	02-2014/02-2034	34	4 [12.963%]
AV							
1024	AV_m1	Passerelle	7,532	1,102	05-2013/01-2038	1,057	77 [8.869%]
CA							
1024	?	?	1,190	900	00-0000/12-2036	870	106 [13.445%]

CL							
512	CL_m1 CL_m2 CL_m3	Routeur	11,336	5,837	00-0000/12-1971	5,807	254 [5.743%]
1024	CL_m4	Routeur	7,349	3,607	00-0000/12-1979	3,510	1,812 [42.033%]
CS							
1024	CS_m1, CS_m2	Pare-feu	10,400	4,021	03-2009/03-2019	2,891	1,191 [48.346%]
	CS_m3, CS_m4	Routeur	14,475	6,077	01-2010/12-2019	6,073	26 [0.857%]
	CS_m5, CS_m6 CS_m7	Pare-feu	999	420	07-2008/07-2018	184	26 [17.017%]
CM							
2048	?	Routeur	724	647	01-2010/01-2020	647	86 [13.950%]
CP							
2048	?	Logiciel	46	46	05-2015/05-2025	15	13 [28.261%]
CT							
1024	CT_m1	Commutateur	585,614	20,371	00-0000/00-0000	20,308	14 [0.003%]
DL							
1024	DL_m3 DL_m4 DL_m5 DL_m6	Routeur	3,663	1,177	03-2009/03-2019	1,176	13 [0.491%]
2048	? ?	Caméra	768,350	246,812	11-2016/11-2036	246,758	1,169 [6.631%]
	DL_m7 DL_m8 DL_m9	Caméra	533,733	164,547	00-0000/12-1979	164,545	565 [0.445%]
	DL_m10 DL_m11 DL_m12	Routeur	5,291	1,534	01-2009/01-2030	1,522	23 [2.041%]
DT							
2048	DT_m1	Routeur	4,753,651	672,454	12-2016/00-0000	672,423	581 [0.044%]
EP							
2048	?	Imprimante	27,481	14,796	05-2015/05-2025	14,796	10 [0.051%]
FT							
2048	FT_m1, FT_m2	Pare-feu	50,974	31,100	11-1999/11-2009	30,807	36 [0.153%]
	FT_m3	Routeur	21	18	01-2019/01-2029	18	12 [57.143%]
HL							
1024	HL_m1	Unité de contrôle	2,029	1,079	03-2008/02-2018	848	30 [2.366%]
2048	HL_m2	d'accès	4,460	2,890	12-1999/12-2009	2,875	13 [0.650%]
HP							
1024	HP_m1, HP_m2	Logiciel	12,105	8,978	12-2002/12-2022	8,976	49 [0.438%]
2048	HP_m3	Imprimante	211	149	07-2016/07-2036	149	15 [9.953%]
HW							
1024	HW_m1	Routeur	7,609	7,300	01-2016/12-2016	4,473	1,651 [44.290%]
	HW_m2	Commutateur	833	624	00-0000/12-2012	569	248 [37.935%]
2048	HW_m1	Routeur	78,095	64,469	01-2014/01-2015	59,708	6,555 [12.569%]
	HW_m5 HW_m6 HW_m7	WAP	642	633	07-2007/07-2015	625	158 [25.545%]
	HW_m2	Commutateur	183,150	46,254	00-0000/12-2012	46,241	30 [0.016%]
	HW_m3 HW_m4	Routeur	66,076	63,730	07-2019/07-2029	34,839	11 [0.215%]
IC							
2048	?	Routeur	653	479	00-0000/00-0000	470	13 [2.603%]

IN							
1024	IN_m1	Pare-feu	251	57	01-2008/12-2017	33	2 [8.765%]
JP							
1024	JP_m1	Pare-feu	14,627	5,171	01-2001/12-2010	5,070	1,292 [18.582%]
KR							
1024	KR_m1	Pointeuse	11,642	11,217	09-2006/01-2037	6,583	1,478 [26.533%]
NG							
1024	NG_m1 NG_m2 NG_m3	Pare-feu	57,994	16,170	01-2013/01-2023	14,952	2,138 [4.930%]
2048	NG_m1	Pare-feu	7,510	3,319	01-2013/01-2023	2,846	54 [3.808%]
NX							
1024	NX_m1 NX_m2	Commutateur	15,310	664	01-2014/12-2023	607	36 [4.540%]
RW							
2048	RW_m1	?	3,104	1,016	12-2009/01-2020	1,016	335 [40.561%]
RT							
512	RT_m1	Commutateur	314	18	01-2011/12-2030	18	9 [8.917%]
1024	RT_m1	Commutateur	5,445	3,731	01-2011/12-2030	2,912	415 [34.252%]
SA							
1024	?	?	10,518	2,134	11-2019/11-2020	2,129	16 [0.589%]
SG							
2048	SG_m1 SG_m2	Logiciel	1,034	819	07-2016/07-2036	814	57 [7.253%]
SF							
1024	SF_m1	Routeur	18,068	5,566	05-2017/05-2027	5,555	41 [0.448%]
ST							
1024	ST_m1 ST_m2, ST_m3	Routeur	1,985	609	01-2010/12-2019	287	126 [57.028%]
SW							
2048	SW_m1	Pare-feu	1,160,869	528,517	00-0000/01-2038	528,517	1,192 [0.224%]
TC							
2048	?	Routeur	27,170	5,798	01-2017/12-2026	5,794	30 [0.475%]
TU							
1024	?	Logiciel	1,095	934	12-2009/01-2020	934	18 [1.918%]
TL							
512	TL_m1	Routeur	156,303	128,550	08-2012/08-2022	119,767	42,631 [37.823%]
1024	TL_m2 TL_m3 TL_m4 TL_m5	Routeur	195	83	01-2006/08-2025	15	7 [26.667%]
TD							
2048	TD_m1	Logiciel	34,712	13,096	03-2019/03-2020	12,998	143 [2.022%]
XR							
1024	XR_m1 XR_m2 XR_m3	Imprimante	1,437	1,424	01-2020/01-2025	1,418	27 [1.949%]
ZX							
512	ZX_ZW2P	Pare-feu	19,019	2,017	01-2000/01-2030	257	4 [0.110%]
1024	ZX_m2 ZX_m3 ZX_m4	Commutateur	334,518	69,369	01-2015/01-2025	69,309	437 [1.580%]
2048	ZX_m5	Passerelle	2,871	970	01-2017/01-2018	496	231 [44.688%]

Dans les dernières colonnes de ces tableaux, le fait que certaines cellules aient un taux de PGCD-vulnérabilité élevé est dû, dans la majorité des cas, à l'utilisation d'un même nombre premier plusieurs fois dans différents modules. Par exemple, pour JP, parmi ses 18.582% de modules distincts PGCD-vulnérables, plus de 2,000 ont un facteur commun : il s'agit du nombre premier suivant.

```
0xc3b1f6b93f3c0e1e5ce06a36863b859112deae455d620c8e781708aac4cd82c2
  a81bac6baad42aa1dfe0448d4180e23f05e9b0f8715e2ba2a18ba0432f4a155b
```

Son poids de Hamming, c'est-à-dire la somme de ses bits, ressemble à celui d'un nombre qui aurait été généré aléatoirement. On a également vérifié qu'il n'est pas un nombre premier *isolé*, c'est-à-dire qu'il n'est pas distant d'autres nombres premiers. De tels nombres ont en effet plus de chance d'être choisis lorsque certaines méthodes de génération de nombres premiers sont utilisées (surtout celles faisant intervenir une fonction de type `nextprime`). D'ailleurs, on n'a trouvé aucune raison évidente quant au choix de ce nombre par ce fabricant de matériels réseaux.

Dans les avant-dernières colonnes du Tableau 5.7, du Tableau 5.8 et du Tableau 5.9, on remarque que les taux de modules distincts de certaines cellules sont bas, c'est-à-dire qu'il y a plus de redondance dans ces cellules. Venant de certains fabricants, ce n'est pas surprenant. Par exemple, il est mentionné dans les politiques de TL de l'époque de la création de la base de données de l'EFF que le même certificat est chargé par défaut dans tous ses matériels réseaux de même version. Cependant, on peut déduire de cette information qu'il y a un problème dans le processus de génération des modules RSA de ce fabricant. En effet, si chaque utilisateur avait gardé le certificat par défaut, il n'y aurait qu'un seul module et on ne parlerait pas de PGCD-vulnérabilité pour ce fabricant. Il est donc fort probable que le remplacement du certificat par défaut par certains utilisateurs ait conduit à la présence des facteurs communs entre différents modules. On verra un deuxième exemple de cette situation chez une autre famille de matériels dans la sous-section 5.3.7.

La Figure 5.5 donne la répartition dans le monde des matériels contenant des certificats (provenant de Rapid7) dont les modules sont PGCD-vulnérables. Comme la Figure 5.1, la Figure 5.5 ressemble aussi à ce que montre la Figure 5.6 de Shodan.

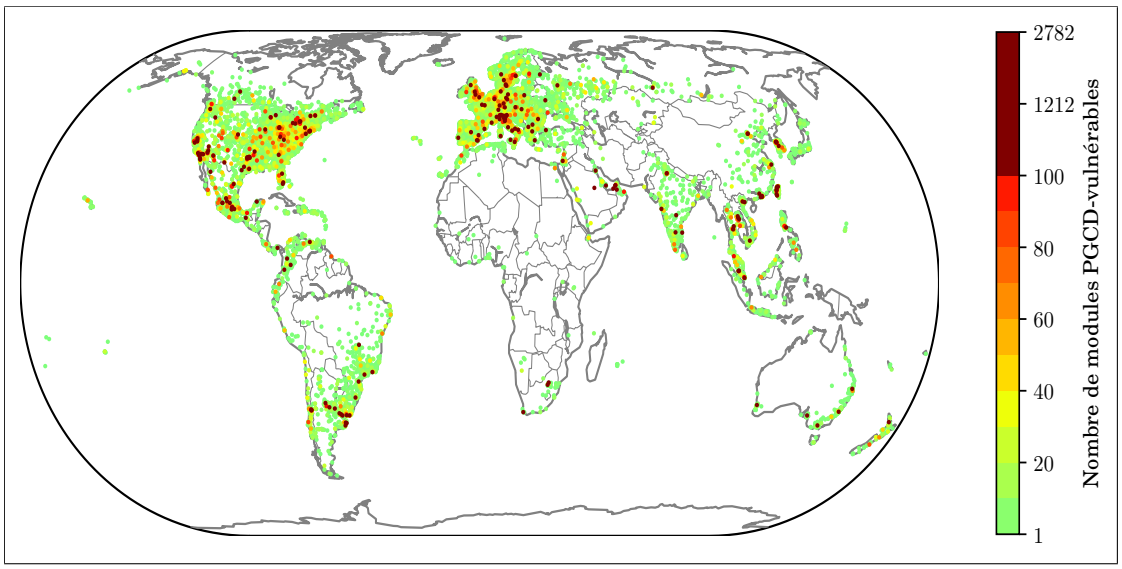


Figure 5.5 – ***Emplacements des certificats provenant de Rapid7 et ayant des modules PGCD-vulnérables.***

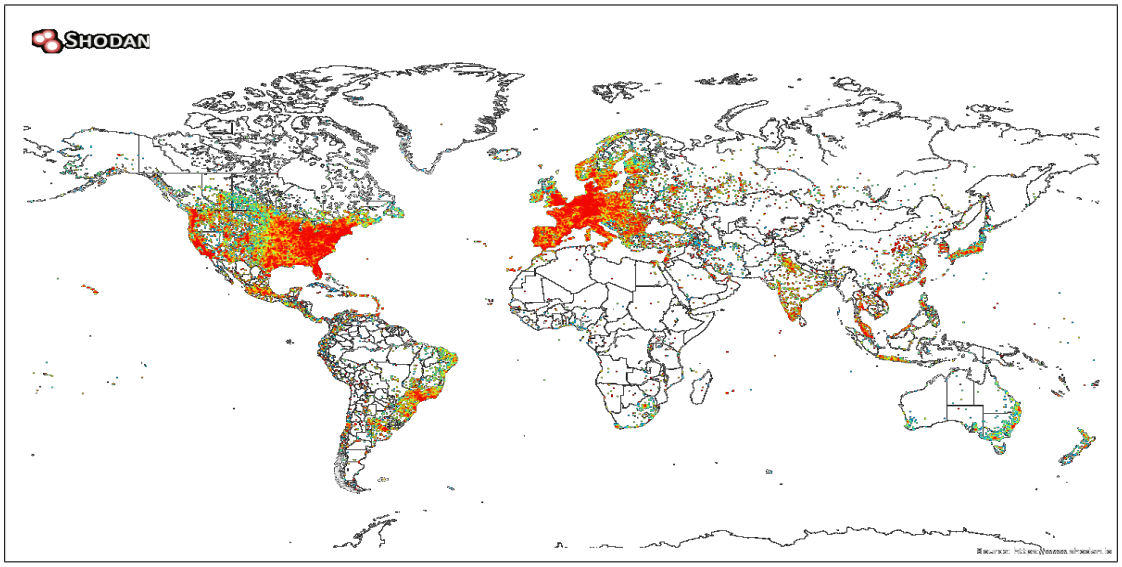


Figure 5.6 – ***Positions d'environ 4 milliards de matériels connectés en 2014.***

5.3.5 Anomalies inattendues

Parmi les modules PGCD-vulnérables de la base de données provenant de l'EFF, il y en a deux de taille 1024 bits qui ont 5 comme facteur premier. On a pu les factoriser sur un i7-4770 CPU@3.40GHz×8 avec la méthode Pollard-Rho de PARI/GP 2.12.0 [117] en moins de trois heures. Les décompositions en produit de facteurs premiers de ces deux modules sont :

$$3 \times 5^2 \times 131 \times 409 \times 3881239 \times 85746852671454053 \times 126870312865714315908719590462499 \times P$$
$$5 \times 71 \times 151 \times 11371301 \times 35208479179312547 \times 30011807242464825487 \times Q$$

où P et Q sont des nombres premiers de tailles respectives 817 bits et 865 bits. Le premier module provient d'un matériel de FT et a un facteur 25, c'est-à-dire 5^2. Même si l'objectif était d'utiliser un module RSA multi-premiers (c'est-à-dire un module composé de plus de deux nombres premiers), *l'usage d'un carré comme facteur n'est pas valide*, car cela engendre des problèmes de chiffrement et de déchiffrement. Quant au second module, il provient d'un matériel d'AT. Malgré le fait qu'il soit friable, il respecte néanmoins les critères du RSA multi-premiers de PKCS#1 [82].

Dans un autre certificat de la même base de données, on a constaté que le module et l'exposant public avaient tous les deux la même taille (1024 bits), ce qui n'est pas conforme, car dans une telle situation, l'attaque de Wiener [124, 19] permet de retrouver efficacement l'exposant privé d, donc de factoriser le module.

d = 0x23
p = 0xe3ca4f7488eda41114c462b33bc99c47eb8c6b727c0c0e97050a540e117f39c7
ab3906f1c287fc94f4386fba0d1e32613b90372d51be6c6407ebb56b35bb515b
q = 0xfd3f7aeaf7a15f28d5cce180fd85ba5aa63a05e94c46247a2da0037362bf6d74
3ca50d4567b3e979f893613f3c25a154f77cabed9a7098354ff4831d6a9c2fff

Cette valeur de d (c'est-à-dire 0x23 ou encore 35 en système décimal) fait partie des valeurs fréquemment utilisées en tant qu'exposent public. On peut donc conclure que les exposants public et privé auraient été probablement échangés par inadvertance dans ce certificat.

Enfin, on a également trouvé dans la base de données provenant de l'ANSSI un certificat dont le module RSA est sur 25 bits. Ce certificat est auto-signé, il aurait été émis en juillet 2001 et devrait expirer en juillet 2034. Son «Subject» est : «CN=MetaKey Cert» et son module est : 0x1020304, c'est-à-dire 16909060 (dont la décomposition est : $2^2 \times 5 \times 7 \times 120779$).

5.3.6 Modules ROCA-vulnérables

Comme mentionné dans la sous-section 4.5.2 du chapitre 4, en voulant générer rapidement des clefs RSA, les développeurs de RSALib choisissaient des modules dont les facteurs premiers étaient uniquement de la forme :

$$k \times M + 65537^a \bmod M$$

où M est un entier connu de tous, et k et a sont des entiers secrets. Un module n généré par cette bibliothèque était alors de la forme :

$$n = (k \times M + 65537^a \bmod M) \times (l \times M + 65537^b \bmod M)$$

Donc $n \equiv 65537^{a+b} \bmod M$, c'est-à-dire que n appartient au sous-groupe multiplicatif de $(\mathbb{Z}/M\mathbb{Z})^\times$ engendré par 65537. L'existence du logarithme discret : $\log_{65537} n \bmod M$ est donc une condition forte pour identifier les modules générés par RSALib. En effet, la probabilité pour qu'un module RSA généré aléatoirement soit dans ce sous-groupe est très faible. Par exemple, pour les modules de 512 bits, on a :

$M = 2 \times 3 \times 5 \times 7 \times \cdots \times 167$
$\phi(M) = \phi(2) \times \phi(3) \times \phi(5) \times \phi(7) \times \cdots \times \phi(167) \approx 2^{216}$ (où ϕ est l'indicatrice d'Euler) [18]
$\mathrm{ord}_M(65537) = 2454106387091158800 \approx 2^{61}$ (obtenu avec `znorder(65537, M)` en PARI/GP) [19]

c'est-à-dire que le cardinal de $(\mathbb{Z}/M\mathbb{Z})^\times$ est de 2^{216} environ alors que celui du sous-groupe engendré par 65537 est d'environ 2^{61}, donc la probabilité pour qu'un module RSA de 512 bits appartienne à ce sous-groupe (lorsque la génération est aléatoire) est autour de 2^{-154}.

Certains modules de nos bases de données vérifient cette condition (voir Tableau 5.10). Pour ceux qui sont seuls dans leurs familles tels que celui de «Nokia Siemens», on pourrait parler de hasard à la rigueur. Mais pour ceux qui sont nombreux dans leurs familles comme ceux de «Bosch Security», la probabilité est telle qu'on peut dire qu'ils viennent de RSALib.

18. $\phi(M)$ est le nombre d'entiers compris entre 1 et M (inclus) et étrangers à M.
19. $\mathrm{ord}_M(65537)$ est le plus petit entier $\alpha > 0$ tel que $65537^\alpha \equiv 1 \bmod M$.

Bases de données	Modules distincts ROCA-vulnérables	Certificats ROCA-vulnérables	Provenances des certificats
EFF SSL Observatory	1	1	Nokia Siemens
	1	1	?
ANSSI	29	32	Kapsch
	10	10	DNIe
	1	1	Nokia Siemens
	2	2	Autres
Rapid7	29	154	Bosch Security
	2	11	Appriss Health
	1	1	Atos
	5	6	Autres

Tableau 5.10 – ***Certificats ayant des modules ROCA-vulnérables.***

Le Groupe Kapsch [20] est une société internationale qui siège à Vienne (Autriche) et qui est spécialisée dans la télématique, les technologies de l'information et les télécommunications. Elle propose des solutions notamment dans les domaines du péage, de la gestion du trafic, de la mobilité urbaine intelligente, de la sécurité routière et des véhicules connectés. Tous les certificats provenant de cette société sont au nombre de 49 et sont présents uniquement dans la base de données de l'ANSSI. Ils ont dans leurs champs «`Subject`» l'extension «.by» (c'est-à-dire celle de la Biélorussie) et tous furent signés par «GlobalSign NV/SA». Parmi les 49 certificats, uniquement les 32 mentionnés dans le Tableau 5.10 sont vulnérables à ROCA. Ces derniers ont été émis entre juillet et août 2017 alors que les 17 autres leur sont antérieurs. Ils devaient expirer en juillet-août 2019, mais au final ils ont été révoqués [21] par leur autorité de certification en novembre 2017 soit environ 9 mois après la découverte de ROCA.

Le DNIe [22] est la carte d'identité électronique mise en service en Espagne depuis 2006. Elle intègre une puce contenant des informations spécifiques au propriétaire et permettant à ce dernier de se connecter à l'Administration, d'utiliser son identité électronique et de signer des documents numériquement. L'émission du premier type de DNIe a été arrêtée en 2015 et un autre type vit le jour en la même année : il s'agit du DNI 3.0, qui, à la différence du premier, permet par exemple une connexion sans fil grâce à la technologie NFC («Near Field Communication»). Les 10 certificats ROCA-vulnérables du Tableau 5.10 provenant de ces cartes ont tous été émis après janvier 2015 et sont actuellement tous expirés (car la durée maximale est de 5 ans), il serait donc fort probable qu'ils proviennent des DNI 3.0. Toutefois, certains d'entre eux avaient déjà été répertoriés et jugés ROCA-vulnérables dès juin 2017 [21].

La société «Bosch Security Systems» [23] propose des solutions dans des secteurs tels que la prévention des accès et des incendies, les annonces publiques et la sécurité vidéo intelligente au service des bâtiments intelligents (surtout les espaces publics et commerciaux). Les 154 certificats provenant de cette société ont été émis en septembre 2018 pour certains et en

20. https://www.kapsch.net.
21. https://misissued.com/batch/28/.
22. Documento Nacional de Identidad electrónico, https://www.dnielectronico.es.
23. https://www.boschsecurity.com.

octobre 2019 pour d'autres. Ils sont tous auto-signés et expireront 10 ans plus tard. Le texte «CN=mydivar» figure dans leurs champs «Subject», donc les matériels utilisés seraient des matériels de vidéosurveillance IP de type Divar [24].

Enfin, on signale qu'on a essayé de factoriser pendant plusieurs mois sans succès certains de ces modules en utilisant une implémentation en C de l'attaque ROCA.

5.3.7 Modules de «ZX_ZW2P»

Lors de la classification des modules PGCD-vulnérables de nos bases de données dans le Tableau 5.7, le Tableau 5.8 et le Tableau 5.9, on a constaté que les facteurs premiers de ceux des certificats provenant d'un matériel de ZX [25] à savoir les pare-feux «ZX_ZW2P» [128] présentaient une anomalie particulière. Les certificats que l'on a et qui proviennent de ce matériel sont tous auto-signés, ils auraient été émis en janvier 2000 et sont supposés expirer en janvier 2030. Toutefois, il est étrange que tous les certificats aient été émis à cette date alors qu'ils ne sont pas des certificats par défaut comme on le verra plus tard. Cette date pourrait donc être une date par défaut que le matériel doit marquer sur le certificat lorsqu'il le génère, ou bien elle serait la date que le matériel prend lors de son premier démarrage.

Les facteurs premiers de quelques uns des modules PGCD-vulnérables contenus dans des certificats provenant des «ZX_ZW2P» sont les suivants :

p_{01} = 0xd1aaebc089f709efd77dd07e3003800ac53f678d6b7bb5a7079f1785359ef001
p_{02} = 0xc9efd77dd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff9d
p_{03} = 0xd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903bfc2e6b
p_{04} = 0xff903bfc2e604d8dd0912ffa77dd3d5a05b07f76677f0fb212a50160a99e7855
p_{05} = 0xcd8dd0912ffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95d17
p_{06} = 0xeffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576e1
p_{07} = 0xdccb73a576b30595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c5b
p_{08} = 0xc595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c4687d309df9ccd
p_{09} = 0xf52e0ff0751acfb461dd0df7429299e6d6f97a296c4687d309df9c101cfd166b
p_{10} = 0xc68b0035b98765e044183db6f8d4ad316b8e98b8351ff41c86152bb00f8e87d5
p_{11} = 0xf51ff41c86152bb00f8e87b07f5ca4271db1e1490b89afe00f1342fa51e7205f
p_{12} = 0xdb1e1490b89afe00f1342fa51e71efa0e22e5d3af0b0d2b92947b0199f02513
p_{13} = 0xef0b0d2b92947b0199f0250e5a604e6d80493bd154eb39b689aac0345e732047

1. En considérant ces nombres premiers comme des chaînes de caractères hexadécimaux, on remarque que certains d'entre eux ont des sous-chaînes en commun (en couleurs). Quand on regarde de plus près, il semble s'agir d'un décalage de caractères, car on peut distinguer deux ordres de succession des couleurs : Rouge-Bleu-Violet-Orange-Cyan et Vert-Brun-Rose.

24. https://www.boschsecurity.com/xc/en/solutions/video-systems/divar/.
25. https://www.zyxel.com.

Figure 5.7 – ***Ordres de succession des couleurs.***

2. On remarque aussi que le premier caractère hexadécimal de chacun de ces nombres premiers est `c` ou `d` ou `e` ou `f`. Ceci pourrait être la conséquence d'un changement volontaire en 1 de leurs premiers ou deux premiers bits de poids fort.
3. Enfin, bien que certains de ces nombres premiers aient des sous-chaînes en commun, les derniers octets (octets de poids faible) des uns diffèrent de ceux des autres. Ceci pourrait s'expliquer par l'usage d'une fonction de type `nextprime` [26] qui ne modifie que certains bits de poids faible.

À partir de ces remarques précédentes, on peut conjecturer l'Algorithme 5.1 sur la façon dont les nombres premiers de 256 bits utilisés par les «ZX_ZW2P» auraient été générés.

Algorithme 5.1 : Génération de nombres premiers des «ZX_ZW2P»

Entrées : une chaîne prédéfinie G de caractères hexadécimaux. *On l'appelle la graine.*
Sorties : un nombre premier p de 256 bits.

```
1 s ← sous-chaîne de G de 64 caractères hexadécimaux // cela fait 256 bits
2 Mettre le premier ou les deux premiers bits de s à 1.
3 p ← nextprime(s)
4 si p est sur 256 bits alors
5 |   retourner p
6 fin
```

Pour vérifier l'exactitude de l'Algorithme 5.1, on s'est servi de l'ordre de succession des couleurs pour retrouver les deux sous-chaînes S_1 et S_2 de la supposée graine G.

$S_1 =$ `1aaebc089f709efd77dd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903bfc2e604d8dd0912ffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576b30595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c4687d309df9c101cfd`

$S_2 =$ `68b0035b98765e044183db6f8d4ad316b8e98b8351ff41c86152bb00f8e87b07f5ca4271db1e1490b89afe00f1342fa51e71efa0e22e5d3af0b0d2b92947b0199f0250e5a604e6d80493bd154eb39b689aac0345e73`

On a ensuite généré tous les nombres premiers pouvant être obtenus à partir des deux sous-chaînes S_1 et S_2 en utilisant l'Algorithme 5.1, ce qui nous a permis de factoriser d'autres modules en effectuant des calculs de PGCD [27] : l'Algorithme 5.1 serait donc correct. On a également supposé que les 32 derniers caractères de S_1 et S_2 pourraient être les 32 premiers caractères des facteurs premiers de certains modules. Puis, on a utilisé la version «the most significant bits» de la technique de Coppersmith [28, 29], ce qui a conduit à la factorisation d'autres modules. À partir des facteurs premiers de ces nouveaux modules, les sous-chaînes S_1 et S_2 ont ainsi pu être étendues. Ce processus a été répété jusqu'à ce qu'il ne soit plus

26. `nextprime`(x) renvoie le plus petit nombre premier p tel que $p \geqslant x$.
27. Plus Grand Commun Diviseur.

possible de factoriser d'autres modules. Les sous-chaînes S_1 et S_2 dans leurs extensions se sont rencontrées et ont donné la sous-chaîne suivante :

S_3 = `c798d9888f377d77d3480be06061eda02af6971b4fdd10543634ab0429dccec6`
`89dc458454d7e81ca0665c6b5620944c5e4123f1e2fa59dd19214304cde799a0`
`68b0035b98765e044183db6f8d4ad316b8e98b8351ff41c86152bb00f8e87b07`
`f5ca4271db1e1490b89afe00f1342fa51e71efa0e22e5d3af0b0d2b92947b019`
`9f0250e5a604e6d80493bd154eb39b689aac0345e73203bcb9c0a768df0b827f`
`99770bce7c7be5b6c002458bc4c64aac5733f3659712f8e7537997e6da98a364`
`1433237a8ac783b78110a2b46d751de23c28fd4b1983b4f1d54ee9030f939f91`
`497c2a86a82c968c595cb14e03137ead10a4ae77f90b8ea9d5dcb61d81aad3ff`
`5382c96bede99e2c844f5cc25df5f16d0c0f4fa7ab04e3191aaebc089f709efd`
`77dd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903bfc2`
`e604d8dd0912ffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95`
`ccb73a576b30595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c468`
`7d309df9c101cfd15fcb6ae781f0c09e5c899f3bd13f8f4857531701d89283c7`
`4a1a0d1095f4deb469cbae2cb501a97a0b0f42a6b859a2029da55613df1ec539`
`cdcd8be42e70914f11241b2252e00829225d52d86f21840ac4caf47c4d239585`
`cb7b6fbfb10790d0899370f6520c815d6d827f13349f438a65d12eb6b52a7141`
`d0548855a724d47270ce438914eb84e1bde4c4573d1491a89a13db1aeef77cba`
`f737a40e89b430b77e26ae6fd2fd15135df9f8e8734c56078385cdba421a50e1`
`42a6bb1a385e8dda120b1cb98434d20438443aa87ccb1c889dd58acdaac474bb`
`e63246e9735ef8c`

Comme nous pouvons le constater dans le Tableau 5.11, les modules qu'on a factorisés sont peu nombreux, mais à cause de leur redondance, ils représentent la majeur partie des certificats provenant des «ZX_ZW2P». À la différence de certains fabricants tels que TL (voir sous-section 5.3.4), ces redondances ne sont pas dues à un certificat par défaut, mais plutôt à l'algorithme de génération employé qui limite considérablement le nombre de nombres premiers à choisir. En effet, dans le guide de l'utilisateur[28] (pages 298-300) de «ZX_ZW2P», il est dit que le certificat par défaut est commun à tous les matériels et que son «`Subject`» est «`CN=ZyWALL 2 Plus Factory Default Certificate`», ce qui n'est pas le cas de la majorité des certificats que nous avons ici.

Bases de données	Tous les certificats	Certificats distincts	Modules distincts	Modules distincts factorisés
EFF SSL Observatory	5,474	5,474	396	67 [83.10%]
ANSSI	8,143	8,143	581	83 [85.74%]
Rapid7	19,019	2,017	257	42 [46.36%]

Tableau 5.11 – ***Certificats provenant des «ZX_ZW2P»*** – *Les pourcentages entre crochets sont les proportions des modules distincts factorisés par rapport aux certificats (non distincts).*

28. https://manualzz.com/doc/3605335/zyxel-zywall-2-plus-user-s-manual.

Certains des modules restants ont pu être factorisés directement avec CADO-NFS [115] et on remarqua que leurs facteurs premiers n'avaient pas de lien évident avec la chaîne S_3. Cela nous a permis de voir que les modules qui avaient été précédemment factorisés provenaient en réalité de matériels appartenant à certaines classes d'adresses MAC (voir Tableau 5.12).

Adresses MAC	**Quantités de matériels vulnérables**		
	EFF SSL Observatory	ANSSI	Rapid7
`00:13:49:XX:XX:XX`	376	447	1,732
`00:19:CB:XX:XX:XX`	2,956	3,338	3,050
`00:23:F8:XX:XX:XX`	1,114	1,553	2,019
`40:4A:03:XX:XX:XX`	103	1,421	1,789
`50:67:F0:XX:XX:XX`	0	223	184
`C8:6C:87:XX:XX:XX`	0	0	44
Totaux	4,549	6,982	8,818

Tableau 5.12 – ***Matériels «ZX_ZW2P» vulnérables.***

Pour affiner davantage les résultats, on a essayé d'obtenir le code source du micrologiciel des «ZX_ZW2P» auprès du fabricant[29], mais malheureusement la réponse fut négative puisque la dernière mise à jour du produit concerné remonte à février 2011 et la licence GPL les oblige à garder le code source public au moins 3 ans seulement.

Il faut aussi noter que des certificats provenant d'un autre fabricant du nom de NT[30] présentent la même anomalie que les certificats «ZX_ZW2P» que l'on vient d'analyser. NT, disparue vers 2013, était une entreprise du secteur des télécommunications dont le siège social se trouvait à Toronto (Canada). Ses matériels concernés (dont 5 provenant de la base de données de l'EFF et 3 provenant de celle de l'ANSSI) ont des certificats dont le «`Subject`» est de la forme «`CN=Business Secure Router MAC`» avec «`MAC`» commençant par `00:19:E1` ou `00:17:D1`. Les modules concernés dans cette famille sont également tous sur 512 bits, cependant les facteurs premiers obtenus n'ont pas permis d'étendre S_3.

5.3.8 Une potentielle tentative d'usurpation d'identité

La recherche des modules PGCD-vulnérables a aussi révélé la présence de modules (de tailles 2048 bits ou 4096 bits) divisibles par de petits nombres premiers. Cela nous paraissant étrange, on s'est particulièrement intéressé aux certificats qui les contiennent. Au premier abord, on pensait qu'il s'agissait de modules RSA multi-premiers. Cependant, on s'est rendu compte que ce n'était pas le cas lorsque la vérification des signatures numériques a échoué. En réalité, ces certificats sont des copies *presque parfaites* d'authentiques certificats. Ils diffèrent des originaux de seulement quelques octets. Prenons l'exemple de la copie *presque parfaite* que l'on a du certificat de «Comodo Japan RSA DV CA». La seule différence qu'il y a entre cette copie et l'original se situe au niveau des modules comme on peut le constater ci-dessous (V étant le module RSA du certificat original et F celui de la copie).

29. https://www.zyxel.com/form/gpl_oss_software_notice.shtml.
30. https://www.nortel-us.com/.

V = 0x85659dd929b5f9b4b84f05ba2d969b1febb19a2f665abeb9ba7788a2c45e073a
37567c621e7fdbac1439d0ffd2d97f0154bc147765fb69f4b383804bb663a014
6ac232ef53f8b0da50b49a293dce96c6f58d175ece0b87cd0e97b1152edc3499
263c24d251a2eccbcbd43e6d05962cdafab6e04fc544164ab6e503885db5fb49
fd820ebd768383aff4f2ea17f16f981943f99f1db6371648369f111b20487695
2ce1f90c3d7718eb0fcf3bbac3bdfe1df7601af4fe2e0a49b0076de65f5d88c4
bd9cd7da61937c506a22f12ab4ef30cda4a33b115f8ca91a264b05db2d3adc38
81b12c27e389bafb8271908f41b26a2351da16b054755492aaa0197bc9a12c09
F = 0x85659dd929b5f9b4b84f05ba2d969b1febb19a2f665abeb9ba7788a2c45e073a
37567c621e7fdbac1439d0ffd2d97f0154bc147765fb69f4b383804bb663a014
6ac232ef53f8b0da50b49a293dce96c6f58d175ece0b87cd0e97b1152edc3499
263c24d251a2eccbcbd43e6d05962cdafab6e04fc544164ab6e503885db5fb49
fd820ebd768383aff4f2ea17f16f981943f99f1db6371648369f111b20487695
2ce1f90c3d7718eb0fcf3bbac3bdfe1df7601af4fe2e0a49b0076de65f5d88c4
bd9cd7da61937c506a22f12ab4ef30cda4a33b115f8ca91a264b05db2d3adc38
81972c27e389bafb8271908f41b26a2351da16b054755492aaa0197bc9a12c09

C'est le changement de «b1» en «97» qui causa la divisibilité de F par de petits nombres premiers. Ce changement se manifeste dans les versions PEM («Privacy Enhanced Mail») des certificats par la modification d'une lettre x en X. Il pourrait donc s'agir d'une erreur de copie du certificat lors de sa collecte. Cependant, le problème n'est observé que chez des modules de 2048 bits ou 4096 bits, contenus dans des certificats appartenant à des autorités de certification telles que «Comodo Japan RSA DV CA», «Go Daddy Root Certificate Authority - G2», «RapidSSL TLS RSA CA G1», «TERENA SSL CA 3» ou encore «DigiCert Global Root CA». Par conséquent, il est fort probable qu'il s'agisse d'une tentative d'usurpation d'identité bien que l'on ignore comment un attaquant pourrait exploiter un tel certificat. Il est à noter qu'aucun de ces modules n'a intégralement pu être factorisé.

5.4 Conclusion

Dans ce chapitre, on a analysé plusieurs centaines de millions de certificats éléctroniques provenant de différentes sources : «EFF SSL Observatory» (2010), ANSSI (2011-2017) et «Sonar Project» (2017-2021). On a utilisé une méthodologie similaire à celle de Hastings, Fried et Heninger [51]. Cette analyse a montré plusieurs anomalies dont certaines sont semblables à celles qui avaient déjà été signalées dans des études précédentes (qu'on a mentionnées dans la section 5.1). Mais à la différence de ces études, on a été en mesure de factoriser plusieurs modules RSA de 2048 bits. Plus précisément, 14,765 modules de 2048 bits (correspondants aux certificats de 18,139 adresses).

On commença par l'identification des modules de tailles inférieures à 768 bits, car de tels modules sont factorisables depuis 2010. Chacune des trois bases de données que l'on a analysées en contient en quantité importante (voir Tableau 5.13). On a aussi constaté (voir Figure 5.2 et Figure 5.3) que les trois bases de données contiennent des certificats invalides mais toujours en activité. Elles contiennent également des certificats dont les modules sont

PGCD-vulnérables, c'est-à-dire des modules que l'on a factorisés à l'aide des calculs de PGCD.

Bases de données	Certificats de modules de tailles inférieures à 768 bits	Certificats de modules PGCD-vulnérables
EFF SSL Observatory	125,370 (3.12%)	12,614 (0.31%)
ANSSI	364,688 (0.20%)	1,187,290 (0.65%)
Rapid7	1,958,278 (0.34%)	1,550,382 (0.27%)

Tableau 5.13 – ***Récapitulatif.***

La Figure 5.1 et la Figure 5.4 donnent les emplacements des certificats ayant ces vulnérabilités. On a observé sur ces figures, que les proportions étaient élevées d'une part dans des endroits très connectés et d'autre part dans des endroits dont les populations ont des faibles revenus. Dans le premier cas, il est évident que plus il y a de matériels connectés (donc plus de certificats) plus la possibilité des vulnérabilités augmente. Dans le second cas, les populations ayant des faibles revenus ont tendance à utiliser des matériels anciens ou des matériels bas de gamme, ce qui augmente également la possibilité des vulnérabilités. On donne dans le Tableau 5.14, une liste de quelques matériels vulnérables avec leurs prix.

Marques	Produits	Dates de sortie - Date de dernière mise à jour	Prix
AD	AD_m1	03/2015 - N/A	< 400 $ (Amazon, Neuf) < 100 $ (Amazon, Occasion)
AC	AC_m1 AC_m2	02/2013 - N/A 10/2013 - 07/2021	< 200 $ (Ebay, Occasion) < 140 $ (voipsupply, Neuf)
CL	CL_m1 CL_m2	07/2010 - 03/2014 04/2009 - 12/2011	< 240 $ (Amazon, Neuf) < 35 $ (Ebay, Occasion)

Tableau 5.14 – ***Prix de certains matériels vulnérables.***

L'existence des modules de petites tailles dans la base de données de l'EFF est probablement due au fait qu'au moment où cette base de données a été constituée, la transition vers des tailles de clefs plus grandes que 1024 bits n'était pas largement appliquée. Quant à ceux qui sont dans les deux autres bases de données, ils viennent d'anciens matériels en activité dont la plupart ne sont plus supportés par leurs fabricants, mais aussi d'anciennes versions de bibliothèques cryptographiques.

Aucun des modules PGCD-vulnérables de la base de données de l'EFF n'a une taille qui dépasse 1024 bits. Par contre, on a constaté que cette vulnérabilité s'est étendue dans les deux autres bases de données à des modules de 2048 bits, car l'utilisation d'une telle taille devient de plus en plus importante. Cependant, comme on le verra dans le chapitre 6, la PGCD-vulnérabilité n'est pas tant liée à l'abondance des modules, mais plutôt aux conditions dans lesquelles les modules sont générés.

Chapitre 6

Recherche des causes de la PGCD-vulnérabilité

Sommaire

Avec les résultats donnés dans la section 4.3 du chapitre 4 sur la distribution des nombres premiers, on sait que si deux nombres premiers relativement grands sont choisis au hasard, la probabilité pour qu'ils soient égaux est presque nulle. Par exemple, la quantité de nombres premiers de 512 bits (resp. 1024 bits) est d'environ 2^{441} (resp. 2^{891}). Mais dans le chapitre 5, l'analyse des trois bases de données ne reflète pas ce résultat compte tenu de la quantité de modules PGCD-vulnérables trouvés. On peut donc affirmer qu'il y eut un dysfonctionnement lors du choix de ces modules.

L'objectif de ce chapitre sera de remonter aux causes de cette vulnérabilité en analysant les éléments ayant contribué à la génération de tels modules.

En guise de rappel, la génération des facteurs premiers d'un module RSA nécessite généralement deux éléments : un générateur de nombres pseudo-aléatoires (ou PRNG pour «Pseudo Random Number Generator») et un algorithme de génération de nombres premiers. Ce dernier transforme la sortie du PRNG en un nombre premier. Donc s'il y a un problème, il ne pourrait venir que de l'un de ces éléments au moins. Étant donné que ces éléments peuvent changer d'une famille de matériels à une autre, et compte tenu du nombre de matériels vulnérables trouvés, on ne va pas étudier le processus de génération de clefs RSA de chacun de ces matériels. On va plutôt exploiter le fait qu'une grande partie d'entre eux utilisent des bibliothèques cryptographiques connues à code source ouvert (ou «Open Source») comme on peut en voir quelques exemples dans le Tableau 6.1. Autrement dit, les études se feront sur des bibliothèques qu'ils utilisent majoritairement bien qu'il puisse y avoir quelques rares familles de matériels qui se servent de bibliothèques développées en interne (le cas des «ZX_ZW2P» dans la sous-section 5.3.7 du chapitre 5 en est un exemple).

La bibliothèque cryptographique utilisée par la plupart des matériels vulnérables trouvés est assurément OpenSSL [116]. D'une part, les informations contenues dans le Tableau 6.1 le confirment ; ces informations proviennent des codes sources des micrologiciels que l'on a pu télécharger sur les sites web des fabricants des matériels concernés.

Matériels		**Bibliothèques**		**Noyaux Linux**	
Nom (Dernière mise à jour)		**Version (Date de sortie)**		**Version (Date de sortie)**	
E2000	(04/2014)	OpenSSL 0.9.8a	(10/2005)	2.4.20	(11/2002)
SRX5308	(03/2017)	OpenSSL 0.9.8zc	(10/2014)	2.6.21	(04/2007)
DCS-935L	(03/2019)	OpenSSL 1.0.1t	(05/2016)	2.6.30	(06/2009)
DCS-5222LB1	(10/2016)	OpenSSL 1.0.1p	(07/2015)	3.0.8	(10/2011)
RV130W	(11/2015)	OpenSSL 1.0.2d	(07/2015)	2.6.31	(09/2009)

Tableau 6.1 – ***Bibliothèques cryptographiques et noyaux Linux utilisés par quelques matériels vulnérables trouvés.*** *Ces informations viennent des codes sources de leurs micrologiciels.*

D'autre part, lors de l'analyse des processus de génération de clefs RSA d'OpenSSL (dans la section suivante), on aura constaté que les facteurs premiers x de chaque module RSA que cette bibliothèque génère, vérifient tous la condition suivante :

$$x \neq 1 \bmod y \text{ pour tout nombre premier } y \text{ tel que } 2 < y < 17881 \tag{6.1}$$

L'immense majorité des facteurs premiers de nos modules PGCD-vulnérables satisfont cette

condition (voir section A.1). On conclût donc qu'ils auraient été générés par OpenSSL. Cela nous a permis de savoir qu'OpenSSL était également utilisée par d'autres familles de matériels dont on ne disposait pas des codes sources des micrologiciels.

Signalons que la condition 6.1 influe aussi sur les modules, car elle impose des contraintes sur leurs facteurs premiers. On peut s'en servir pour savoir s'il est probable qu'un matériel utilise OpenSSL pour générer ses clefs RSA. Par exemple, p et q étant congrus à 2 modulo 3, alors $n = p \times q$ sera congru à 1 modulo 3. Il est donc probable qu'une famille de matériels utilise OpenSSL quand tous ses modules sont congrus uniquement à 1 modulo 3. Cette probabilité est même grande malgré l'existence des facteurs premiers RSA de type Dirichlet, c'est-à-dire des nombres premiers de la forme $ak+b$ où a et b sont étrangers. De tels facteurs sont en effet autorisés par certaines normes telles que la norme IEEE-P1363 [3], mais leur utilisation reste rare.

Utilisent-elles OpenSSL ?	
Oui	**Non**
2W NG	AC
CL RT	HW
CS ST	JP
DL SW	KR
DT TL	ZX

Tableau 6.2 – ***Classification des familles selon qu'elles utilisent ou non OpenSSL.***

6.1 Versions d'OpenSSL à analyser

Au vu des informations contenues dans le Tableau 6.1, notre étude portera sur les versions postérieures à la version 0.9.8. De cette version à la version 1.1.0l, on verra, dans la section 6.2, qu'il n'y a pas eu de changements importants au niveau du générateur de clefs, et que c'est à partir de la version 1.1.1 (voir section 6.3) que le changement fut radical. L'historique des versions [1] d'OpenSSL est donné dans le Tableau 6.3.

Version	Date de sortie	Dernière version mineure	
0.9.8	08/2005	0.9.8zh	(12/2015)
1.0.0	05/2010	1.0.0t	(12/2015)
1.0.1	03/2012	1.0.1u	(09/2016)
1.0.2	01/2015	1.0.2u	(12/2019)
1.1.0	08/2016	1.1.0l	(09/2019)
1.1.1	09/2018	N/A	

Tableau 6.3 – ***Versions majeures d'OpenSSL.***

1. https://www.openssl.org/source/old/.

Au cours de l'étude des différentes versions du générateur, on supposera que le système d'exploitation utilisé est un Linux 32 bits, car cela correspond au cas de la totalité des matériels vulnérables que l'on a pu identifier.

6.2 Génération des clefs RSA par les versions allant de 0.9.8 à 1.1.0l

Pour générer une clef RSA en utilisant OpenSSL avec les paramètres par défaut, il suffit d'utiliser la commande suivante : `openssl genrsa`.

Algorithme 6.1 : La fonction `genrsa`

```
1 S ← stat(RFILE)
2 si la version est entre 0.9.8 et 0.9.8e alors rand_add(S, 0)
3 si RFILE existe alors
4 |   si la version est entre 0.9.8f et 1.1.0l alors rand_add(S, 0)
5 |   rand_add(RFILE, 1024)
6 fin
7 rand_poll()
8 Générer les paramètres de la clef RSA.
```

La fonction `genrsa` est implémentée dans le fichier `apps/genrsa.c`, et des détails de son contenu sont donnés par l'Algorithme 6.1. Lorsqu'elle est appelée, dans un premier temps, elle se sert du fichier `RFILE` (d'emplacement `$HOME/.rnd`) comme source d'entropie pour modifier l'état interne du générateur en utilisant la fonction `rand_add` avec les propriétés et le contenu de ce fichier. Ces propriétés sont obtenues avec la fonction `stat` [2] de C et leur utilisation dépend de la version d'OpenSSL employée. Dans l'Algorithme 6.1, la ligne **4** concerne les versions allant de 0.9.8f à 1.1.0l, c'est-à-dire que ces versions n'ajoutent les propriétés de `RFILE` à l'état interne du générateur que si `RFILE` existe. Par contre, la ligne **2** (se trouvant dans les versions comprises entre 0.9.8 et 0.9.8e) ajoute les octets renvoyés par `stat` même si `RFILE` est absent. On verra lors de nos expérimentations que l'absence de ce fichier a des conséquences sur les modules générés.

Algorithme 6.2 : La fonction `rand_poll`

```
1 R ← 32 octets de /dev/urandom
2 rand_add(R, 32)
3 rand_add(getpid(), 0)
4 rand_add(getuid(), 0)
5 T ← temps écoulé en secondes depuis le 01/01/1970
6 rand_add(T, 0)
```

Ensuite, avant de choisir les paramètres de la clef RSA, on s'assure que le générateur a accumulé suffisamment d'entropie en appelant la fonction `rand_poll` (voir Algorithme 6.2).

2. `stat` est une fonction permettant d'obtenir l'état (propriétés) du fichier passé en paramètre.

Cette fonction modifie l'état interne du générateur avec le temps système (en secondes), l'identifiant du processus en cours (ou PID pour «Process ID»), l'identifiant de l'utilisateur (ou UID pour «User ID») ainsi que 32 octets de données extraites de `/dev/urandom`.

Enfin, la dernière étape consistera à générer les paramètres de la clef. La valeur par défaut de l'exposant public e est 65537. Le module aura des facteurs premiers p et q (où $p \neq q$) tels que $(p-1) \times (q-1)$ et e sont étrangers. Chaque facteur premier est généré par la fonction `BN_generate_prime_ex` (voir Algorithme 6.3) du fichier `crypto/bn/bn_prime.c`.

Algorithme 6.3 : Génération d'un facteur premier d'un module RSA par OpenSSL

Entrées : la taille k en octets du nombre premier qui doit être généré.
Sorties : un nombre premier B facteur d'un module RSA.

1 T ← temps écoulé en secondes depuis le 01/01/1970.
2 rand_add(T, 0)
3 A ← rand_bytes(k) et mettre le bit de poids fort de A à 1.
4 Prendre le nombre impair B (où $B \geqslant A$) tel que B et $(B-1)$ ne soient divisibles par aucun des 2047 premiers nombres premiers impairs.
5 Vérifier la primalité de B en utilisant des tests de Miller-Rabin.
6 **si** B n'est pas premier **alors** aller à l'étape **3**.
7 **return** B

La taille par défaut du module dépend de la version d'OpenSSL utilisée : elle est de 512 bits entre 0.9.8 et 1.0.1c, 1024 bits entre 1.0.1d et 1.0.1u, et 2048 bits entre 1.0.2 et 1.1.0l.

Algorithme 6.4 : La version de `rand_add` exécutée lors de l'appel à `genrsa`

Entrées : une chaîne $b = b_0 \| b_1 \| \cdots \| b_{n-1}$ où chaque b_i fait 20 octets sauf éventuellement pour b_{n-1}.

1 oldmd ← mdgst
2 **pour** $i \in [0; n-1]$ **faire**
3 m ← mdgst
4 s ← size(b_i) // taille du i-ème bloc
5 k ← (stidx+s)-1023
6 **si** k > 0 **alors**
7 m ← m ‖ state[stidx...1022] ‖ state[0...(k-1)]
8 **sinon**
9 m ← m ‖ state[stidx...(stidx+s-1)]
10 **fin**
11 mdgst ← SHA1(m ‖ b_i ‖ mdctr)
12 mdctr[1] ← mdctr[1] + 1
13 **pour** $j \in [0; s-1]$ **faire**
14 state[stidx] ← state[stidx] ⊕ mdgst[j] // mise à jour de state
15 stnum ← Max(Min(stnum, 1023), stidx+1)
16 stidx ← (stidx+1) mod 1023
17 **fin**
18 **fin**
19 mdgst ← mdgst ⊕ oldmd
20 **si** ntrpy < 32 **alors** ntrpy ← ntrpy + add

Le nombre d'itérations du test de Miller-Rabin que l'on effectue à la ligne **5** est donné par le tableau 4.4 de «Handbook of Applied Cryptography» [78, p.148] pour une probabilité d'erreur inférieure à 2^{-80}.

Algorithme 6.5 : La version de `rand_bytes` exécutée lors de l'appel à `genrsa`

Sorties : une chaîne $b = b_0 \| b_1 \| \cdots \| b_{n-1}$ de num octets. Chaque b_i est un bloc de 10 octets sauf éventuellement pour b_{n-1}.

```
1  oldmd ← mdgst
2  /* Mélanger l'état interne du générateur lors du premier appel     */
3  si mdctr[0] = 0 alors
4  |  pour n ∈ [0;51] faire rand_add("....................", 20, 0)
5  fin
6  /* Construction de b                                               */
7  pour i ∈ [0; n − 1] faire
8  |  m ← ""
9  |  si i = 0 alors
10 |  |  m ← m ‖ getpid()
11 |  |  si la version est entre 1.1.0 et 1.1.0l alors
12 |  |  |  u ← temps écoulé depuis le 01/01/1970 en secondes
13 |  |  |  v ← temps écoulé depuis le 01/01/1970 en microsecondes
14 |  |  |  m ← m ‖ u ‖ v ‖ rdrand() ‖ rdrand() ‖ rdrand() ‖ rdrand()
15 |  |  fin
16 |  fin
17 |  m ← m ‖ mdgst ‖ mdctr
18 |  si la version est entre 0.9.8 et 1.0.2u alors m ← m ‖ b_i
19 |  k ← (stidx+10)-stnum
20 |  si k > 0 alors
21 |  |  m ← m ‖ state[stidx...(stnum-1)] ‖ state[0...(k-1)]
22 |  sinon
23 |  |  m ← m ‖ state[stidx...(stidx+9)]
24 |  fin
25 |  mdgst ← SHA1(m)
26 |  pour j ∈ [0;9] faire
27 |  |  state[stidx] ← state[stidx] ⊕ mdgst[j]     // mise à jour de state
28 |  |  stidx ← (stidx+1) mod stnum
29 |  fin
30 |  b_i ← mdgst[10...(9+size(b_i))]
31 fin
32 mdgst ← SHA1(mdctr ‖ mdgst ‖ oldmd)
33 mdctr[0] ← mdctr[0] + 1
```

Pour expliquer le fonctionnement de ces algorithmes (Algorithme 6.4 et Algorithme 6.5), commençons d'abord par donner les rôles de certaines variables globales qu'ils contiennent.

— `state` : est un tableau de 1023 octets représentant l'état interne du générateur.

— `stnum` : est l'indice maximal de la zone de `state` ayant été mise à jour.

— `stidx` : est l'indice courant du réservoir `state`.

— `mdctr` : est un tableau composé de 2 entiers. Le premier compte le nombre d'appels à `rand_bytes` et le second correspond au nombre de blocs d'octets ajoutés à `state` depuis le lancement du programme.

— `mdgst` : est un tableau de 20 octets servant à stocker un condensat SHA1.

— `ntrpy` : compte l'entropie fournie au générateur et sa valeur maximale est de 32 octets.

Dans l'Algorithme 6.4, il n'y a eu aucune modification algorithmique dans `rand_add` entre les versions 0.9.8 et 1.1.0l. Les données à ajouter sont découpées en des blocs de 20 octets sauf éventuellement pour le dernier qui en ferait moins. Le tableau `state` (qui est circulaire) est aussi découpé en des blocs de 20 octets à partir de `stidx`. Chaque couple de blocs composé d'un bloc de données et d'un bloc de `state` correspondants est concaténé puis le résultat est condensé avec le SHA1. Les blocs de `state` ayant été utilisés ainsi que les compteurs sont ensuite mis à jour. Une fois que tous les blocs sont traités, `mdgst` est également mis à jour.

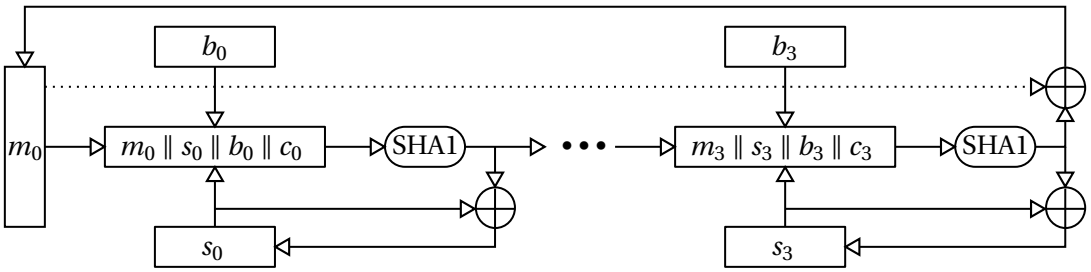


Figure 6.1 – ***Exemple de `rand_add` avec une donnée de 512 bits*** *– Les variables c_i, m_i et s_i représentent respectivement les valeurs de `ctr`, de `mdgst` et `state`. La taille étant de 512 bits ou 64 octets, alors b_0, b_1 et b_2 sont de 20 octets chacun, et b_3 est de 4 octets.*

En revanche, des modifications ont été apportées à `rand_bytes` entre les versions 0.9.8 et 1.1.0l, d'où la présence des deux suites de lignes rouges dans l'Algorithme 6.5. Ces lignes concernent notamment des sources d'entropie que l'on trouve dans le Tableau 6.4. Les b_i sont des blocs de 10 octets (sauf pour le dernier éventuellement) non initialisés en C. Il est donc difficile de prédire à l'avance les valeurs qu'ils contiennent, d'où leur utilisation comme source d'entropie. La fonction `rand_bytes` découpe `state` en des blocs de 10 octets à partir de `stidx`. Chaque couple de blocs composé d'un bloc b_i et d'un bloc de `state` est concaténé puis condensé. On utilise les 10 premiers octets de ce résultat pour mettre à jour le bloc de `state` ayant été pris, et les 10 derniers octets sont affectés au b_i correspondant. Il est à noter que le principe consistant à diviser le condensat SHA1 en deux blocs de 10 octets permet d'éviter entre autres l'apparition des motifs dans les nombres premiers générés. Une fois que tous les b_i auront été remplis, on met à jour `mdgst` également.

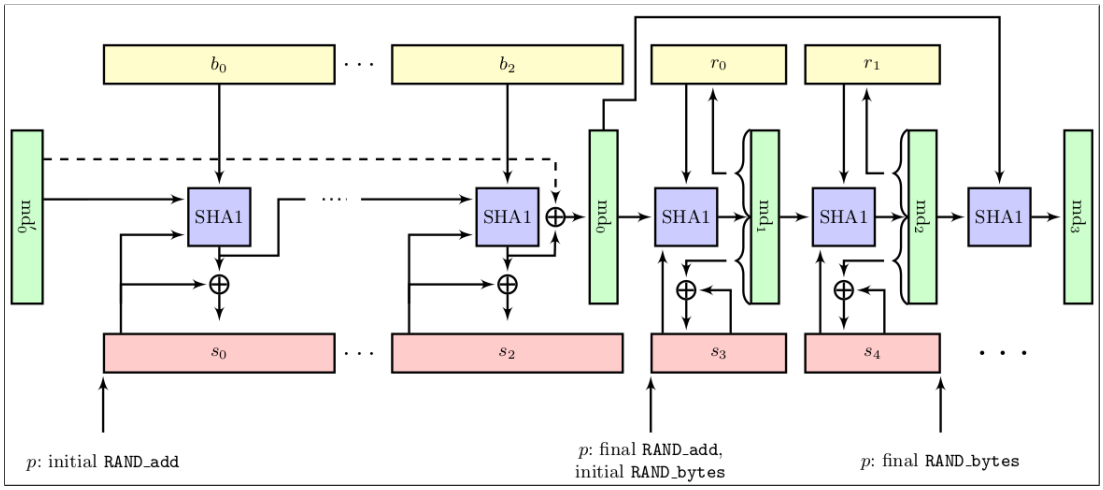


Figure 6.2 – ***Exemple de*** `rand_add` ***avec une donnée de 160 bits suivi de*** `rand_bytes` *– Les variables b_0, b_1, b_2, s_1, s_2 et s_2 sont sur 20 octets chacun. Les variables r_0, r_1, s_3 et s_4 sont sur 10 octets chacun. (Source : [113])*

6.2.1 Conditions d'expérimentations

Pour que les résultats de nos expérimentations soient cohérents avec les vulnérabilités observées, on va se mettre dans les les mêmes conditions (ou presque) que celles des matériels vulnérables trouvés. Cela implique non seulement d'utiliser les mêmes versions d'OpenSSL et de noyau Linux présentes sur ces matériels, mais aussi de mettre quasiment dans les mêmes états les éléments qui impactent la génération d'une clef RSA, à savoir les sources d'entropie. À cet effet, le constat suivant est important, car il a des conséquences directes sur les sources d'entropie citées dans le Tableau 6.4.

Constat. Pour chaque famille de matériels dont des modules sont PGCD-vulnérables et qui utiliseraient OpenSSL, on a remarqué que les certificats dont ces modules provenaient étaient émis à la même date (à la seconde près).

L'existence des modules PGCD-vulnérables entraîne que ces certificats, bien qu'ils aient été émis à la même date, ne sont pas des certificats par défaut. En effet, dans le cas contraire, on n'aurait qu'un seul module par famille, et donc pas de PGCD-vulnérabilité. Cela pourrait alors signifier que ces certificats ont été générés presque au même moment après le tout premier démarrage des matériels dont ils proviennent.

Au premier démarrage d'un système Linux, le fichier `RFILE` est absent, donc lorsqu'on sollicite OpenSSL pour la toute première fois, le contenu de ce fichier ne peut être utilisé en tant que source d'entropie. De plus, pour les versions d'OpenSSL comprises entre 0.9.8f et 1.1.0l, les propriétés de ce fichier ne peuvent pas non plus être utilisées. C'est après cette première invocation d'OpenSSL que `RFILE` est créé en y mettant une donnée de 1 Mo extraite du générateur pseudo-aléatoire par la fonction `rand_bytes`. Cette donnée est ensuite utilisée pour initialiser le réservoir du générateur lors de la prochaine utilisation d'OpenSSL.

<table>
<tr><th>Sources d'entropie</th><th>0.9.8 – 0.9.8e</th><th>0.9.8f – 1.0.2u</th><th>1.1.0 – 1.1.0l</th></tr>
<tr><td>stat(RFILE)</td><td>Oui</td><td colspan="2">Oui si RFILE existe</td></tr>
<tr><td>RFILE</td><td colspan="3">Oui si RFILE existe</td></tr>
<tr><td>/dev/urandom</td><td colspan="3" rowspan="4">Oui</td></tr>
<tr><td>Identifiant du processus</td></tr>
<tr><td>Identifiant de l'utilisateur</td></tr>
<tr><td>Temps système en secondes</td></tr>
<tr><td>Les b_i non initialisés de rand_bytes</td><td colspan="2">Oui</td><td>Non</td></tr>
<tr><td>Temps système en microsecondes</td><td colspan="2" rowspan="2">Non</td><td rowspan="2">Oui</td></tr>
<tr><td>La fonction rdrand() d'OpenSSL</td></tr>
</table>

Tableau 6.4 – ***Sources d'entropie utilisées par les versions d'OpenSSL comprises entre 0.9.8 et 1.1.0l*** – *La fonction rdrand d'OpenSSL est une version personnalisée de celle d'Intel.*

D'autres facteurs favorisent également l'apparition de la PGCD-vulnérabilité, à savoir l'obsolescence des noyaux Linux utilisés (voir Tableau 6.1), ou encore l'absence de certaines activités telles que les mouvements de souris et de clavier. Cela rend en effet difficile la mise de données suffisantes dans le fichier /dev/urandom au démarrage.

Compte tenu des remarques ci-dessus, les expérimentations ont été faites sur plusieurs Raspberry Pi 4 dotés de Ubuntu Server 20.04.3 LTS (32 bits). Le choix d'un tel environnement permet de simuler un matériel connecté sans interface et ayant peu de ressources. On a également imposé les conditions suivantes.

— Remplacer /dev/urandom par un fichier de contenu statique tel que /dev/zero. Cela simule la situation où il est difficile pour certains matériels de mettre suffisamment d'entropie dans /dev/urandom à leur démarrage.

— Fixer l'identifiant du processus qui génère la clef RSA. Si les certificats sont générés presque au même moment après le démarrage, il est alors probable qu'un processus de même identifiant soit utilisé pour les matériels appartenant à la même famille.

— Quant au UID, il est déjà fixé à 1000.

— Avant chaque génération de clefs RSA, effacer RFILE et réinitialiser l'horloge. Ces deux actions correspondent au fait que RFILE est absent au premier démarrage d'un Linux et que les certificats vulnérables ont été émis à la même date.

Ces conditions ont été appliquées en apportant des modifications directement dans le code source d'OpenSSL. La dernière condition peut aussi s'appliquer en utilisant des commandes SHELL. Cependant, réinitialiser l'horloge d'un Raspberry plusieurs millions de fois de cette façon pourrait causer des défaillances dans sa carte micro SD [3].

3. https://www.switchdoc.com/2016/01/tutorial-repairing-corrupted-sd-cards-for-the-raspberry-pi-on-mac/.

6.2.2 Expérimentations

Les conditions d'expérimentations évoquées dans la section précédente permettent de contrôler le comportement de certaines sources d'entropie du Tableau 6.4. Mais notons que ces conditions n'ont pas d'impact sur toutes les sources à la fois. Cela nous évite de se retrouver toujours avec le même module.

Versions allant de 0.9.8 à 0.9.8e Il reste deux sources d'entropie non contrôlées, à savoir les propriétés du fichier `RFILE` et les tableaux b de `rand_bytes` initialisés par `malloc`. Ces deux sources apportent peu d'entropie comme les développeurs d'OpenSSL le reconnaissent à travers un commentaire dans le code source de la fonction `rand_bytes`. Les valeurs renvoyées par chacune de ces sources varient, mais leurs périodes de répétition sont relativement courtes. En effet, la valeur de retour de `stat(RFILE)` est une structure (`struct` en C) contenant 88 octets de données. Lorsque `RFILE` est absent (ce qui est notre cas), chacun des 76 premiers premiers octets est initialisé avec zéro et les 12 restants sont sous la forme suivante :

x	y	z	-65	$x-4$	$y+4$	z	-65	$x-4$	y	z	-65

Les octets x, y et z peuvent être vus comme des entiers compris entre -128 et 127 (le type `char` en C). Lors des expérimentations, on remarqua que x a 2^4 valeurs possibles, à savoir les $16k+4$ avec $k \in \mathbb{Z}$ et $-8 \leqslant k \leqslant 7$. Les octets y et z peuvent prendre respectivement 2^8 et (2^7+1) valeurs. D'où le nombre maximal de valeurs que `stat(RFILE)` pourrait renvoyer est :

$$2^4 \times 2^8 \times (2^7+1) = 2^{19} + 2^{12}.$$

Quant aux tableaux b initialisés par `malloc`, on parvint à faire la même analyse, mais pas avec la même précision. On a pu trouver que leur période moyenne de répétition était majorée par 2^5, c'est-à-dire qu'en initialisant 32 tableaux b avec `malloc`, au moins deux d'entre eux seront égaux en moyenne.

```
struct stat  {
  __dev_t st_dev;             /* Device.  */
  unsigned short int __pad1;
  __ino_t st_ino;             /* File serial number.  */
  __mode_t st_mode;           /* File mode.  */
  __nlink_t st_nlink;         /* Link count.  */
  __uid_t st_uid;             /* User ID of the file's owner. */
  __gid_t st_gid;             /* Group ID of the file's group.*/
  __dev_t st_rdev;            /* Device number, if device.  */
  unsigned short int __pad2;
  __off_t st_size;                    /* Size of file, in bytes.  */
  __blksize_t st_blksize;     /* Optimal block size for I/O.  */
  __blkcnt_t st_blocks;               /* Number 512-byte blocks allocated. */
  __time_t st_atime;                  /* Time of last access.  */
  __syscall_ulong_t st_atimensec;     /* Nscecs of last access.  */
  __time_t st_mtime;                  /* Time of last modification.  */
  __syscall_ulong_t st_mtimensec;     /* Nsecs of last modification.  */
  __time_t st_ctime;                  /* Time of last status change.  */
  __syscall_ulong_t st_ctimensec;     /* Nsecs of last status change.  */
  unsigned long int __glibc_reserved4;
  unsigned long int __glibc_reserved5;
};
```

Versions allant de 0.9.8f à 1.0.2u Les tableaux b sont les seuls éléments pouvant apporter un peu d'entropie au générateur. Comme dans le cas précédent, ils se répètent en moyenne au bout de 32 fois puisqu'ils sont initialisés par `malloc`.

Versions allant de 1.1.0 à 1.1.0l Les sources d'entropie sont le temps système et `rdrand()`. Dans le cas des versions précédentes, le temps système était compté en secondes, son apport entropique était donc insignifiant (car deux clefs de même taille se génèrent généralement à la seconde près). Dans le cas présent, il est en microsecondes, ce qui donne la possibilité à des fluctuations pouvant apporter plus d'entropie que dans la situation précédente. Quant à la fonction `rdrand()` qui est censée extraire de l'aléa sur des composants physiques, elle est absente sur les matériels que l'on analyse.

Dans chacune de ces trois plages de versions d'OpenSSL, une version fut arbitrairement choisie. Il s'agit de : 0.9.8, 1.0.1 et 1.1.0. Pour chacune de ces versions, 16 millions de modules RSA ont été générés, dont 7 millions sur 512 bits, 5 millions sur 1024 bits et 4 millions sur 2048 bits. Les résultats des analyses faites sur ces modules sont donnés dans le Tableau 6.5.

Tailles des modules	Versions d'OpenSSL	Modules	Modules distincts	Modules distincts PGCD-vulnérables	Nombres premiers	Nombres premiers distincts	Ratio
512	0.9.8	7M	6,995,870 [99.94%]	685,689 [9.80%]	14M	13,358,303 [95.42%]	52.37%
	1.0.1		246,800 [3.53%]	137,846 [55.85%]		364,185 [2.60%]	67.77%
	1.1.0		6,999,550 [99.99%]	1,542,324 [22.03%]		12,532,681 [89.52%]	55.85%
1024	0.9.8	5M	4,998,918 [99.98%]	1,534 [0.03%]	10M	9,997,069 [99.97%]	50.00%
	1.0.1		490,172 [9.80%]	263,928 [53.84%]		730,800 [7.31%]	67.07%
	1.1.0		4,999,783 [99.99%]	376,427 [7.53%]		9,661,356 [96.61%]	51.75%
2048	0.9.8	4M	3,999,924 [99.99%]	656 [0.02%]	8M	7,999,521 [99.99%]	50.00%
	1.0.1		3,762,313 [94.06%]	3,597,199 [89.93%]		4,399,845 [54.99%]	85.51%
	1.1.0		3,999,990 [99.99%]	118,370 [2.96%]		7,893,883 [98.67%]	50.67%

Tableau 6.5 – ***Résultats des expérimentations*** – *Les pourcentages entre crochets sont les proportions des nombres de la colonne concernée par rapport à ceux de la colonne précédente. La lettre M désigne "millions".*

La dernière colonne du Tableau 6.5 correspond aux ratios des modules distincts par rapport aux nombres premiers distincts. La valeur idéale de ce ratio est de 50% et correspond au cas où il n'y a pas de redondance ni dans les modules ni dans les nombres premiers. Cette colonne nous montre donc que la version 1.0.1 a plus de redondance que la version 1.1.0 qui en a plus que la version 0.9.8. On interprète les résultats davantage dans la sous-section 6.2.3.

6.2.3 Interprétation

Pour chaque version analysée, on note $p_{\min}$ et $p_{\max}$ le plus petit et le plus grand nombres premiers générés. Pour mieux visualiser les résultats sur un graphique, on a divisé l'intervalle de bornes $p_{\min}$ et $p_{\max}$ en 2 000 intervalles réguliers. Tous les nombres premiers générés ont ensuite été repartis entre ces intervalles. On considère uniquement le cas des modules de 512 bits. On obtient des résultats similaires pour les modules de 1024 bits et 2048 bits.

Pour les modules de 512 bits (donc des nombres premiers de 256 bits) et pour chacune des trois versions, l'amplitude de chacun des 2 000 intervalles est de l'ordre de

$$2^{243} + 2^{237} + 2^{236}.$$

La Figure 6.3, la Figure 6.4 et la Figure 6.5 donnent respectivement les répartitions pour les versions 0.9.8, 1.0.1 et 1.1.0 des nombres premiers entre 2 000 intervalles réguliers.

Nombres premiers	Versions d'OpenSSL	Quantités minimales	Quantités maximales	Quantités moyennes	Écarts types
Tous	0.9.8	6,645	7,423	7,000	110
	1.0.1	711	80,193		4,541
	1.1.0	6,089	268,912		8,903
Distincts	0.9.8	6,412	6,968	6,679	84
	1.0.1	136	261	182	15
	1.1.0	6,023	6,515	6,266	79

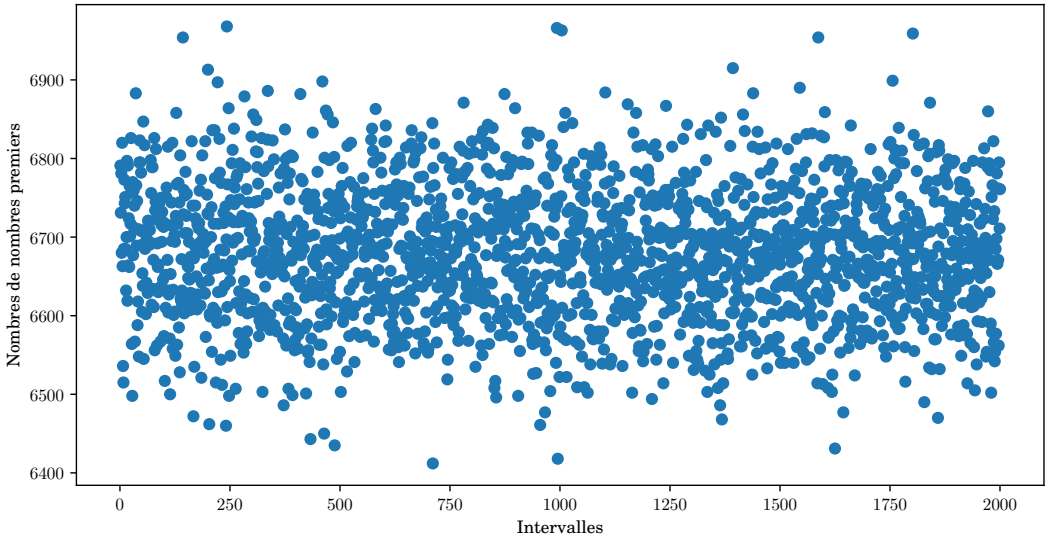
Tableau 6.6 – ***Moyennes et écart-types des données d'expérimentations.***

Version 0.9.8 : Sur la Figure 6.3a, on observe peu de variations dans les quantités de nombres premiers distincts par intervalle. Cela se traduit, dans le Tableau 6.6, par un écart-type égal à 84. La quantité moyenne est 6 679 et les quantités minimale et maximale sont 6 412 et 6 968. La comparaison avec la Figure 6.3b qui illustre la répartition des nombres premiers (non distincts) par intervalle, montre qu'il y a de la redondance en appui à des résultats du Tableau 6.5. Les quantités minimale et maximale passent à 6 645 et 7 423. Mais malgré cela, on observe qu'aucun intervalle n'est significativement plus sollicité qu'un autre (l'écart-type étant de 110), c'est-à-dire que les nombres premiers restent repartis presque également entre les intervalles.

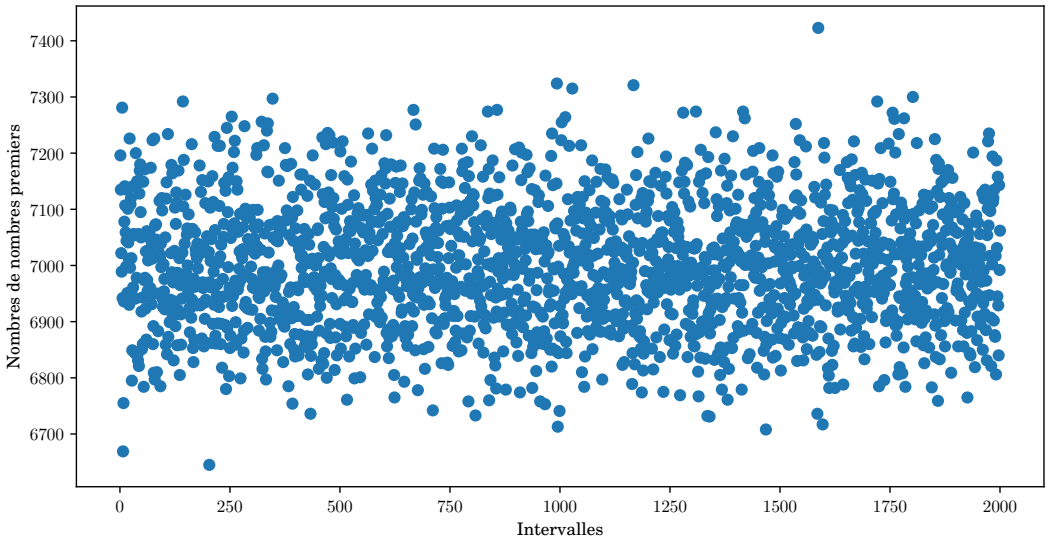
Version 1.0.1 : La Figure 6.4a montre une bande bleue semblable à celle de la Figure 6.3a, et donne l'impression que quelques valeurs se démarquent de la moyenne plus que dans le cas précédent. Mais, cette impression vient de la comparaison immédiate des deux figures sans tenir compte des échelles (qui sont différentes). D'ailleurs, l'écart-type du cas présent est de 15, donc plus petit que celui du cas de la version 0.9.8. La quantité moyenne est 182 et les quantités minimale et maximale sont 136 et 261. Ces faibles quantités de nombres premiers distincts viennent sans doute du fait que cette version a moins de sources d'entropie que les deux autres versions.

Par contre, sur la Figure 6.4b qui donne la répartition des nombres premiers (non distincts), on observe non seulement qu'il y a plus de redondance que dans le cas de la version 0.9.8, mais aussi qu'une quinzaine d'intervalles (dans l'ovale rouge) se distinguent nettement plus que les autres. En effet, la quantité moyenne est passée à 7 000, et les quantités minimale et maximale à 711 et 80 193. Chacun des 15 intervalles contient plus de 40 000 nombres premiers, alors qu'aucun des 1985 intervalles restants n'en contient plus de 15 000. La présence de ces intervalles à valeurs extrêmes conduit à un écart-type élevé de 4 541, et sans eux, l'écart-type dévient 2 147 soit moins de la moitié de sa valeur initiale. Comme le montre la Figure 6.4a, ces intervalles ne contiennent pas plus de nombres premiers distincts que les autres intervalles. Leurs taux de redondance sont liés aux états initiaux choisis pour les sources d'entropie lors des expérimentations. Si l'on vient à changer ces valeurs initiales, on observerait le même phénomène, mais pas nécessairement avec les mêmes intervalles.

Version 1.1.0 : Similairement aux cas précédents, sur la Figure 6.5a, on observe aussi que les nombres premiers distincts sont répartis presque également entre les intervalles avec un écart-type de 79, une quantité moyenne de 6 266, et des quantités minimale et maximale de 6 023 et 6 515. Quant à la Figure 6.5b, elle ressemble à la Figure 6.4b du cas de la version 1.0.1, mais avec des quantités encore plus élevées que celles qui apparaissent sur cette dernière (quantités minimale et maximale à 6 089 et 268 912). Une douzaine d'intervalles se distinguent des autres avec plus de 18 000 nombres premiers chacun, alors qu'aucun des 99.70% des autres intervalles n'en contient plus de 8 000. Sans ces intervalles, la quantité moyenne serait 6 487 et l'écart-type serait égal à 313 soit 28 fois plus petit que l'écart-type initial. Aussi, ces intervalles ne contiennent pas plus de nombres premiers distincts que les autres intervalles comme on peut l'observer sur la Figure 6.5a.

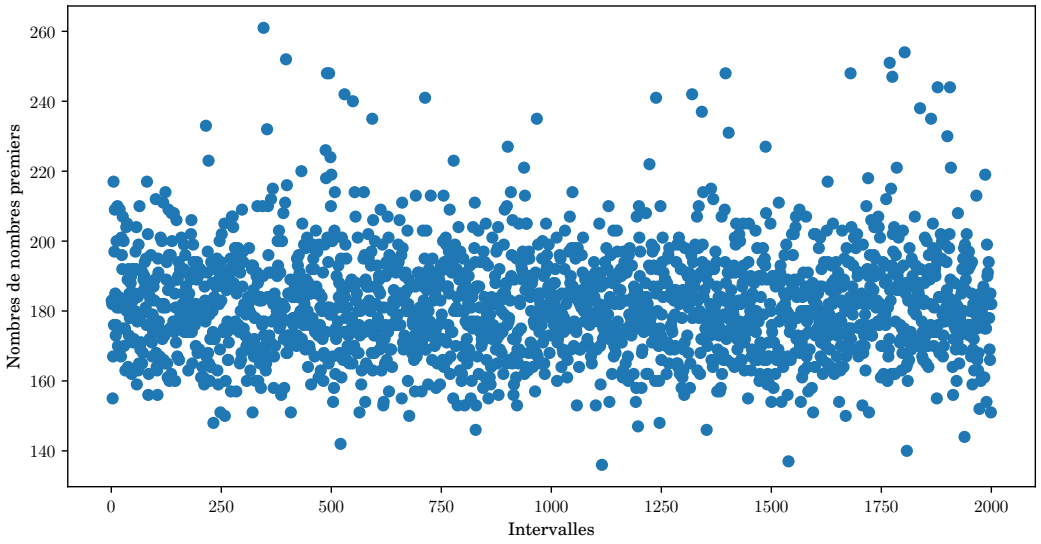


(a) ***Nombres premiers distincts.***

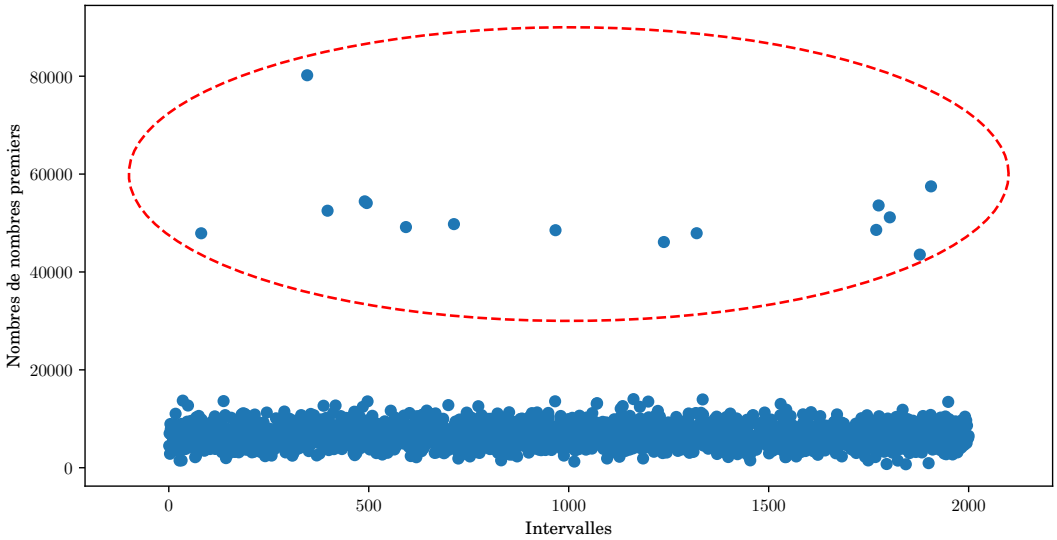


(b) ***Tous les nombres premiers.***

Figure 6.3 – ***Répartition sur 2 000 intervalles réguliers des facteurs premiers de 14 millions de modules RSA de 512 bits générés par OpenSSL 0.9.8 sous les conditions de la sous-section 6.2.1*** – *Chacun des intervalles a une amplitude d'environ* 2^{243}.

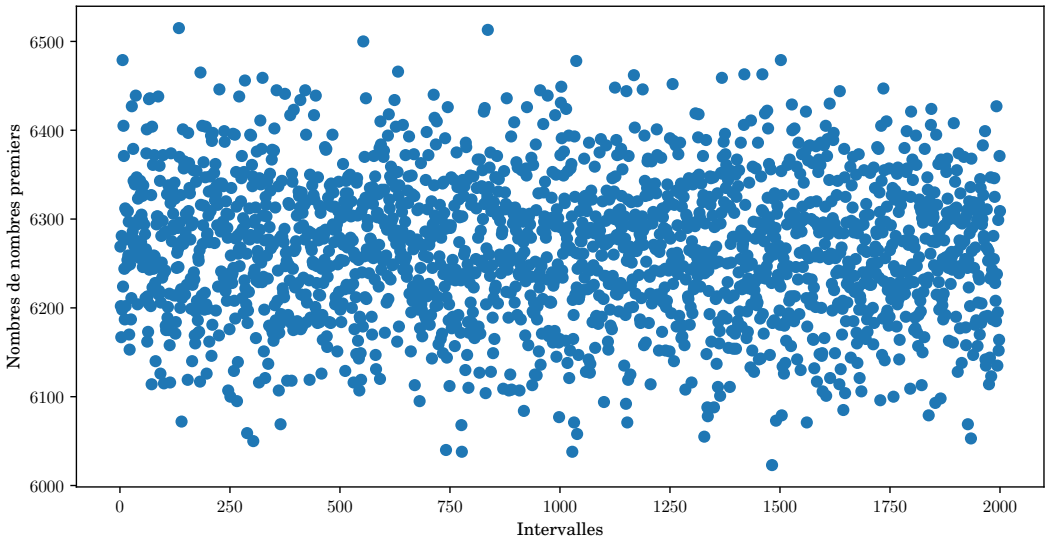


(a) ***Nombres premiers distincts.***

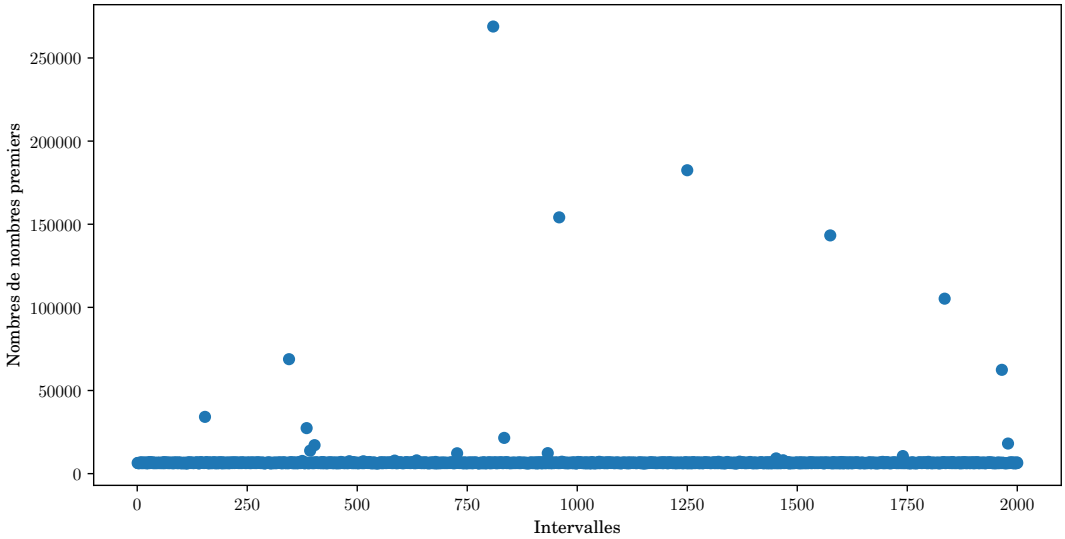


(b) ***Tous les nombres premiers.***

Figure 6.4 – ***Répartition sur 2000 intervalles réguliers des facteurs premiers de 14 millions de modules RSA de 512 bits générés par OpenSSL 1.0.1 sous les conditions de la sous-section 6.2.1*** – *Chaque intervalle a une amplitude d'environ* 2^{243}. *Les quantités anormales sont entourées en rouge.*



(a) ***Nombres premiers distincts.***



(b) ***Tous les nombres premiers.***

Figure 6.5 – ***Répartition sur 2000 intervalles réguliers des facteurs premiers de 14 millions de modules RSA de 512 bits générés par OpenSSL 1.1.0 sous les conditions de la sous-section 6.2.1*** – *Chaque intervalle a une amplitude d'environ* 2^{243}.

6.2.4 Conclusion

Sur la Figure 6.4b, la plupart des nombres premiers sont redondants avec des nombres d'apparition presque égaux. Leur redondance est significative, car la quantité moyenne de nombres premiers distincts est 182. Par contre, cette moyenne est 6 266 pour la Figure 6.5b,

la redondance n'est donc pas aussi significative que dans le cas précédent. En résumé, pour la version 1.1.0, on a peu de nombres premiers redondants, mais avec de grands nombres d'occurrences. Alors que pour la version 1.0.1, on a plusieurs nombres premiers redondants, mais dont les nombres d'occurrences ne sont pas aussi grands que ceux de la version 1.1.0.

Pour les versions 1.0.1 et 1.1.0, le choix de ces nombres premiers redondants est lié aux conditions initiales des expérimentations telles que le PID ou encore l'instant utilisé pour réinitialiser l'horloge. Si l'on vient à changer ces conditions, on observerait le même phénomène, mais pas nécessairement avec les mêmes nombres premiers.

Aussi, les nombres premiers les plus redondants sont généralement les facteurs ayant été générés en premier lors de la génération des modules. Lorsque OpenSSL génère un module RSA, elle génère deux nombres premiers a et b dans cet ordre. Le plus grand des deux sera p et le plus petit q. Lors des expérimentations, il a été constaté que les collisions sont plus fréquentes sur les a que sur les b. En effet, le réservoir du générateur pseudo-aléatoire aura accumulé plus d'entropie lors de la génération de b que lors de la génération de a. Cela explique aussi pourquoi les taux de nombres premiers redondants sont plus élevés que ceux des modules redondants, c'est-à-dire qu'il est plus facile d'avoir deux modules ayant un facteur commun que d'avoir deux modules égaux.

6.3 Génération des clefs RSA par les versions allant de 1.1.1 à 1.1.1j

À partir de ces versions, le générateur de nombres pseudo-aléatoires d'OpenSSL a changé radicalement. On est passé à un générateur pseudo-aléatoire appelé «CTR-DRBG» (présenté dans la sous-section 6.3.1), basé sur des chiffrements par blocs et proposé par la norme SP 800-90A [14, p.48–58] du NIST. Des modifications ont également été apportées à l'algorithme de génération des clefs RSA. Par exemple, contrairement aux versions précédentes, on a maintenant la possibilité d'utiliser des modules RSA composés de plus de deux facteurs premiers (ou modules RSA multi-premiers). Toutefois, les nombres de facteurs premiers des modules générés par ces versions récentes d'OpenSSL sont choisis conformément aux valeurs données dans le Tableau 6.7.

Tailles des modules RSA	[512 ; 1024[	[1024 ; 4096[	[4096 ; 8192[	[8192 ; 16384[
Facteurs premiers possibles	2	[2 ; 3]	[2 ; 4]	[2 ; 5]

Tableau 6.7 – ***Nombres de facteurs premiers possibles en fonction de la taille des modules.***

Les modules RSA générés vérifient également d'autres critères. Leurs tailles minimale et maximale valent 512 bits et 16384 bits ; la taille par défaut vaut 2048 bits et le nombre de facteurs premiers est de 2 par défaut. Les facteurs premiers sont distincts et auront la même taille si leur nombre divise la taille du module. Sinon, c'est-à-dire si cette division admet un reste r non nul, alors la taille de r facteurs premiers est augmentée de 1. Par exemple, si l'on veut un module RSA de 2048 bits composé de trois facteurs premiers, on aura deux nombres premiers de 683 bits chacun et un nombre premier de 682 bits.

Chaque nombre premier généré vérifie la condition 6.1 et sa primalité est testée par le test de Miller-Rabin [94] (que l'on présenta dans la sous-section 4.2.3). Les nombres d'itérations

de ce test, en fonction de la taille du nombre à tester, sont donnés dans le Tableau 6.8. Notons que les développeurs d'OpenSSL ont créé ce tableau à l'aide d'un algorithme provenant de la norme FIPS 186-4 [59, app.F.1, p.117] pour une probabilité d'erreur de 2^{-128}.

Tailles des nombres	[55 ; 308[	[308 ; 347[	[347 ; 400[	[400 ; 476[	[476 ; 1345[	[1345 ; 3747[	[3747 ; →[
Nombres de vérification	27	8	7	6	5	4	3

Tableau 6.8 – ***Nombres d'itérations du test de Miller-Rabin pour OpenSSL.***

6.3.1 Générateur pseudo-aléatoire CTR-DRBG du NIST

On sait que les générateurs de nombres aléatoires jouent un rôle important en cryptographie, notamment dans la génération des clefs cryptographiques. Il existe principalement deux familles de générateurs aléatoires. La première est celle des générateurs de nombres pseudo-aléatoires (notés PRNG pour «PseudoRandom Number Generators» ou DRBG pour «Deterministic Random Bit Generators») qui génèrent des bits de façon déterministe avec un algorithme. L'autre famille est celle des générateurs de nombres aléatoires physiques (notés TRNG pour «True Random Number Generators» ou NRBG pour «Non-deterministic Random Bit Generators») qui utilisent des sources physiques pour produire des bits *à priori* aléatoires.

Dans cette section, on va présenter le CTR-DRBG, un PRNG recommandé par la norme SP 800-90A [14, p.48–58] du NIST et basé sur un chiffrement par bloc en mode CTR [37]. Dans cette norme, on trouve aussi deux autres PRNG : le Hash-DRBG (basé sur une fonction de hachage) et le HMAC-DRBG (reposant sur un HMAC [84]). Le CTR-DRBG est plus utilisé que les deux autres dans les bibliothèques cryptographiques, en l'occurrence OpenSSL. Il fait également partie du générateur de nombres aléatoires des processeurs récents d'Intel [4].

On pourra trouver plus d'informations sur les générateurs aléatoires de façon générale dans le deuxième volume de «Art of Computer Programming» [63, chap.4] de Donald Knuth. On trouvera également des informations supplémentaires dans la norme SP 800-90A [14] sur les trois générateurs de nombres pseudo-aléatoires recommandés par le NIST.

L'utilisation du CTR-DRBG nécessite une source d'entropie pour fournir des bits *à priori* aléatoires et les sources d'entropie utilisées sont typiquement des TRNG. Cependant, l'usage d'autres sources est également possible selon les besoins. Les algorithmes de chiffrement par bloc autorisés sont le 3DES (avec trois clefs) et l'AES (avec une clef de 128 bits, 192 bits ou 256 bits). Les quatre paramètres associés à l'algorithme sont les suivants.

outlen. C'est la taille d'un bloc que renvoie la fonction de chiffrement utilisée.

keylen. Il s'agit de la taille de la clef utilisée dans l'algorithme de chiffrement.

seedlen. Il représente la taille de la graine qui est utilisée pour initialiser le PRNG. Notons que *seedlen* = *outlen* + *keylen*. Une graine (ou «seed») est une chaîne de bits qui est utilisée pour définir l'état initial d'un PRNG. Elle détermine aussi une partie de l'état interne du PRNG et son entropie doit être suffisante pour assurer le niveau de sécurité du PRNG.

4. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html, Section 3.2.3.

reseed_interval. Le nombre maximal de blocs produits par la fonction de chiffrement avant la mise à jour du générateur avec une nouvelle graine.

Paramètres	3DES	AES-128	AES-192	AES-256
outlen	64	128	128	128
keylen	168	128	192	256
seedlen	232	256	320	384
reseed_interval	$\leqslant 2^{32}$	$\leqslant 2^{48}$	$\leqslant 2^{48}$	$\leqslant 2^{48}$

Tableau 6.9 – ***Paramètres de CTR-DRBG.***

Trois fonctions agissent sur ce PRNG : celle qui l'initialise, celle qui génère des nombres pseudo-aléatoires et celle qui effectue sa mise à jour.

Initialisation. Un chiffrement par bloc en mode CTR nécessite une clef de chiffrement K et un compteur noté V dans le NIST SP 800-90A [14]. L'ensemble constitué de K et V représente la graine. Au démarrage du générateur, K et V doivent être initialisés avec des valeurs qui peuvent être arbitrairement choisies. Ces valeurs sont utilisées pour produire une suite de blocs avec V qui augmente de 1 après chaque chiffrement. Le processus continue jusqu'à ce qu'au moins *seedlen* bits soient générés. Les *seedlen* bits à gauche de la sortie sont XORés à *seedlen* bits d'entropie (notés *provided_data*) pour produire une nouvelle graine. Les *keylen* bits de poids fort de cette graine forment la nouvelle clef K et les *outlen* bits de poids faible donnent la nouvelle valeur de V.

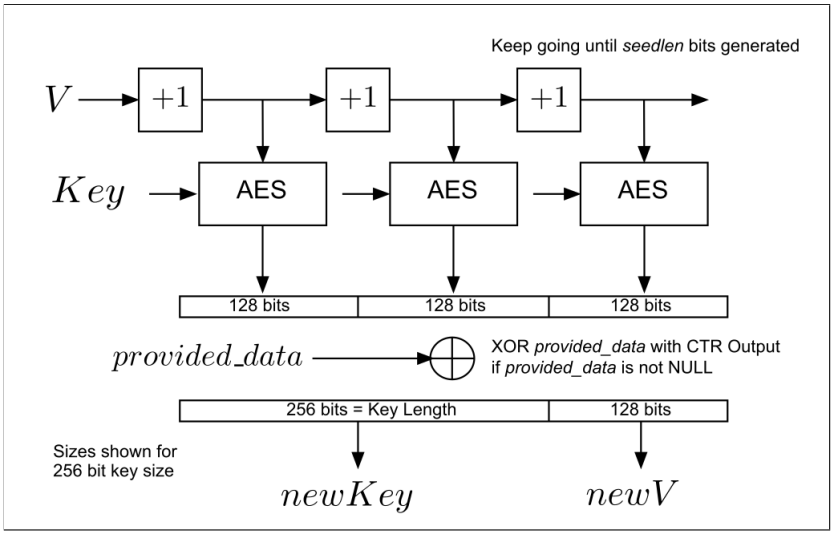


Figure 6.6 – ***Processus d'initialisation de CTR-DRBG.*** *(Source : [55])*

Génération. Une fois que K et V sont obtenus, le PRNG peut être utilisé pour générer des bits pseudo-aléatoires (un bloc à la fois). Chaque itération utilise la même clef et incrémente de 1 la valeur de V.

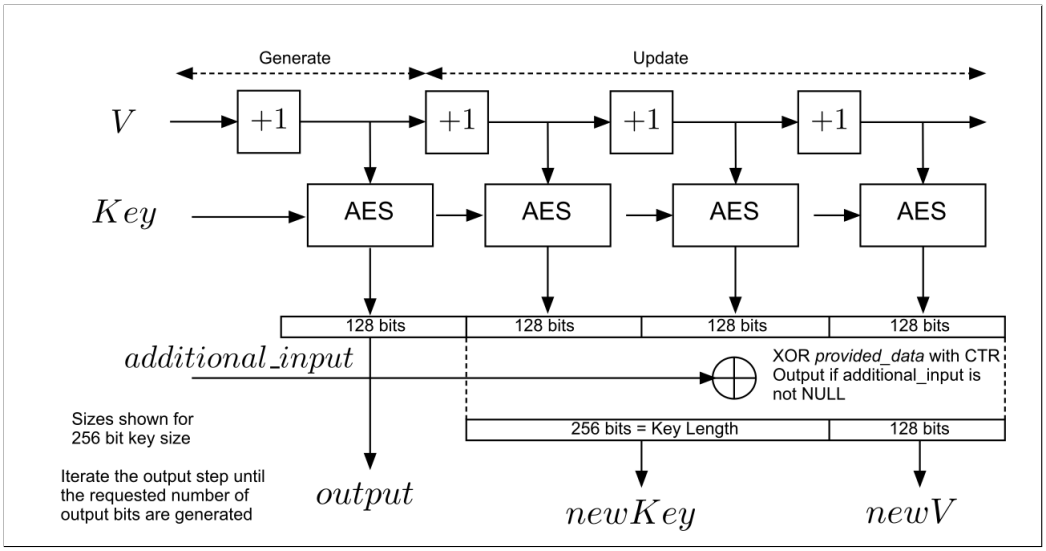


Figure 6.7 – ***Processus de génération de CTR-DRBG.*** *(Source : [55])*

Mise à jour **:** Pour des raisons de sécurité, le nombre de blocs générés doit être limité. Le CTR-DRBG utilise le paramètre *reseed_interval* pour fixer cette limite. Après chaque génération, un compteur mesurant le nombre de *reseed* augmente de 1 et dès que ce compteur atteint *reseed_interval*, la fonction de mise à jour est appelée. Elle opère de la même façon que la fonction d'initialisation à la différence que les dernières valeurs de K et V sont utilisées comme paramètres de la fonction de mise à jour.

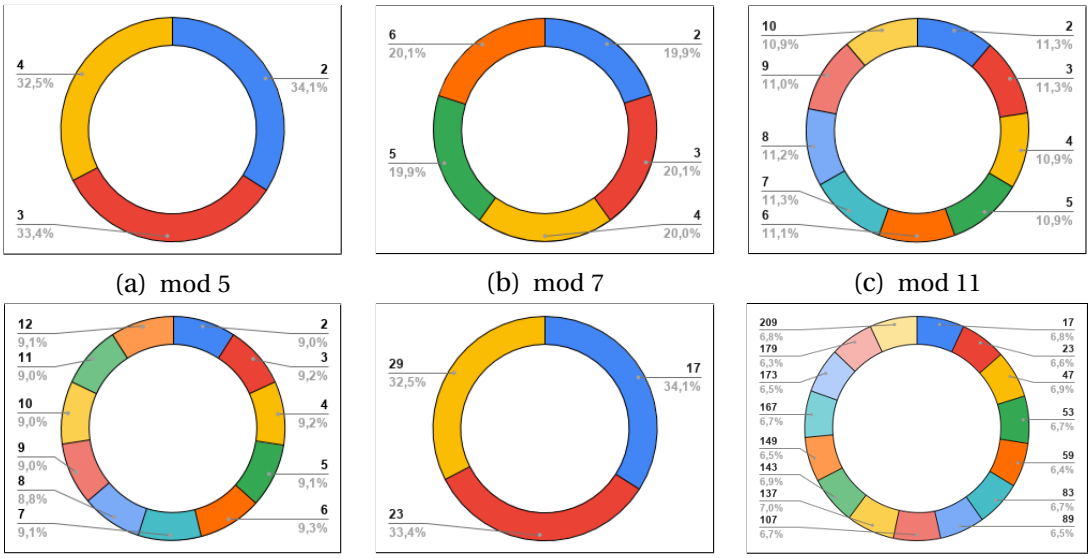
6.3.2 Conditions d'expérimentations

Les paramètres du CTR-DRBG implémenté par les versions récentes d'OpenSSL sont ceux figurant dans la dernière colonne du Tableau 6.9. La graine (c'est-à-dire le couple $(K;V)$) est choisie en utilisant les fichiers `/dev/urandom` et `RANDFILE`. Les expérimentations, faites sur des Raspberry Pi 4, se feront dans les mêmes conditions que celles de la sous-section 6.2.1. Cependant, à la différence des anciennes versions, les nouvelles versions ont une variable `provided_data` (présente sur la Figure 6.6 et la Figure 6.7) qui ajoute de l'aléa (extrait de `/dev/urandom`) au générateur après chaque génération de données aléatoires.

6.3.3 Résultats des expérimentations

Avec OpenSSL 1.1.1 sous les conditions évoquées ci-dessus, on a généré 7 millions de modules de 512 bits, 5 millions de modules de 1024 bits et 3 millions de modules de 2048 bits. D'une part, on n'observa aucune redondance de nombres premiers. Ceci est dû, comme on vient de le mentionner ci-dessus, à la variable `provided_data` qui rafraîchit le générateur avec de l'aléa extrait de `/dev/urandom`. Ce rafraîchissement est effectué après chaque sollicitation du générateur (y compris durant la génération de nombres pseudo-aléatoires dans les itérations du test de Miller-Rabin). D'autre part, on donne sur la Figure 6.8, la répartition des

restes des nombres premiers générés modulo les entiers 5, 7, 11, 13, 30 et 210. On remarque que l'entier 1 ne fait pas partie des restes du fait de la condition 6.1. On constate aussi que, pour chaque modulo, la répartition est faite presque également entre les restes ; les déviations n'apparaissant pas exploitables pour des attaques.



(a) mod 5 (b) mod 7 (c) mod 11

(d) mod 13 (e) mod 30 (f) mod 210

Figure 6.8 – ***Proportions des restes des facteurs premiers (OpenSSL).***

Les modules PGCD-vulnérables de certains fabricants de matériels réseaux (cités dans les Tableau 5.7, Tableau 5.8 et Tableau 5.9 du chapitre 5) ne remplissent pas la condition 6.1, ils ne proviennent donc pas d'OpenSSL. Ils devraient, en conséquence, provenir des bibliothèques cryptographiques (telles que MbedTLS [12]) qui sont actuellement répandues dans les systèmes embarqués.

6.4 Génération des clefs RSA avec PolarSSL

Entre 2009 et 2011, MbedTLS [12], appelé à l'époque PolarSSL, utilisait un algorithme HAVEGE [106, 47, 68] pour la génération de ses nombres aléatoires. La seule source d'aléa utilisée dans cet algorithme est la fonction `hardclock`. Sa sortie correspond :

— au nombre de tic-tacs (ou «Ticks») écoulés depuis la dernière remise à zéro (ou «Reset») du processeur pour les `i386`, `amd64`, `x86_64`, `powerpc`, `sparc`, `alpha` et `ia64`.

— au nombre de micro-secondes écoulées depuis le lancement du programme de génération de la clef RSA pour les autres types de processeurs.

Le générateur aléatoire puise dans un réservoir de taille 1024 représenté par un tableau dont chaque élément est sur 32 bits. Lors du lancement du programme de génération de la clef RSA, le contenu du réservoir est initialisé avec des zéros pour ensuite être mélangé aux

résultats de 8192 appels à `hardlock` en utilisant des opérations fondamentales (et aucune fonction de hachage n'est utilisée). Au delà de 512 sollicitations du réservoir, son contenu (non remis à zéro) est de nouveau mélangé à la sortie de `hardlock` 8192 fois en utilisant les mêmes opérations avant d'être réutilisé.

L'algorithme qui choisit les nombres premiers est assez similaire à celui d'OpenSSL : une fois que le nombre aléatoire est fourni par le générateur pseudo-aléatoire, ses deux premiers bits sont transformés en 1, puis le nombre premier qui le suit («next prime») est retourné si celui-ci est sur la taille demandée. La primalité d'un nombre est testée en passant un crible puis des itérations du test de Miller-Rabin. Ce crible, composé des nombres premiers impairs et inférieurs à 1000 (soit 167 nombres premiers), est utilisé dans le but de détecter d'éventuels petits facteurs premiers avant de déployer un algorithme plus efficace. Le crible est suivi du test de Miller-Rabin dont les nombres d'itérations (en fonction de la taille du nombre à tester) sont données dans le Tableau 6.10. Signalons que ce tableau fut tiré, par les développeurs de PolarSSL, de l'ouvrage «Handbook of Applied Cryptography» [78, p.148] et qu'il assure une probabilité d'erreur d'au plus 2^{-80}.

Tailles des nombres	[100 ; 150[	[150 ; 250[	[250 ; 350[	[350 ; 650[	[650 ; 850[	[850 ; 1300[	[1300 ; →[
Nombres de vérification	27	18	12	8	4	3	2

Tableau 6.10 – ***Nombres d'itérations du test de Miller-Rabin pour PolarSSL.***

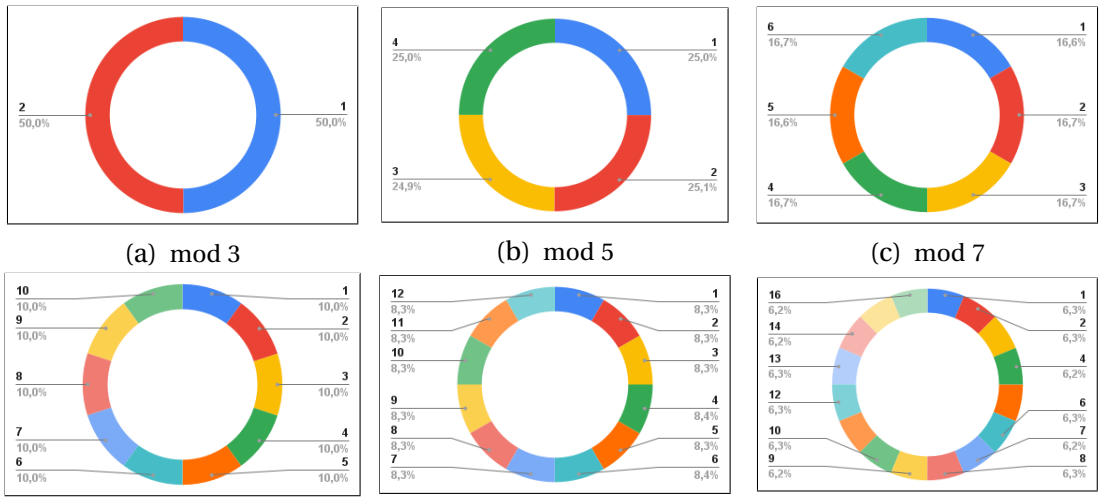
6.4.1 Conditions d'expérimentations

La version de PolarSSL que l'on a utilisée pour les expérimentations est la version 1.0.0, car celle-ci coïncide avec la période où la base de données de l'EFF fut constituée. Puisque les expérimentations ont lieu sur des Raspberry Pi 4 avec des processeurs ARM, alors selon ce qui vient d'être mentionné ci-dessus, la seule source d'aléa est le temps système. Durant ces expérimentations, on réinitialisera donc l'horloge avant chaque génération de clefs afin de se mettre dans la situation des matériels vulnérables trouvés.

6.4.2 Résultats des expérimentations

Sous les conditions citées dans la section précédente, on a généré 7 millions de modules RSA de 512 bits, 5 millions de modules de 1024 bits et 3 millions de modules de 2048 bits. Aucun des nombres premiers généré n'apparaît plus d'une fois, c'est-à-dire qu'il n'y a pas de redondance. La répartition de leurs restes modulo les entiers $5, 7, 11, 13, 30$ et 210 est illustrée par la Figure 6.9. Sur cette figure, on constate que les répartitions sont meilleures que celles qui apparaissent sur la Figure 6.8. Cette différence pourrait venir de la condition 6.1 à laquelle les nombres premiers générés par OpenSSL obéissent.

Si n_1 et n_2 sont deux modules générés par cette bibliothèque. Pour espérer trouver un facteur commun entre n_1 et n_2 en tenant compte de la façon dont le générateur est conçu, il suffit, par exemple, que : lors de la génération de n_1, la suite des 8192 sorties de `hardlock` qui initialise le réservoir soit égale à la suite des 8192 sorties de `hardlock` qui initialise le réservoir lors de la génération de n_2.



(a) mod 3 (b) mod 5 (c) mod 7

(d) mod 11 (e) mod 13 (f) mod 17

Figure 6.9 – ***Proportions des restes des facteurs premiers (PolarSSL).***

6.5 Génération des clefs RSA avec MbedTLS

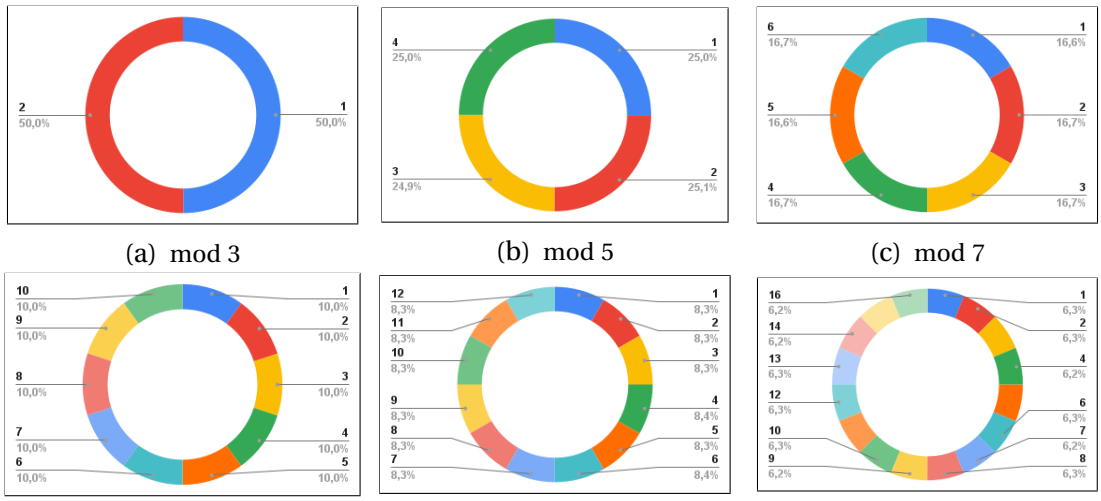
À partir de 2012, la bibliothèque PolarSSL est devenue MbedTLS. Par la même occasion, elle suivit les recommandations de la norme FIPS 186-4 [59] du NIST (voir sous-section 4.6.2) en changeant à la fois son générateur pseudo-aléatoire en CTR-DRBG [14] (présenté dans la sous-section 6.3.1) et son algorithme de génération de nombres premiers. Cependant, contrairement à OpenSSL et à PolarSSL, la bibliothèque MbedTLS n'utilise pas de crible lors de la vérification de la primalité d'un entier donné. Elle utilise directement le test de Miller-Rabin dont elle choisit le nombre d'itérations d'un tableau issu de l'ouvrage «Handbook of Applied Cryptography» [78, p.147] pour garantir une probabilité d'erreur d'au plus 2^{-100}. On doit aussi noter que MbedTLS n'utilise pas une méthode de type «next prime». Dès lors qu'un nombre pseudo-aléatoire ne passe pas le test de Miller-Rabin, il est jeté et un autre est généré.

6.5.1 Conditions d'expérimentations

Puisqu'il s'agit du même générateur de nombres pseudo-aléatoires qui est utilisé par les versions récentes d'OpenSSL, on effectue les expérimentations dans les mêmes conditions que celles mentionnées dans la section 6.4.

6.5.2 Résultats des expérimentations

On a généré 7 millions de modules RSA de 512 bits, 5 millions de modules de 1024 bits et 3 millions de modules de 2048 bits. On n'a observé aucune répétition de nombres premiers. La répartition des restes modulo, illustrée par la Figure 6.10, est semblable à celle de PolarSSL donnée par la Figure 6.9.



(a) mod 3 (b) mod 5 (c) mod 7

(d) mod 11 (e) mod 13 (f) mod 17

Figure 6.10 – ***Proportions des restes des facteurs premiers (MbedTLS).***

6.6 Conclusion

Dans ce chapitre, l'objectif était de remonter aux origines de la PGCD-vulnérabilité. Cela est passé par l'analyse des générateurs de clefs RSA des deux bibliothèques cryptographiques OpenSSL et MbedTLS (avec un traitement plus détaillé pour la première). La raison du choix d'analyser OpenSSL vient du fait que la plupart des modules PGCD-vulnérables trouvés dans le chapitre 5 vérifient la condition 6.1 (qui est connue comme étant une sorte d'empreinte d'OpenSSL). Quant à la bibliothèque MbedTLS, elle a été choisie pour la simple raison qu'elle est actuellement répandue dans les systèmes embarqués.

On a fait des expérimentations sur des Raspberry Pi 4 (avec Ubuntu Server 32 bits), car ces matériels sont similaires à ceux identifiés dans la chapitre 5 et dont les modules étaient PGCD-vulnérables. Par similarité, on entend des matériels ayant peu de ressources et n'ayant pas de sources d'entropie importantes (notamment due à une absence de clavier, de souris et d'activité disque). Lors des expérimentations, on s'est aussi mis presque dans les mêmes conditions que les matériels vulnérables identifiés. Plusieurs versions des bibliothèques analysées ont été étudiées en raison des différences existant entre leurs générateurs de clefs RSA. Ces versions vont de celles publiées au moment de la création de la base de données de l'EFF jusqu'à maintenant. Dans le cas d'OpenSSL, on les a reparties entre deux principaux groupes : 0.9.8 à 1.1.0l et 1.1.1 à 1.1.1j. Pour le premier groupe, il n'y a pas de différence majeure entre les générateurs de clefs RSA, mais il est sujet à la PGCD-vulnérabilité sous les conditions que l'on a imposées. Quant au second groupe, le changement du générateur fut radical avec le passage aux recommandations du NIST, et on ne parvint pas à trouver de modules PGCD-vulnérables lors des expérimentations.

Chapitre 7

Gestion et traitement des données analysées

Sommaire

Le but de ce chapitre est de présenter les codes sources des implémentations faites dans le chapitre 5 (principalement avec les outils PARI/GP et Python), la façon dont les données ont été traitées et les ressources (espace de stockage, mémoire RAM, temps d'exécution) nécessaires à leur manipulation.

Rappelons que les bases de données analysées dans le chapitre 5 sont au nombre de trois. La première provient du projet «EFF SSL Observatory», la seconde a été fournie par l'ANSSI, et la dernière est composée à la fois des données de Rapid7 et des données que nous avons nous-mêmes obtenues en parcourant l'Internet.

7.1 Forme des données provenant de l'EFF

Les données téléchargées sur la page web du projet [1] correspondent à la sauvegarde (avec la fonction `mysqldump`) des tables d'une base de données MySQL [2]. Pour l'utiliser, on a dû créer une base de données MySQL (du nom de `obseravtory`). La taille du fichier est de 16 Go, mais devient 4.5 Go après une compression en format `tar.gz` en mode par défaut [3].

Cette base de données est composée de plusieurs tables, mais celle qui nous intéresse est la table `all_certs` qui contient tous les certificats collectés et qui comprend plusieurs colonnes dont celles listées dans le Tableau 7.1.

Field	Type	Null	Key	Default	Extra
fingerprint	char(80)	NO	UNI	NULL	
ip	varchar(15)	YES		NULL	
Issuer	varchar(1893)	YES		NULL	
RSA Public Key:Modulus	varchar(6146)	YES		NULL	
RSA_Modulus_Bits	varchar(5)	YES		NULL	
Subject	varchar(5045)	YES		NULL	
valid	tinyint(1)	YES		NULL	
Validity:Not After	varchar(25)	YES		NULL	
Validity:Not Before	varchar(25)	YES		NULL	

Tableau 7.1 – ***Colonnes de la table*** `all_certs` ***pertinentes pour notre analyse.***

La colonne «`ip`» nous fournit les adresses IPv4 auxquelles les certificats furent collectés, «`RSA Public Key:Modulus`» et «`RSA_Modulus_Bits`» donnent respectivement les modules et leurs tailles. La colonne «`valid`» indique si le certificat était valide au moment où il a été recueilli. Les autres colonnes correspondent aux autres champs d'un certificat X.509 discutés dans la sous-section 3.4.2 du chapitre 3.

1. https://www.eff.org/fr/observatory.
2. https://www.mysql.com/.
3. `tar czvf observatory-dec-2010.sql.tar.gz observatory-dec-2010.sql`.

7.2 Forme des données provenant de l'ANSSI

Les données fournies par l'ANSSI sont un ensemble de 198 fichiers dont chacun contient environ un million de certificats X.509 disposés sous forme de DER concaténés. La norme DER (ou «Distinguished Encoding Rules») [18] est une norme d'encodage (en format binaire) pour les données respectant une syntaxe ASN.1 [70, 71]. Dans le cas présent, on l'utilise dans la norme X.509 [18] pour les certificats électroniques. Afin de comprendre le traitement des données provenant de l'ANSSI, on va présenter, dans la sous-section 7.2.1, le format utilisé par la norme DER. Toutefois, le lecteur pourra trouver plus de détails dans le RFC 5280 [18] et dans l'ouvrage «SSL and TLS Essentials : Securing the Web» [118] de Stephen Thomas.

7.2.1 La norme ASN.1 et l'encodage DER des certificats X.509

Comme le langage C, la notation ASN.1 a des types primitifs tels que `BOOLEAN`, `INTEGER` et `NULL`. Elle possède également des méthodes qui permettent aux utilisateurs de définir des combinaisons de ces types pour créer des objets plus complexes. Dans les certificats X.509, les constructions qui apparaissent sont principalement `SEQUENCE`, `SET` et `CHOICE`. Une autre caractéristique de la notation ASN.1 est le *balisage* (ou «Tagging») qui associe une valeur unique (appelée *balise* ou «Tag») à un objet ASN.1 ou un élément d'un objet ASN.1. Ces balises qui permettent de distinguer les objets ou les éléments ASN.1 les uns des autres sont réparties entre quatre classes : «Universal», «Application-Specific», «Context-Specific», «Private-Use». Ci-dessous, on présente la classe «Universal», les autres n'étant pas pertinentes pour les certificats X.509. Cette classe est utilisée pour distinguer les objets primitifs des objets construits. La norme ASN.1 définit des valeurs de balises «Universal» spécifiques pour tous les types primitifs et les opérations de construction. Par exemple, pour les objets de types `BOOLEAN`, `INTEGER`, `NULL` et `SEQUENCE`, les valeurs sont respectivement 1, 2, 5 et 16.

Lorsque la notation ASN.1 est utilisée pour définir des objets transmis à travers un réseau de communication, la norme ASN.1 définit plusieurs approches dont l'encodage DER (ou «Distinguished Encoding Rules») qui est utilisé notamment dans le cas des certificats X.509. Comme illustré sur la Figure 7.1, l'encodage en DER de la plupart des objets ASN.1 suit un format TLV (*balise* ou «Tag», *taille* ou «Length», *valeur* ou «Value») dans lequel chaque champ contient un ou plusieurs octets.

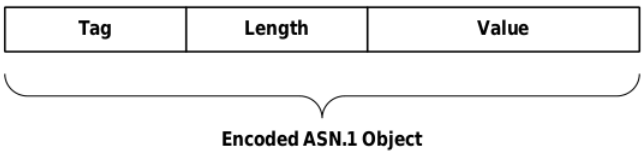


Figure 7.1 – ***Encodage d'un objet ASN.1 en DER.*** *(Source : [118])*

Les balises sont encodées de deux façons différentes selon que leurs valeurs numériques sont inférieures ou supérieures à l'entier 30. La Figure 7.2 illustre l'exemple d'une balise dont la valeur est inférieure ou égale à 30.

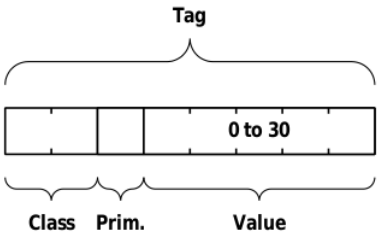


Figure 7.2 – ***Une balise de valeur numérique inférieure ou égale à 30.*** *(Source : [118])*

Les deux bits de poids fort indiquent la classe de la balise (voir Tableau 7.2), le bit suivant indique si l'objet est primitif ou construit, et les cinq autres représentent la valeur numérique de la balise. Par exemple, la balise d'un objet de type `SEQUENCE` est représentée en binaire par `0b00110000` et en hexadécimal par `0x30`. Les deux premiers bits `00` indiquent qu'il s'agit de la classe «Universal». Le bit suivant est `1` pour indiquer que l'objet est construit (et non primitif), et les cinq bits restants correspondent à l'entier 16 (valeur numérique de la balise).

Bits	Classes
00	«Universal»
01	«Application-Specific»
10	«Context-Specific»
11	«Private-Use»

Tableau 7.2 – ***Différentes classes de balise.***

Lorsque la valeur numérique de la balise est strictement supérieure à 30, la manière dont la balise est encodée est illustrée par la Figure 7.3. Les bits indiquant la classe de la balise et le caractère primitif ou construit de l'objet ASN.1 sont les mêmes que ceux de la Figure 7.2, mais les autres bits du premier octet sont tous égaux à 1. La valeur numérique de la balise est présente sur les octets suivants. Le bit de poids fort de ces octets est le bit d'*extension*. Ce bit vaut 1 dans tous les octets sauf le dernier. Les bits restants, lorsqu'ils sont concaténés, fournissent la valeur numérique de la balise.

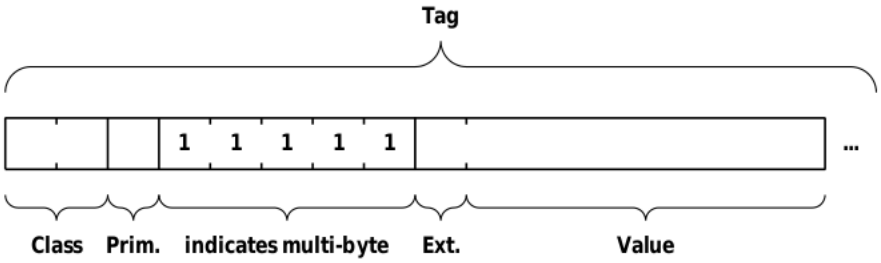


Figure 7.3 – ***Une balise de valeur numérique strictement supérieure à 30.*** *(Source : [118])*

Le champ *taille* correspond à la taille (en octets) du contenu du champ *valeur*. Son remplissage suit une stratégie similaire (mais non identique) à celui de la balise. Si la taille est inférieure ou égale à 127 octets, elle est représentée sur un seul octet (dont le bit de poids fort est évidemment égal à 0). Mais lorsqu'elle est strictement supérieure à 127 bits, plusieurs octets sont utilisés pour son encodage. Le bit de poids fort du premier octet vaut 1 et le reste de cet octet donne le nombre minimal d'octets k nécessaires à l'encodage de la taille. Les k octets suivants sont ensuite utilisés pour représenter la taille. En guise d'exemple, sur la Figure 7.4, le premier octet (0x30) indique qu'il s'agit d'un objet de type `SEQUENCE`; l'octet 0x82 montre que la taille est supérieure à 127 et qu'elle sera représentée par les deux octets suivants, à savoir 0x07AF (donc soit 1967 octets).

Figure 7.4 – ***Contenu partiel en DER d'un certificat X.509*** – *On a le contenu ASN.1 à gauche et la sortie de la commande `hexdump` à droite. Le décodage a été réalisé avec un outil*[4] *basé sur JavaScript et le certificat utilisé est celui du site de l'Institut Fourier*[5].

7.2.2 Extraction des informations des certificats X.509 fournis par l'ANSSI

À partir des explications données sur l'encodage DER dans la sous-section 7.2.1, on peut maintenant extraire les informations des certificats X.509 fournis par l'ANSSI en utilisant l'Algorithme 7.1 dont l'implémentation en Python est donnée par le Code-Source 7.1.

4. https://lapo.it/asn1js/.
5. https://www-fourier.univ-grenoble-alpes.fr/.

Algorithme 7.1 : Extraction des champs des certificats X.509 fournis par l'ANSSI

Entrées : les 198 fichiers provenant de l'ANSSI.
Sorties : trois fichiers MySQL dont `anonymous_RSA_dump.sql`.

```
1  début
2      pour chacun des 198 fichiers faire
3          v ← 0.
4          tant que la lecture des octets est possible faire
5              v ← v + 2 et u ← v.
6              b ← octet de rang v et v ← v + 1.
7              si b > 127 alors
8                  t ← v + b − 128.
9                  b ← octets de rangs : v, v + 1, ··· , t.
10                 v ← t + b.
11             c ← octets de rangs : u, u + 1, ··· , v.
12             si le certificat est du format X.509 et non corrompu alors
13                 si le certificat utilise le chiffrement RSA alors
14                     Extraire les champs qui nous intéressent.
15                     Puis les ajouter au fichier anonymous_RSA_dump.sql.
16                 sinon
17                     Ajouter le certificat au fichier anonymous_non_RSA_dump.sql.
18             sinon
19                 Ajouter le certificat au fichier anonymous_corrupted_dump.sql.
20         fin
21     fin
22 fin
```

Dans l'Algorithme 7.1, les variables u et v servent à indiquer les indices de début et de fin des octets qui constituent un certificat particulier. Après la lecture de l'octet de rang v à la ligne **6**, la représentation des données change selon que la valeur de b est inférieure ou supérieure à 127. Comme mentionné dans le dernier paragraphe de la sous-section 7.2.1, les deux possibilités qui se présentent sont les suivantes.

— Si b est inférieur ou égal à 127, il correspond à la taille du contenu du certificat.

— Sinon son premier bit est 1 et les bits restants donnent le nombre minimal d'octets k nécessaires à l'encodage de la taille. Puis on retrouve la taille sur les k octets suivants. C'est ce second cas qui convient plus à notre situation, car la plupart des certificats qui existent actuellement contiennent plus de 128 octets de données.

Dès lors que l'on a le début et la fin d'un certificat, on le reconstitue à la ligne **11**. S'il est corrompu, il est sauvegardé dans le fichier `anonymous_corrupted_dump.sql`. Par contre, s'il n'est pas corrompu, deux options se présentent : soit il ne contient aucune clef RSA et on le sauvegarde dans le fichier `anonymous_non_RSA_dump.sql`, soit il a une clef RSA et on extrait certains de ses champs puis on les ajoute au fichier `anonymous_RSA_dump.sql`.

Notons que le préfixe `anonymous` s'inspire du fait que les certificats fournis par l'ANSSI sont anonymes, c'est-à-dire que l'on ignore les adresses IP auxquelles ils ont été collectés. Notons également qu'à la ligne **5** de l'Algorithme 7.1, lorsque l'on passe d'un certificat à un autre on augmente de 2 la valeur de v pour ignorer l'octet nul qui préfixe chaque certificat.

Code-Source 7.1 – ***Extraction des champs des certificats X.509 fournis par l'ANSSI.***

```
#!/usr/bin/env python

# anssi_sql_dump_gen.py (2021-06-08)
# Copyright (C) 2018-2022 Mohamed Traoré (mohamed.traore1@univ-grenoble.fr)

# This program extracts some fields of the certificates that come from ANSSI and
 ↪  that use the RSA cryptosystem.
# It creates three SQL files : anonymous_RSA_dump.sql, anonymous_non_RSA_dump.sql
 ↪  and anonymous_corrupted_dump.sql

import os
import sys
import json
import glob
from OpenSSL import crypto

# global variables
tbnaA = "anonymous_RSA"
tbnaB = "anonymous_non_RSA"
tbnaC = "anonymous_corrupted"
dumpA = tbnaA + "_dump.sql"
dumpB = tbnaB + "_dump.sql"
dumpC = tbnaC + "_dump.sql"
l = 5000

# dumps initialization
with open(dumpA, 'w') as f:
  s  = "SET sql_mode = ALLOW_INVALID_DATES;\n"
  s += "DROP TABLE IF EXISTS %s;\n" % tbnaA
  s += "CREATE TABLE `%s` (\n" % tbnaA
  s += "`id` INTEGER AUTO_INCREMENT,\n"
  s += "PRIMARY KEY(`id`),\n"
  s += "`fingerprint` VARCHAR(40),\n"
  s += "`version` TINYINT,\n"
  s += "`serial number` TEXT,\n"
  s += "`signature algorithm` TEXT,\n"
  s += "`issuer` TEXT,\n"
  s += "`valid` BOOL,\n"
  s += "`not before` TIMESTAMP DEFAULT '1975-01-01 00:00:00',\n"
  s += "`not after` TIMESTAMP DEFAULT '1975-01-01 00:00:00',\n"
  s += "`subject` TEXT,\n"
  s += "`key size` SMALLINT,\n"
  s += "`modulus` TEXT,\n"
  s += "`exponent` TEXT) CHARACTER SET = utf8mb4;"
  f.write(s)

```

```
with open(dumpB, 'w') as f, open(dumpC, 'w') as g:
  b  = "DROP TABLE IF EXISTS %s;\n" % tbnaB
  b += "CREATE TABLE `%s` (\n" % tbnaB
  c  = "DROP TABLE IF EXISTS %s;\n" % tbnaC
  c += "CREATE TABLE `%s` (\n" % tbnaC
  s  = "`id` INTEGER AUTO_INCREMENT,\n"
  s += "PRIMARY KEY(`id`),\n"
  s += "`fingerprint` VARCHAR(40),\n"
  s += "`certificate` TEXT) CHARACTER SET = utf8mb4;"
  f.write(b + s)
  g.write(c + s)

# parse certs and save them into mysql files
def cert_saving():
  sqlA = "\nINSERT INTO `" + tbnaA + "` VALUES "
  sqlB = "\nINSERT INTO `" + tbnaB + "` VALUES "
  sqlC = "\nINSERT INTO `" + tbnaC + "` VALUES "
  ctrA = ctrB = ctrC = 0

  # open files
  outfA = open(dumpA, "a")
  outfB = open(dumpB, "a")
  outfC = open(dumpC, "a")

  # start writing
  d = glob.glob("certs/*")
  for fic in d:
    f = open(fic, 'rb')
    a = f.read(1)
    while a == b'0':
      k = 1
      try:
        b = int.from_bytes(f.read(1), byteorder='big')
        k+= 1
        if b > 127:
          k+= b - 128
          b = int.from_bytes(f.read(b - 128), byteorder='big')
        f.seek(-1*k, 1)
        der = f.read(b+k)
        cert = crypto.load_certificate(crypto.FILETYPE_ASN1, der)
        fing = cert.digest("sha1").decode("utf-8").replace(":", "").lower()
        publicKey = cert.get_pubkey()
        if publicKey.type() == crypto.TYPE_RSA:
          publicKey           = publicKey.to_cryptography_key()
          version             = cert.get_version()
          serialNumber        = hex(cert.get_serial_number()).replace("L","")
          signatureAlgorithm  = (cert.get_signature_algorithm()).decode("utf-8")
          issuer              = cert.get_issuer().get_components()
          # convert issuer into the right format
          s = ""
          for i in issuer:
            if i != issuer[len(issuer) - 1]:
              s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8") + ", "
```

```
98              else:
99                s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8")
100           issuer    = s.replace("\\", "\\\\").replace("\'", "\\\'")
101           valid     = int(not(cert.has_expired()))
102           notB      = list(cert.get_notBefore().decode("utf-8"))
103           notBefore = "".join(str(i) for i in notB[0:4]) + "-" + "".join(str(i)
              ↪  for i in notB[4:6]) + "-" + "".join(str(i) for i in notB[6:8]) + " "
              ↪  + "".join(str(i) for i in notB[8:10]) + ":" + "".join(str(i) for i
              ↪  in notB[10:12]) + ":" + "".join(str(i) for i in notB[12:14])
104           notA      = list(cert.get_notAfter().decode("utf-8"))
105           notAfter  = "".join(str(i) for i in notA[0:4]) + "-" + "".join(str(i)
              ↪  for i in notA[4:6]) + "-" + "".join(str(i) for i in notA[6:8]) + " "
              ↪  + "".join(str(i) for i in notA[8:10]) + ":" + "".join(str(i) for i
              ↪  in notA[10:12]) + ":" + "".join(str(i) for i in notA[12:14])
106           subject   = cert.get_subject().get_components()
107           # convert subject into the right format
108           s = ""
109           for i in subject:
110             if i != subject[len(subject) - 1]:
111               s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8") + ", "
112             else:
113               s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8")
114           subject  = s.replace("\\", "\\\\").replace("\'", "\\\'")
115           size     = publicKey.key_size
116           modulus  = hex(publicKey.public_numbers().n).replace("L","")
117           exponent = publicKey.public_numbers().e
118           # SQL code updating
119           sqlA += "(" + ", ".join(["NULL", "'"+fing+"'", "'"+str(version)+"'",
              ↪  "'"+serialNumber+"'", "'"+signatureAlgorithm+"'", "'"+issuer+"'",
              ↪  "'"+str(valid)+"'", "'"+notBefore+"'", "'"+notAfter+"'",
              ↪  "'"+subject+"'", "'"+str(size)+"'", "'"+modulus+"'",
              ↪  "'"+str(exponent)+"'"]) + "), \n"
120           ctrA += 1
121
122           # RSA certs saving
123           if ctrA > 1:
124             sqlA = sqlA[:-3] + ";"
125             outfA.write(sqlA)
126             sqlA = "\nINSERT INTO `" + tbnaA + "` VALUES "
127             ctrA = 0
128         else:
129           cr_dump = crypto.dump_certificate(crypto.FILETYPE_PEM, cert)
130           cr_dump = cr_dump.decode("utf-8").replace("-----BEGIN CERTIFICATE-----",
              ↪  "").replace("-----END CERTIFICATE-----", "").replace("\n", "")
131           sqlB += "(" + ", ".join(["NULL", "'"+fing+"'", "'"+cr_dump+"'"]) + "),
              ↪  \n"
132           ctrB += 1
133
134           # non-RSA certs saving
135           if ctrB > 1:
136             sqlB = sqlB[:-3] + ";"
137             outfB.write(sqlB)
138             sqlB = "\nINSERT INTO `" + tbnaB + "` VALUES "
```

```
139             ctrB = 0
140         except:
141           cr_dump = crypto.dump_certificate(crypto.FILETYPE_PEM, cert)
142           cr_dump = cr_dump.decode("utf-8").replace("-----BEGIN CERTIFICATE-----",
              ↪  "").replace("-----END CERTIFICATE-----", "").replace("\n", "")
143           sqlC += "(" + ", ".join(["NULL", "'"+fing+"'", "'"+cr_dump+"'"]) + "), \n"
144           ctrC += 1
145
146           # corrupted certs saving
147           if ctrC > 1:
148             sqlC = sqlC[:-3] + ";"
149             outfC.write(sqlC)
150             sqlC = "\nINSERT INTO `" + tbnaC + "` VALUES "
151             ctrC = 0
152         # go to the next certificate
153         a = f.read(1)
154       # close the current file
155       f.close()
156     # RSA certs saving
157     if ctrA > 0:
158       sqlA = sqlA[:-3] + ";"
159       outfA.write(sqlA)
160       sqlA = "\nINSERT INTO `" + tbnaA + "` VALUES "
161       ctrA = 0
162
163     # non-RSA certs saving
164     if ctrB > 0:
165       sqlB = sqlB[:-3] + ";"
166       outfB.write(sqlB)
167       sqlB = "\nINSERT INTO `" + tbnaB + "` VALUES "
168       ctrB = 0
169
170     # corrupted certs saving
171     if ctrC > 0:
172       sqlC = sqlC[:-3] + ";"
173       outfC.write(sqlC)
174       sqlC = "\nINSERT INTO `" + tbnaC + "` VALUES "
175       ctrC = 0
176
177     # close files
178     outfA.close()
179     outfB.close()
180     outfC.close()
181
182 if __name__ == "__main__":
183     # no need to create a dictionary to make certificate distinct
184     # they already are
185     cert_saving()
```

L'exécution du Code-Source 7.1 crée les fichiers SQL : `anonymous_corrupted_dump.sql`, `anonymous_non_RSA_dump.sql` et `anonymous_RSA_dump.sql` qui permettent de créer trois tables SQL : `anonymous_corrupted`, `anonymous_non_RSA` et `anonymous_RSA`. Chaque ligne

(sauf éventuellement la dernière) de chacun des trois fichiers permet d'insérer, avec une seule commande, 5000 lignes dans la table SQL correspondante. Le choix de ce nombre permet à la fois d'alléger les fichiers générés (en évitant plusieurs `INSERT INTO`) et d'être au-dessous de la quantité maximale de données pouvant être insérées avec une seule commande. Cette quantité, contenue dans la variable MySQL : `max_allowed_packet` [6], est de 16 Mo par défaut avec MariaDB 10.4.21 (version que l'on utilise) et doit être comprise entre 1 Mo et 1 Go.

Une ligne des tables `anonymous_corrupted` et `anonymous_non_RSA` est composée de l'empreinte SHA1 d'un certificat et le contenu PEM de ce certificat. Quant à `anonymous_RSA`, elle est composée de plusieurs champs comme on peut le constater dans le Tableau 7.3.

```
MariaDB [observatory]> DESC anonymous_RSA;
+---------------------+-------------+------+-----+---------------------+----------+
| Field               | Type        | Null | Key | Default             | Extra    |
+---------------------+-------------+------+-----+---------------------+----------+
| id                  | int(11)     | NO   | PRI | NULL                | auto_inc |
| fingerprint         | varchar(40) | YES  |     | NULL                |          |
| version             | tinyint(4)  | YES  |     | NULL                |          |
| serial number       | text        | YES  |     | NULL                |          |
| signature algorithm | text        | YES  |     | NULL                |          |
| issuer              | text        | YES  |     | NULL                |          |
| valid               | tinyint(1)  | YES  |     | NULL                |          |
| not before          | timestamp   | NO   |     | 1975-01-01 00:00:00 |          |
| not after           | timestamp   | NO   |     | 1975-01-01 00:00:00 |          |
| subject             | text        | YES  |     | NULL                |          |
| key size            | smallint(6) | YES  |     | NULL                |          |
| modulus             | text        | YES  |     | NULL                |          |
| exponent            | text        | YES  |     | NULL                |          |
+---------------------+-------------+------+-----+---------------------+----------+
```

Tableau 7.3 – ***Structure de la table*** *anonymous_RSA*.

Dans le Tableau 7.4, on donne la taille (avant et après compression en `tar.gz` en mode par défaut [7]) des données brutes et de chacun des fichiers créés par le Code-Source 7.1.

Données	**Tailles des données**	
	Avant compression	Après compression
Données brutes (en DER concaténés)	253 Go	124 Go
`anonymous_RSA_dump.sql`	147 Go	70 Go
`anonymous_non_RSA_dump.sql`	30 Go	10 Go
`anonymous_corrupted_RSA_dump.sql`	64 Mo	29 Mo
Totaux	430 Go	204 Go

Tableau 7.4 – ***Espace occupé par les données provenant de l'ANSSI.***

6. https://mariadb.com/kb/en/server-system-variables/#max_allowed_packet.
7. `tar czvf nom_fichier.sql.tar.gz nom_fichier.sql`.

7.3 Forme des données provenant de Rapid7

Les données du projet «Project Sonar» sont collectées en effectuant une requête TCP (ou «Transmission Control Protocol») [2] à des adresses IPv4 sur le port HTTPS en l'occurrence, suivie de l'exécution d'un script de collecte sur chaque matériel qui répond positivement. Les données recueillies sont ensuite comparées à celles des collectes précédentes. Toutes les nouvelles données sont téléchargées vers le projet et sauvegardées dans trois fichiers dont les noms sont préfixés par la date de collecte : `${date}_certs.gz`, `${date}_hosts.gz` et `${date}_names.gz`.

— Chaque ligne de `${date}_certs.gz` est composée du condensat SHA1 d'un certificat X.509 suivi de l'encodage base64 de ce certificat, c'est-à-dire que ce fichier fournit une correspondance entre les encodages base64 des certificats et leurs empreintes SHA1.

— Chaque ligne de `${date}_hosts.gz` comprend une adresse IPv4 et le condensat SHA1 du certificat trouvé à cette adresse, c'est-à-dire que ce fichier établit une correspondance entre des adresses IPv4 et les empreintes SHA1 des certificats présents à ces adresses. Si plusieurs certificats sont trouvés sur un hôte, les condensats SHA1 de ces certificats seront affichés dans l'ordre dans lequel ils ont été reçus. Il est en effet courant qu'un serveur SSL/TLS fournisse tous les certificats entrant dans la chaîne de vérification d'un certificat donné.

— Chaque ligne de `${date}_names.gz` contient le condensat SHA1 d'un certificat suivi du «`Common Name`» ou de l'une des entrées de «`SubjectAltName`». Notons qu'il est possible qu'un seul certificat soit associé à plusieurs noms.

Parmi les trois types de fichiers listés ci-dessus, uniquement les deux premiers nous intéressent ; le troisième pouvant être obtenu à partir des deux premiers. On a téléchargé ceux qui ont été collectés entre janvier 2017 et janvier 2021. À cet effet, on a dû créer un compte sur le site internet de Rapid7 [8]. Ce compte n'offrait par défaut que 30 téléchargements par jour. Compte tenu de la quantité de données à télécharger, et donc de l'impact négatif de cette limite sur nos travaux, on fit une demande d'augmentation auprès des équipes [9] de Rapid7. Leur réponse fut positive et la limite a été doublée. Cela nous a permis d'acquérir 181 fichiers de type `${date}_hosts.gz` et les 181 fichiers de type `${date}_certs.gz` correspondants.

Après l'acquisition des données, on est passé par une étape d'unification, car plusieurs données se répètent notamment les certificats de CA. D'une part, avec le Code-Source 7.2 et les fichiers `${date}_hosts.gz`, on créa un dictionnaire (`hf_dict.json`) dont chaque clef est l'empreinte SHA1 d'un certificat et la valeur associée à cette clef est une liste d'adresses IPv4 auxquelles on trouve ce certificat.

8. https://www.rapid7.com/.
9. research@rapid7.com.

Code-Source 7.2 – ***Création du dictionnaire*** *hf_dict.json*.

```
#!/usr/bin/env python

# hf_dict_creation.py (2021-02-09)
# Copyright (C) 2018-2022 Mohamed Traoré (mohamed.traore1@univ-grenoble.fr)

# This program creates a dictionary saves it in the file hf_dict.json.
# A key is the SHA1 fingerprint of a certificate; the value associated to that key
↪  is a list of IP addr where that certificate is found.

import sys
import glob
import json

# create hf_dict
def hf_dict_creation(hosts_dir):
  # hosts loading lines
  hf_list = glob.glob(hosts_dir + "/*")
  lines = set()
  for f in hf_list:
    with open(f, "r") as file:
      lines |= set(file.readlines())

  # make fing-ipadd dictionary
  hf_dict = dict()
  for line in lines:
    [hf_ipadd, hf_fing] = line.strip().split(",")
    try:
      hf_dict[hf_fing] += [hf_ipadd]
    except:
      hf_dict[hf_fing]  = [hf_ipadd]

  # save hf_dict
  with open("hf_dict.json", "w") as f:
    json.dump(hf_dict, f, indent=0)

if __name__ == "__main__":
  hf_dict_creation(sys.argv[1])
```

D'autre part, à partir des fichiers `${date}_certs.gz` et du Code-Source 7.3, un second dictionnaire (`cf_dict.json`) fut créé; chaque clef étant l'empreinte SHA1 d'un certificat et la valeur associée à cette clef étant le contenu PEM de ce certificat.

Code-Source 7.3 – ***Création du dictionnaire*** *cf_dict.json*.

```
#!/usr/bin/env python

# cf_dict_creation.py (2021-02-09)
# Copyright (C) 2018-2022 Mohamed Traoré (mohamed.traore1@univ-grenoble.fr)

# This program creates a dictionary saves it in the file cf_dict.json.
# A key is the SHA1 fingerprint of a certificate; the value associated to that key
↪  is the PEM content of that certificate.

import sys
import glob
import json

# create cf_dict
def cf_dict_creation(certs_dir):
  # certs loading lines
  cf_list = glob.glob(certs_dir + "/*")
  lines = set()
  for f in cf_list:
    with open(f, "r") as file:
      lines |= set(file.readlines())

  # make fing-certs dictionary
  cf_dict = dict()
  for line in lines:
    [cf_fing, cf_cert]  = line.strip().split(",")
    cf_dict[cf_fing]  = cf_cert

  # save cf_dict
  with open("cf_dict.json", "w") as f:
    json.dump(cf_dict, f, indent=0)

if __name__ == "__main__":
  cf_dict_creation(sys.argv[1])
```

Dès lors que les données ont été unifiées, l'étape qui a suivi consistait à parcourir les données, de disséquer les certificats X.509 utilisant le chiffrement RSA puis de les sauvegarder dans le fichier `opendata_RSA_dump.sql`. Les autres certificats (n'utilisant pas le chiffrement RSA ou corrompus) sont sauvegardés respectivement dans `opendata_non_RSA_dump.sql` et `opendata_corrupted_dump.sql`. Le préfixe `opendata` a été choisi pour faire référence au nom qui est attribué à ces données par Rapid7 (c'est-à-dire «Rapid7 Open Data»). La création des fichiers SQL a été effectuée avec le Code-Source 7.4 qui est similaire au Code-Source 7.1 notamment la phase de dissection des certificats.

Code-Source 7.4 – ***Extraction des champs des certificats X.509 provenant de Rapid7.***

```
#!/usr/bin/env python

# opendata_sql_dump_gen.py (2021-02-10)
# Copyright (C) 2018-2022 Mohamed Traoré (mohamed.traore1@univ-grenoble.fr)

# This program extracts some fields of the certificates that come from Rapid7 and
 ↪  that use the RSA cryptosystem.
# It creates three SQL files : opendata_RSA_dump.sql, opendata_non_RSA_dump.sql
 ↪  and opendata_corrupted_dump.sql

import os
import sys
import json
from OpenSSL import crypto

# global variables
tbnaA = "opendata_RSA"
tbnaB = "opendata_non_RSA"
tbnaC = "opendata_corrupted"
dumpA = tbnaA + "_dump.sql"
dumpB = tbnaB + "_dump.sql"
dumpC = tbnaC + "_dump.sql"
crtf  = "temp"
l = 5000

# dumps initialization
with open(dumpA, 'w') as f:
  s  = "SET sql_mode = ALLOW_INVALID_DATES;\n"
  s += "DROP TABLE IF EXISTS %s;\n" % tbnaA
  s += "CREATE TABLE `%s` (\n" % tbnaA
  s += "`id` INTEGER AUTO_INCREMENT,\n"
  s += "PRIMARY KEY(`id`),\n"
  s += "`ip number` INTEGER,\n"
  s += "`ip list` MEDIUMTEXT,\n"
  s += "`fingerprint` VARCHAR(40),\n"
  s += "`version` TINYINT,\n"
  s += "`serial number` TEXT,\n"
  s += "`signature algorithm` TEXT,\n"
  s += "`issuer` TEXT,\n"
  s += "`valid` BOOL,\n"
  s += "`not before` TIMESTAMP DEFAULT '1975-01-01 00:00:00',\n"
  s += "`not after` TIMESTAMP DEFAULT '1975-01-01 00:00:00',\n"
  s += "`subject` TEXT,\n"
  s += "`key size` SMALLINT,\n"
  s += "`modulus` TEXT,\n"
  s += "`exponent` TEXT) CHARACTER SET = utf8mb4;"
  f.write(s)

with open(dumpB, 'w') as f, open(dumpC, 'w') as g:
  b  = "DROP TABLE IF EXISTS %s;\n" % tbnaB
```

```
  b += "CREATE TABLE `%s` (\n" % tbnaB
  c  = "DROP TABLE IF EXISTS %s;\n" % tbnaC
  c += "CREATE TABLE `%s` (\n" % tbnaC
  s += "`id` INTEGER AUTO_INCREMENT,\n"
  s += "PRIMARY KEY(`id`),\n"
  s += "`ip number` INTEGER,\n"
  s += "`ip list` MEDIUMTEXT,\n"
  s += "`fingerprint` VARCHAR(40),\n"
  s += "`certificate` TEXT) CHARACTER SET = utf8mb4;"
  f.write(b + s)
  g.write(c + s)

# parse certs and save them
def cert_saving(cf_dict, hf_dict, keys):
  sqlA = "\nINSERT INTO `" + tbnaA + "` VALUES "
  sqlB = "\nINSERT INTO `" + tbnaB + "` VALUES "
  sqlC = "\nINSERT INTO `" + tbnaC + "` VALUES "
  ctrA = ctrB = ctrC = 0
  certfile = crtf + "_" + str(os.getpid())

  # open files
  outfA = open(dumpA, "a")
  outfB = open(dumpB, "a")
  outfC = open(dumpC, "a")

  # start writing
  for fing in keys:
    with open(certfile, "w") as f:
      f.write("-----BEGIN CERTIFICATE-----\n")
      f.write(cf_dict[fing] + "\n")
      f.write("-----END CERTIFICATE-----")
    try:
      cert = crypto.load_certificate(crypto.FILETYPE_PEM, open(certfile).read())
      publicKey = cert.get_pubkey()
      if publicKey.type() == crypto.TYPE_RSA:
        publicKey           = publicKey.to_cryptography_key()
        version             = cert.get_version()
        serialNumber        = hex(cert.get_serial_number()).replace("L","")
        signatureAlgorithm  = (cert.get_signature_algorithm()).decode("utf-8")
        issuer              = cert.get_issuer().get_components()
        # bien convertir issuer
        s = ""
        for i in issuer:
          if i != issuer[len(issuer) - 1]:
            s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8") + ", "
          else:
            s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8")
        issuer    = s.replace("\\", "\\\\").replace("\'", "\\\'")
        valid     = int(not(cert.has_expired()))
        notB      = list(cert.get_notBefore().decode("utf-8"))
        notBefore = "".join(str(i) for i in notB[0:4]) + "-" + "".join(str(i) for
        ↪  i in notB[4:6]) + "-" + "".join(str(i) for i in notB[6:8]) + " " +
        ↪  "".join(str(i) for i in notB[8:10]) + ":" + "".join(str(i) for i in
        ↪  notB[10:12]) + ":" + "".join(str(i) for i in notB[12:14])
```

```
100         notA      = list(cert.get_notAfter().decode("utf-8"))
101         notAfter  = "".join(str(i) for i in notA[0:4]) + "-" + "".join(str(i) for
             ↪  i in notA[4:6]) + "-" + "".join(str(i) for i in notA[6:8]) + " " +
             ↪  "".join(str(i) for i in notA[8:10]) + ":" + "".join(str(i) for i in
             ↪  notA[10:12]) + ":" + "".join(str(i) for i in notA[12:14])
102         subject   = cert.get_subject().get_components()
103         # bien convertir subject
104         s = ""
105         for i in subject:
106           if i != subject[len(subject) - 1]:
107             s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8") + ", "
108           else:
109             s = s + (i[0]).decode("utf-8") + "=" + (i[1]).decode("utf-8")
110         subject  = s.replace("\\", "\\\\").replace("\'", "\\\'")
111         size     = publicKey.key_size
112         modulus  = hex(publicKey.public_numbers().n).replace("L","")
113         exponent = publicKey.public_numbers().e
114         ip_list   = str(hf_dict[fing]).strip("[]").replace("'", "")
115         ip_numb   = str(len(hf_dict[fing]))
116         #sql code updating
117         sqlA += "(" + ", ".join(["NULL", "'"+ip_numb+"'", "'"+ip_list+"'",
             ↪  "'"+fing+"'", "'"+str(version)+"'", "'"+serialNumber+"'",
             ↪  "'"+signatureAlgorithm+"'", "'"+issuer+"'", "'"+str(valid)+"'",
             ↪  "'"+notBefore+"'", "'"+notAfter+"'", "'"+subject+"'",
             ↪  "'"+str(size)+"'", "'"+modulus+"'", "'"+str(exponent)+"'"]) + "), \n"
118         ctrA += 1
119
120         # RSA certs saving
121         if ctrA > 1:
122           sqlA = sqlA[:-3] + ";"
123           outfA.write(sqlA)
124           sqlA = "\nINSERT INTO `" + tbnaA + "` VALUES "
125           ctrA = 0
126       else:
127         ip_list   = str(hf_dict[fing]).strip('[]').replace("'", "")
128         ip_numb   = str(len(hf_dict[fing]))
129         sqlB += "(" + ", ".join(["NULL", "'"+ip_numb+"'", "'"+ip_list+"'",
             ↪  "'"+fing+"'", "'"+cf_dict[fing]+"'"]) + "), \n"
130         ctrB += 1
131
132         # Non-RSA certs saving
133         if ctrB > 1:
134           sqlB = sqlB[:-3] + ";"
135           outfB.write(sqlB)
136           sqlB = "\nINSERT INTO `" + tbnaB + "` VALUES "
137           ctrB = 0
138     except:
139       ip_list   = str(hf_dict[fing]).strip('[]').replace("'", "")
140       ip_numb   = str(len(hf_dict[fing]))
141       sqlC += "(" + ", ".join(["NULL", "'"+ip_numb+"'", "'"+ip_list+"'",
           ↪  "'"+fing+"'", "'"+cf_dict[fing]+"'"]) + "), \n"
142       ctrC += 1
143
144       # Corrupted certs saving
```

```
145         if ctrC > 1:
146           sqlC = sqlC[:-3] + ";"
147           outfC.write(sqlC)
148           sqlC = "\nINSERT INTO `" + tbnaC + "` VALUES "
149           ctrC = 0
150
151     # RSA certs saving
152     if ctrA > 0:
153       sqlA = sqlA[:-3] + ";"
154       outfA.write(sqlA)
155       sqlA = "\nINSERT INTO `" + tbnaA + "` VALUES "
156       ctrA = 0
157
158     # Non-RSA certs saving
159     if ctrB > 0:
160       sqlB = sqlB[:-3] + ";"
161       outfB.write(sqlB)
162       sqlB = "\nINSERT INTO `" + tbnaB + "` VALUES "
163       ctrB = 0
164
165     # Corrupted certs saving
166     if ctrC > 0:
167       sqlC = sqlC[:-3] + ";"
168       outfC.write(sqlC)
169       sqlC = "\nINSERT INTO `" + tbnaC + "` VALUES "
170       ctrC = 0
171
172     # close files
173     outfA.close()
174     outfB.close()
175     outfC.close()
176
177 if __name__ == "__main__":
178     # dict creation
179     #os.system("./cf_dict_creation.py " + sys.argv[1])
180     #os.system("./hf_dict_creation.py " + sys.argv[2])
181
182     # dict loading
183     cf_dict = dict()
184     hf_dict = dict()
185     with open("cf_dict.json", "r") as f, open("hf_dict.json", "r") as g:
186       cf_dict = json.load(f)
187       hf_dict = json.load(g)
188
189     # common fing
190     keys = set(cf_dict.keys()) & set(hf_dict.keys())
191
192     # update record_file
193     with open("record_file", "a") as f:
194       f.write("#keys : " + str(len(keys)) +"\n")
195
196     # saving certificates
197     cert_saving(cf_dict, hf_dict, keys)
198     os.system("rm temp*")
```

Dans le Tableau 7.5, on donne la structure de la table `opendata_RSA`. Elle est similaire à celle figurant dans le Tableau 7.3 ; la différence étant la présence des colonnes «`ip number`» et «`ip list`» dans la table `opendata_RSA`. Le premier donne le nombre d'adresses IPv4 auxquelles se trouve un certificat et le second fournit la liste de ces adresses.

```
MariaDB [observatory]> DESC opendata_RSA;
+---------------------+-------------+------+-----+---------------------+----------+
| Field               | Type        | Null | Key | Default             | Extra    |
+---------------------+-------------+------+-----+---------------------+----------+
| id                  | int(11)     | NO   | PRI | NULL                | auto_inc |
| ip number           | int(11)     | YES  |     | NULL                |          |
| ip list             | mediumtext  | YES  |     | NULL                |          |
| fingerprint         | varchar(40) | YES  |     | NULL                |          |
| version             | tinyint(4)  | YES  |     | NULL                |          |
| serial number       | text        | YES  |     | NULL                |          |
| signature algorithm | text        | YES  |     | NULL                |          |
| issuer              | text        | YES  |     | NULL                |          |
| valid               | tinyint(1)  | YES  |     | NULL                |          |
| not before          | timestamp   | NO   |     | 1975-01-01 00:00:00 |          |
| not after           | timestamp   | NO   |     | 1975-01-01 00:00:00 |          |
| subject             | text        | YES  |     | NULL                |          |
| key size            | smallint(6) | YES  |     | NULL                |          |
| modulus             | text        | YES  |     | NULL                |          |
| exponent            | text        | YES  |     | NULL                |          |
+---------------------+-------------+------+-----+---------------------+----------+
```

Tableau 7.5 – ***Structure de la table*** *opendata_RSA*.

Enfin, dans le Tableau 7.6, on donne la taille (avant et après compression en `tar.gz` en mode par défaut) des données brutes et de chacun des fichiers créés par le Code-Source 7.4.

Données	**Tailles des données**	
	Avant compression	Après compression
Fichiers `${date}_hosts.gz`	414 Go	146 Go
Fichiers `${date}_certs.gz`	802 Go	430 Go
`hf_dict.json`	24 Go	10 Go
`cf_dict.json`	243 Go	131 Go
`anonymous_RSA_dump.sql`	130 Go	66 Go
`anonymous_non_RSA_dump.sql`	4 Go	2 Go
`anonymous_corrupted_RSA_dump.sql`	17 Mo	10 Mo
Totaux	1,617 Go	785 Go

Tableau 7.6 – ***Espace occupé par les données provenant de Rapid7.***

7.4 Batch GCD

Dans cette section, on va présenter l'algorithme «Batch GCD» évoqué plusieurs fois dans le chapitre 5. Cet algorithme est utilisé pour déterminer les plus grands diviseurs communs (ou PGCD) des modules RSA lorsque l'on a une collection de modules RSA.

Supposons qu'on a deux modules RSA distincts $n_1 = p_1 \times q_1$ et $n_2 = p_2 \times q_2$. En calculant le PGCD de n_1 et n_2 (dont la complexité est en $\mathcal{O}(\log\min(n_1, n_2))$), on a : soit il vaut 1 (donc n_1 et n_2 sont étrangers), soit il correspond à un facteur commun de n_1 et n_2. On pensait que la seconde situation ne s'observait que si les facteurs premiers étaient relativement petits. On rappelle en effet qu'avec la distribution des nombres premiers (présentée dans la section 4.3 du chapitre 4), la probabilité de choisir *aléatoirement* le même nombre premier relativement grand comme facteur pour deux modules différents est quasiment nulle. Par exemple, la quantité de nombres premiers de 512 bits (resp. 1024 bits) est à peu près 2^{441} (resp. 2^{891}). Mais on a vu dans certaines études présentées dans la section 5.1 du chapitre 5 (dont celle de Heninger *et al.* [52]) que ce résultat n'est pas toujours observé en pratique. Pour aboutir à cette conclusion, ils ont constitué une collection de modules RSA puis ont vérifié si certains d'entre eux avaient des facteurs communs.

Soit C une collection de m modules RSA distincts $n_1, n_2, \cdots, n_m$. Une méthode naïve de déterminer les PGCD serait de calculer le PGCD de chaque couple $(n_i; n_j)$ pour $i < j$. Asymptotiquement, le calcul du PGCD de deux nombres de k bits étant en $\mathcal{O}(k^2)$, alors le calcul des PGCD de tous les couples $(n_i; n_j)$ sera en $\mathcal{O}\left(\frac{m}{2}k^2\right)$. Cependant, l'algorithme fourni dans les études de Heninger *et al.* [52] et de Hastings, Fried et Heninger [51] (à savoir «Batch GCD») peut le faire en temps quasi-linéaire.

L'idée de cet algorithme est assez simple; il décompose la résolution du problème en deux phases : calculer le produit de tous les modules $n = n_1 \times n_2 \times \cdots \times n_m$ puis tester si n_i a un facteur commun avec d'autres modules en vérifiant si $\text{PGCD}\left({n_i}^2, n\right) > n_i$.

Étape 1. Calculer $a \times b \times c \times d$ en faisant $(a \times b) \times (c \times d)$ est plus rapide que la multiplication de gauche à droite $(((a \times b) \times c) \times d)$. Cette méthode est appelée *arborescence de produits* (ou «product tree»). Elle permet de maintenir les nombres à multiplier sur la même taille et on rappelle qu'un produit de deux nombres de k bits se fait en $\mathcal{O}(k \times \log k \times \log\log k)$ avec une multiplication par transformée de Fourier rapide (comme l'algorithme de Schönhage-Strassen [104]).

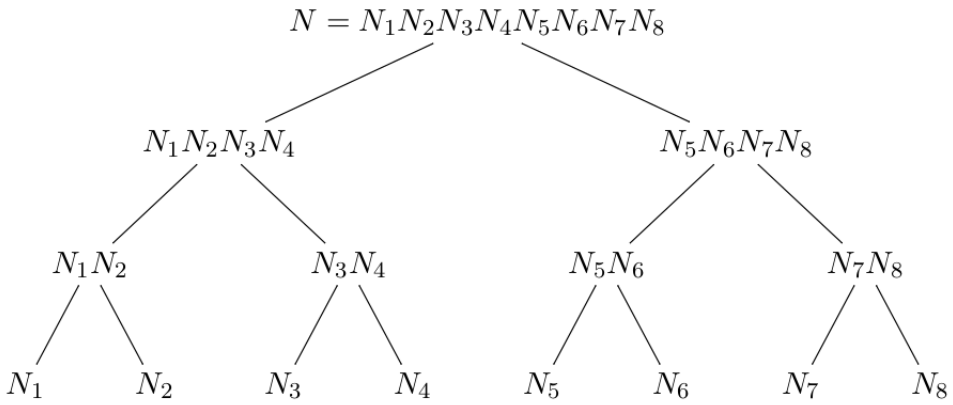


Figure 7.5 – ***Arborescence de produits avec 8 nombres.***

Étape 2. Une fois que le produit $n = n_1 \times n_2 \times \cdots \times n_m$ est calculé avec une arborescence de produits, on devra déterminer le PGCD$\left({n_i}^2, n\right)$ pour chaque i. Il s'agit là d'un calcul de PGCD où un nombre est largement plus grand que l'autre. En effet, ${n_i}^2$ est sur $2k$ bits alors que n est sur km bits. Cependant, on peut se ramener au calcul de PGCD de deux nombres de $2k$ bits en remarquant que PGCD$\left({n_i}^2, n\right)$ = PGCD$\left({n_i}^2, n \bmod {n_i}^2\right)$. Pour les m réductions modulaires, on utilise l'équation $n \bmod a = (n \bmod ab) \bmod a$ qui suggère aussi l'usage d'un arbre. Il est similaire à l'arborescence de produits, mais contrairement à ce dernier, il part de la racine aux feuilles. On l'appelle *arborescence de restes* et on l'illustre sur la Figure 7.6.

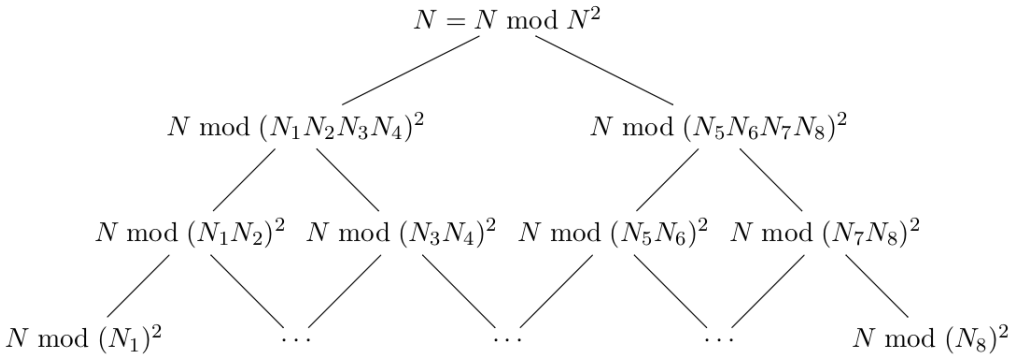


Figure 7.6 – ***Arborescence de restes avec 8 nombres.***

En résumé, on calcule le produit $n = \prod_{i=1}^{m} n_i$ avec une arborescence de produits, détermine les restes $r_i = n \bmod {n_i}^2$ avec une arborescence de restes puis calcule le PGCD$\left(n_i, \dfrac{r_i}{n_i}\right)$ pour tout i. Une implémentation en PARI/GP de l'algorithme est donnée par le Code-Source 7.5.

Code-Source 7.5 – ***Implémentation de «Batch GCD» en PARI/GP.***

```
/*
  batch_gcd.gp (2021-06-08)
  Copyright (C) 2018-2022 Mohamed Traoré (mohamed.traore1@univ-grenoble.fr)

  This program computes the common factors of RSA moduli.
  It produces two files : mods (which contains the factored moduli) and gcds (the
   ↪  common factors found).
*/

/*******************
  global variables
*******************/
PATH_F    = "batchGCD/";
INPUT_F    = Str(PATH_F, "input.txt");
OUTPUT_F  = Str(PATH_F, "output.txt");

```

```
/******************
  batch GCD
*******************/
product_tree() = {
  my(l = 0, V, W, f, T = getwalltime());
  V = Set(readvec(INPUT_F));
  while(#V > 1,
    print("Level ", l);
    W = vector((#V + 1) \ 2);
    parfor(i = 1, #V \ 2, i, j, W[j] = V[2*j - 1] * V[2*j]);
    if(#V % 2, W[#W] = V[#V]);
    s = Str(PATH_F, "P", l, ".gp");
    f = fileopen(s, "w");
    filewrite(f, W);
    fileclose(f);
    V = W;
    l++;
  );
  print("Product tree took ", getwalltime() - T, " ms.");
  return(l);
}

remainder_tree(level) = {
  my(l = level, s, P, V, W, f, T = getwalltime());
  s = Str(PATH_F, "P", l, ".gp");
  P = read(s);
  while(l > 0,
    print("Level ", l);
    l--;
    s = Str(PATH_F, "P", l, ".gp");
    V = read(s);
    parfor(i = 1, #V, i, j, V[j] = P[(j+1) \ 2] % ((V[j])^2));
    P = V;
  );
  V = Set(readvec(INPUT_F));
  W = vector(#V);
  parfor(i = 1, #V, i, j,
    W[j] = P[(j+1) \ 2] % ((V[j])^2);
    W[j] = W[j] \ V[j];
    W[j] = gcd(W[j], V[j]);
  );
  f = fileopen(OUTPUT_F, "w");
  filewrite(f, W);
  fileclose(f);
  print("Remainder tree took ", getwalltime() - T, " ms.");
}

batch_gcd() = {
  my(l, V, W, mods = [], gcds = [], g, m, T = getwalltime(), P, j);
  l = product_tree() - 1;
  remainder_tree(l);
  V = Set(readvec(INPUT_F));
  W = read(OUTPUT_F);
```

```
69    g = fileopen(Str(PATH_F, "gcds"), "w");
70    m = fileopen(Str(PATH_F, "mods"), "w");
71    for(i = 1, #W,
72      if(W[i] != 1,
73        filewrite(g, W[i]);
74        filewrite(m, V[i]);
75      );
76    );
77    fileclose(g);
78    fileclose(m);
79
80
81    P = readvec(Str(PATH_F, "gcds"));
82    g = fileopen(Str(PATH_F, "gcds"), "w");
83    for(i = 1, #P,
84      if(ispseudoprime(P[i]),
85        filewrite(g, P[i]),
86        \\else
87        j = 1;
88        while( (P[i] == P[j]) || (gcd(P[i], P[j]) == 1), j++);
89        filewrite(g, gcd(P[i], P[j]));
90      );
91    );
92    fileclose(g);
93    print("Batch GCD took ", getwalltime() - T, " ms.");
94  }
95
96  from_eff(filename, n) = {
97    my(a, b, i = 0);
98    a = fileopen(filename, "r");
99    b = fileopen(Str(PATH_F, "input.txt"), "w");
100   while(i < n && s = filereadstr(a),
101     filewrite(b, s);
102     i++;
103   );
104   fileclose(a);
105   fileclose(b);
106 }
```

Le Code-Source 7.5 a été utilisé pour déterminer les modules PGCD-vulnérables de nos trois bases de données dans le chapitre 5. On donne dans le Tableau 7.7 des informations sur la taille des données ainsi que les ordinateurs et ressources utilisés.

Sources	**Fichiers des modules**		**Mémoire GCD**	**Temps GCD**	**Ordinateurs**
	Non compressé	Compressé			
EFF	0.7 Go	1.2 Go	32 Go	3 H	Intel Xeon Platinum 8180M CPU 2.50GHz ×112
ANSSI	71 Go	41 Go	512 Go	130 H	
Rapid7	65 Go	37 Go	512 Go	169 H	AMD EPYC 7601 32-Core Processor

Tableau 7.7 – ***Ressources nécessaires à la détermination des modules PGCD-vulnérables.***

7.5 Localisation des matériels vulnérables

Dans le chapitre 5, on a observé quelques types de vulnérabilités, et pour certains types (modules de petites tailles et modules PGCD-vulnérables), on a fourni des figures montrant les positions géographiques des matériels ayant ces vulnérabilités. Le Code-Source 7.6 permit la réalisation de ces figures. Puisque la localisation repose sur les adresses IP, cela n'a été possible que pour les certificats vulnérables provenant de la base de données de Rapid7. On a récupéré les adresses IP associées aux certificats vulnérables puis on a utilisé l'outil `geoip2` [10] qui est une bibliothèque Python permettant de localiser une adresse IPv4. Les coordonnées géographiques obtenues à la suite de cette opération sont utilisées par le Code-Source 7.6.

Code-Source 7.6 – ***Localisation des matériels sur une carte.***

```
#!/usr/bin/env python3

# draw_points_on_map.py (2021-06-08)
# Copyright (C) 2018-2022 Mohamed Traoré (mohamed.traore1@univ-grenoble.fr)

# This program draws some points on the world map.
# It takes a set of (lat, long) coordinates and creates a map

# Load libraries
import numpy
import pandas
import matplotlib
import matplotlib.colors as colors
import matplotlib.pyplot as plt
from matplotlib import cm
from numpy import linspace
from mpl_toolkits.basemap import Basemap

# Define font similar to fourier
plt.rcParams.update({
  "text.usetex": True,
  "font.family": "serif",
  "font.serif": ["New Century Schoolbook"],
})

# Load data from ods file
cities = pandas.read_csv('GCD_city_lat_long.csv').sort_values(by=['Nombre'],
 ↪  ascending=True) # sort in order to have red on the top
#cities = pandas.read_csv('small_city_lat_long.csv').sort_values(by=['Nombre'],
 ↪  ascending=True)

# Variables
exp = 1/3 #1.5 # 1/2.0
alp = 1
col = 6 # 9

```

10. https://pypi.org/project/geoip2/.

```
35 # Extract the data
36 lat = cities['Latitude'].values
37 lon = cities['Longitude'].values
38 population = cities['Nombre'].values
39 area = len(population) * [1]
40
41 # Draw the map background
42 fig = plt.figure(figsize=(10, 8))
43 m = Basemap(projection='eck4', lon_0=0, resolution='c')
44 m.drawcoastlines(color='gray')
45 m.drawcountries(color='gray')
46
47 # Create nonlinear colorbar
48 leng = 100
49 #leng = 200
50
51 cm_subsection = linspace(0.5, 0.9, leng)
52 temp = [ cm.jet(x) for x in cm_subsection ]
53 collist = temp + ( max(population) - leng ) * [ cm.jet(1.0) ] # temp[leng-1] ]
54 cmap = colors.ListedColormap(collist)
55
56 # Scatter city data, with color reflecting population and size reflecting area
57 m.scatter(lon, lat, latlon=True, c=population, s=area, cmap=cmap, alpha=alp,
   ↪ zorder=2)
58
59 # Draw colorbar and legend
60 leng = max(population)
61 bound = [1] + [10*(i+1) for i in range(9) ] + list(range(100, leng, (leng-1) //
   ↪ (col-1)))[:-1] + [leng]
62 #bound = [1] + [20*(i+1) for i in range(9) ] + list(range(200, leng, (leng-1) //
   ↪ (col-1)))[:-1] + [leng]
63 plt.colorbar(label='Nombre de modules de tailles inférieures à 768 bits',
   ↪ boundaries=bound, shrink=0.5)
64
65 # Save as pgf
66 plt.savefig("final/pdf/GCD_city_lat_long.pdf", bbox_inches='tight')
67 #plt.savefig("final/pdf/small_city_lat_long.pdf", bbox_inches='tight')
```

7.6 Identification des matériels de modules PGCD-vulnérables

Dès lors que l'on a obtenu les modules PGCD-vulnérables dans la sous-section 5.3.4 du chapitre 5, on les a classifiés selon les matériels dont ils proviennent. Cette classification s'est faite de différentes façons. L'origine de certains certificats de modules PGCD-vulnérables a pu être connue à travers les informations contenues dans leurs champs «`Subject`». Pour ceux qui ne figuraient pas dans cette catégorie, on a été obligé d'utiliser soit un navigateur web, soit l'outil `Nmap` [76] sur les adresses auxquelles ces certificats furent collectés.

Après toutes ces investigations, on a établi le Tableau 7.8 faisant une correspondance entre les matériels et les contenus du champ «`Subject`» de leurs certificats respectifs.

Tableau 7.8 – ***Correspondance entre fabricants et contenus de «Subject»*** *– Le symbole «%» figurant dans les «`Subject`» représente des informations spécifiques au matériel utilisé telles qu'une adresse MAC ou un numéro de série.*

Fabricants	Contenus de «Subject»
2Wire	`C=US, O=2Wire, OU=Gateway Device, serialNumber=%, CN=Gateway Authentication`
Adtran	`C=US, ST=AL, L=Huntsville, O=ADTRAN, Inc., CN=NetVanta`
B+B SmartWorx	`C=CZ, ST=Czech Republic, O=Advantech B+B SmartWorx s.r.o., CN=%`
Alarm.com	`C=US, ST=Virginia, L=Virginia, O=Alarm.com, OU=Alarm.com, CN=www.alarm.com`
Asus	`C=US`
	`C=US, CN=192.168.1.1`
	`C=US, CN=router.asus.com`
AudioCodes	`CN=ACL_%` `O=AudioCodes, CN=420HD_%`
AVM	`%.myfritz.net`
CalAmp	`O=CalAmp Corp., OU=Wireless Networks Group, CN=CalAmp Corp.`
Cisco Linksys	`C=US, CN=Linksys, OU=Division, O=Cisco-Linksys,LCC, L=Irvine, ST=California`
	`C=US, ST=California, L=Irvine, O=Cisco-Linksys, LLC, OU=Division, CN=Linksys, emailAddress=support@linksys.com`
Cisco Systems	`CN=%, OU=RV%, O=Cisco Systems, Inc., C=US`
	`O=Cisco Systems, Inc., OU=Cisco Small Business, serialNumber=PID:RV130-WB-K9-G5 SN:%`
	`CN=CISCO-serialNumber/UDI:PID=SA520-K9,SN=%, OU=SA520-K9, O=Cisco Systems, Inc., C=US`
Comset	`C=US, CN=Comset, CN=Router`
	`C=US, CN=Comset, CN=Router, CN=Quicksilver, CN=5`
cPanel	`C=US, ST=TX, L=Houston, O=cPanel, Inc., CN=cPanel, Inc. Certification Authority`
CTSystem	`CN=localhost`
D-Link	`O=D-Link`
	`C=TW, ST=Taiwan, L=Taipie, O=D-LINK, OU=DHPD Dept., CN=www.dlink.com`
	`CN=%, OU=Certificate for DSR (Self-Signed), O=D-Link Corporation, C=TW`
	`C=TW, ST=Asia, L=Asia, O=D-Link Corporation, OU=D-Link Corporation, CN=www.dlink.com`
	`C=CN, ST=TAIWAN, L=TAIWAN, O=D-Link Corporation, OU=D-Link Corporation, CN=D-Link Corporation`
DrayTek	`C=TW, ST=HsinChu, L=HuKou, O=DrayTek Corp., OU=DrayTek Support, CN=Vigor Router`
Epson	`C=JP, CN=%, O=SEIKO EPSON CORP.`
Fortinet	`CN=FGT30B%, O=Fortinet Ltd.`
	`CN=FGT50B%, O=Fortinet Ltd.`
	`O=Fortinet Ltd., CN=FAP14C%`

Honeywell	C=US, ST=Wisconsin, L=135_West_Forest_Hill_Avenue,Oak_Creek,53154, O=Honeywell, OU=Access_Systems, CN=%, emailAddress=info@honeywell.com
HP	O=HP, OU=HP, CN=%, ST=STATE, L=HP, C=CN, emailAddress=%.Local
	C=US, ST=Texas, L=Houston, O=Hewlett-Packard Company, OU=ISS, CN=%
Huawei	CN=%, emailAddress=root@%
	name=%-Self-Signed-Certificate-%
	unstructuredName=AR101GW-Lc-S-Self-Signed-Certificate-%
	C=In, O=Huawei, OU=HTIPL, dnQualifier=LSW, ST=Karnataka, CN=Quidway
	C=CN, O=HUAWEI, OU=Switch & Enterprise Gateway Product Line, CN=%.huawei.com
	C=In, O=Huawei, OU=HTIPL, dnQualifier=DN qualifier, ST=Karnataka, CN=EndEntity, serialNumber=serial, L=locality, title=title, SN=surname, GN=givenName, initials=ini, pseudonym=pseudo, generationQualifier=gen, emailAddress=email@@@, DC=domainComponent
Icotera	CN=%.icotera
Innominate	C=DE, ST=Berlin, L=Berlin, O=Innominate Security Technologies AG, CN=%, emailAddress=mguard@innominate.com
Juniper	CN=self-signed, CN=system generated, CN=%
Kronos	CN=%, O=Kronos, OU=Kronos Incorporated, L=Chelmsford, ST=MA, C=USA
Netgear	OU=Certificate for SRX5308 (Self-Signed), O=Netgear, C=US
	CN=Netgear VPN Firewall , OU=Netgear Prosafe, O=Netgear Inc., C=US
Netonix	C=US
RSAWeb	C=US, CN=OlarmBase1
RubyTech	CN=PSGS-2314J
Sagem	C=FR, ST=France, L=Paris, O=SAGEM, OU=URD3
Samsung	O=SAMSUNG, OU=SAMSUNG, CN=%, ST=STATE, L=SAMSUNG, C=CN, emailAddress=%
Sangfor	C=CN, ST=guangdong, L=shenzhen, O=sangfor, OU=sslvpn, CN=sslvpn, emailAddress=ssl@sangfor.com
Schmid Tel	C=IN, ST=India, L=New Delhi, O=Schmid Telecom Ind Pvt. Ltd, OU=Sales, CN=www.schmid-telecom.in
SonicWALL	C=US, ST=California, L=Sunnyvale, O=HTTPS Management Certificate for SonicWALL (self-signed), OU=HTTPS Management Certificate for SonicWALL (self-signed), CN=%
Technicolor	CN=Technicolor, OU=Technicolor, O=Technicolor, L=Hsinchu, ST=Taiwan, C=TW
TomatoUSB	C=US, CN=TomatoUSB
TP-Link	CN=tplinkwifi.net
	C=CN, ST=GZ, L=SZ, O=TPLINK, CN=SMB
	C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R4_%, emailAddress=service@mail.com

Tridium	`CN=Niagara4, O=Tridium, C=US`
Xerox	`C=US, ST=Oregon, L=Wilsonville, CN=%, O=Xerox Corporation, OU=Xerox Office Business Group`
	`C=US, ST=Oregon, L=Wilsonville, CN=%, O=Xerox Corporation, OU=Xerox Office Business Group, CN=%, CN=%`
	`C=US, ST=Oregon, L=Wilsonville, CN=%, O=Xerox Corporation, CN=%, OU=Xerox Office Business Group, CN=%`
ZyXEL	`CN=ZyWALL 2 Plus%`
	`CN=My Name, OU=My Organization`
	`C=TW, ST=TWN, O=ZyXEL, OU=ZyXELcert, CN=ZyXELcert`

7.7 Conclusion

Dans ce chapitre, on a discuté des implémentations ayant été faites pour la réalisation des analyses effectuées principalement dans le chapitre 5. On a également présenté les formes ainsi que les tailles des données analysées. Il a été nécessaire de faire un rappel de la notation ASN.1 et du format DER pour comprendre le traitement des données provenant de l'ANSSI (qui étaient sous forme de DER concaténés). Dans la section 7.4, on a exposé l'algorithme «Batch GCD» que l'on a utilisé pour déterminer les modules PGCD-vulnérables.

Quant à la section 7.5, son objectif était d'expliquer la façon dont on a localisé des matériels vulnérables pour ensuite les placer sur une carte. Enfin, dans la section 7.6, on a fourni un tableau de correspondance entre les fabricants de matériels vulnérables trouvés et les contenus des champs «`Subject`» de leurs certificats.

Conclusion

Au cours de cette thèse, notre but principal était d'analyser des certificats SSL/TLS X.509 utilisant le chiffrement RSA et provenant de matériels connectés à la recherche d'éventuelles anomalies. Ces analyses, effectuées dans le chapitre 5 en suivant une méthodologie similaire à celle de Hastings, Fried et Heninger [51], nous ont permis de détecter plusieurs vulnérabilités : modules de petites tailles, modules redondants, certificats invalides mais toujours en usage, modules vulnérables à l'attaque ROCA ainsi que des modules PGCD-vulnérables. Pour la base de données de Rapid7 (qui est la plus récente), on trouva 1,958,278 certificats qui ont des modules dont les tailles sont inférieures à 768 bits, soit 0.34% du nombre de total de certificats. Pour la même base de données, on a pu trouver 1,550,382 certificats dont les modules sont PGCD-vulnérables, soit 0.27% du nombre total de certificats.

Bien que les études présentées dans la section 5.1 du chapitre 5 aient déjà signalé l'existence des modules PGCD-vulnérables, elles n'ont pu factoriser aucun module de 2048 bits contrairement à la notre. En effet, on a pu factoriser 14,765 modules de 2048 bits (correspondants aux certificats de 18,139 adresses), ce qui, à notre connaissance, n'a jamais été fait.

À partir des modules PGCD-vulnérables issus des pare-feux «ZX_ZW2P», on a constaté que cette famille de matériels générait ses modules RSA de 512 bits d'une manière particulière. Des analyses approfondies ont conduit, d'une part, à une rétro-conception partielle du générateur utilisé, et d'autre part, à la factorisation instantanée des modules (de 512 bits) de plusieurs milliers de certificats provenant de cette famille. Pour la base de données de Rapid7, les modules provenant des «ZX_ZW2P» et que l'on a pu factoriser instantanément sont au nombre 42, mais à cause de la redondance de ces modules, cette quantité représente les certificats provenant de 8,817 adresses IPv4.

Après avoir remarqué que la plupart des modules PGCD-vulnérables avaient été générés par la bibliothèque OpenSSL, on a analysé, dans le chapitre 6, les processus de génération de clefs RSA de plusieurs versions de cette bibliothèque. En se mettant quasiment dans les mêmes conditions que les matériels vulnérables identifiés, les expérimentations nous ont permis de remonter aux causes de la PGCD-vulnérabilité.

En terme de perspectives, certains points peuvent être améliorés. Il serait intéressant d'envisager d'effectuer les mêmes types d'analyses sur d'autres chiffrements asymétriques tels que le chiffrement ElGamal ou encore les chiffrements basés sur les courbes elliptiques. L'anomalie que l'on a signalée dans la sous-section 5.3.8 et qu'on a qualifiée de *potentielle tentative d'usurpation d'identité* mérite aussi d'être analysée de façon approfondie. Quant aux analyses effectuées dans le chapitre 6, on pourrait faire de même pour d'autres biblio-

thèques cryptographiques qui se rependent de plus en plus. Un autre point intéressant consisterait à proposer des contre-mesures *raisonnables* à la PGCD-vulnérabilité causée par les anciennes versions d'OpenSSL. Par raisonnables, on entend des contre-mesures pouvant être utilisées sans nécessiter d'importantes ressources. En effet, comme on l'a déjà signalé dans le Tableau 5.14 du chapitre 5, la plupart des matériels affectés par la PGCD-vulnérabilité et que l'on a pu identifier sont des matériels bas de gamme. On pourrait donc s'attendre à ce que leurs fabricants ne veuillent pas consacrer beaucoup de ressources pour apporter des correctifs afin d'éviter cette vulnérabilité.

Bibliographie

[1] OpenMPI High Performance Message Passing Library. https://www.open-mpi.org. [57]

[2] Transmission Control Protocol. RFC 793, Sept. 1981. [126]

[3] IEEE Standard Specifications for Public-Key Cryptography. *IEEE Std 1363-2000*, pages 1–228, 2000. https://doi.org/10.1109/IEEESTD.2000.92292. [3, 39, 44, 58, 93]

[4] M. Agrawal, N. Kayal, and N. Saxena. PRIMES Is in P. *Annals of Mathematics*, 160(2) :781–793, 2004. https://doi.org/10.4007/annals.2004.160.781. [45]

[5] W. Alford, A. Granville, and C. Pomerance. There are Infinitely Many Carmichael Numbers. *Annals of Mathematics*, 139, 10 1995. [42]

[6] C. Allen and T. Dierks. The TLS Protocol Version 1.0. RFC 2246, Jan. 1999. [33]

[7] F. Arnault. Rabin-Miller Primality Test : Composite Numbers Which Pass It. *Mathematics of Computation - Math. Comput.*, 64 :355–361, 01 1995. [44]

[8] A. B. Association. ANSI X9.31-1998 Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA), 1998. [v, 41, 44, 45, 60]

[9] A. O. L. Atkin and F. Morain. Elliptic Curves and Primality Proving. *Mathematics of Computation*, 61(203) :29–68, 1993. [45]

[10] E. Bach. Explicit bounds for primality testing and related problems. *Mathematics of Computation*, 55(191) :355–380, 1990. [44]

[11] R. Baillie and S. S. Wagstaff. Lucas Pseudoprimes. *Mathematics of Computation*, 35(152) :1391–1417, 1980. [44]

[12] P. Bakker. MbedTLS. https://tls.mbed.org/. [110]

[13] E. Barker. SP 800-57 (Part 1 Rev. 5), Recommendation for Key Management - Part 1 : General. Technical report, Gaithersburg, MD, USA, 2020. https://doi.org/10.6028/NIST.SP.800-57pt1r5. [60]

[14] E. Barker and J. Kelsey. SP 800-90A (Rev. 1), Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, Gaithersburg, MD, USA, 2015. https://doi.org/10.6028/NIST.SP.800-90Ar1. [60, 106, 107, 108, 112]

[15] E. Barker and J. Kelsey. SP 800-90C, Recommendation for Random Bit Generator (RBG) Constructions. Technical report, Gaithersburg, MD, USA, 2016. [60]

[16] R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0. RFC 7568, June 2015. [33]

[17] D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. van Someren. Factoring RSA Keys from Certified Smart Cards : Coppersmith in the Wild. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, pages 341–360, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-42045-0_18. [2, 55, 66, 68, 73]

[18] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008. [3, 12, 27, 117]

[19] D. Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2) :203–213, 1999. [39, 81]

[20] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. [48]

[21] D. Boneh, G. Durfee, and Y. Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In K. Ohta and D. Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 25–34, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-49649-1_3. [56]

[22] D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT'98*, pages 59–71, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0054117. [39]

[23] BSI - Technical Guideline. Cryptographic Mechanisms : Recommendations and Key Lengths, 2020. [3, 60]

[24] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security Privacy*, 7(1) :78–81, 2009. [34]

[25] E. Canfield, P. Erdös, and C. Pomerance. On a problem of Oppenheim concerning "factorisatio numerorum". *Journal of Number Theory*, 17(1) :1–28, 1983. [54]

[26] S. Choudhury, K. Bhatnagar, and W. Haque. *Public Key Infrastructure Implementation and Design*. ITPro collection. Wiley, 2002. [v, 12, 30]

[27] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer Publishing Company, Incorporated, 2010. [38, 43, 45, 51]

[28] D. Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10(4) :233–260, 1997. https://doi.org/10.1007/s001459900030. [3, 55, 56, 68, 73, 85]

[29] J.-S. Coron. Finding Small Roots of Bivariate Integer Polynomial Equations Revisited. In C. Cachin and J. L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 492–505, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-24676-3_29. [55, 85]

[30] J.-S. Coron and A. May. Deterministic Polynomial-Time Equivalence of Computing the RSA Secret Key and Factoring. *J. Cryptol.*, 20(1) :39–50, Jan. 2007. https://doi.org/10.1007/s00145-006-0433-6. [39]

[31] C. de La Vallée Poussin. *Recherches analytiques sur la théorie des nombres premiers*. Number vol. 1 à 5 in Recherches analytiques sur la théorie des nombres premiers. Hayez, 1897. [47]

[32] B. de Weger. Cryptanalysis of RSA with Small Prime Difference. *Applicable Algebra in Engineering, Communication and Computing*, 13 :17–28, 01 2002. [52]

[33] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Apr. 2006. [33]

[34] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, 1976. https://doi.org/10.1109/tit.1976.1055638. [15, 38]

[35] J. Dumas, P. Lafourcade, and P. Redon. *Architectures de sécurité pour internet - 2e éd. : Protocoles, standards et déploiement*. Dunod, 2020. [12]

[36] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 291–304, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2504730.2504755. [68]

[37] M. J. Dworkin. SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation : Methods and Techniques. Technical report, Gaithersburg, MD, USA, 2001. https://doi.org/10.6028/NIST.SP.800-38A. [7, 107]

[38] M. J. Dworkin. SP 800-38B, Recommendation for Block Cipher Modes of Operation : the CMAC Mode for Authentication. Technical report, Gaithersburg, MD, USA, 2005. https://doi.org/10.6028/NIST.SP.800-38B. [8]

[39] M. J. Dworkin. SP 800-38A Addendum. Recommendation for Block Cipher Modes of Operation : Three Variants of Ciphertext Stealing for CBC Mode. Technical report, Gaithersburg, MD, USA, 2010. https://doi.org/10.6028/NIST.SP.800-38A-Add. [7]

[40] Electronic Frontier Foundation. EFF SSL Observatory, 2010. https://www.eff.org/fr/observatory. [66, 67, 68, 70]

[41] D. T. Elgamal and K. E. Hickman. The SSL Protocol. Internet-Draft draft-hickman-netscape-ssl-00, Internet Engineering Task Force, Apr. 1995. Work in Progress. [33]

[42] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4) :469–472, 1985. https://doi.org/10.1109/tit.1985.1057074. [38, 67]

[43] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering : Design Principles and Practical Applications*. Wiley Publishing, 2010. https://doi.org/10.1002/9781118722367. [7, 8]

[44] H. Finney, L. Donnerhacke, J. Callas, R. L. Thayer, and D. Shaw. OpenPGP Message Format. RFC 4880, Nov. 2007. [3, 12, 15, 29, 67]

[45] A. O. Freier, P. Karlton, and P. C. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, Aug. 2011. [33]

[46] J. Friedlander and H. Iwaniec. *Opera de Cribro*. American mathematical society colloquium publications. American Mathematical Society, 2010. [38]

[47] F. Goichon, C. Lauradoux, G. Salagnac, and T. Vuillemin. Entropy transfers in the Linux Random Number Generator. Research Report RR-8060, INRIA, Sept. 2012. [110]

[48] J. S. Hadamard. Etude sur les propriétés des fonctions entières et en particulier d'une fonction considérée par Riemann. *Journal de Mathématiques Pures et Appliquées*, pages 171–216. [47]

[49] J. S. Hadamard. Sur la distribution des zéros de la fonction $\zeta(s)$ et ses conséquences arithmétiques. *Bulletin de la Société Mathématique de France*, 24 :199–220. [47]

[50] J. Håstad. Solving Simultaneous Modular Equations of Low Degree. *SIAM J. Comput.*, 17 :336–341, 1988. [55]

[51] M. Hastings, J. Fried, and N. Heninger. Weak Keys Remain Widespread in Network Devices. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 49–63, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2987443.2987486. [4, 66, 68, 70, 72, 88, 134, 143]

[52] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs : Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 35, USA, 2012. USENIX Association. [2, 66, 67, 68, 70, 73, 76, 134]

[53] T. Howes, S. Kille, and W. Yeong. X.500 Lightweight Directory Access Protocol. RFC 1487, July 1993. [17]

[54] N. Howgrave-Graham. Finding small roots of univariate modular equations revisited. In M. Darnell, editor, *Crytography and Coding*, pages 131–142, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. [55]

[55] D. Johnston. *Random Number Generators–Principles and Practices : A Guide for Engineers and Programmers*. De Gruyter, Incorporated, 2018. [vi, 108, 109]

[56] B. Kaliski. Timing Attacks on Cryptosystems. RSA Laboratories Bulletin, 1996. [48]

[57] B. Kaliski and M. Robshaw. The Secure Use of RSA. CryptoBytes, 1995. [48]

[58] J. Kelsey, S.-j. Chang, and R. Perlner. SP 800-185, SHA-3 Derived Functions : cSHAKE, KMAC, TupleHash and ParallelHash. Technical report, Gaithersburg, MD, USA, 2016. https://doi.org/10.6028/NIST.SP.800-185. [8]

[59] C. F. Kerry and C. Romine. FIPS PUB 186-4 Federal Information Processing Standards Publication : Digital Signature Standard (DSS), 2013. [3, 41, 44, 59, 60, 67, 107, 112]

[60] J. Kilgallin and R. Vasko. Factoring RSA Keys in the IoT Era. In *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 184–189, 2019. [70]

[61] S. Kille. A String Representation of Distinguished Names. RFC 1779, Mar. 1995. [22]

[62] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-Bit RSA Modulus. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 333–350, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14623-7_18. [72]

[63] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.) : Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997. [107]

[64] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48 :203–209, 1987. [38]

[65] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-68697-5_9. [48]

[66] L. M. Kohnfelder. Towards a Practical Public-key Cryptosystem. 1978. https://dspace.mit.edu/bitstream/handle/1721.1/15993/07113748-MIT.pdf. [8, 15]

[67] D. Koukoulopoulos. *The Distribution of Prime Numbers*. Graduate Studies in Mathematics. American Mathematical Society, 2019. [38, 44, 47]

[68] C. Lauradoux, J. Ponge, and A. Roeck. Online Entropy Estimation for Non-Binary Sources and Applications on iPhone. Research Report RR-7663, INRIA, June 2011. [110]

[69] J. V. Leeuwen, Warwick, A. R. Meyer, and M. Nival. *Handbook of Theoretical Computer Science : Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990. [45]

[70] D. S. Legg. Abstract Syntax Notation X (ASN.X). RFC 4912, July 2007. [21, 117]

[71] D. S. Legg. Abstract Syntax Notation X (ASN.X) Representation of Encoding Instructions for the Generic String Encoding Rules (GSER). RFC 4913, July 2007. [21, 117]

[72] D. H. Lehmer and R. E. Powers. On factoring large numbers. *Bull. Amer. Math. Soc.*, 37(10) :770–776, 10 1931. https://projecteuclid.org:443/euclid.bams/1183495051. [51]

[73] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. https://eprint.iacr.org/2012/064. [2, 66, 67]

[74] H. W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(3) :649–673, 1987. https://www.jstor.org/stable/1971363. [67]

[75] H. W. Lenstra. Primality Testing with Gaussian Periods. In M. Agrawal and A. Seth, editors, *FST TCS 2002 : Foundations of Software Technology and Theoretical Computer Science*, pages 1–1, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. [46]

[76] G. Lyon. Nmap Network Scanner. https://nmap.org/. [70, 139]

[77] A. Melnikov. Updated Transport Layer Security (TLS) Server Identity Check Procedure for Email-Related Protocols. RFC 7817, Mar. 2016. [33]

[78] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996. [6, 7, 8, 45, 96, 111, 112]

[79] G. L. Miller. Riemann's Hypothesis and Tests for Primality. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, pages 234–239, New York, NY, USA, 1975. Association for Computing Machinery. https://doi.org/10.1145/800116.803773. [43, 44]

[80] V. S. Miller. Use of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-39799-X_31. [38]

[81] K. Moriarty and S. Farrell. Deprecating TLS 1.0 and TLS 1.1. RFC 8996, Mar. 2021. [33]

[82] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1 : RSA Cryptography Specifications Version 2.2. RFC 8017, nov 2016. https://rfc-editor.org/rfc/rfc8017.txt. [39, 81]

[83] M. A. Morrison and J. Brillhart. A method of factoring and the factorization of F_7. *Mathematics of Computation*, 29 :183–205, 1975. [53]

[84] National Institute of Standards and Technology. FIPS 198-1, The Keyed-Hash Message Authentication Code (HMAC). Technical report, Gaithersburg, MD, USA, 2008. https://doi.org/10.6028/NIST.FIPS.198-1. [8, 107]

[85] M. Nemec, M. Sys, P. Svenda, D. Klinec, and V. Matyas. The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1631–1648, New York, NY, USA, 2017. Association for Computing Machinery. [2, 55, 57, 66]

[86] R. Oppliger. *SSL and TLS : Theory and Practice, Second Edition*. Artech House information security and privacy series. Artech House Publishers, 2016. [12]

[87] A. Pellegrini, V. Bertacco, and T. Austin. Fault-Based Attack of RSA Authentication. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 855–860, Leuven, BEL, 2010. European Design and Automation Association. https://doi.org/10.1109/DATE.2010.5456933. [48]

[88] R. Plymen. *"The Great Prime Number Race"*. Student Mathematical Library. American Mathematical Society, United States, aug 2020. [38]

[89] J. M. Pollard. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society*, 76(3) :521–528, 1974. [49]

[90] J. M. Pollard. A Monte Carlo Method For Factorization. *BIT Numerical Mathematics*, pages 331–334, 1975. [73]

[91] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The Pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151) :1003–1026, 1980. [44]

[92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, USA, 1988. [56]

[93] Python Software Foundation. Python Programming Language. https://www.python.org/. [70]

[94] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1) :128 – 138, 1980. http://www.sciencedirect.com/science/article/pii/0022314X80900840. [39, 43, 58, 106]

[95] Rapid7 Open Data. Project Sonar, 2013. https://opendata.rapid7.com/. [66, 68]

[96] E. Rescorla. HTTP Over TLS. RFC 2818, May 2000. [33]

[97] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018. [12, 33]

[98] E. Rescorla and T. Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008. [33]

[99] P. Ribenboim. *The New Book of Prime Number Records*. Springer-Verlag New York, 1996. https://doi.org/10.1007/978-1-4612-0759-7. [48]

[100] B. Riemann. Ueber die Anzahl der Primzahlen unter einer gegebenen Grösse. *Monatsberichte der Berliner Akademie*, pages 671–680, November 1859. [47]

[101] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2) :120–126, Feb. 1978. https://doi.org/10.1145/359340.359342. [3, 38, 39]

[102] K. Schmeh. *Cryptography and Public Key Infrastructure on the Internet*. Wiley, 2006. [12]

[103] B. Schneier and P. Sutherland. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., USA, 2nd edition, 1995. [6, 8, 41]

[104] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7 :281–292, 2005. [134]

[105] A. Selberg. An Elementary Proof of Dirichlet's Theorem About Primes in an Arithmetic Progression. *Annals of Mathematics*, 50(2) :297–304, 1949. http://www.jstor.org/stable/1969454. [61]

[106] A. Seznec and N. Sendrier. Havege : A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(4) :334–346, 2003. [110]

[107] R. W. Shirey. Internet Security Glossary, Version 2. RFC 4949, Aug. 2007. [14]

[108] D. Simon. The Private Communication Technology Protocol. Internet-Draft draft-benaloh-pct-00, Internet Engineering Task Force, Nov. 1995. Work in Progress. [33]

[109] R. Solovay and V. Strassen. A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing*, 6(1) :84–85, 1977. https://doi.org/10.1137/0206006. [42]

[110] R. Solovay and V. Strassen. Erratum : A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing*, 7(1) :118–118, 1978. https://doi.org/10.1137/0207009. [42]

[111] I. Soprunov. A Short Proof of the Prime Number Theorem for Arithmetic Progressions. 01 2010. https://academic.csuohio.edu/soprunov_i/pdf/primes.pdf. [61]

[112] A. Stiglic. *Strong Prime*, pages 1265–1266. Springer US, Boston, MA, 2011. https://doi.org/10.1007/978-1-4419-5906-5_480. [58]

[113] F. Strenzke. An Analysis of OpenSSL's Random Number Generator. In *Proceedings, Part I, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9665*, page 644–669, Berlin, Heidelberg, 2016. Springer-Verlag. [98]

[114] G. Tenenbaum and M. Mendès France. *The Prime Numbers and Their Distribution*. Student mathematical library. American Mathematical Soc., 2000. [38]

[115] The CADO-NFS Development Team. CADO-NFS, An Implementation of the Number Field Sieve Algorithm, 2017. Release 2.3.0. http://cado-nfs.gforge.inria.fr/. [57, 87]

[116] The OpenSSL Project. OpenSSL - Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. [2, 4, 39, 41, 68, 92]

[117] The PARI Group, Univ. Bordeaux. *PARI/GP version 2.12.0*, 2019. available from http://pari.math.u-bordeaux.fr/. [4, 44, 49, 52, 73, 76, 81]

[118] S. Thomas. *SSL and TLS Essentials : Securing the Web*. John Wiley & Sons, 2000. [vi, 12, 117, 118]

[119] M. S. Turan, E. Barker, J. Kelsey, K. McKay, M. Baish, and M. Boyle. SP 800-90B, Recommendation for the Entropy Sources Used for Random Bit Generation. Technical report, Gaithersburg, MD, USA, 2018. https://doi.org/10.6028/NIST.SP.800-90B. [60]

[120] R. Van Der Meulen. Gartner, Inc., 2017. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016. [2]

[121] H. von Mangoldt. Zu riemanns abhandlung „ueber die anzahl der primzahlen unter einer gegebenen grösse". *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1895 :255 – 305. [47]

[122] P. Švenda, M. Nemec, P. Sekan, R. Kvašnovsky, D. Formánek, D. Komárek, and V. Matyáš. The Million-Key Question : Investigating the Origins of RSA Public Keys. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 893–910, USA, 2016. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/svenda. [62]

[123] S. S. Wagstaff. *Cryptanalysis of Number Theoretic Ciphers*. CRC Press, Inc., USA, 2002. https://doi.org/10.1201/9781315275765. [49]

[124] M. J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36(3) :553–558, 1990. https://doi.org/10.1109/18.54902. [81]

[125] S. William. *Cryptography and Network Security - Principles and Practice, 7th Edition*. Pearson Education India. [12, 20]

[126] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When Private Keys Are Public : Results from the 2008 Debian OpenSSL Vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 15–27, New York,

NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1644893.1644896. [67]

[127] T. Ylonen, B. Thomas, B. Lampson, C. Ellison, R. L. Rivest, and W. S. Frantz. SPKI Certificate Theory. RFC 2693, Sept. 1999. [32]

[128] ZyXEL. ZyWALL 2 Plus. https://www.zyxel.com/uk/en/products_services/zywall_2_plus.shtml. [84]

Annexe A

Quelques statistiques des données

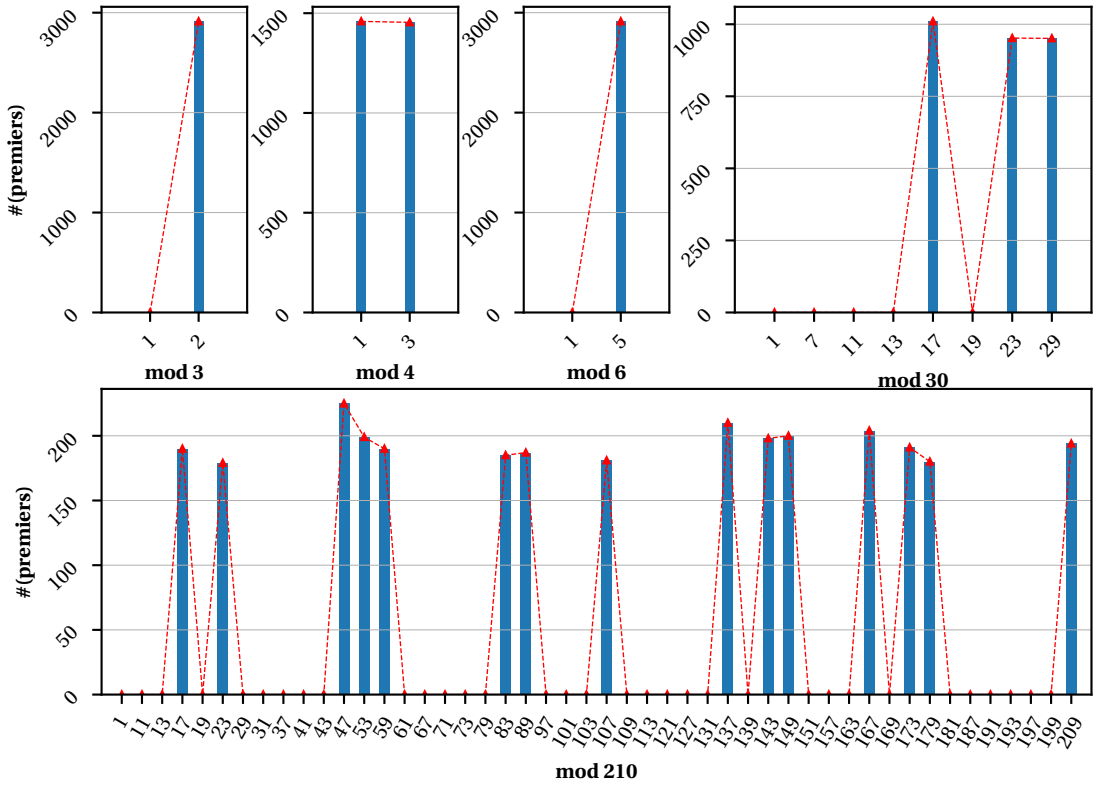
Sommaire

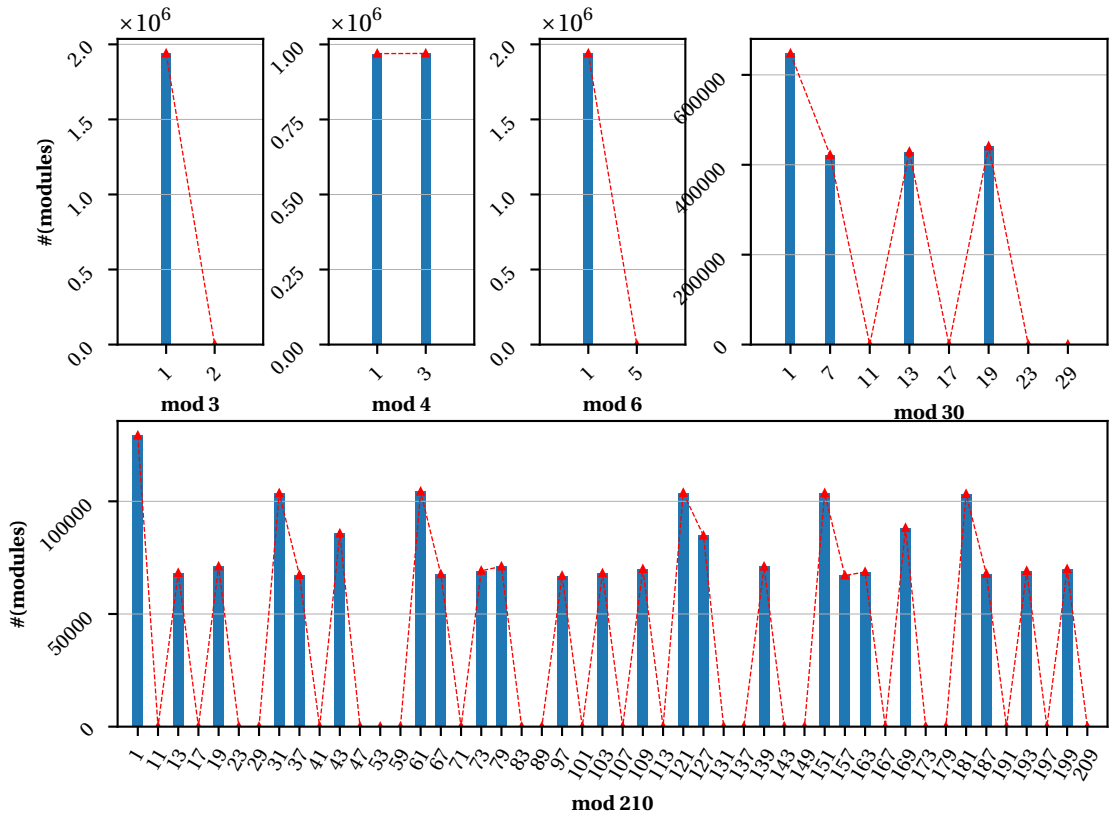
On va donner, dans la section A.1, la répartition des restes modulo les entiers 3, 5, 6, 30 et 210 des nombres premiers trouvés à la suite de la recherche des modules PGCD-vulnérables dans la sous-section 5.3.4 du chapitre 5. Cela permettra d'identifier facilement les familles qui utilisent ou non la bibliothèque OpenSSL. Il y a également la répartition des restes modulo les mêmes entiers des modules RSA provenant des familles affectés par la PGCD-vulnérabilité.

D'autre part, on donne dans la section A.2, les nombres premiers provenant des pare-feux «ZX_ZW2P» avec leurs fréquences d'apparition dans des certificats distincts.

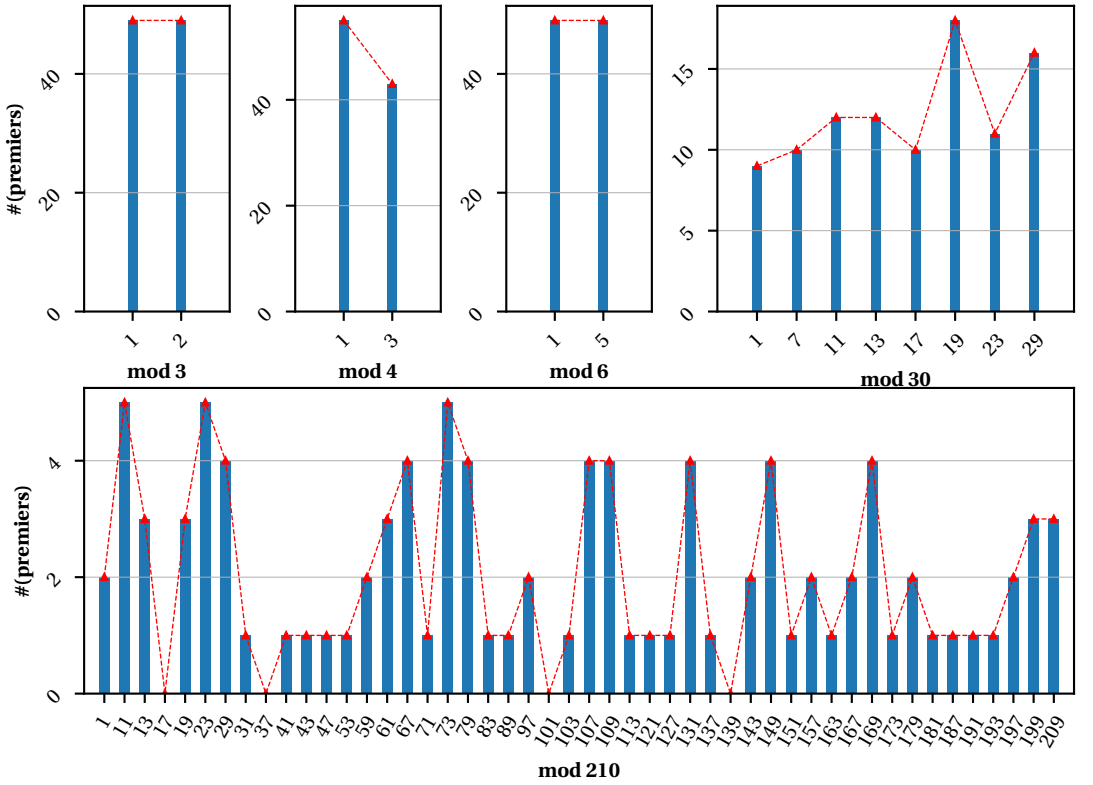
A.1 Répartition des restes modulo

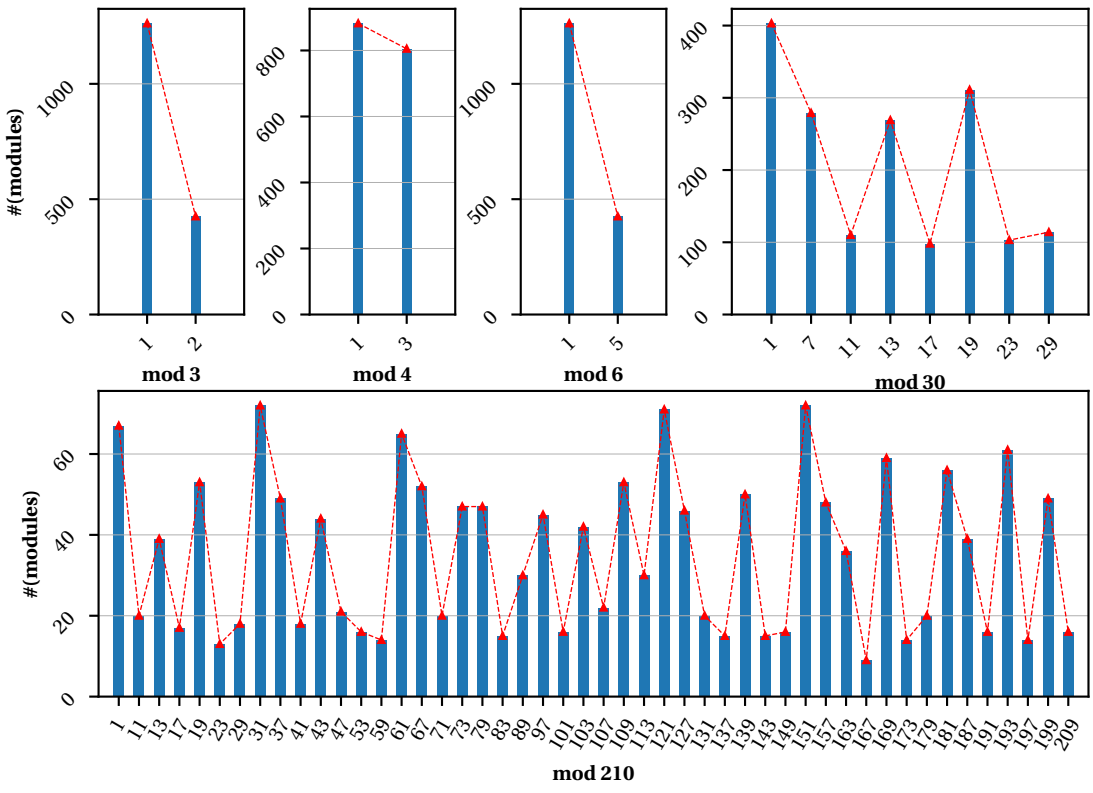
A.1.1 2W – (RSA-1024)



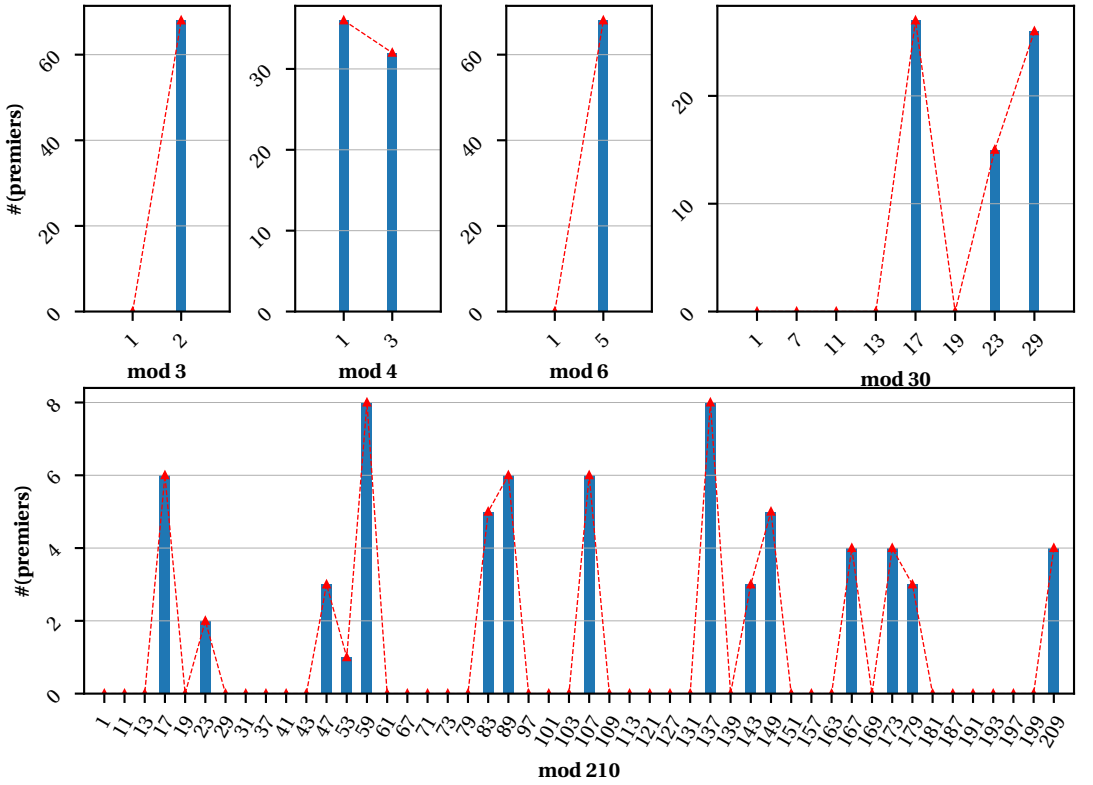


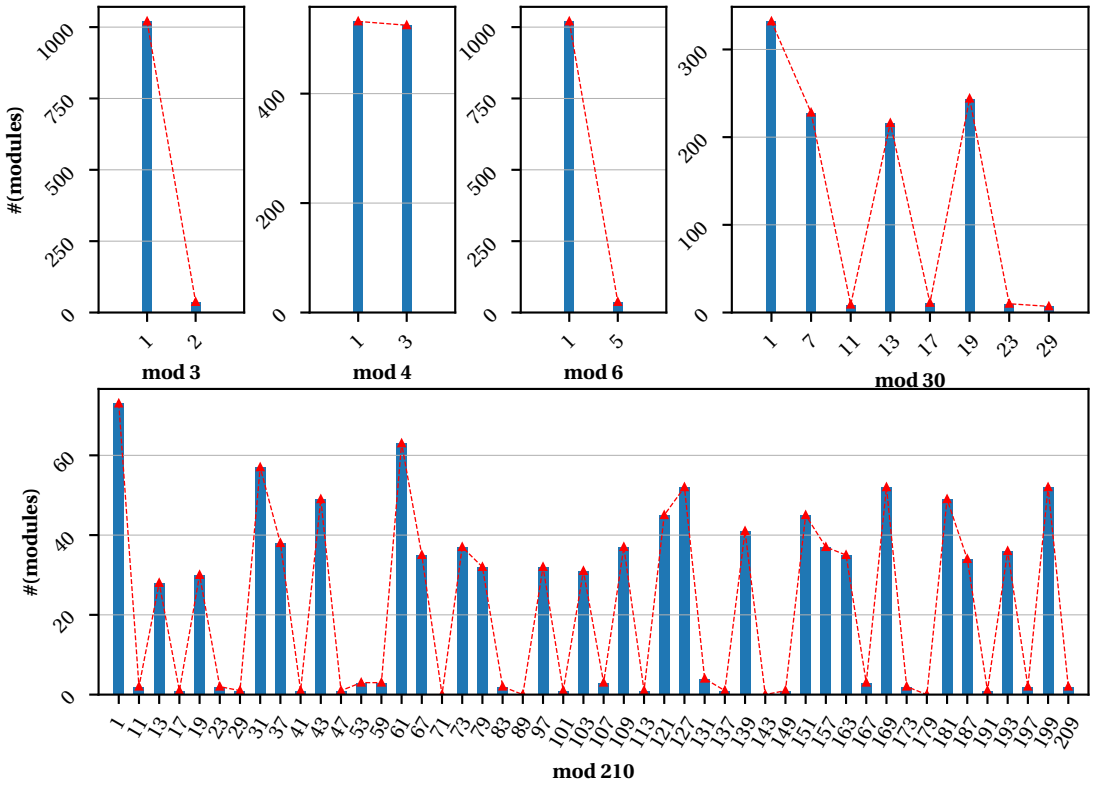
A.1.2 AC – (RSA-1024)



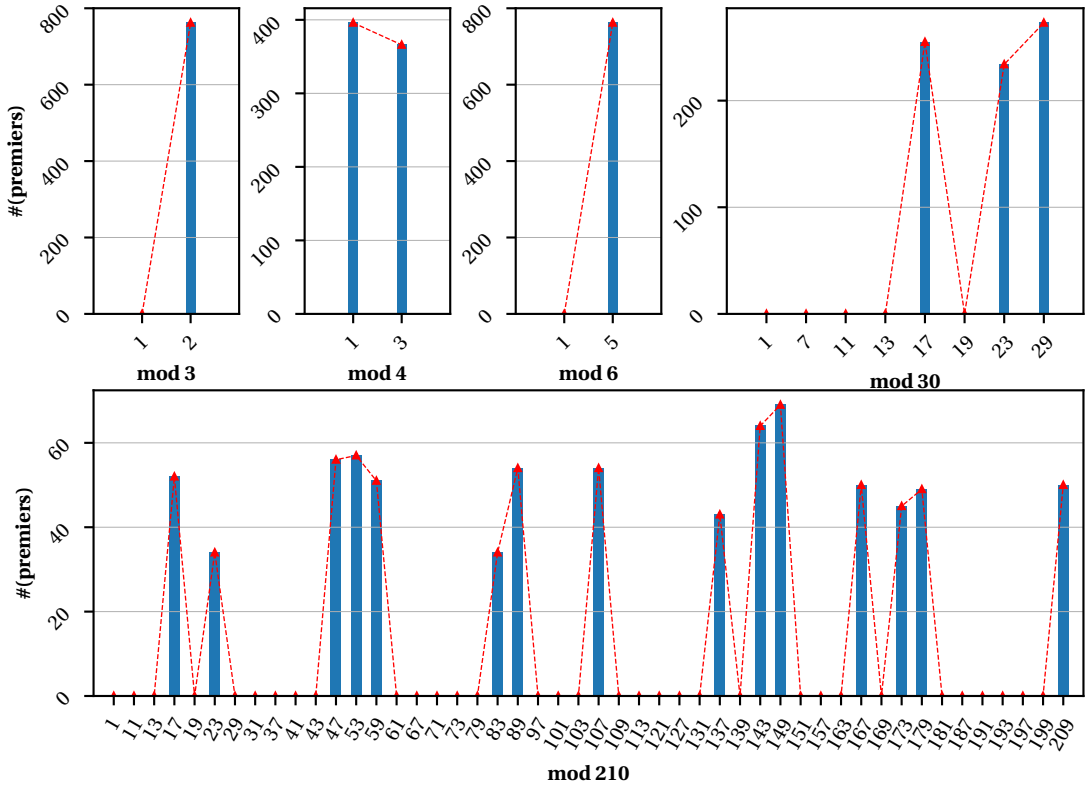


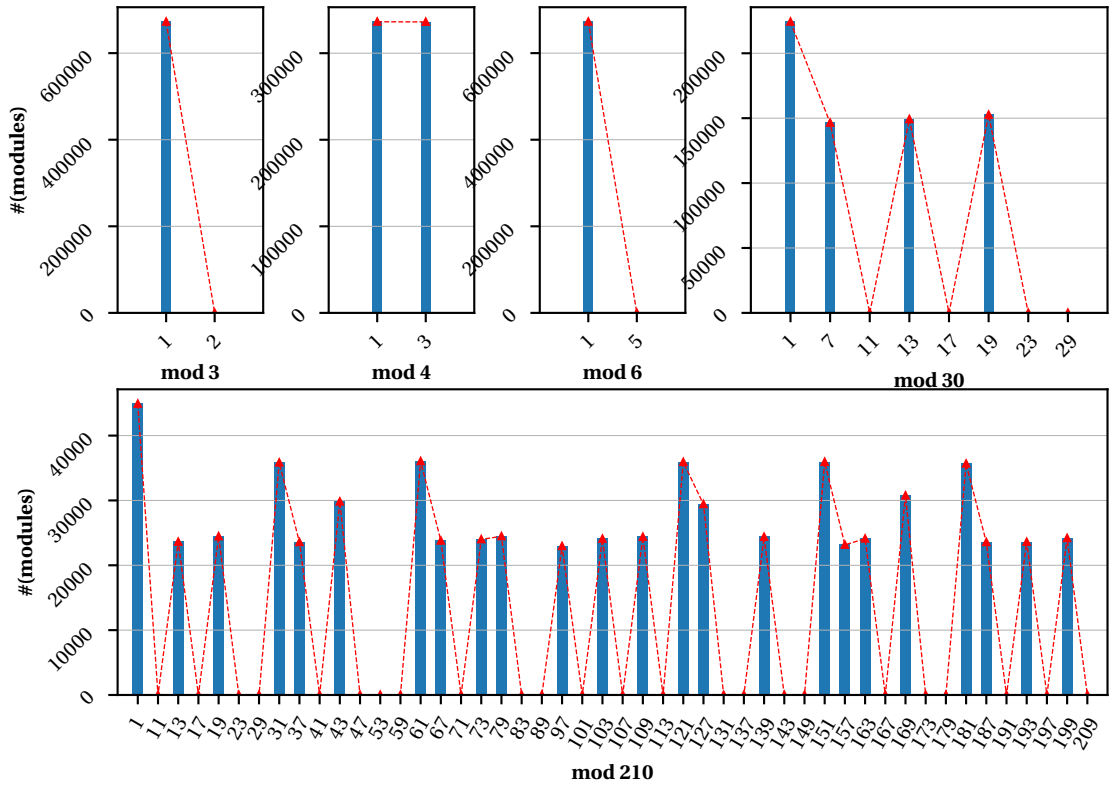
A.1.3 AV – (RSA-1024)



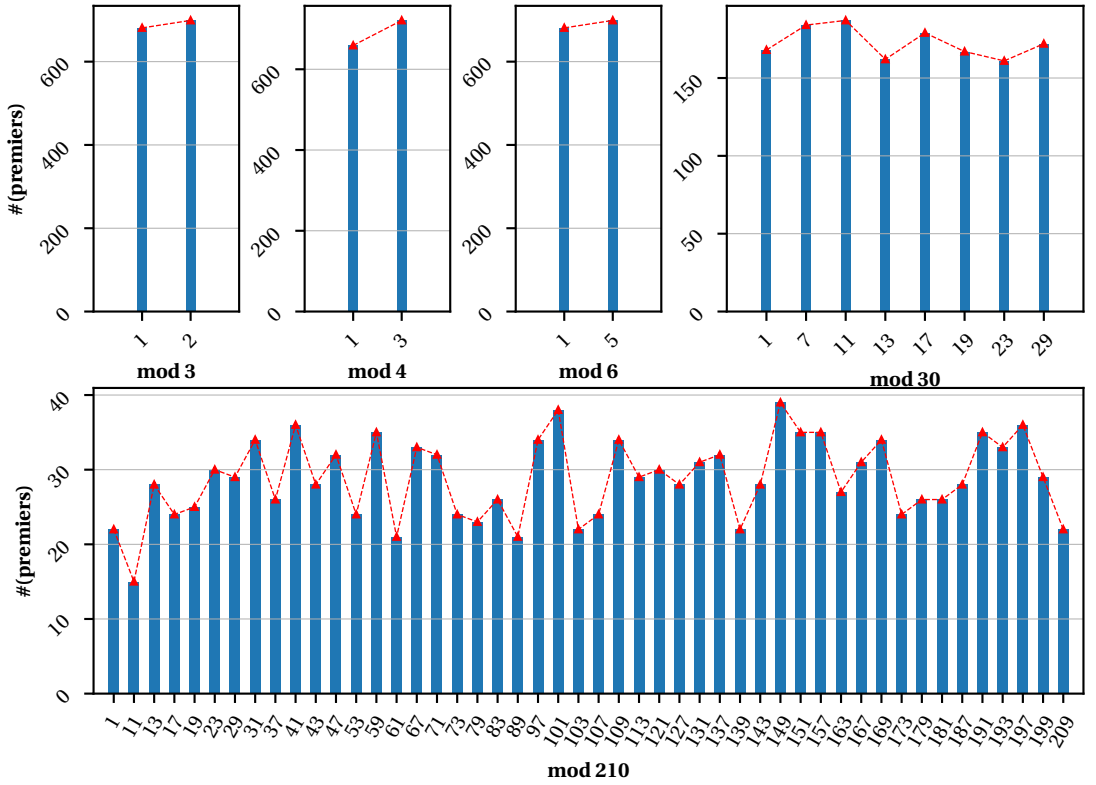


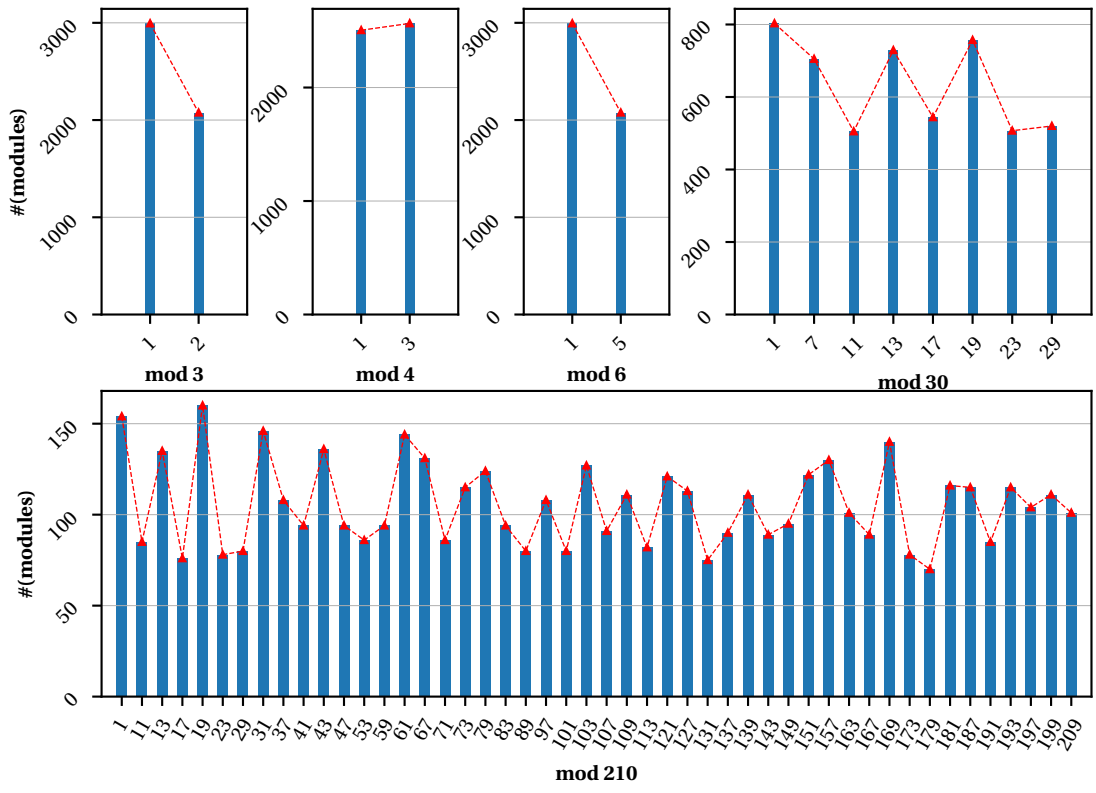
A.1.4 DT – (RSA-2048)



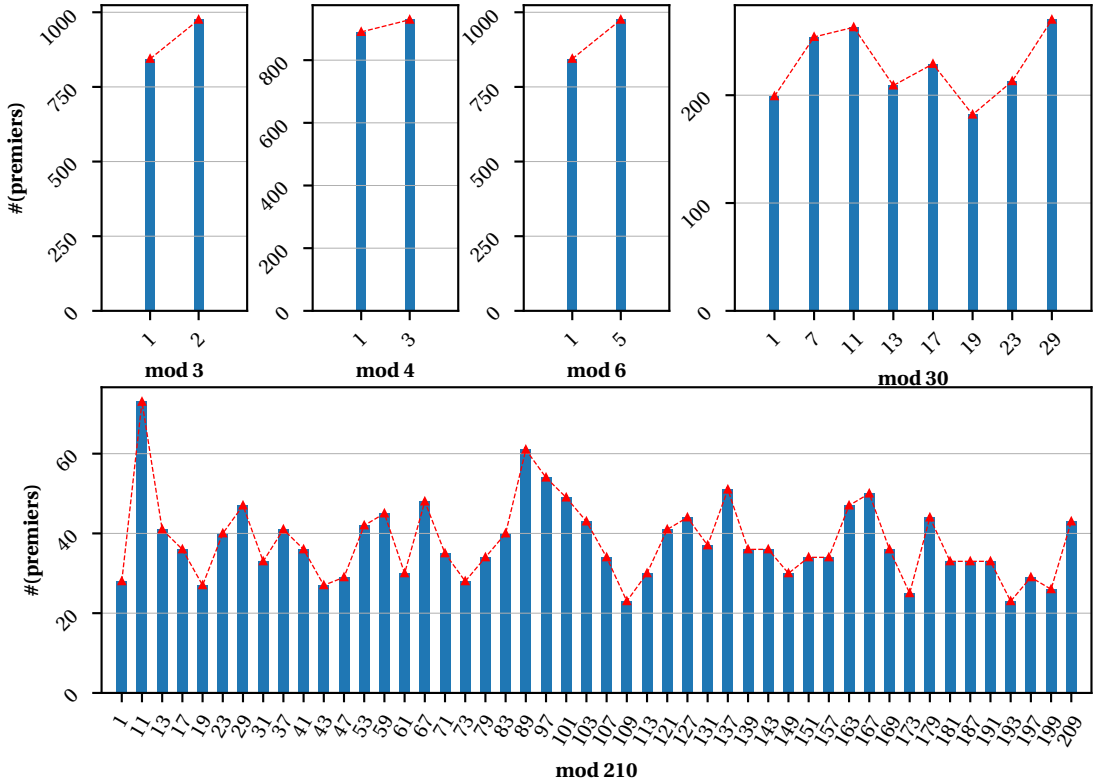


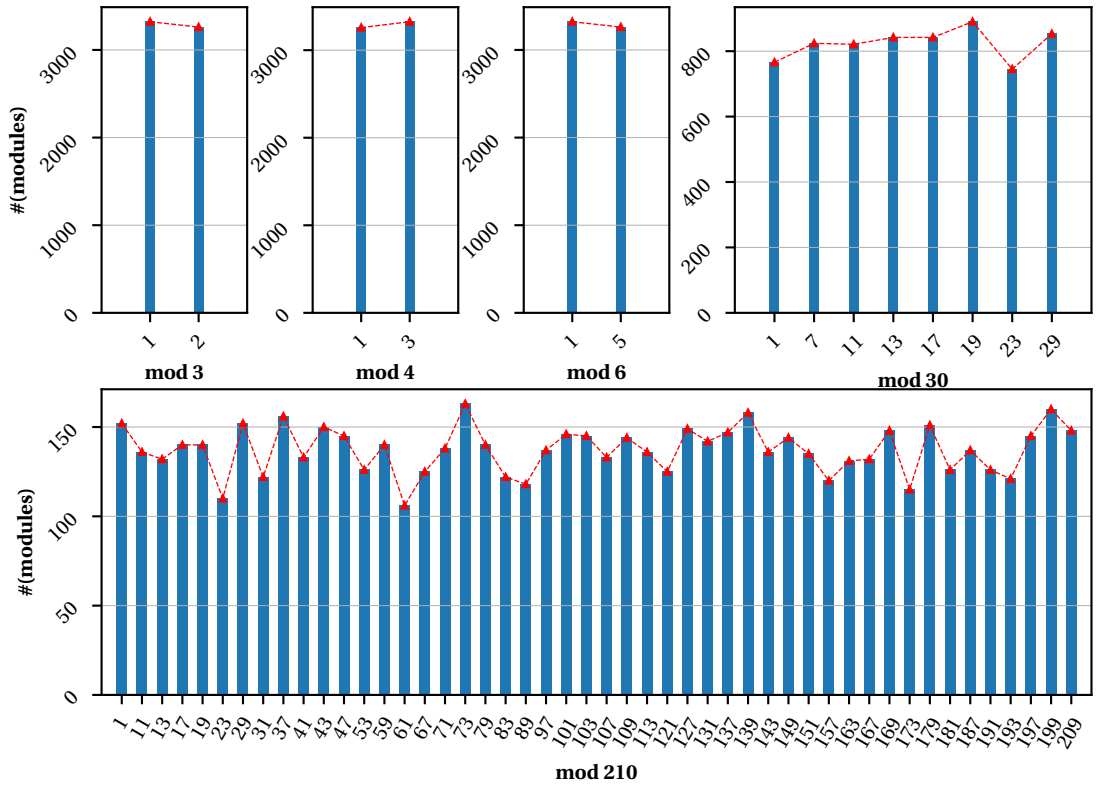
A.1.5 JP – (RSA-1024)



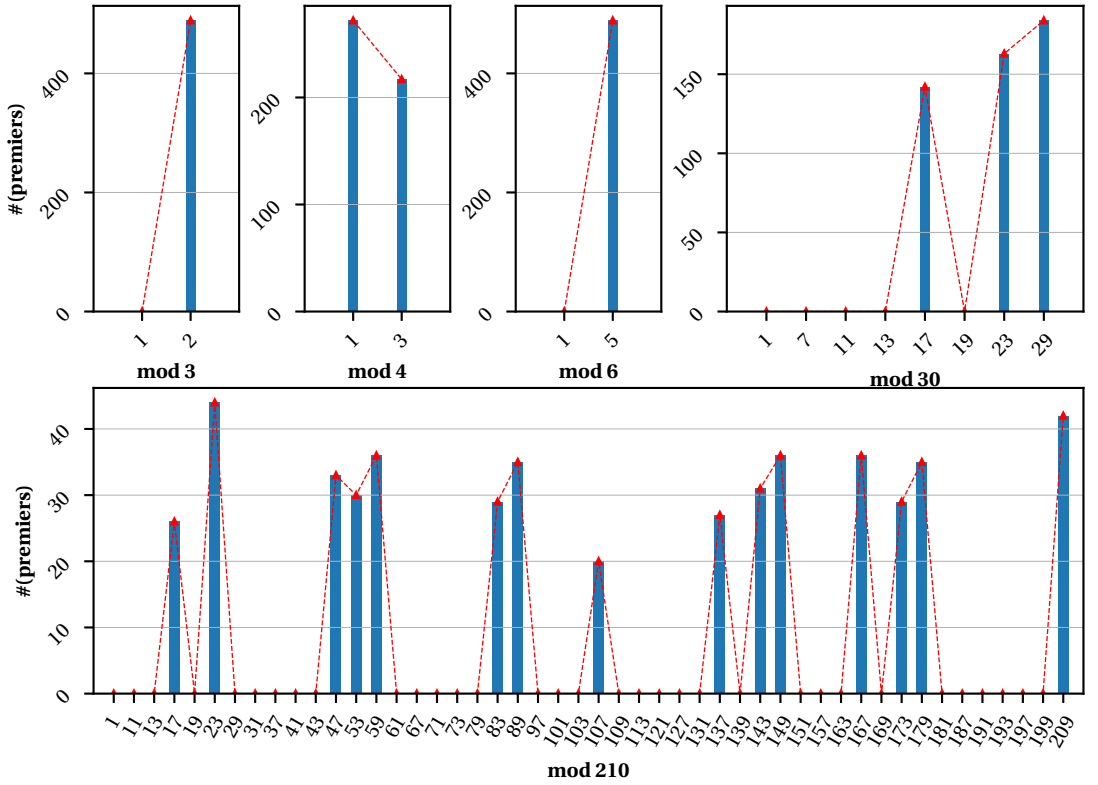


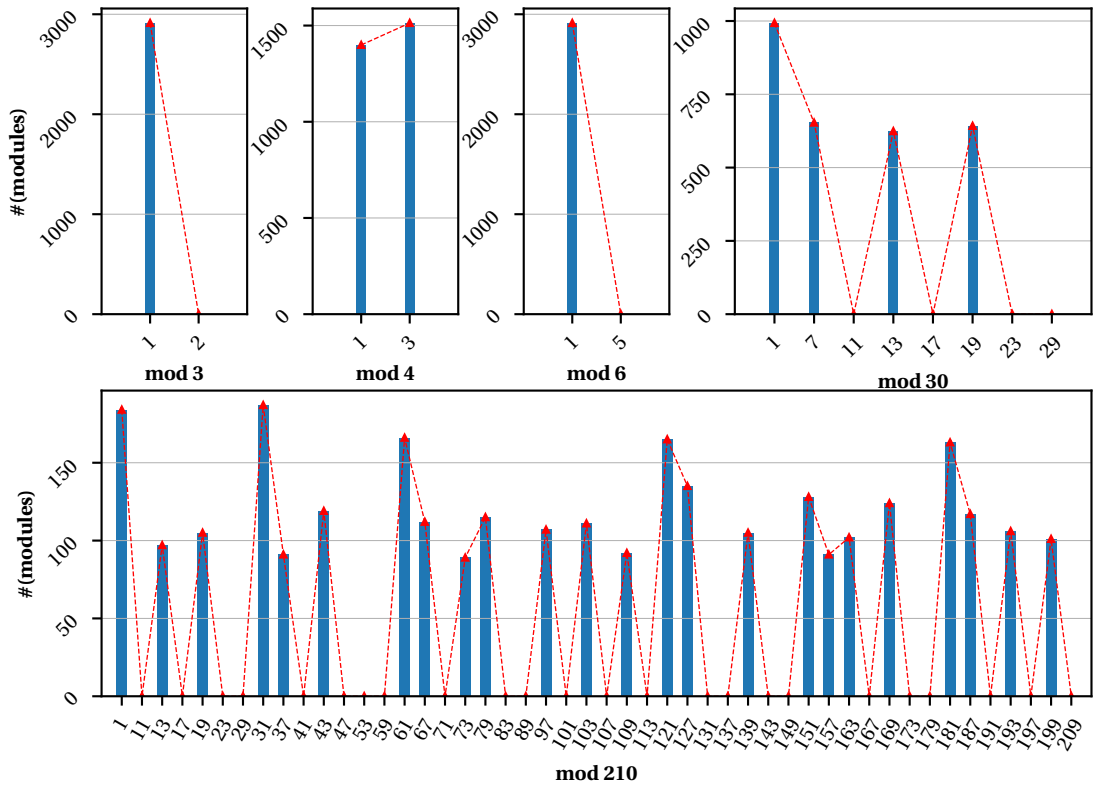
A.1.6 KR – (RSA-1024)



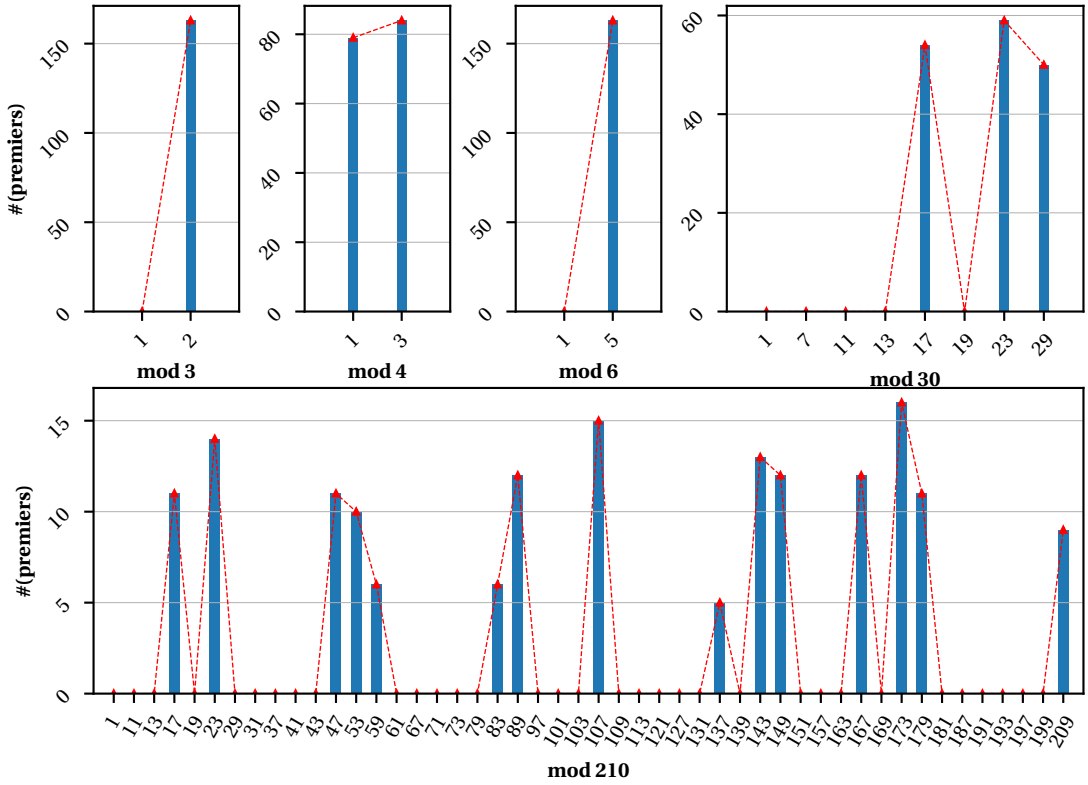


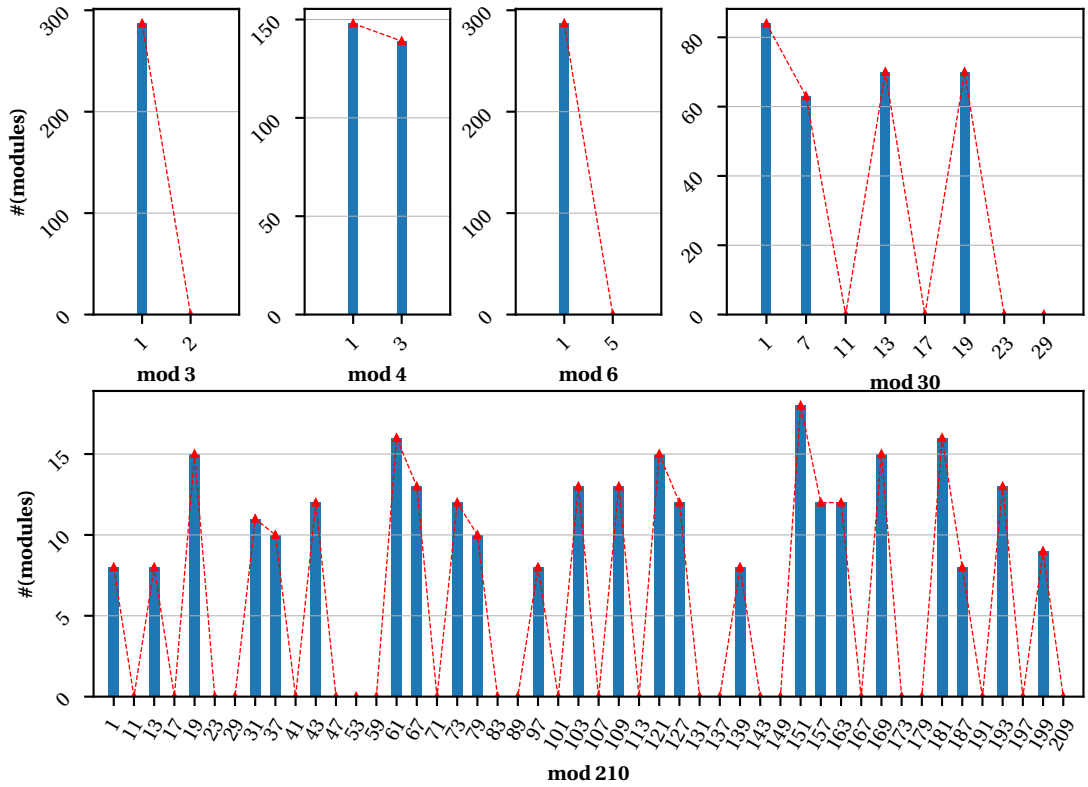
A.1.7 RT – (RSA-1024)



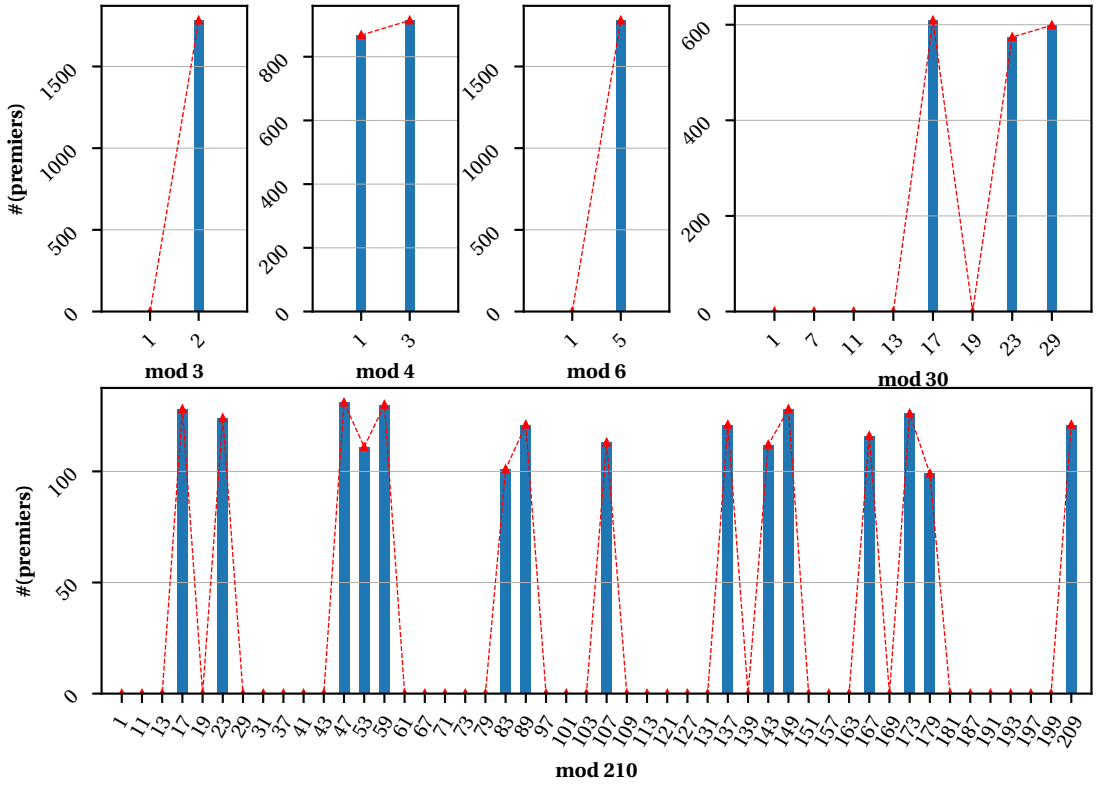


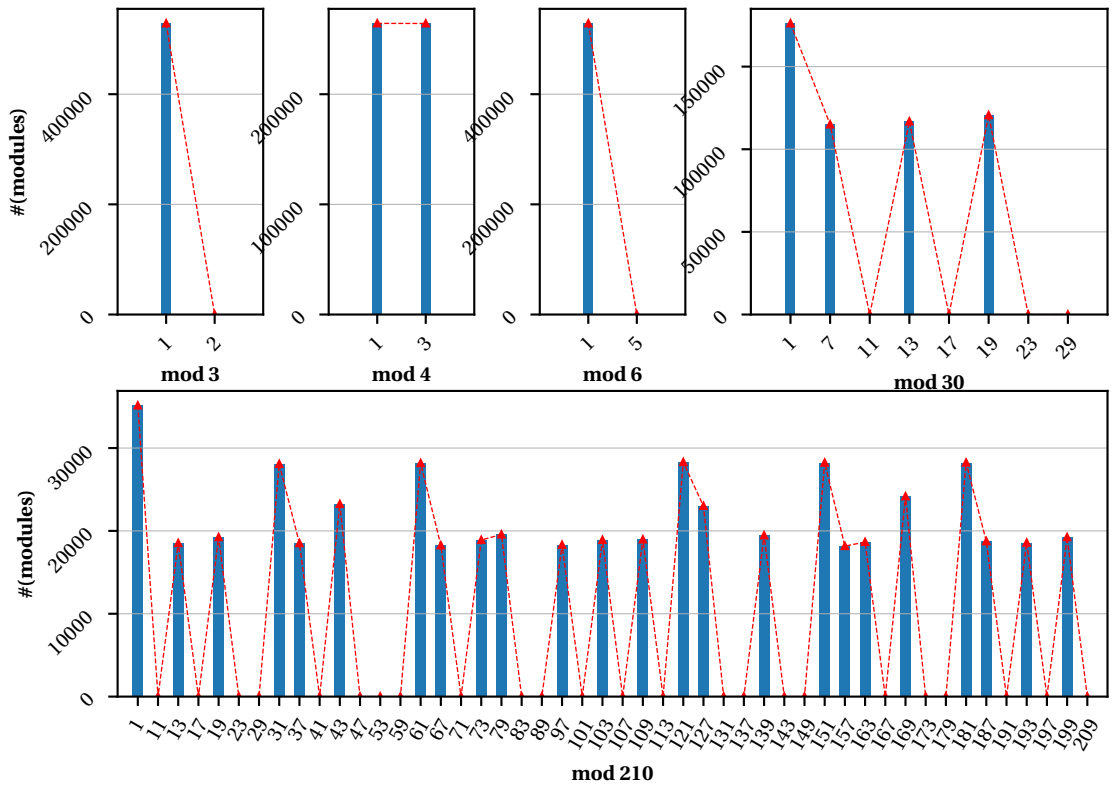
A.1.8 ST – (RSA-1024)



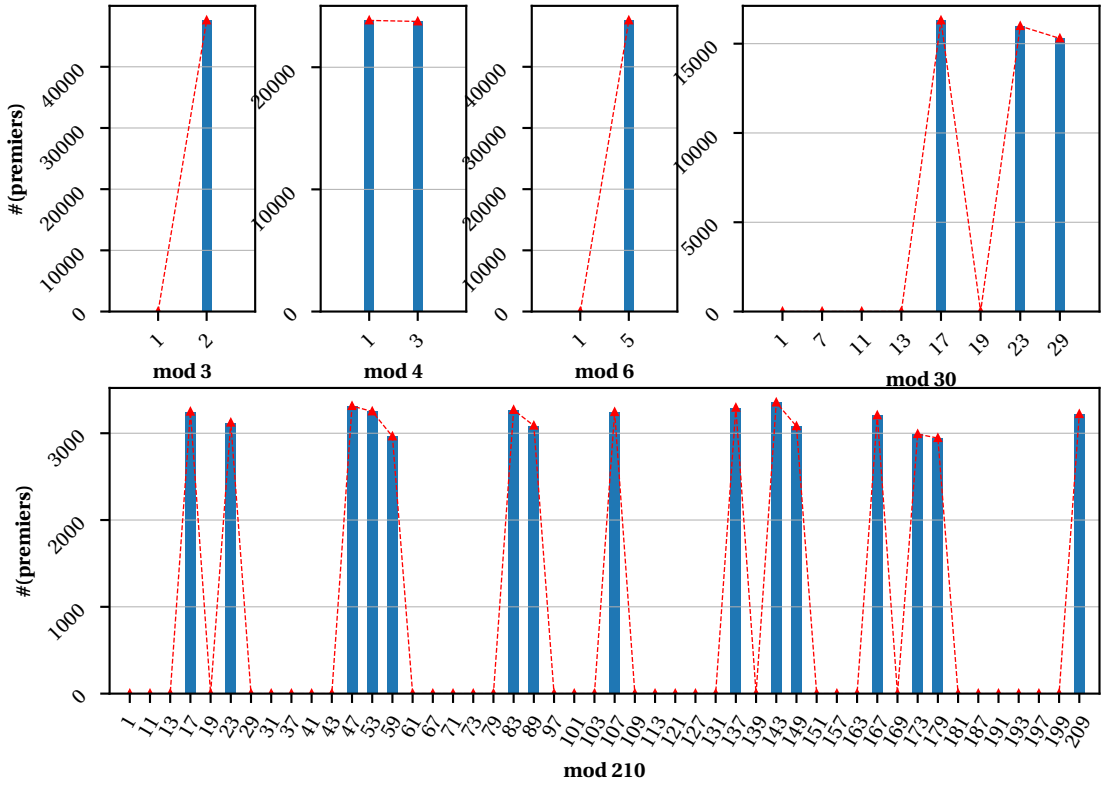


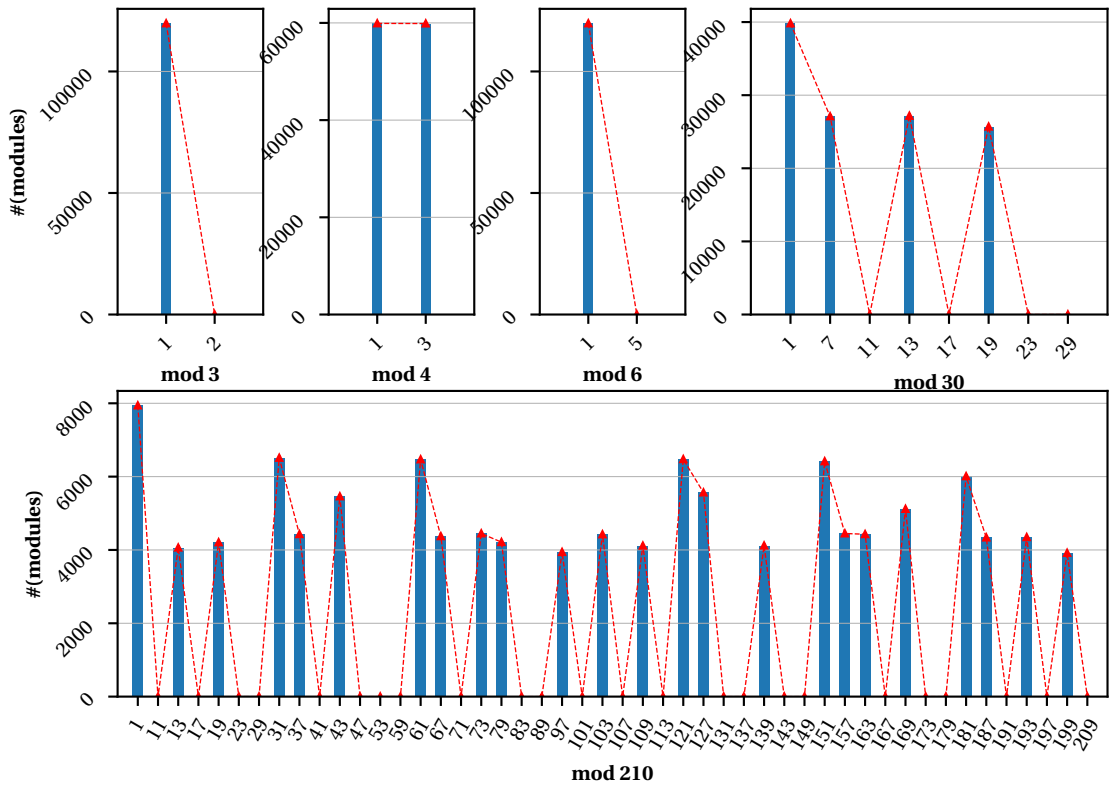
A.1.9 SW – (RSA-2048)



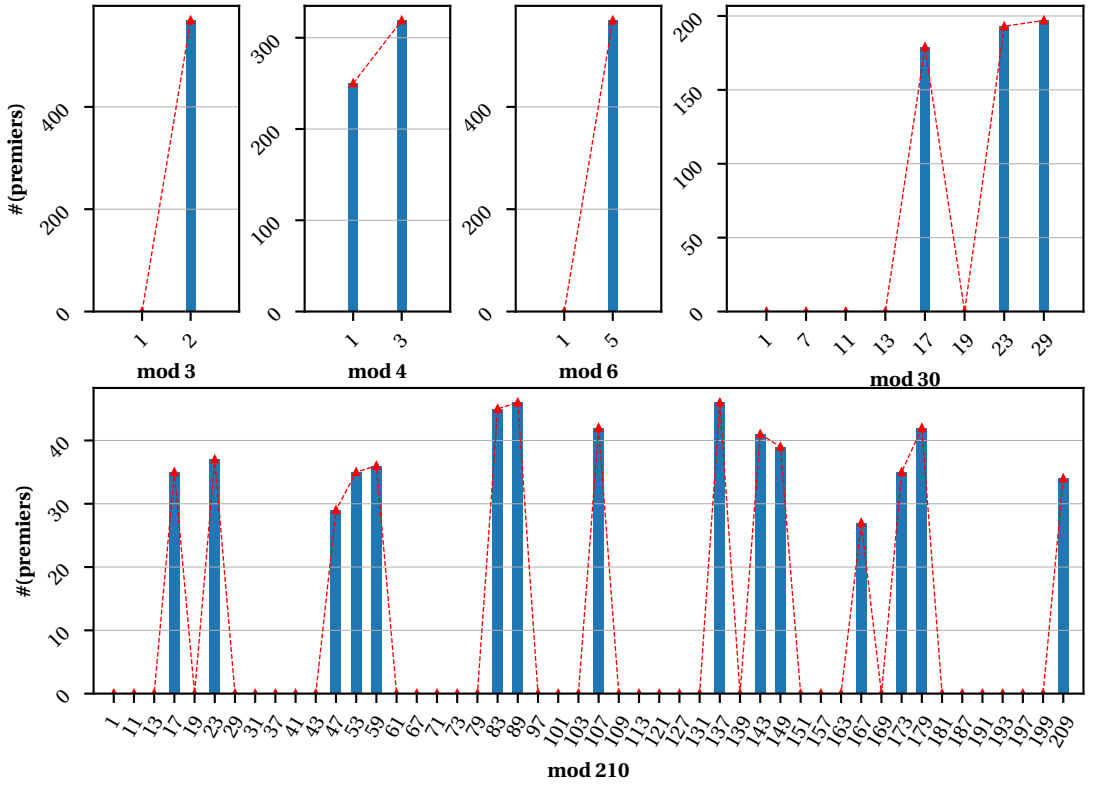


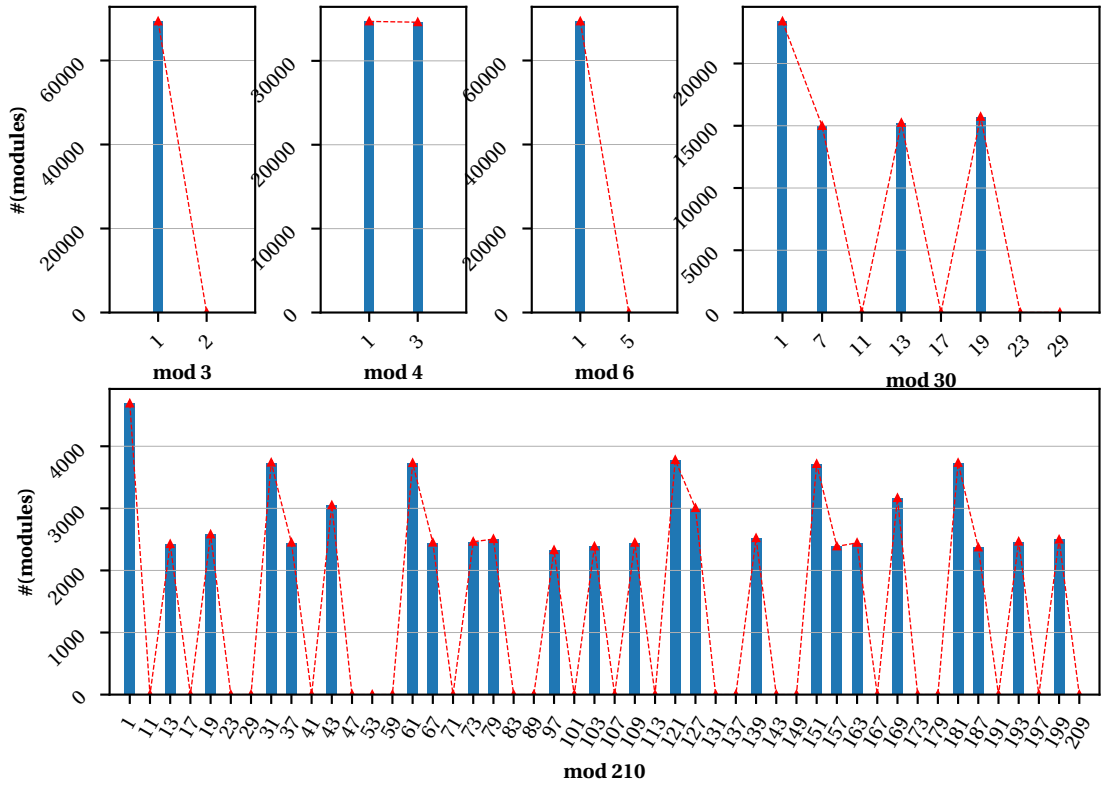
A.1.10 TL – (RSA-512)



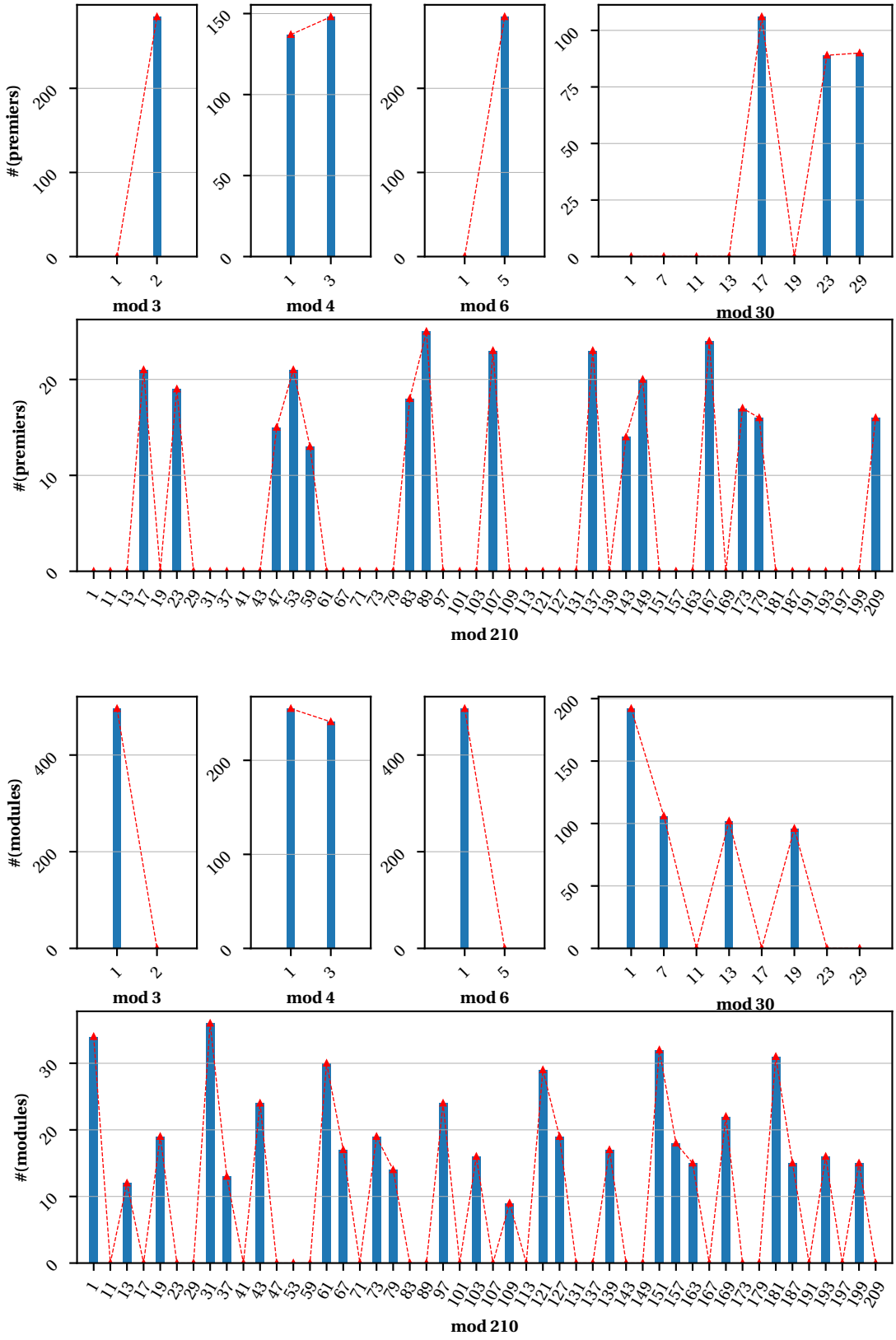


A.1.11 ZX – (RSA-1024)





A.1.12 ZX – (RSA-2048)



A.2 Nombres premiers provenant des «ZX_ZW2P»

Tableau A.1 – ***Nombres premiers provenant de matériels «ZX_ZW2P».***

Nombres premiers	Certificats distincts		
	EFF	ANSSI	Rapid7
0xc54d7e81ca0665c6b5620944c5e4123f1e2fa59dd19214304cde799a068b00a9	3,769	5,779	1,355
0xdc798d9888f377d77d3480be06061eda02af6971b4fdd10543634ab0429dced1	3,769	5,779	1,355
0xebb00f8e87b07f5ca4271db1e1490b89afe00f1342fa51e71efa0e22e5d3af55	449	774	222
0xfb0199f0250e5a604e6d80493bd154eb39b689aac0345e73203bcb9c0a768f97	449	774	222
0xc526edadf5136816e4f249dd751a6f87bb0b838eb357abcbb65c54247584662f	592	647	92
0xebc03c175c798d9888f377d77d3480be06061eda02af6971b4fdd10543634b83	592	647	92
0xea3bb511cbba9dbe0730be9fa8b4c954a35e2ab24a83e717ba32feeb82264d6b	60	62	6
0xf41c86152bb00f8e87b07f5ca4271db1e1490b89afe00f1342fa51e71efa0ed1	60	62	6
0xc68b0035b98765e044183db6f8d4ad316b8e98b8351ff41c86152bb00f8e87d5	31	39	12
0xcae77f90b8ea9d5dcb61d81aad3ff5382c96bede99e2c844f5cc25df5f16d157	30	32	3
0xce3191aaebc089f709efd77dd07e3003800ac53f678d6b7bb5a7079f178535f7	30	32	3
0xef0b0d2b92947b0199f0250e5a604e6d80493bd154eb39b689aac0345e732047	21	22	10
0xf51ff41c86152bb00f8e87b07f5ca4271db1e1490b89afe00f1342fa51e7205f	20	21	10
0xc29dccec689dc458454d7e81ca0665c6b5620944c5e4123f1e2fa59dd19214a3	14	20	10
0xd497c2a86a82c968c595cb14e03137ead10a4ae77f90b8ea9d5dcb61d81aaf4b	13	16	11
0xfede99e2c844f5cc25df5f16d0c0f4fa7ab04e3191aaebc089f709efd77dd085	13	16	11
0xddb1e1490b89afe00f1342fa51e71efa0e22e5d3af0b0d2b92947b0199f02513	17	19	2
0xc595cb14e03137ead10a4ae77f90b8ea9d5dcb61d81aad3ff5382c96bede99f7	14	18	4
0xd9712f8e7537997e6da98a3641433237a8ac783b78110a2b46d751de23c29087	10	18	8
0xdd54ee9030f939f91497c2a86a82c968c595cb14e03137ead10a4ae77f90ba6b	10	18	8
0xe8ac783b78110a2b46d751de23c28fd4b1983b4f1d54ee9030f939f91497c2b1	14	18	4
0xd1e71efa0e22e5d3af0b0d2b92947b0199f0250e5a604e6d80493bd154eb3a85	13	16	3
0xde73203bcb9c0a768df0b827f99770bce7c7be5b6c002458bc4c64aac573410b	13	16	3
0xff903bfc2e604d8dd0912ffa77dd3d5a05b07f76677f0fb212a50160a99e7855	7	15	9
0xb7ead10a4ae77f90b8ea9d5dcb61d81aad3ff5382c96bede99e2c844f5cc26d9	16	13	1
0xd1aaebc089f709efd77dd07e3003800ac53f678d6b7bb5a7079f1785359ef001	7	15	8
0xdedd314fd91f0d6de4f9652056f3690d9f8f5371926a5bf839bd9a6b2eaa4f2b	16	13	1
0xc5733f3659712f8e7537997e6da98a3641433237a8ac783b78110a2b46d7522b	10	13	3
0xd4eb39b689aac0345e73203bcb9c0a768df0b827f99770bce7c7be5b6c0024c7	10	11	3
0xd77dd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903c43	4	15	5
0xd81aad3ff5382c96bede99e2c844f5cc25df5f16d0c0f4fa7ab04e3191aaed87	4	15	5
0xcf8e87b07f5ca4271db1e1490b89afe00f1342fa51e71efa0e22e5d3af0b0d41	10	11	1
0xd19214304cde799a068b0035b98765e044183db6f8d4ad316b8e98b8351ff455	10	11	1
0xd785359eed1c6adbaf2bff903bfc2e604d8dd0912ffa77dd3d5a05b07f766bcd	7	10	4
0xdf16d0c0f4fa7ab04e3191aaebc089f709efd77dd07e3003800ac53f678d6c91	7	9	4
0xc6d751de23c28fd4b1983b4f1d54ee9030f939f91497c2a86a82c968c595cbc7	4	9	1

0xca3641433237a8ac783b78110a2b46d751de23c28fd4b1983b4f1d54ee90324d	4	7	3
0xca768df0b827f99770bce7c7be5b6c002458bc4c64aac5733f3659712f8e76a9	4	7	3
0xec002458bc4c64aac5733f3659712f8e7537997e6da98a3641433237a8ac7865	4	9	1
0xcd8dd0912ffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95d17	4	8	1
0xe99e77a8215dadd95ccb73a576b30595cbbcf52e0ff0751acfb461dd0df74293	2	5	6
0xf59eed1c6adbaf2bff903bfc2e604d8dd0912ffa77dd3d5a05b07f76677f1063	2	5	6
0xe03137ead10a4ae77f90b8ea9d5dcb61d81aad3ff5382c96bede99e2c844f5fb	5	6	1
0xf8110a2b46d751de23c28fd4b1983b4f1d54ee9030f939f91497c2a86a82ca05	5	6	1
0xc9efd77dd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff9d	3	7	1
0xcbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c4687d309df9c101d41	4	5	1
0xf81f0c09e5c899f3bd13f8f4857531701d89283c74a1a0d1095f4deb469cbb63	4	5	1
0xc307775fb05d042713676d3b2954c6af0033def4e738f5770b090dac5b7c1d4f	5	3	0
0xc6253879caef0d30a295808ea362a03387fde399d790cbbabfa15927d90da9af	3	4	1
0xca2b46d751de23c28fd4b1983b4f1d54ee9030f939f91497c2a86a82c968c817	4	2	2
0xcd28ae994331ebf7e591d8b7f05878f081c01fbb536c83e4b192bda8be916f4b	3	4	1
0xeb8e98b8351ff41c86152bb00f8e87b07f5ca4271db1e1490b89afe00f13430b	5	3	0
0xf7ead10a4ae77f90b8ea9d5dcb61d81aad3ff5382c96bede99e2c844f5cc25f9	4	2	2
0xfb10790d0899370f6520c815d6d827f13349f438a65d12eb6b52a7141d0548ef	3	3	2
0xc00ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903bfc2e604d8dd0ef	3	2	2
0xc95f4deb469cbae2cb501a97a0b0f42a6b859a2029da55613df1ec539cdcd95d	1	3	3
0xec96bede99e2c844f5cc25df5f16d0c0f4fa7ab04e3191aaebc089f709efd99b	3	2	2
0xeeef77cbaf737a40e89b430b77e26ae6fd2fd15135df9f8e8734c56078385d95	1	3	3
0xeffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576e1	3	4	0
0xf11241b2252e00829225d52d86f21840ac4caf47c4d239585cb7b6fbfb107aa9	1	3	3
0x8f1342fa51e71efa0e22e5d3af0b0d2b92947b0199f0250e5a604e6d80493c69	3	3	0
0xcdf7429299e6d6f97a296c4687d309df9c101cfd15fcb6ae781f0c09e5c89a2b	2	2	2
0xdcdcd8be42e70914f11241b2252e00829225d52d86f21840ac4caf47c4d23a49	2	2	2
0xe77f0fb212a50160a99e77a8215dadd95ccb73a576b30595cbbcf52e0ff07601	2	2	2
0xedd95ccb73a576b30595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97b87	2	3	1
0xfbcb9c0a768df0b827f99770bce7c7be5b6c002458bc4c64aac5733f3659727d	2	4	0
0xfe6da98a3641433237a8ac783b78110a2b46d751de23c28fd4b1983b4f1d553b	2	4	0
0x10fc84bd74bcbe4ecdab4ac2b37d0794788f2689c4fcbc400f38a99b660e45423	3	3	0
0xcfd4b1983b4f1d54ee9030f939f91497c2a86a82c968c595cb14e03137ead20d	3	2	0
0xdb558a404b1d3567aa5300cc7862e9918b2e36904ac1103c553b8d3dcb4128e5	1	2	2
0xdd0548855a724d47270ce438914eb84e1bde4c4573d1491a89a13db1aeef77d1	1	1	3
0xe4aac5733f3659712f8e7537997e6da98a3641433237a8ac783b78110a2b4735	3	2	0
0xe7c7be5b6c002458bc4c64aac5733f3659712f8e7537997e6da98a3641433257	2	3	0
0xef2bff903bfc2e604d8dd0912ffa77dd3d5a05b07f76677f0fb212a50160ac1b	2	3	0
0x9e2fa59dd19214304cde799a068b0035b98765e044183db6f8d4ad316b8e99cb	2	2	0
0xa03bcb9c0a768df0b827f99770bce7c7be5b6c002458bc4c64aac5733f36599d	2	2	0
0xc0345e73203bcb9c0a768df0b827f99770bce7c7be5b6c002458bc4c64aac5b7	1	1	2

0xc1433237a8ac783b78110a2b46d751de23c28fd4b1983b4f1d54ee9030f93b1d	2	2	0
0xc4d239585cb7b6fbfb10790d0899370f6520c815d6d827f13349f438a65d1305	0	1	3
0xcb501a97a0b0f42a6b859a2029da55613df1ec539cdcd8be42e70914f112428d	1	3	0
0xd42a6bb1a385e8dda120b1cb98434d20438443aa87ccb1c889dd58acdaac4875	1	3	0
0xd5fcb6ae781f0c09e5c899f3bd13f8f4857531701d89283c74a1a0d1095f4e13	1	3	0
0xda6d20b746a26f2e4c356d5e7624d2544cdebb47c0ffb2b85fbc7c180635f213	2	2	0
0xe1dd0df7429299e6d6f97a296c4687d309df9c101cfd15fcb6ae781f0c09e6a9	1	3	0
0xe2603407b73279a05c1e6db55dd9309cbcc66d59f3ab612e5cc01c74b588c32f	2	2	0
0xee49267b175c945ca27f8469c242616a53ef674582f3aade6897caa4b1ca50cb	2	2	0
0xef8e7537997e6da98a3641433237a8ac783b78110a2b46d751de23c28fd4b1af	1	1	2
0xf52e0ff0751acfb461dd0df7429299e6d6f97a296c4687d309df9c101cfd166b	2	2	0
0x103099921e45d97639752b0a4a8648d1a65e0407dcf336fa97c4ce917d211aec7	2	2	0
0xa1f8b0ce5c432e33fe0f8dddd03dccd322279fa33e04d51e210505dc3c1ba6d9	1	1	1
0xaffa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a57811	1	1	1
0xc160a99e77a8215dadd95ccb73a576b30595cbbcf52e0ff0751acfb461dd0e5f	0	3	0
0xc5b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576b30595cbbcf559	1	2	0
0xccde799a068b0035b98765e044183db6f8d4ad316b8e98b8351ff41c86152be5	1	2	0
0xcfb461dd0df7429299e6d6f97a296c4687d309df9c101cfd15fcb6ae781f0d3b	1	2	0
0xd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903bfc2e6b	1	2	0
0xd6f97a296c4687d309df9c101cfd15fcb6ae781f0c09e5c899f3bd13f8f48585	0	2	1
0xd8be42e70914f11241b2252e00829225d52d86f21840ac4caf47c4d239585e4b	1	2	0
0xdccb73a576b30595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c5b	1	1	1
0xe65d12eb6b52a7141d0548855a724d47270ce438914eb84e1bde4c4573d1494f	0	0	3
0xebc089f709efd77dd07e3003800ac53f678d6b7bb5a7079f1785359eed1c6bd7	1	1	1
0xf2179cb701890a254cbd31b84eb1a4285fb10ccd290cc0b25b3f5ca5b0696ca1	1	2	0
0xf8f4857531701d89283c74a1a0d1095f4deb469cbae2cb501a97a0b0f42a6c5f	1	2	0
0xf90d0899370f6520c815d6d827f13349f438a65d12eb6b52a7141d0548855abf	1	2	0
0xfe4754aac0c76d04364275571cda2a62aeef028bb4af625a24ae5509a3a3a665	0	2	1
0x110c89933cafb6cfb6436d06961c6873ea3c5f3b15ac60f1969bdd0048e7d2f33	1	2	0
0x1343b1fe79e10fdc5e6aa6ab9e733079b9d3e2a2903bd04e92c366cdc697b5987	1	1	1
0x9c101cfd15fcb6ae781f0c09e5c899f3bd13f8f4857531701d89283c74a1a1c5	1	1	0
0xa304b8638eef8d95ef04f2ee24508d5804c50419955cf629c1c42e8e257fa017	1	1	0
0xb45e73203bcb9c0a768df0b827f99770bce7c7be5b6c002458bc4c64aac5745f	1	1	0
0xc595cbbcf52e0ff0751acfb461dd0df7429299e6d6f97a296c4687d309df9ccd	1	1	0
0xc968c595cb14e03137ead10a4ae77f90b8ea9d5dcb61d81aad3ff5382c96bf03	1	1	0
0xcc09e5c899f3bd13f8f4857531701d89283c74a1a0d1095f4deb469cbae2cc71	1	1	0
0xcd23c204392cdb42f911580192dddd955c55b0bc1fdb8eb62c2016cd6123a287	0	1	1
0xcdf0b827f99770bce7c7be5b6c002458bc4c64aac5733f3659712f8e753799df	1	1	0
0xd241b2252e00829225d52d86f21840ac4caf47c4d239585cb7b6fbfb10790d89	1	0	1
0xda05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576b30595cbbd57	1	1	0
0xdacfb461dd0df7429299e6d6f97a296c4687d309df9c101cfd15fcb6ae781f97	1	1	0

0xdb6611590a15491190ab6661f2a27f3625f5f22a429718537734cbd6117b7f93	1	1	0
0xdf4deb469cbae2cb501a97a0b0f42a6b859a2029da55613df1ec539cdcd8bf17	1	0	1
0xe0a99e77a8215dadd95ccb73a576b30595cbbcf52e0ff0751acfb461dd0df7f1	1	1	0
0xe65a10bb23d2ed59f0eff20ca5455aa93215150659f9f394d2bf92b7e2fa7b73	0	2	0
0xe6c46b8973ac0731939ef31024a6c913b168ad87b58252cc161c8e07ad13bdad	1	1	0
0xe70ce438914eb84e1bde4c4573d1491a89a13db1aeef77cbaf737a40e89b4379	1	1	0
0xe89b430b77e26ae6fd2fd15135df9f8e8734c56078385cdba421a50e142a6e29	1	1	0
0xf003800ac53f678d6b7bb5a7079f1785359eed1c6adbaf2bff903bfc2e604e71	1	1	0
0xf42a6b859a2029da55613df1ec539cdcd8be42e70914f11241b2252e00829353	1	1	0
0xf5cc25df5f16d0c0f4fa7ab04e3191aaebc089f709efd77dd07e3003800ac611	1	1	0
0xf7dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576b305e9	1	1	0
0xfa77dd3d5a05b07f76677f0fb212a50160a99e77a8215dadd95ccb73a576b3b1	1	1	0
0xff90b8ea9d5dcb61d81aad3ff5382c96bede99e2c844f5cc25df5f16d0c0f507	1	1	0
0xfffa2e9cf62bc921ffb13f13bdaa9fb91a6b6408660da719bb1c6ea310dfcb49	0	0	2
0x11f1cfb3368016c4b43186fee62ef720a0c92d3a642158c0638374f18e5416a45	1	1	0
0x12a67a0bf0dd8e42ef6cbb18e2e1e854dc60c87873a961a08e87009fb9562e003	0	2	0
0x13e279ee5b202a34e4aff53113c1185ee6b860fcbf2c06d69e6c1578b5ad405a3	1	1	0
0x1a09298358ab0d701c11e0486a7172d7e34da1a95731dbe0533f6d585d82003c9	1	1	0
0xa7a34c9b28f20ca512634c4f0c6f7791df2c070b999dfe87e03aa994a18e9865	1	0	0
0xc2d43b507233005c125d166a0667603a7cd7a8f8d19a6d39ff90fc8e4d6b05a9	0	1	0
0xc3aa87ccb1c889dd58acdaac474bbe63246e9735ef8c3d2193e3f17c28446adf	0	0	1
0xc64e39acacac6d627bee3a5e9b9fa92c304cc9833fd5f3af314b51a88b577b47	1	0	0
0xc9df9c101cfd15fcb6ae781f0c09e5c899f3bd13f8f4857531701d89283c74e3	0	0	1
0xd9f0250e5a604e6d80493bd154eb39b689aac0345e73203bcb9c0a768df0b87b	0	1	0
0xdf2a1c81d7bae2fcfe146b6496a5f4439781e85e1dfc1d7aa3733638c3638da7	0	0	1
0xe15dadd95ccb73a576b30595cbbcf52e0ff0751acfb461dd0df7429299e6d6f9	1	0	0
0xe83c74a1a0d1095f4deb469cbae2cb501a97a0b0f42a6b859a2029da55613df5	0	0	1
0xe9da55613df1ec539cdcd8be42e70914f11241b2252e00829225d52d86f2187d	0	1	0
0xeadbaf2bff903bfc2e604d8dd0912ffa77dd3d5a05b07f76677f0fb212a50179	1	0	0
0xec539cdcd8be42e70914f11241b2252e00829225d52d86f21840ac4caf47c7bf	1	0	0
0xf1701d89283c74a1a0d1095f4deb469cbae2cb501a97a0b0f42a6b859a202a1b	1	0	0
0xff76677f0fb212a50160a99e77a8215dadd95ccb73a576b30595cbbcf52e1067	0	1	0

– Résumé –

Dans ces travaux, on analyse des certificats SSL/TLS X.509 utilisant le chiffrement RSA issus de centaines de millions de matériels connectés, à la recherche d'anomalies et on étend les travaux de Hastings, Fried et Heninger (2016) notamment. Notre étude a été réalisée sur trois bases de données provenant de l'EFF (2010-2011), de l'ANSSI (2011-2017) et de Rapid7 (2017-2021). Plusieurs vulnérabilités affectant des matériels de fabricants connus furent détectées : modules de petites tailles, modules redondants, certificats invalides mais toujours en usage, modules vulnérables à l'attaque ROCA ainsi que des modules dits «PGCD-vulnérables» (i.e. des modules ayant des facteurs communs).

On a identifié 1,550,382 certificats dont les modules sont PGCD-vulnérables, permettant de factoriser 14,765 modules de 2048 bits ce qui, à notre connaissance, n'a jamais été fait. En analysant certains modules PGCD-vulnérables, nous avons pu rétro-concevoir en partie le générateur de modules (de 512 bits) utilisé par certaines familles de pare-feux, ce qui a permis la factorisation instantanée de 42 modules de 512 bits, correspondant aux certificats provenant de 8,817 adresses IPv4. Dans la dernière partie de nos travaux, nous analysons les codes sources et les méthodes en charge du processus de génération de clefs RSA des versions d'OpenSSL (couvrant la période 2005 à 2021) et avons pu remonter aux causes de plusieurs de ces vulnérabilités.
Mots clés : Génération de paramètres cryptographiques, Certificats SSL/TLS, Déviation statistique dans les RNG, Cryptologie, OpenSSL, Matériels connectés.

– Abstract –

In this work, we analyze SSL/TLS X.509 certificates using RSA encryption from hundreds of millions of connected devices, looking for anomalies and we extend the work of Hastings, Fried and Heninger (2016) in particular. Our study was carried out on three databases from EFF (2010-2011), ANSSI (2011-2017) and Rapid7 (2017-2021). Several vulnerabilities affecting devices from well-known manufacturers were detected : small moduli, redundant moduli, invalid certificates but still in use, moduli vulnerable to the ROCA attack as well as so-called "GCD-vulnerable" moduli (i.e. moduli with common factors).

We identified 1,550,382 certificates whose moduli are GCD-vulnerable, allowing to factorize 14,765 moduli of 2048 bits which, to our knowledge, has never been done. By analyzing some GCD-vulnerable moduli, we were able to partially reverse-engineer the 512-bit modulus generator used by some families of firewalls, which allowed the instantaneous factorization of 42 512-bit moduli, corresponding to the certificates from 8,817 IPv4 addresses. In the last part of our work, we analyze the source codes and the methods in charge of the RSA key generation process of the versions of OpenSSL (covering the period 2005 to 2021) and were able to trace the causes of several of these vulnerabilities.
Keywords : Generation of cryptographic parameters, SSL/TLS Certificates, Statistical biases of RNG, Cryptology, OpenSSL, Network devices.