Abstract

SSL/TLS, a 20-year old security protocol, has become a major component securing network communications, from HTTPS e-commerce and social network sites to Virtual Private Networks, from e-mail protocols to virtually every possible protocol. In the recent years, SSL/TLS has received a lot of attentions, leading to the discovery of many security vulnerabilities, and to protocol improvements. In this thesis, we explore the SSL/TLS ecosystem at large using IPv4 HTTPS scans, while proposing collection and analysis methodologies to obtain reproducible and comparable results across different measurement campaigns. Beyond these observations, we focus on two key aspects of TLS security: how to mitigate Record Protocol attacks, and how to write safe and efficient parsers. We also build on the numerous implementation flaws in almost all TLS stacks in the last years, and we propose some thoughts about the challenges in writing a secure TLS library.

The first part presents a brief history of SSL/TLS and contains a state of the art of the known attacks devised on the SSL/TLS protocols. It then focuses on the Record protocol. By analysing cryptographic attacks affecting this layer published since 2011, we show a pattern common to all of the underlying flaws: the repetition of a secret within and across different TLS sessions. We also propose generic countermeasures to defend against this kind of attacks in the case of HTTPS connections; to this aim, we mask the targeted secret values with random values. Even if our mechanisms are not supposed to replace structural fixes added to the protocol by updates, they allow to save time against an attacker in the meantime. As a matter of fact, after we proposed our defense-in-depth countermeasures, the POODLE attack was published, and would have been thwarted.

In the second part, we describe a set of campaigns we launched between 2010 and 2015 to scan and analyse the IPv4 HTTPS server at wild. Before discussing the results across the different campaigns, we first present our collection methodology; we also compare our way of enumerating hosts, i.e. using full IPv4 scans, with other techniques. Moreover, we propose a framework to analyse SSL/TLS data in a reproducible manner, called `concerto`. This proved useful to compare results between our first article published in 2012 and those compiled for this manuscript, even though several analyses had evolved in this period of time. Concerning the results themselves, our work has two distinctive features. First, we sent multiple stimuli to each contacted hosts in 2011 and 2014, allowing us to get an interesting insight into server behaviour. Next, we studied several subsets of the TLS servers, based on trusted hosts according to different certificate trust stores.

The third part presents several aspects of the implementation challenges a developer has to face when writing a TLS stack. We first describe our approach concerning the parsing step for TLS messages and X.509 certificates. To this aim, we show different ways of writing binary parsers, and then focus on a framework, `parsifal`, that we developed to produce robust and efficient parsers. This part also contains an extensive analysis of recent implementation bugs affecting TLS stacks. This inventory allows us to identify the root causes of recurring flaws and to propose possible responses to improve the situation in the long term.

The thesis is a study of the TLS ecosystem, considered from different points of view: first an analysis based on the specifications, then an experimental observation of HTTPS servers world-wide, between 2010 and 2015, and finally several thoughts and proposals regarding the implementation aspects. Beyond our work, these three axis offer several perspectives. Concerning the specifications, TLS 1.3 and its security analysis will be an important step. Building on our collection and reproducible analysis framework, new and regular TLS campaigns could be launched; they should include multiple stimuli to help observe the evolution of the ecosystem in practice. Finally, to improve the security of implementations, more work could be done on programming languages and on the related software engineering.

Résumé

SSL/TLS est un protocole de sécurité datant de 1995 qui est devenu aujourd'hui une brique essentielle pour la sécurité des communications, depuis les sites de commerce en ligne ou les réseaux sociaux jusqu'aux réseaux privés virtuels (VPN), en passant par la protection des protocoles de messagerie électronique, et de nombreux autres protocoles. Ces dernières années, SSL/TLS a été l'objet de toutes les attentions, menant à la découverte de nombreuses failles de sécurité et à des améliorations du protocole. Dans cette thèse, nous explorons l'écosystème SSL/TLS sur Internet en énumérant les serveurs HTTPS sur l'espace IPv4 ; nous proposons pour cela des méthodologies de collecte et d'analyse permettant d'obtenir des résultats reproductibles et comparables entre différentes campagnes de mesure. Au-delà de ces observations, nous nous sommes intéressés en détail à deux aspects essentiels de la sécurité TLS : comment parer les attaques sur le *Record Protocol*, et comment implémenter des *parsers* sûrs et efficaces. En nous basant sur les nombreuses failles d'implémentation qui ont affecté presque toutes les piles TLS ces dernières années, nous tirons quelques enseignements concernant les difficultés liées à l'écriture d'une bibliothèque TLS de confiance.

La première partie présente un rapide historique des protocoles SSL/TLS et propose un état de l'art des attaques connues sur ces protocoles. Nous nous concentrons ensuite sur le *Record Protocol*. En analysant les attaques cryptographiques affectant cette couche publiées depuis 2011, nous montrons qu'elles présentent toutes un point commun : la répétition d'un secret au sein et entre sessions TLS. Nous présentons également des contre-mesures génériques pour se protéger de ce type d'attaque dans le cas de HTTPS ; pour cela, nous proposons de masquer les valeurs secrètes visées avec des valeurs aléatoires. Même si ces mécanismes n'ont pas vocation à remplacer les corrections structurelles apportées par des mises à jour du protocole, ils permettent néanmoins de gagner du temps face à un attaquant dans l'attente du correctif. En pratique, après la mise au point de nos contre-mesures, l'attaque POODLE a été publiée et était déjà contrée, confirmant que notre proposition participait au principe de défense en profondeur.

Dans la seconde partie, nous décrivons des campagnes de mesures que nous avons menées entre 2010 et 2015 pour analyser les serveurs HTTPS de l'espace d'adressage IPv4. Avant la présentation proprement dite des résultats sur ces différentes campagnes, nous expliquons la démarche de collecte mise en œuvre ; nous comparons également notre méthode d'énumération des hôtes TLS (en l'occurrence le parcours de l'ensemble des adresses IPv4) avec d'autres techniques. Ensuite, nous proposons un ensemble de logiciels, `concerto`, pour analyser les données SSL/TLS de manière reproductible. Cette méthodologie d'analyse s'est montrée utile pour comparer des résultats entre notre premier article publié en 2012 et ceux compilés pour ce manuscrit, alors que certains algorithmes avaient évolué entre temps. Concernant les résultats eux-mêmes, notre travail présente deux particularités. Tout d'abord, lors de nos campagnes, nous avons envoyé des stimuli multiples en 2011 et 2014, nous permettant une meilleure compréhension du comportement des serveurs. De plus, les résultats sont présentés pour des sous-ensembles de serveurs TLS validés par différents magasins de certificats.

La troisième partie présente différents aspects liés aux défis que représente l'implémentation d'une pile TLS. Nous commençons par décrire notre approche concernant l'étape de *parsing* des messages TLS et des certificats X.509. Pour cela, nous montrons différentes méthodes pour disséquer les contenus binaires, avant de nous concentrer sur `parsifal`, notre solution pour écrire des *parsers* robustes et efficaces. Cette partie contient aussi une analyse détaillée des récentes failles d'implémentation affectant les piles TLS. Cet inventaire nous permet d'identifier les causes profondes de failles récurrentes et de proposer des réponses pour améliorer la situation à long terme.

Cette thèse est une étude de l'écosystème TLS, vu de différents points de vue : d'abord en se basant sur les spécifications, puis en reposant sur l'observation expérimentale des serveurs HTTPS entre 2010 et

2015, et enfin en s'attachant aux aspects liés à l'implémentation. Au-delà de notre travail, ces trois axes offrent chacun des perspectives pour de futures recherches. Du point de vue des spécifications, TLS 1.3 et son analyse de sécurité seront une étape importante. En s'appuyant sur nos méthodologies de mesure et d'analyse, de nouvelles campagnes pourraient être lancées de manière régulière ; celles-ci devraient inclure plusieurs stimuli pour aider à mieux comprendre l'évolution de l'écosystème en pratique. Enfin, pour améliorer la sécurité des implémentations, des travaux pourraient être menés sur les langages de programmation et l'ingénierie logicielle associée.

Synthèse

SSL (*Secure Sockets Layer*) et TLS (*Transport Layer Security*) sont deux variantes d'un même protocole. Leur objectif est de fournir différents services pour sécuriser un canal de communication : authentification unilatérale du serveur ou mutuelle, confidentialité et intégrité des données échangées de bout en bout. Cette couche de sécurité peut être appliquée à toute couche de transport fiable (c'est-à-dire garantissant la transmission des données de façon ordonnée). En pratique, SSL/TLS est surtout utilisé sur la couche transport TCP, afin de proposer des versions sécurisées de protocoles existants (par exemple HTTPS pour HTTP, IMAPS pour IMAP, etc.). Cette thèse présente différents travaux sur le protocole TLS. La présente synthèse suit le même plan que le manuscrit en anglais.

Dans un premier temps, nous étudions les spécifications du protocole, l'état de l'art des attaques, et les contre-mesures associées. La section 1 présente un rapide historique du protocole, décrit le fonctionnement du protocole, et donne un aperçu des grandes vulnérabilités affectant TLS. La section 2 détaille les attaques sur le *Record Protocol*, qui spécifie comment sont protégées en confidentialité et en intégrité les données véhiculées par TLS. Pour contrer de manière générique cette classe d'attaques, nous proposons également des mécanismes de défense en profondeur.

La seconde partie de cette thèse consiste en une série d'expérimentations pour découvrir l'écosystème des serveurs HTTPS de manière concrète. La section 3 présente les différentes méthodes disponibles pour contacter les serveurs HTTPS. Elle se focalise en particulier sur la méthode que nous avons utilisée : l'énumération exhaustive des adresses IPv4 routables. La section 4 décrit ensuite les outils que nous avons utilisés pour traiter cet ensemble volumineux de données. Elle insiste sur notre méthodologie, que nous avons voulu reproductible dans le temps et compatible avec des données issues de sources diverses. Enfin, la section 5 contient les résultats obtenus à l'aide de cette méthododologie d'analyse sur différents jeux de données.

La dernière partie de nos travaux sur TLS a porté sur l'implémentation de piles TLS. La section 6 s'attache en particulier au problème du *parsing* des formats binaires, tels que ceux utilisés dans TLS. Elle décrit notre solution à ce problème, `parsifal`, un outil pour écrire rapidement des *parsers* robustes et efficaces, que nous avons mis en œuvre dans nos outils d'analyse. Au-delà du problème d'interprétation des messages, la section 7 fait un état de l'art détaillé des failles d'implémentations des piles TLS. Nous y présentons également des idées pour améliorer la sécurité de ces briques logicielles critiques.

1 Présentation du protocole TLS

SSL (*Secure Sockets Layer*) est un protocole mis au point par Netscape à partir de 1994 pour permettre l'établissement d'une connexion sécurisée (chiffrée, intègre et authentifiée). La première version publiée est la version 2 [Hic95], rendue disponible en 1995. SSLv2 fut rapidement suivi d'une version 3 [FKK11], qui corrige des failles conceptuelles importantes. Bien qu'il existe un mode de fonctionnement de compatibilité, les messages de la version 3 diffèrent de ceux de la version 2.

Entre 1999 et 2001, SSL a fait l'objet d'une standardisation par l'IETF (*Internet Engineering Task Force*) et a été renommé TLS (*Transport Layer Security*). Contrairement au passage de SSLv2 à SSLv3, TLS n'a alors pas été l'objet de changements structurels.

Depuis 2001, TLS a connu quelques évolutions. Dès 2003, un cadre permettant des extensions dans le protocole TLS a été décrit [BWNH+03]. Ce cadre a été réactualisé en 2006 et 2011 [BWNH+06, 3rd11]. Ces extensions sont aujourd'hui essentielles pour permettre de faire évoluer le standard de façon souple, mais requièrent l'abandon de la compatibilité avec SSLv2 et SSLv3. La version la plus récente du protocole est actuellement TLS 1.2, publiée en 2008 [DR08].

1.1 Détails d'une connexion TLS classique

Afin de permettre l'établissement d'un canal de communication chiffré et intègre, les deux parties doivent s'entendre sur les algorithmes et les clés à utiliser. Dans cette étape de négociation, plusieurs messages sont échangés. La figure 1 présente un exemple complet. On suppose pour l'exemple que la suite cryptographique négociée utilise l'échange de clé par chiffrement RSA. Des variantes de cet échange de clé mettant en œuvre un échange Diffie-Hellman sont également possibles.

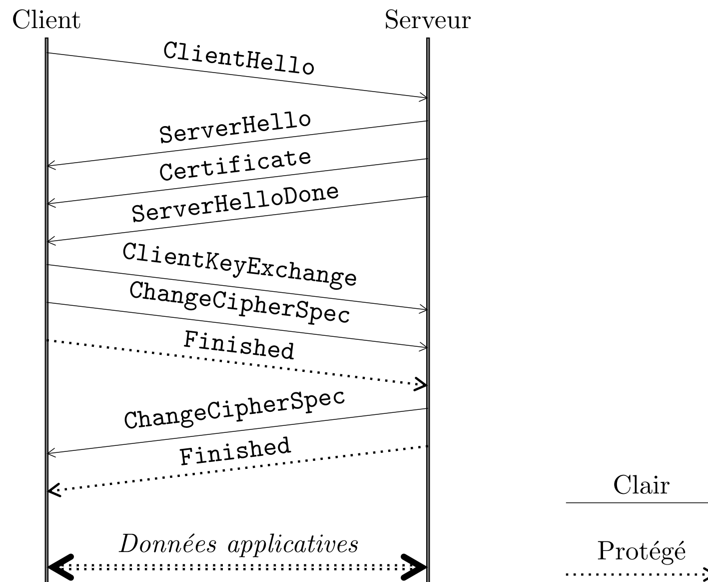

FIGURE 1 : Exemple de négociation TLS avec l'échange de clé par chiffrement RSA.

Envoi du `ClientHello`

La négociation commence avec l'envoi par le client d'un message `ClientHello`. Dans ce message, le client propose un ensemble de suites cryptographiques qu'il est capable de mettre en œuvre. Chaque suite cryptographiques décrit les mécanismes cryptographiques qui seront utilisés pour les fonctions suivantes : l'échange de clés, l'authentification du serveur[1], et la protection des données applicatives, en confidentialité et en intégrité.

Ce message contient d'autres paramètres qui doivent être négociés : la version du standard utilisée (SSLv3, TLS 1.0, TLS 1.1 ou TLS 1.2) et le mécanisme de compression éventuellement appliqué sur les données applicatives. Enfin, il comporte un champ `client_random`, un aléa fourni par le client qui sera utilisé pour la dérivation des clés.

Réponse du serveur

Lorsque le serveur reçoit le `ClientHello`, deux cas peuvent se produire :

- aucune des propositions du client n'est jugée acceptable. Le serveur met alors fin à la connexion avec un message de type `Alert` ;
- dans le cas contraire, le serveur choisit une suite cryptographique parmi celles proposées par le client et émet le message `ServerHello` qui fait état de son choix. Ce message contient également une valeur aléatoire, nommée `server_random`, utilisée lors de la dérivation des clés.

Le serveur envoie ensuite son certificat (dans le message `Certificate`) et termine par un message `ServerHelloDone` pour indiquer qu'il attend maintenant une réponse du client.

[1] L'authentification du client est possible, mais elle se fait de manière indépendante.

Dans l'exemple donné, on suppose que le serveur choisit la suite `TLS_RSA_WITH_AES_128_CBC_SHA` :

- `RSA` décrit ici à la fois la méthode d'authentification du serveur et la méthode d'établissement des secrets de session : une fois que le client aura reçu le certificat du serveur, il tirera un secret de session au hasard, le *pre-master secret*, et le chiffrera en utilisant la clé publique contenue dans le certificat. Cet aléa chiffré sera ensuite envoyé dans le message `ClientKeyExchange`, que seul le serveur pourra déchiffrer. Il s'agit donc d'une authentification implicite du serveur ;
- `AES_128` indique que l'algorithme de chiffrement par bloc AES va être utilisé en mode CBC avec une clé de 128 bits pour chiffrer le canal de communication ;
- `SHA` concerne enfin la protection en intégrité des données : HMAC SHA-1 sera employé.

Fin de la négociation

Une fois la suite choisie et le certificat reçu, le client vérifie la chaîne de certification. Si le certificat n'est pas validé, le client émet une alerte qui met fin à la connexion. Sinon, il poursuit et envoie un message, `ClientKeyExchange`, contenant le *pre-master secret* chiffré.

À ce stade, le client et le serveur disposent tous les deux du *pre-master secret* (le client l'a généré, et le serveur peut déchiffrer le message `ClientKeyExchange`), ainsi que d'éléments aléatoires publics échangés lors des messages `ClientHello` et `ServerHello`. Ces éléments partagés sont alors dérivés pour fournir les clés symétriques déstinées à protéger le trafic en confidentialité et en intégrité.

Des messages `ChangeCipherSpec` sont échangés dans chaque sens pour indiquer l'activation des paramètres (algorithmes et clés) négociés. Les messages `Finished` sont donc les premiers à être protégés cryptographiquement, et contiennent un haché de l'ensemble des messages échangés pendant la négociation, afin de garantir a posteriori l'intégrité de la négociation.

1.2 Modèle d'attaquant

Il existe deux modèles classiques pour l'analyse de sécurité d'un protocole de communication :

- l'**attaquant réseau passif**, capable d'observer les messages échangés entre entités légitimes ;
- l'**attaquant réseau actif**, qui peut de plus ajouter, modifier ou supprimer des messages entre des entités légitimes. Un tel attaquant peut également jouer le rôle de client ou de serveur TLS auprès d'acteurs légitimes.

Dans le cas de TLS, où HTTPS reste le cas d'usage prépondérant, on considère généralement également un attaquant prenant en compte les spécifités du protocole HTTP. L'**attaquant HTTPS actif** peut ainsi tirer profit du fait que le protocole applicatif mélange des éléments secrets avec des données qu'il peut contrôler. Malgré la présence d'un mécanisme de sécurité permettant de limiter les échanges entre les origines différentes, la *Same Origin Policy* [Bar11], ce cloisonnement est loin d'être absolu.

1.3 Aperçu des vulnérabilités affectant TLS

Failles concernant la négociation

La première vulnérabilité envisageable pendant la phase de négociation est le choix de paramètres cryptographique faibles. On peut citer par exemple la sélection d'une suite dite EXPORT, qui limite les clés de chiffrement symétriques à 40 bits ; ces suites, aujourd'hui obsolètes, demeurent cependant supportées par de nombreuses piles TLS. Un autre exemple est l'utilisation de paramètres cryptographiques asymétriques de petite taille (clé RSA de 512 bits, groupe Diffie-Hellman sur 256 ou 512 bits).

SSL/TLS a également été l'objet de vulnérabilités structurelles dans la spécification de sa phase de négociation. Ainsi, SSLv2 ne garantit pas l'intégrité a posteriori des messages de négociation, ce qui permet à un attaquant d'altérer les messages de négociation pour amener un client et un serveur à sélectionner une suite cryptographique plus faible que celle qu'ils auraient choisie en marche normale. Plus récemment, deux attaques ont montré que les garanties de sécurité attendues du protocole n'étaient pas assurées lors de l'utilisation de la renégociation ou de la reprise de session : l'attaque sur la renégociation [Ris09] et le *Triple Handshake* [BDF+14], qui permettaient à un attaquant d'injecter du contenu dans une connexion authentifiée sans que la confusion ne soit détectée.

Attaques sur le *Record Protocol*

Une fois la négociation effectuée, les messages sont protégés en intégrité et en confidentialité à l'aide des algorithmes négociés : il s'agit du *Record Protocol*. La section 2 détaille les attaques à l'encontre de ce protocole. Ces dernières peuvent être classées de la manière suivantes :

- les attaques contre le mode de chiffrement par bloc CBC : BEAST [DR11], Lucky 13 [AP13] et POODLE [MDK14] ;
- les attaques exploitant les biais statistiques de l'algorithme RC4 [IOWM13, ABP$^+$13] ;
- les attaques utilisant la compression comme un canal d'information auxiliaire : CRIME [RD12], TIME [BS13] et BREACH [PHG13].

Problèmes liés aux certificats

Dans le protocole TLS, l'authentification du serveur (et optionnellement celle du client) repose généralement sur des certificats. La sécurité des échanges dépend donc de la qualité de ceux-ci. L'utilisation d'une mauvaise source d'aléa lors de la génération d'un clé cryptographique peut remettre en cause la sécurité. Deux exemples documentés sont la vulnérabilité d'OpenSSL contenue dans la distribution Linux Debian [Deb08] et l'étude des certificats présentés par des équipements réseau [HDWH12].

Un autre problème récurrent concernant les certificats est l'utilisation illégitime d'une autorité de certification pour signer des certificats illégitimes. Cette utilisation peut être frauduleuse, comme dans le cas de l'attaque à l'encontre de Comodo [Com11] ou de Diginotar [Vas11, FI12], ou accidentelle, lorsqu'une autorité de confiance est utilisée à tort dans des systèmes de *data leak prevention*.

Enfin, les certificats souffrent d'un autre problème, qui aggrave les précédents : les mécanismes de révocation associés aux certificats X.509 ne fonctionnent pas en pratique, ce qui force l'utilisation de mécanismes ad-hoc telles que des listes noires embarquées dans les navigateurs.

Erreurs d'implémentation

Comme tout composant logiciel, une pile TLS peut contenir des erreurs d'implémentation, et certaines peuvent mener à des failles de sécurité. Une étude poussée de ces failles est proposée dans la section 7. Parmi les grandes catégories de vulnérabilités TLS résultant d'une erreur d'implémentation, on peut citer les erreurs liées à la gestion de la mémoire comme les dépassements de tampon, les erreurs de logique dans la validation de certificats ou encore les problèmes dans les automates d'état.

1.4 TLS 1.3

Depuis 2013, le groupe de travail TLS de l'IETF travaille sur la prochaine version du standard, TLS 1.3. La motivation initiale était d'accélérer l'établissement de session TLS. Au fur et à mesure que la spécification avançait, le groupe de travail a également pris en compte les nombreuses vulnérabilités publiées en 2014 et 2015 pour améliorer durablement la sécurité du protocole.

Bien que TLS 1.3 ne soit pas encore complètement défini (notre analyse repose sur le 12e brouillon de la RFC [Res16], publié en mars 2016), les points structurants du nouveau standard ne devraient plus changer désormais. TLS 1.3 est en particulier l'occasion d'un nettoyage majeur du protocole.

Tout d'abord, la structure même de la négociation a été repensée pour qu'il soit possible de monter une session en un aller-retour sur le réseau (on parle de RTT, pour *Round-Time Trip*) en mode nominal, alors que les versions précédentes nécessitaient 2 RTT par défaut. Pour cela, TLS 1.3 supprime l'échange de clé par chiffrement RSA, et ne conserve que les modes Diffie-Hellman éphémère sur des groupes prédéfinis[2], ce qui garantit la propriété de sécurité persistante (*forward secrecy*).

Du point de vue cryptographique, ce nouvel échange de clé est plus robuste, puisque l'ensemble du transcript de la négociation est désormais signé par le serveur avec ses paramètres Diffie-Hellman, et non uniquement les aléas publics.

[2] D'autres modes, reposant sur un secret symétrique partagé, ont aussi été conservés. Ils servent notamment à la reprise de session.

Cette nouvelle version du standard a également été l'occasion de supprimer des algorithmes et constructions obsolètes :

- comme vu ci-dessus, le chiffrement RSA disparaît comme mécanisme d'échange de clé ;
- la signature RSA, qui reposait sur PKCS#1 v1.5 est remplacée, au sein du protocole, par la version 2.1 du standard (PSS, *Probabilistic Signature Scheme*) ;
- le mode CBC et l'algorithme RC4 disparaissent au profit des modes combinés tels que GCM.

Certains aspects du standard sont encore sujets à discussion, en particulier le mode 0 RTT. Dans ce mode, un client pourrait envoyer des données chiffrées dès le premier paquet TLS, ce qui serait un excellent argument pour l'utilisation généralisée de TLS. Cependant, ce mode ne peut pas, par nature, offrir les mêmes garanties qu'une négociation standard. Il faudra donc vraisemblablement l'utiliser avec parcimonie, en fonction d'une analyse de risque par protocole applicatif et par emploi.

2 Analyse de sécurité du *Record Protocol*

Depuis 2011, de nombreuses attaques ont été présentées sur le *Record Protocol*, la couche du protocole TLS chargée de protéger en confidentialité et en intégrité le trafic applicatif. Nous analysons dans cette section ces failles, et cette analyse fait ressortir un élément commun à toutes les preuves de concept démontrant les failles : elles reposent sur l'existence d'un secret répété plusieurs fois au sein de différentes sessions TLS. De tels secrets sont omniprésents dans le monde HTTPS : il peut notamment s'agir des *cookies* d'authentification, que le client retransmet à chaque requête.

Afin de contrer ces attaques, nous proposons d'utiliser un mécanisme classique dans le monde des attaques par canaux auxiliaires : le masquage. En effet, en masquant les éléments secrets de manière systématique avec des chaînes de caractères aléatoires, les attaques ne fonctionnent plus. Il s'agit cependant d'un mécanisme de défense en profondeur, qui ne saurait remplacer la correction des vulnérabilités sous-jacentes.

2.1 BEAST

En 1995, Rogaway décrit une attaque à clair choisi adaptatif contre le mode CBC, dès que l'IV utilisé est prédictible [Rog95]. En 2002, Möller remarque que TLS 1.0 remplit les conditions [Möl04]. Cependant, l'attaque n'est alors pas considérée réaliste, étant donné les hypothèses nécessaires (l'attaquant doit partiellement maîtriser le clair). La situation change en 2011 avec la publication de l'attaque BEAST *(Browser Exploit Against SSL/TLS)*, présenté par Duong et Rizzo [DR11].

On suppose que l'attaquant peut choisir le premier bloc du texte clair que le *Record Protocol* va chiffrer. Cela lui permet de tester si un bloc chiffré correspond au bloc clair choisi[3]. Afin de rendre l'attaque pratique, il est même possible de tester un bloc de clair octet par octet, puisqu'une partie des en-têtes est connu de l'attaquant. L'attaque permet donc de retrouver un secret avec 128 essais en moyenne par octet.

La meilleure contre-mesure à cette attaque est de rendre l'IV imprédictible pour chaque *record*, comme le spécifie TLS 1.1. On peut aussi utiliser un algorithme de chiffrement par flot pour éviter d'utiliser le mode CBC, mais cela revient en pratique à utiliser RC4, ce qui pose d'autres problèmes. Une astuce consistant à découper les *records* de longueur n en deux *records* de longueur 1 et $n-1$ a été implémenté dans les navigateurs pour rendre l'attaque inefficace, même avec TLS 1.0.

2.2 CRIME, TIME et BREACH

En 2012, Duong and Rizzo ont publié une autre attaque intitulée CRIME *(Compression Ratio Info-leak Made Easy)* [RD12]. Cette fois encore, l'objectif est de récupérer la valeur d'un *cookie* d'authentification. L'attaque repose sur l'étape de compression, et suppose que l'attaquant peut choisir une partie du texte clair envoyé en même temps que le *cookie*, par exemple le chemin dans l'URL. L'année suivante, une autre équipe de recherche a présenté TIME (*Timing Info-leak Made Easy*) [BS13], une variante de CRIME, utilisant une méthode d'observation différente.

[3] En réalité, ce n'est pas *exactement* le même bloc clair, mais un bloc altéré d'une manière connue de l'attaquant (l'application d'un XOR avec d'autres blocs connus).

Le fonctionnement de ces attaques repose sur le fait que l'attaquant peut déclencher le chiffrement d'un message contenant à la fois un secret (le *cookie*) et des éléments qu'il maîtrise (l'URL ou d'autres en-têtes). Il peut ainsi tester un mot de passe en l'incluant dans le message. Si l'hypothèse sur le mot de passe est bonne, il y aura répétition dans le clair et le message sera mieux compressé. CRIME et TIME utilisent deux méthodes différentes pour observer la différence de taille du *record*. CRIME suppose que l'attaquant peut observer la taille des paquets chiffrés sur le réseau. TIME mesure en revanche le délai de transmission des réponses.

De manière similaire, les auteurs de l'attaque TIME ont proposé une attaque pour récupérer des éléments secrets envoyés de manière répétée par le serveur, comme les jetons anti-CSRF. En 2013, une adaptation de cette dernière attaque, utilisant la compression HTTP pour récupérer les jetons émis par le serveur, a été présentée : BREACH [PHG13].

La seule contre-mesure efficace pour les attaques utilisant la compression TLS est de désactiver cette fonctionnalité. Pour l'attaque côté serveur reposant sur la compression HTTP, il est nécessaire de restreindre la compression HTTP lors des requêtes émises par un site tiers ; pour cela, il faut vérifier l'en-tête *Referer*. En effet, désactiver complètement la compression HTTP dégraderait fortement les performances et augmenterait la bande passante utilisée.

2.3 Lucky 13

Lorsque TLS est utilisé avec un algorithme de chiffrement par bloc, le message clair est concaténé au motif d'intégrité calculé sur le clair à l'aide d'un HMAC, puis le résultat est aligné à la taille d'un bloc (étape de *padding*), et enfin, la primitive de chiffrement par bloc est appelé en mode CBC. Cette construction, *MAC-then-Encrypt*, est connue pour permettre des attaques : les attaques par oracle de *padding*, décrites par Vaudenay en 2002 [Vau02]. Dès qu'un attaquant peut distinguer une erreur liée au *padding* d'une erreur liée au motif d'intégrité, que ce soit à l'aide d'un message d'erreur spécifique, ou à cause d'une différence dans le temps de traitement, il y a une fuite d'information exploitable.

Lors du déchiffrement des *records*, le récepteur doit vérifier que le *padding* est correct : les p derniers octets du bloc doivent avoir la même valeur $p-1$. Ainsi, les blocs terminant par 00, 01 01 ou encore 02 02 02 sont acceptables. Ensuite, le motif d'intégrité est extrait et vérifié. Si l'attaquant peut distinguer entre une erreur d'intégrité et une erreur de *padding*, ce dernier cas permet de détecter le contenu du bloc, octet par octet.

Concrètement, dans le cas de TLS, l'attaquant doit exploiter des différences dans le temps de traitement. Par exemple, en cas de *padding* invalide, le MAC peut ne pas être vérifié, ce qui mène à une réponse du serveur plus rapide. C'est pourquoi TLS 1.1 [DR06] indique que les implémentations doivent s'assurer que le temps de traitement des *records* est essentiellement le même dans tous les cas. AlFardan et Paterson ont néanmoins montré qu'une *timing attack* était possible sur la majorité des implémentations TLS [AP13]. Pour cela, il était nécessaire que le secret à retrouver soit répété dans différentes sessions TLS, puisque chaque essai met fin à la connexion en cours.

2.4 Biais statistiques de RC4

RC4 est un algorithme de chiffrement par flot conçu par Rivest en 1987. Cette primitive est très simple à implémenter et a de très bonnes performances. Depuis 1995, plusieurs biais statistiques ont été identifiés sur les premiers octets de la suite chiffrante produite par RC4.

En 2013, deux équipes de recherche ont présenté des attaques pratiques permettant de retrouver un texte clair s'il avait été chiffré un grand nombre de fois avec différentes clés [IOWM13, ABP+13]. Depuis, de nouveaux travaux ont permis d'améliorer l'attaque [GPvdM15] pour récupérer un mot de passe avec seulement 2^{26} connexions. Suite à ces attaques, l'IETF a publié début 2015 une RFC interdisant l'utilisation de RC4 dans TLS [Pop15], pour envoyer un signal fort quant aux dangers de cet algorithme.

2.5 POODLE

En 2014, Möller, Duong et Kotowicz ont présenté POODLE (*Padding Oracle on Downgraded Legacy Encryption*) [MDK14], une nouvelle attaque sur le *padding* CBC utilisé dans SSLv3. En effet, SSLv3 utilise une méthode de *padding* différente de TLS : quand n octets doivent être ajoutés pour compléter un bloc, *seul le dernier octet du bloc* doit contenir $n-1$, les autres octets de *padding* pouvant prendre

des valeurs arbitraires. Un attaquant peut exploiter ce laxisme dans la vérification pour obtenir un oracle de *padding*. L'attaque nécessite une maîtrise des messages clairs pour aligner les blocs de manière adéquate, mais toute implémentation SSLv3 conforme donne accès à un oracle parfaitement fiable.

2.6 Une contre-mesure générique : le masquage des secrets

Toutes les attaques présentées permettent de retrouver un secret répété dans différentes sessions TLS. Il est possible de les modéliser comme des attaques par canaux auxiliaire de premier ordre [CJRR99, PR13]. À chaque essai, l'attaquant gagne de l'information sur un morceau contigu du clair (typiquement, l'attaquant apprend si un octet donné du clair vaut une valeur donnée).

Une contre-mesure classique dans le monde des canaux auxiliaires est de *masquer* un tel secret. Pour protéger un secret s, cela revient à tirer un aléa de la même taille que s à chaque émission, le masque m, et de transmettre $(m, m \oplus s)$ au lieu de s. Ainsi, le destinataire légitime peut retrouver la valeur du secret de manière évidente (en appliquant l'opération $\oplus$), mais l'attaquant n'apprend plus rien d'utile. En effet, le masque étant à usage unique, la connaissance de l'attaquant concernant une connexion ne lui est d'aucune utilité pour trouver s, et il ne peut pas la combiner avec la connaissance apprise d'une autre connexion puisqu'elle utilise un masque différent.

Implémentation du masquage pour TLS

Afin de tester cette contre-mesure, nous avons implémenté plusieurs formes de masquage. La première est une implémentation naïve au niveau de TLS, et consiste à remplacer l'étage de compression par un masquage global du *record*. Cette implémentation, nommée *TLS Scrambling*, rend les attaques inopérantes et consomme peu de ressources, mais elle présente deux inconvénients majeurs. D'une part, l'étage de compression étant retiré en pratique de TLS 1.3, une mesure reposant sur cette fonctionnalité n'a aucun avenir. D'autre part, puisque nous masquons l'ensemble du message clair, et non uniquement le secret, il ne s'agit pas d'un mécanisme de masquage au sens strict. En conséquence, cette mesure n'offre pas les garanties du masquage : certaines attaques de premier ordre pourraient fonctionner en présence de ce mécanisme.

Les *cookies* HTTP masqués

La seconde implémentation que nous proposons consiste à masquer les *cookies* en modifiant les en-têtes HTTP envoyés. Il est en effet possible de changer la représentation des *cookies* sensibles dans les en-têtes, sans que cela ne perturbe le fonctionnement du client. Ces *cookies* sont en effet généralement définis avec l'attribut `httpOnly`, ce qui signifie qu'ils ne sont pas accessibles au client.

Une première version de nos *MCookies* repose uniquement sur la modification du serveur web : à chaque fois qu'un *cookie* sensible est envoyé au client, il est remplacé par sa version masquée par un aléa frais. Puis, à chaque fois que le client présente ce *cookie* masqué, un nouveau masque est généré et le *cookie* est redéfini. L'avantage de cette solution est qu'elle répond complètement au besoin du masquage, tout en ne nécessitant qu'une modification minime du serveur web. En particulier, ni le client ni l'application web n'ont besoin d'être modifiés. Cependant, redéfinir les *cookies* dans chaque réponse a un impact non négligeable sur la bande passante. De plus, comme le *cookie* est redéfini par le serveur, un attaquant actif peut bloquer les réponses du serveur et forcer le client à renvoyer plusieurs fois le *cookie* masqué avec un même aléa, rendant les attaques à nouveau possibles.

Pour résoudre ces deux inconvénients, une seconde version des *MCookies* a été développée. Elle consiste à faire réaliser le masquage par le client. Cela nécessite donc que le client soit modifié, mais cela permet alors de résoudre efficacement les problèmes de bande passante et de réponse aux attaques actives. De plus, cette évolution des *MCookies* est compatible avec la première version : si le client ne supporte pas le masquage, le serveur passera dans un mode dégradé et effectuera le masquage.

Bien que les mécanismes proposés aportent une protection contre des attaques réelles, il est important de retenir qu'il s'agit d'outils pour assurer la défense en profondeur et qu'ils doivent être implémentés *en complément* des véritables corrections du protocole TLS. Une pile TLS 1.2 à jour et bien configurée permet de répondre à ces menaces.

3 Méthodologie de collecte de données pour les campagnes de mesures HTTPS

Afin de comprendre l'écosystème HTTPS, nous avons choisi d'observer le comportement des serveurs HTTPS accessibles sur Internet. Nous décrivons ici les différentes méthodes existantes pour réaliser ce genre de mesures, en insistant sur les choix que nous avons retenus.

3.1 Énumération des serveurs HTTPS

Il existe plusieurs méthodes pour observer des données HTTPS :

- en énumérant l'ensemble des adresses IPv4 routables pour trouver les machines avec un port 443/tcp ouvert, puis en établissant une session SSL/TLS avec ces hôtes ;
- en contactant un ensemble de serveurs HTTPS à partir d'une liste de noms de machine ;
- en collectant le trafic HTTPS réel d'utilisateurs consentants.

La première méthode est a priori exhaustive, puisqu'elle permet de traiter l'ensemble des adresses IP dans le monde. Elle permet en particulier de contacter des implémentations exotiques, qui ne seraient pas disponibles autrement. Cependant, il existe de nombreuses machines avec un port 443/tcp donnant accès à un service autre que HTTPS. De plus, cette méthode ne tient pas compte de la popularité des hôtes contactés, puisqu'elle ne discrimine pas le serveur `www.google.com` d'un hôte tel que `randomhost.dyndns.org` ou même d'une machine sans nom de domaine.

La seconde option est plus restrictive, mais si la liste de noms de domaine utilisée est bien choisie, elle peut mieux représenter l'usage concret d'HTTPS. De plus, cette méthode est compatible avec l'extension TLS SNI (*Server Name Indication* [3rd11]), qui permet à un client de s'adresser à des hôtes virtuels portés par la même adresse.

Enfin, la dernière méthode a l'avantage d'être passive et de représenter fidèlement le trafic des utilisateurs. Pour la mettre en œuvre, il est essentiel d'avoir accès à un certain volume de trafic pour obtenir des données pertinentes. Cependant, contrairement aux deux premières méthodes, il n'est pas possible de sélectionner les stimuli utilisés pour sonder les serveurs HTTPS.

Pour notre étude, nous avons retenu la première méthode, avec une gestion des campagnes en deux phases, menées en parallèle : dans un premier temps, une énumération brute des serveurs avec le port 443/tcp ouvert ; puis dans un second temps, une ou plusieurs montées de session TLS avec les hôtes énumérés.

Un des contraintes que nous avons dû intégrer est de ne pas surcharger les liens réseau. En effet, une telle surcharge peut mener à la perte de paquets, ou au déclenchement de règles bloquant nos requêtes, qui pourraient être perçues comme des attaques. À l'inverse, il faut éviter que la mesure dure trop longtemps, car certains serveurs HTTPS sont hébergés sur des adresses dynamiques, et notre objectif est d'obtenir un instantané de l'écosystème à un moment donné. C'est pourquoi nous avons choisi un compromis : répartir nos mesures sur deux à trois semaines.

3.2 Le choix des stimuli

Comme nous avons choisi une méthode de collecte active, nous pouvons choisir les stimuli que nous souhaitons utiliser pour sonder les serveurs HTTPS. Contrairement à d'autres mesures réalisées sur l'espace d'adressage IPv4 entier, nous avons ainsi retenu *plusieurs* messages `ClientHello` pour découvrir les fonctionnalités offertes par les différents serveurs. Là encore, pour ne pas surcharger ces serveurs, nous avons limité le nombre de stimuli à une dizaine. Au-delà de messages `ClientHello` standard, il existe plusieurs aspects que nous avons souhaité étudier dans nos mesures :

- le support pour différentes versions du protocole (SSLv2, TLS 1.2) ;
- le support de certaines suites cryptographiques, en envoyant des messages ne proposant que des sous-ensembles restreints d'algorithmes (courbes elliptiques, Diffie-Hellman éphémère) ;
- le support d'extensions TLS.

3.3 Les différents types de réponses obtenues

Pour chaque hôte avec un port 443/tcp ouvert, nous avons donc émis les stimuli retenus, et enregistré les réponses obtenues. Ces réponses peuvent être rangées en trois catégories.

Tout d'abord, environ la moitié des serveurs ne répondent pas à nos stimuli par des messages SSL/TLS valides. Cela peut facilement s'expliquer par le fait que le port 443/tcp est souvent détourné pour héberger d'autres services que HTTPS, puisque ce port n'est généralement pas filtré par les équipements réseau. Certaines de ces réponses sont vides. D'autres correspondent à d'autres protocoles identifiables, tels que des messages HTTP en clair ou des bannières du protocole SSH.

Dans certains cas, les serveurs contactés ne peuvent pas répondre à notre sollicitation, puisqu'ils n'implémentent pas les options proposées dans notre stimulus. Ces serveurs émettent alors des alertes.

Enfin, nous avons également obtenu, comme attendu, de nombreuses réponses incluant des messages de négociation SSL/TLS. Ces réponses commencent par un message `ServerHello` et se terminent par un message `ServerHelloDone`, comme présenté dans la section 1. Après avoir enregistré cette première salve de messages de la part d'un serveur, nous mettons fin à la connexion pour ne pas faire consommer plus de ressources à notre interlocuteur. Il est intéressant de noter que certaines réponses contenaient des paramètres incohérents, qui n'avaient pas été proposés par notre client. Ce comportement, non conforme au standard TLS, est une manifestation possible de l'intolérance des serveurs à certaines options.

4 `concerto` : une méthodologie d'analyse reproductible

Cette section présente un ensemble d'outils, `concerto`, permettant l'analyse des campagnes de données TLS. Il peut s'agir de nos jeux de données, présentés à la section précédente, mais également d'autres jeux de données publics.

Initialement, `concerto` était un outil permettant d'extraire et d'analyser les chaînes de certificats présentés par un sous-ensemble restreint de serveurs. Il a ensuite été étendu pour pouvoir traiter des campagnes TLS entières et intégrer les autres paramètres contenus dans nos jeux de données, en particulier la version du protocole et la suite cryptographique sélectionnées par le serveur.

4.1 L'origine de `concerto` : l'analyse des certificats X.509

L'authentification des serveurs TLS repose généralement sur des certificats X.509. À cette fin, les serveurs envoient un message `Certificate`, contenant le certificat du serveur, ainsi que toute la chaîne de certification permettant d'en vérifier la validité.

Le premier certificat, lié à l'identité du serveur HTTPS, contient la clé publique qui sera utilisée pendant la phase de négociation (soit pour chiffrer un secret dans le cas de l'échange de clé par chiffrement RSA, soit pour vérifier une signature calculée par le serveur). Ensuite, en partant du certificat du serveur, chaque certificat du message est censé être signé par l'autorité dont le certificat vient juste après dans le message. Cette chaîne de certification se termine normalement par un certificat auto-signé correspondant à une autorité de confiance[4].

La réalité est toute autre, puisque de nombreux serveurs envoient des messages `Certificate` dans le désordre. La plupart des implémentations s'accommodent de tels messages. Il existe même des cas légitimes de ne pas suivre la spécification sur ce point, lorsqu'il existe plusieurs chaînes de certification : un certificat serveur peut en effet remonter à deux autorités de certification distinctes. Dans ce cas, pour maximiser la compatibilité du site avec des clients variés, le message peut contenir l'ensemble des certificats pour reconstituer les différentes chaînes de certification. Concrètement, cette contrainte imposant l'ordre des certificats devrait être levée dans TLS 1.3.

Un autre problème avec les messages `Certificate` est que certains serveurs omettent d'envoyer des certificats d'autorité intermédiaires. Face à de tels messages, le client ne peut pas a priori reconstituer la chaîne de certification, et devrait donc rejeter la connexion. Il existe des moyens de contourner ce problème, par exemple en maintenant dans le client un cache des autorités intermédiaires rencontrées ou en téléchargeant les certificats manquants en suivant des liens depuis une extension X.509[5].

[4] Ce dernier certificat peut être omis en pratique. En effet, soit l'autorité est reconnue de confiance par le client, et ce dernier connaît déjà le certificat ; soit le client n'a pas confiance en cette autorité et la chaîne doit être rejetée.

[5] Cependant, c'est une mauvaise pratique d'initier de telles connexions, puisque la source de l'information n'est pas encore authentifiée.

Pour permettre à des administrateurs de repérer ce genre de problèmes dans des déploiements réels, `concerto` a pour objectif d'analyser les chaînes de certification transmises. Pour cela, l'outil essaie de construire les chaînes possibles, et de noter chaque chaîne en fonction de sa qualité : la chaîne est-elle complète ? de confiance ? ordonnée ?

Après quelques expérimentations sur de petits jeux de données, nous avons étendu `concerto` pour pouvoir traiter des volumes de données plus importants et pour prendre en compte d'autres paramètres que les certificats.

4.2 Fonctionnement de `concerto`

Pour cela, `concerto` repose désormais sur de nombreux outils élémentaires, pour mieux passer à l'échelle. Les étapes de dissection (*parsing*) des données binaires sont réalisées à l'aide de `parsifal`, qui est présenté dans la section 6. Le fonctionnement de `concerto` se décompose en différentes phases.

Dans un premier temps, il faut préparer le contexte d'une analyse, en injectant un certain nombre de métadonnées : les stimuli utilisés pendant la ou les campagnes traitées (cette information permettra d'identifier les anomalies dans les réponses des serveurs, comme la sélection d'une suite cryptographique qui n'avait pas été proposée), et le ou les magasins de certificats à utiliser pour les analyses de certificats.

L'étape suivante consiste en l'injection des réponses des serveurs. L'objectif est ici de *parser* les réponses brutes des serveurs pour noter les paramètres sélectionnés et stocker l'ensemble des certificats présentés. `concerto` propose plusieurs outils pour réaliser cette opération, en fonction de la source des données brutes.

Ensuite, les certificats extraits sont analysés. Lors de cette phase, les informations pertinentes sont extraites de chaque certificat X.509. Puis, l'ensemble des liens possibles entre certificats est vérifié. Enfin, les chaînes de certification sont construites, en tenant compte à la fois des certificats envoyés par les serveurs, mais aussi en s'autorisant à prélever des certificats intermédiaires extérieurs.

Une fois toutes ces données extraites et analysées, quelques outils permettent de produire des statistiques sur les campagnes traitées. En parallèle, d'autres outils permettent d'explorer les résultats via une interface web.

La figure 2 présente la cinématique complète de `concerto`, depuis l'injection de données jusqu'à la production de statistiques. Les rectangles blancs représentent les différents outils. Les parallélogrammes gris sont les tables CSV produites par `concerto` et les éléments ovales en blanc sont des fichiers. Tout commence en haut du schéma par le stimulus (`ClientHello`), les données collectées *(answer dumps)* et le magasin de certificats extrait des sources de la bibliothèque NSS[6] à partir de la date des mesures.

4.3 Quelques réflexions sur les choix d'implémentation

L'ensemble des données traitées par `concerto` est aujourd'hui stockées dans de grands fichiers texte au format CSV. En effet, les traitements nécessaires aux différentes phases présentées ci-dessus nécessitent d'accéder aux données de deux manières différentes : soit en flux (en traitant les fichiers ligne par ligne), soit de manière globale (ce qui nécessite la lecture de l'ensemble des données). Ces deux types d'accès sont compatibles avec de simples fichiers CSV. Une fois les données extraites et analysées, il est cependant possible de les injecter dans une base de données SQL pour permettre des requêtes plus fines. C'est ce qui est fait par exemple pour l'interface web incluse dans `concerto`, `piccolo`.

L'analyse de grands volumes de données TLS est un défi pour lequel certains choix sont dimensionnants pour le passage à l'échelle, en particulier pour l'analyse des certificats. Sans rentrer dans les détails, on peut citer deux exemples. Tout d'abord, lors de la construction de toutes les chaînes possibles, nous avons introduit une limitation sur le nombre de certificats d'autorités intermédiaires à considérer en dehors du message `Certificate` transmis. En effet, il existe des autorités de certificats se signant de manière croisée ; certaines d'entre elles sont de plus associées à des certificats multiples (mais partageant la même clé publique). Sans cette limitation sur les autorités intermédiaires extérieures, le nombre de chaînes que l'on peut construire explose, sans apporter de nouvelle information intéressante.

Un autre problème menant à une explosion combinatoire est la gestion des certificats X.509v1. La première version du standard ne spécifiant pas d'extensions, il n'est pas possible de distinguer un certificat terminal d'un certificat d'autorité. Historiquement les certificats X.509v1 étaient donc

[6] La bibliothèque *Network Security Services* est un composant des produits Mozilla tels que le navigateur Firefox ou le client de messagerie Thunderbird. Elle est en particulier responsable de la gestion des certificats et des connexions TLS.

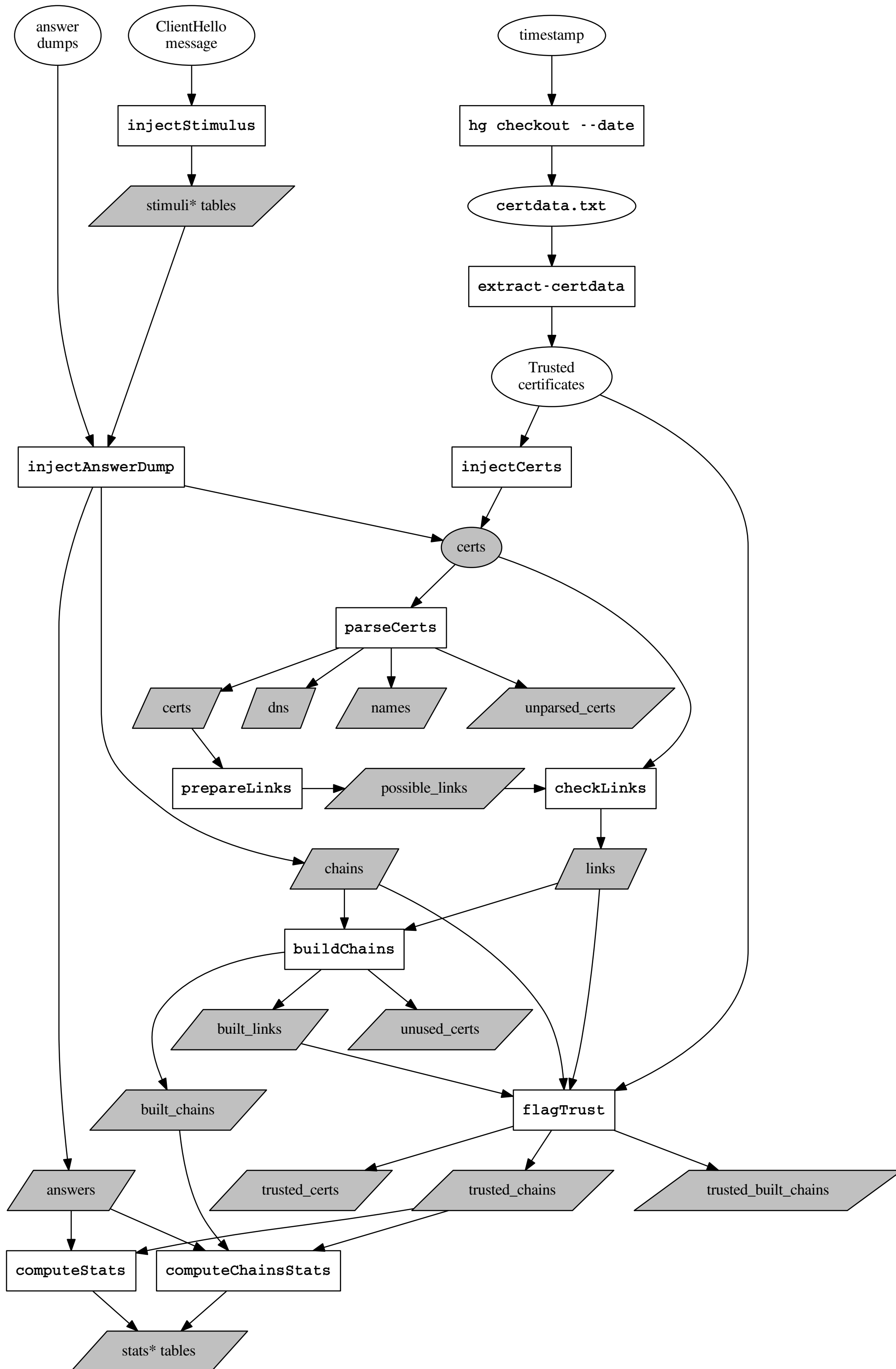

FIGURE 2 : Cinématique complète de `concerto` sur une campagne.

tous reconnus comme des autorités de certification potentielles, ce qui peut mener à une quantité importante de liens inintéressants à vérifier. Par exemple, il existe des équipements réseau et des logiciels de virtualisation générant des certificats X.509v1 ayant tous une forme similaire. En particulier, nous avons observé un groupe de 140 000 certificats auto-signés distincts présentant la même identité ; sans information complémentaire, il faut examiner les 140 000^2 liens possibles. Pour éviter ces calculs inutiles, seuls les certificats X.509v1 contenus dans le magasin de confiance sont considérés comme des autorités de certification, ce qui est conforme avec le comportement des piles TLS actuelles.

5 Analyse comparative de l'écosystème HTTPS

La méthodologie d'analyse reposant sur `concerto`, décrite dans la section précédente, a été appliquée sur les données des campagnes TLS présentées à la section 3 et sur des jeux de données indépendants. Dans cette section, nous étudions tout d'abord les jeux de données pertinents pour notre étude. Ensuite, les résultats concernant différents paramètres sont analysés pour évaluer la qualité des connexions TLS et leur évolution dans le temps.

5.1 Sélection des jeux de données

En marge des campagnes que nous avons menées en 2010, 2011 et 2014, plusieurs études ont été menées depuis 2010 sur TLS. Pour certaines d'entre elles, les données brutes ont été rendues publiques. Afin de pouvoir appliquer `concerto`, les campagnes retenues doivent respecter certains critères :

- les mesures doivent avoir été réalisées selon un protocole rigoureux et documenté, depuis une adresse IP source unique ;
- idéalement, les messages `ClientHello` utilisés comme stimuli doivent être connus ;
- afin de pouvoir valider les chaînes de certification au moment de leur envoi, les réponses des serveurs doivent être horodatées.

Les données collectées à Télécom SudParis en utilisant notre méthodologie de collecte respectent ces conditions. Ces jeux de données contiennent des campagnes correspondant à juillet 2010 (un stimulus), juillet 2011 (7 stimuli) et mars 2014 (10 stimuli).

Parallèlement à nos travaux en 2010, l'EFF *(Electronic Frontier Foundation)* a réalisé des mesures équivalentes en août et décembre 2010 [EB10a, EB10b]. Les données brutes correspondantes ont été publiées, ainsi que les outils utilisés pour la collecte. Bien que certains éléments de la méthodologie n'aient pas été détaillés (en particulier l'articulation entre la phase d'énumération des ports 443/tcp ouverts et la phase de montée de session TLS), nous avons choisi d'utiliser ces données dans notre analyse.

Depuis 2013, des chercheurs de l'université du Michigan ont publié des articles concernant des campagnes de mesures sur Internet [DWH13, DKBH13]. Les outils sur lesquels reposent ces mesures, ZMap et ZGrab, ont été rendus disponibles. De même, de nombreux jeux de données ont été publiés sur la plate-forme `scans.io`. Ces jeux de données de qualité ont été intégrés dans notre corpus.

Enfin, d'autres initiatives concernant des campagnes de mesures TLS ont été présentées depuis 2010, mais nous ne les avons pas retenues pour nos travaux. Dans certains cas, les données brutes n'étaient pas disponibles publiquement [HBKC11, Ris10]. Dans d'autres cas, la qualité des données n'était pas suffisante pour notre étude [Bot12, KHF14].

5.2 Analyse des paramètres TLS

Après traitement par `concerto`, les premiers éléments que nous avons analysés sur les jeux de données à notre disposition concernent les paramètres TLS choisis par le serveur.

Version du protocole

Tout d'abord, nous avons étudié les versions du protocole supportées par les serveurs. Jusqu'en 2014, les stimuli standard utilisés proposent au mieux d'utiliser TLS 1.0. Pour 2014 et 2015, nous disposons également de stimuli TLS 1.2. La figure 3 présente les résultats pour des campagnes en 2010, 2011, 2014

(avec un stimulus TLS 1.0 et un stimulus TLS 1.2) et 2015. Elle contient également, pour information, les résultats pour les serveurs de la liste Top Alexa 1 Million, pour lesquels les données sont disponibles sur le site `scans.io`.

Pour des messages `ClientHello` standard limités à TLS 1.0, nous observons le même résultat entre 2010 et 2014 : TLS 1.0 est préféré par 96 % des serveurs à SSLv3. En 2014, la proportion des serveurs HTTPS compatibles avec TLS 1.2 était de 30 %. La situation s'est améliorée en 2015, puisque près de la moitié des serveurs choisit la dernière version du protocole.

Il est également intéressant de constater que peu de serveurs choisissent TLS 1.1 lorsqu'ils se voient proposer TLS 1.2. Cela semble indiquer que la majorité des serveurs compatibles avec TLS 1.1 supporte également TLS 1.2.

Bien que la situation soit plutôt favorable, il reste encore une proportion non négligeable de serveurs sélectionnant SSLv3, une version obsolète du protocole. De même, TLS 1.2 datant de 2006, il est décevant que la proportion des serveurs supportant cette version en 2015 n'atteigne pas 50 %.

Suites cryptographiques

Un autre paramètre que nous avons naturellement étudié est la suite cryptographique sélectionnée par le serveur. Plutôt que de s'intéresser aux choix précis faits par les différents serveurs, nous nous sommes intéressés à des catégories de suites cryptographiques. En particulier, nous avons étudié le critère de sécurité persistante (*forward secrecy*).

Dans chacune des campagnes considérées, le message `ClientHello` présentait des suites cryptographiques offrant la propriété de *forward secrecy*. La figure 4 montre la proportion des serveurs ayant choisi une suite cryptographique assurant cette propriété. Les résultats sont donnés pour l'ensemble des serveurs *(TLS hosts)*, ainsi que pour des sous-ensembles de serveurs considérés comme de confiance vis-à-vis de deux magasins de confiance issus de la bibliothèque NSS (*Trusted hosts* pour le magasin complet et *EV hosts* pour les certificats racines *Extended Validation*).

Entre 2010 et 2014, cette proportion a baissé, pour finalement s'améliorer en 2015. Cette dernière évolution est clairement liée à l'augmentation nette du support des suites cryptographiques proposant un échange de clé Diffie-Hellman sur courbes elliptiques constaté dans les données.

5.3 Qualité des chaînes de certification

Nous avons également étudié la qualité des chaînes de certification envoyées par les serveurs en 2010, 2011, 2014 et 2015. La figure 5 montre la répartition des chaînes envoyées par les serveurs selon la classification suivante :

- les chaînes compatibles avec la RFC *(RFC compliant)*, qui contiennent l'ensemble des certificats pour remonter à une autorité de confiance, dans l'ordre ;
- les chaînes désordonnées *(Unordered)*, qui contiennent l'ensemble des certificats nécessaires à la construction d'une chaîne de confiance, mais dans le désordre et avec d'éventuels certificats dupliqués ou inutiles ;
- les chaînes transvalides *(Transvalid)*, pour lesquelles il a été nécessaire d'utiliser des certificats en dehors du message envoyé par le serveur pour construire une chaîne complète ;
- les chaînes incomplètes *(Incomplete)*, pour lesquelles aucune chaîne n'a pu être construite.

En faisant abstraction des chaînes incomplètes, on observe deux tendances. D'une part la proportion de chaînes transvalides, clairement non conformes à la spécification, est faible mais non nulle, dans toutes les campagnes. D'autre part, la proportion de chaînes désordonnées a augmenté en 2014, ce qui peut s'expliquer par le besoin pour certains serveurs de présenter une double chaîne de certification pour rester compatible avec d'anciens clients (qui ne supportent que la fonction de hachage SHA-1 par exemple) tout en proposant une chaîne de certification moderne (reposant sur des autorités de certification utilisant la fonction de hachage SHA-256) aux autres clients.

5.4 Analyse du comportement des serveurs

L'envoi de plusieurs stimuli à l'ensemble des serveurs dans nos campagnes de mesures nous a permis d'analyser le comportement des serveurs face à des options non standard. En effet, nous avons considéré

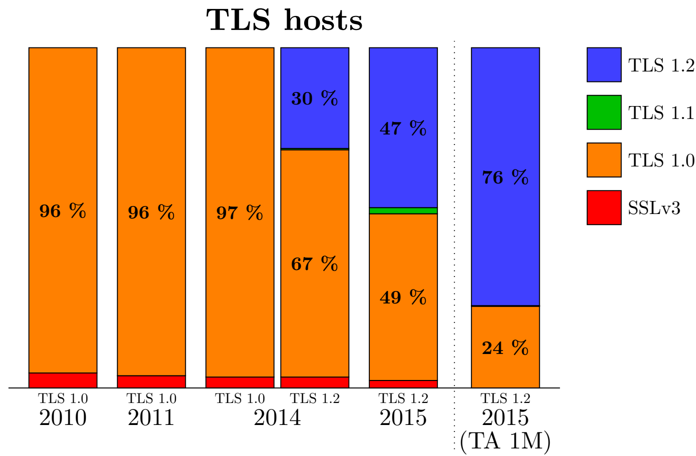

FIGURE 3 : Évolution de la version du protocole choisie par les serveurs entre 2010 et 2015. Les cinq premières colonnes correspondent à des campagnes sur tout l'espace IPv4, alors que la dernière colonne concerne le sous-ensemble constitué des serveurs du Top Alexa 1 Million.

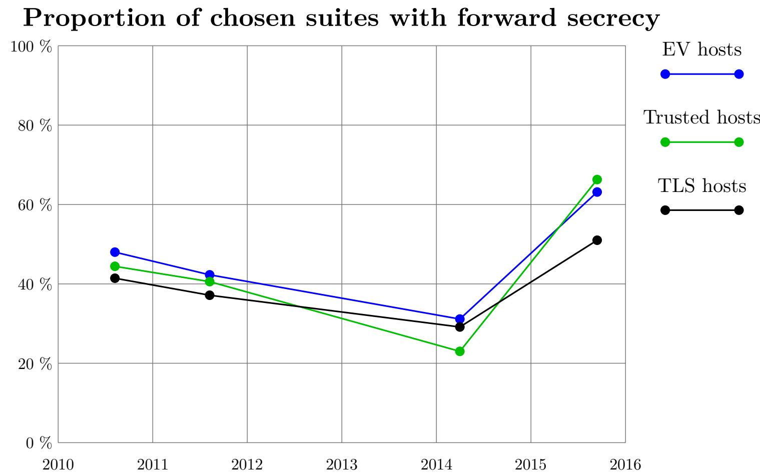

FIGURE 4 : Proportion des suites cryptographiques choisies par les serveurs entre 2010 et 2015 assurant la confidentialité persistante .

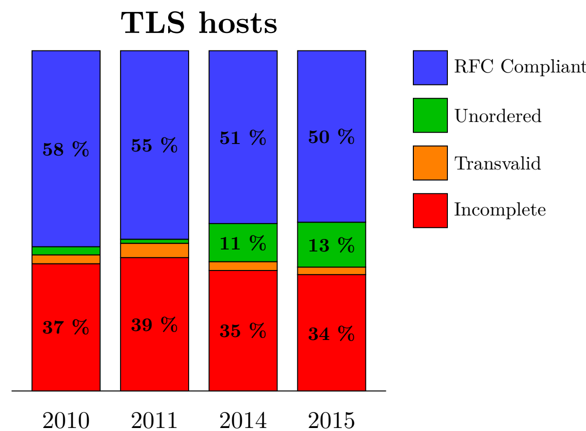

FIGURE 5 : Évolution de la qualité des chaînes de certificats envoyées entre 2010 et 2015.

l'ensemble des serveurs ayant répondu à un `ClientHello` TLS 1.0 standard, puis nous avons analysé le comportement de ces serveurs face à des stimuli plus exotiques.

En particulier, nous nous sommes intéressés à l'intolérance de ces serveurs à certaines fonctionnalités. Par exemple, en 2011, 23 % des serveurs étaient intolérants à un `ClientHello` ne proposant que des suites DHE. Cette intolérance se manifeste soit par des réponses ne contenant pas de messages TLS, soit par des `ServerHello` incompatibles avec les propositions envoyées par le client, à la place d'un message d'alerte. La situation semble s'améliorer avec le temps, puisque cette proportion tombe à 14 % en 2014. La même analyse avec un `ClientHello` ne contenant que des algorithmes mettant en œuvre les courbes elliptiques donne des résultats similaires : 18 % d'intolérance en 2011 contre 7 % en 2014.

Nous avons également mené cette analyse avec un stimulus TLS 1.2 similaire au stimulus de référence TLS 1.0 (seule la version du protocole changeait entre les deux messages). Cette fois, la proportion de serveurs intolérants était de moins d'1 %, ce qui est une bonne nouvelle pour le déploiement de TLS 1.2. Une étude similaire utilisant la version 1.3 serait utile.

6 `parsifal` : une solution pour l'écriture de *parsers* binaires

Dans le cadre de notre étude des campagnes HTTPS décrites dans les chapitres précédents, nous avons été amenés à utiliser des *parsers* (ou dissecteurs) pour analyser les messages binaires renvoyés par les serveurs contactés. L'expérience a montré qu'il fallait disposer d'outils robustes et maîtrisés pour bien comprendre les comportements d'un protocole donné, en particulier pour en caractériser les anomalies. Les implémentations disponibles sont en effet parfois limitées (refus de certaines options), laxistes (acceptation silencieuse de paramètres erronés) ou fragiles (terminaison brutale du programme pour des valeurs inattendues, qu'elles soient licites ou non). Ce constat nous a conduit au développement d'outils, l'objectif étant de développer *rapidement* des dissecteurs *robustes* et *performants*. Cette section décrit brièvement `parsifal`, une implémentation générique de *parsers* binaires reposant sur le pré-processeur `camlp4` d'OCaml, qui a servi à écrire les *parsers* utilisés dans `concerto`.

L'analyse des données collectées pose plusieurs difficultés. Tout d'abord, les fichiers à analyser représentent un volume conséquent (plusieurs centaines de giga-octets dans le cas de notre analyse de TLS). Ensuite, les informations à extraire sont contenues dans des messages de structures complexes. Enfin, il s'agit de données brutes, non filtrées, dont la qualité, voire l'innocuité, laisse parfois à désirer.

6.1 Description de `parsifal`

`parsifal` est issu des besoins identifiés et de l'expérience acquise dans l'écriture de *parsers* pour des formats binaires dans différents langages (C++, python, OCaml).

Le concept de base de `parsifal` est la définition de « types enrichis », les $\mathcal{PT}$ypes, qui sont simplement des types OCaml quelconques pour lesquels certaines fonctions sont fournies. Ainsi, un $\mathcal{PT}$ype est défini par les éléments suivants :

- un type OCaml `t` décrivant le contenu à *parser* ;
- une fonction `parse_t` pour disséquer les objets depuis une chaîne de caractères ;
- des fonctions pour exporter les objets sous forme binaire (`dump_t`) ou dans une représentation haut niveau utile aux fonctions d'affichage (`value_of_t`).

On peut distinguer trois sortes de $\mathcal{PT}$ypes. Tout d'abord, la bibliothèque standard fournit des $\mathcal{PT}$ypes de base (entiers, chaînes de caractères, listes, objets ASN.1 encodés avec la représentation DER[7], etc.). Ensuite, il est possible de construire des $\mathcal{PT}$ypes à partir de mots clés tels que `struct`, `union`, `enum` ; pour ceux-ci, une description suffit au pré-processeur pour générer automatiquement le type OCaml et les fonctions correspondantes. Enfin, dans certains cas, il est nécessaire d'écrire le type OCaml et les fonctions `parse_t`, `dump_t` et `value_of_t` à la main, pour gérer des cas particuliers. Pour illustrer les deux premiers types de $\mathcal{PT}$ypes, voici une implémentation rudimentaire des messages TLS à l'aide de `parsifal` :

[7]Comme son nom l'indique, ASN.1 *(Abstract Syntax Notation One)* est une représentation abstraite des données. Les *Distinguished Encoding Rules* (DER) en sont une représentation concrète canonique.

```
enum tls_content_type (8, Exception) =
  | 0x14 -> CT_ChangeCipherSpec      | 0x15 -> CT_Alert
  | 0x16 -> CT_Handshake             | 0x17 -> CT_ApplicationData

union record_content (Unparsed_Record) =
  | CT_Alert            -> Alert of array(2) of uint8
  | CT_Handshake        -> Handshake of binstring
  | CT_ChangeCipherSpec -> ChangeCipherSpec of uint8
  | CT_ApplicationData  -> ApplicationData of binstring

struct tls_record = {
  content_type : tls_content_type;
  record_version : uint16;
  record_content : container[uint16] of record_content (content_type)
}
```

Le dernier bloc de code décrit ce qu'est un *record* TLS, c'est-à-dire un enregistrement (décrit à l'aide du mot clé `struct`) contenant quatre champs : le type du contenu, la version du protocole, la taille du contenu et le contenu lui-même. Pour le premier champ, il existe 4 types de contenu, qui sont décrits par l'énumération `tls_content_type` (annoncée par le mot clé `enum` du premier bloc). Cette valeur est stockée sur un entier 8 bits, et si la lecture de ce champ donne une valeur non énumérée, une exception sera levée ; c'est le sens des paramètres de l'énumération (`8` et `Exception`).

La version TLS est stockée sur 16 bits : on utilise donc le $\mathcal{PT}$ype prédéfini `uint16`. Comme il existe certaines versions connues, on pourrait utiliser une énumération ici également, avec un comportement plus laxiste face aux valeurs inconnues (ajout d'un constructeur avec `UnknownVal`) :

```
enum tls_version (16, UnknownVal UnknownVersion) =
  | 0x0002 -> SSLv2        | 0x0300 -> SSLv3
  | 0x0301 -> TLSv1        | 0x0302 -> TLSv1_1
  | 0x0303 -> TLSv1_2
```

Les deux derniers champs du *tls_record* sont décrits ensemble par un conteneur dont la longueur, variable, tient sur 16 bits. Le contenu du message lui-même est décrit par le $\mathcal{PT}$ype `record_content`, qui prend un argument (`content_type`). En effet, `record_content` est une `union`, dont le contenu dépend d'un discriminant, ici le type de contenu. En l'occurrence, une alerte TLS contient deux octets, un message de type `ChangeCipherSpec` contient un octet et les messages de type `ApplicationData` contiennent des données applicatives à interpréter en fonction du protocole utilisant TLS ; pour le type restant, le contenu exact du *record* est plus complexe, et nécessiterait la définition de nouveaux $\mathcal{PT}$ypes.

6.2 Bilan de l'application de `parsifal` à divers cas d'usage

Après avoir écrit plusieurs implémentations dans différents langages (Python, C++, OCaml), nous avons développé `parsifal`, une implémentation générique de *parsers* binaires reposant sur un pré-processeur, qui répond à nos besoins : la possibilité d'exprimer des formats complexes, la rapidité d'écriture, et la production de *parsers* robustes et performants.

Au-delà de son objet initial, l'écriture de *parsers* de messages TLS et de certificats X.509, nous avons réutilisé `parsifal` pour d'autres usages depuis 2013. En voici deux exemples :

- le protocole DNS, qui met en œuvre une forme de compression spécifique, nécessite de maintenir un état lors de la dissection ou de la confection d'un paquet. DNS fut également l'occasion d'introduire les champs de bits dans `parsifal` ;
- le format de fichier PE *(Portable Executable)*, utilisé notamment dans la spécification UEFI (*Unified Extensible Firmware Interface*, le remplaçant du BIOS, *Basic Input/Output System*). Même si le support dans `parsifal` est limité, il a tout de même mis en évidence l'aspect non-linéaire de ce format, qui contient des pointeurs internes au fichier.

L'étude de ces protocoles et formats nous a permis de mieux appréhender les propriétés attendues d'un bon format binaire. Un exemple de construction facile à *parser* est l'utilisation de structures

TLV (*Tag/Length/Value*, c'est-à-dire un triplet contenant le type de données, sa longueur, et la valeur proprement dite). À l'inverse, certaines propriétés comme l'utilisation de pointeurs internes rendent la dissection (et a fortiori la validation) d'un fichier plus complexe.

Enfin, une des raisons du succès de `parsifal` au sein de notre équipe vient du fait qu'il automatise la génération de morceaux de code répétitifs et inintéressants. Cela laisse le développeur libre de s'attacher aux éléments les plus complexes du développement. Un autre aspects positif de `parsifal` est qu'il permet par construction une description progressive d'un format de fichier ou d'un protocole. Il est alors naturel de n'écrire que les éléments pertinents pour l'analyse recherchée.

7 Les défis à relever dans l'implémentation d'une pile TLS

Après avoir écrit `parsifal` pour disséquer les paquets TLS, l'étape suivante était logiquement d'écrire une pile TLS. L'implémentation rudimentaire développée autour de `parsifal` nous a permis d'appréhender la complexité de la tâche. En parallèle, de nombreuses failles d'implémentation ont été découvertes dans les piles TLS. Nous en proposons ici une analyse, en triant les vulnérabilités par catégories. Nous étudions également les axes d'amélioration qui permettraient de ne pas reproduire ces erreurs.

7.1 Erreurs de programmation classiques

Comme tout développement logiciel, une pile TLS contient des erreurs de programmation. Dans le contexte d'un protocole de sécurité, les plus simples d'entre elles peuvent mener à des failles de sécurité.

En février 2014, Apple a publié un avis de sécurité concernant son implémentation du protocole TLS : la répétition d'une instruction `goto fail` transformait la majeure partie d'une fonction en code mort (CVE-2014-1266[8]). La conséquence était catastrophique du point de vue de la sécurité du protocole, puisque le code éludé avait comme objet la vérification d'une signature cryptographique. Un attaquant pouvait alors se faire passer pour n'importe quel serveur auprès d'un client vulnérable.

Quelques jours plus tard, le projet libre GnuTLS publiait à son tour un avis de sécurité qui permettait également à un attaquant d'usurper l'identité d'un serveur (CVE-2014-0092). Le problème venait de la valeur de retour d'une fonction vérifiant les certificats X.509. Celle-ci était supposée retourner 0 si le certificat était invalide, 1 sinon. Cependant, en cas d'erreur de *parsing*, la fonction retournait un entier négatif. Ce cas n'était pas documenté, et était en pratique traité comme le cas du certificat valide.

En 2014, des vulnérabilités classiques dans la gestion mémoire ont également affecté des piles TLS. La plus médiatisée, *Heartbleed*, était un simple dépassement de tampon en lecture, menant à la divulgation par un serveur TLS de nombreuses informations ; dans le cas d'un serveur HTTPS, on pouvait par exemple obtenir les éléments d'authentification d'autres utilisateurs, mais aussi la clé privée du serveur.

Ces vulnérabilités ont fait couler beaucoup d'encre, même s'il s'agit d'erreurs de programmation extrêmement répandues. Il est donc légitime de se demander comment éviter de telles erreurs. Pour les contrer, une première réponse est d'utiliser correctement les outils disponibles pour détecter un maximum d'erreurs. Cela passe par l'activation et la prise en compte des avertissements du compilateur et par l'utilisation d'outils d'analyse statique.

Une autre piste est l'utilisation de langages de programmation plus robustes que le langage C. Par exemple, la faille affectant GnuTLS peut être vue comme la conséquence de l'utilisation d'un entier pour représenter une valeur booléenne. Cependant, un entier peut contenir plus que deux valeurs, et utiliser une valeur négative pour signaler une exception n'est pas forcément pertinent. De même, il existe des langages permettant d'automatiser la gestion de la mémoire, rendant inapplicables des classes entières de vulnérabilités ; cependant cette amélioration de la sécurité a en général un revers : il n'est plus possible de gérer de manière fine la mémoire, en particulier l'effacement en mémoire des secrets.

Enfin, un dernier axe d'amélioration est d'ajouter plus de tests dans les méthodes de développement. D'une part, il serait bon de tester les aspects négatifs de la spécification : au-delà de s'assurer que ce qui doit fonctionner fonctionne, il est nécessaire du point de vue de la sécurité de vérifier que ce qui doit échouer échoue effectivement. D'autre part, il est étonnant de retrouver les mêmes vulnérabilités dans des implémentations indépendantes, à plusieurs années d'écart ; il serait donc utile que les tests de non-régression soient partagés entre implémentations.

[8] CVE *Common Vulnerability and Exposures* est un annuaire des vulnérabilités logicielles. Les références permettent d'identifier de manière non ambiguë une faille connue.

7.2 Failles dans les *parsers*

Dans cette section, nous décrivons quelques vulnérabilités d'implémentation trouvées dans le code des *parsers*. Ces failles peuvent résulter d'une confusion dans la spécification ou d'une mauvaise interprétation lors de l'écriture du code.

Considérons un premier exemple concernant l'interprétation des chaînes de caractères dans les certificats et a été présentée par Marlinspike en 2009 [Mar09]. Les caractères nuls sont interdits dans la majorité des chaînes de caractères ASN.1. Cependant, un attaquant peut demander à faire signer un certificat pour un nom de domaine comportant un caractère nul, par exemple `www.mybank.com\0.evil.com`. Au lieu de le rejeter, la majorité des implémentations accepte cette chaîne, mais les interprétations peuvent diverger. Face à une telle requête, une autorité de certification peut ainsi considérer que le nom de la demande dépend d'`evil.com`, et solliciter une confirmation en envoyant un courrier électronique à ce domaine, contrôlé par l'attaquant. Une fois le certificat obtenu, l'attaquant peut le présenter à un client écrit en C, qui interprète naturellement le caractère nul comme la fin de la chaîne de caractères. L'attaquant peut alors usurper l'identité du site `www.mybank.com`.

Dans un autre registre, OpenSSL a récemment été l'objet d'une attaque permettant de négocier la version de TLS à la baisse. En effet, lorsqu'un serveur OpenSSL reçoit un `ClientHello` éclaté en *records* de taille inférieure à 5 octets (ce qui est autorisé par la spécification), il ne peut pas déduire la version proposée par le client, puisque ce champ est présent dans à partir du 5e octet. Plutôt que de reporter sa décision quant à la version à choisir, OpenSSL forçait alors la version à TLS 1.0. Le problème soulevé s'est révélé si compliqué à résoudre que les développeurs ont finalement choisi d'interdire une telle fragmentation du `ClientHello`.

Une autre incohérence entre implémentations a été observée dans la spécification de l'extension TLS *Encrypt-then-MAC* [Gut14]. Bien qu'il s'agisse d'une extension de sécurité, et malgré les tests d'interopérabilité entre deux implémentations avant la publication du document, lorsqu'une troisième implémentation a ajouté le support de cette extension, le code s'est révélé incompatible avec les deux précédentes. Sans porter directement atteinte à la sécurité du protocole, cet exemple montre la difficulté de spécifier de manière simple et précise des protocoles comme TLS.

La complexité de certaines spécifications comme l'ASN.1 ou l'absence de descriptions formelles comme dans le cas de TLS mènent à des ambiguïtés dans le standard. Il en résulte des différences d'interprétation entre piles TLS. Dans certains cas, un attaquant peut se servir de cette brèche pour remettre en cause la sécurité du protocole, comme le montrent les deux premiers exemples.

Afin de réduire les risques liés à ces problèmes, il est important d'améliorer les spécifications pour qu'elles ne prêtent plus à confusion. Un autre axe d'amélioration est là encore l'utilisation de tests aux limites, de préférence dans une base de tests partagée entre implémentations.

7.3 Les réels dangers de la cryptographie obsolète sur la sécurité

Dans la section 2, nous avons vu que le mode CBC dans sa construction *MAC-then-Encrypt*, était sujet à des oracles de *padding*. Comme en témoignent les différents articles théoriques, ces problèmes étaient connus dès 2002. Cependant, malgré la publication de deux versions de TLS entre 2002 et aujourd'hui, ce mode dangereux n'a jamais été remis en cause. Au lieu de proposer d'utiliser la construction *Encrypt-then-MAC* (finalement standardisée en 2014) ou d'imposer l'utilisation du chiffrement authentifié introduit avec TLS 1.2, la responsabilité de traiter le problème a été confiée aux développeurs. En effet, une note d'implémentation dans la spécification de TLS 1.1 indique que le traitement des *records* doit prendre essentiellement le même temps, que le *padding* soit correct ou non. L'esquisse d'une implémentation est même proposée.

Malheureusement, cette mesure s'est révélée incorrecte, puisqu'une attaque a été décrite à l'encontre de DTLS [PA12], suivie de l'attaque Lucky 13 [AP13]. L'attaque est même réapparue plus tard, dans une implémentation écrite pour prendre en compte la vulnérabilité [AP15].

Il existe un cas semblable dans le monde asymétrique. En 1998, Bleichenbacher a montré qu'il était possible d'exploiter un oracle de *padding* lors du déchiffrement RSA d'un message pour retrouver un message clair chiffré avec la même clé. Cette attaque, surnommée *Million Message Attack*, est décrite dans la RFC 3218 [Res02]. Ce document contient de plus trois contre-mesures possibles :

- regrouper tous les cas d'erreurs possible en un unique signal, afin que les erreurs de *padding* ne puissent être distinguées des autres erreurs ;

- lorsque c'est possible, ignorer toutes les erreurs de manière silencieuse en remplaçant le message déchiffré par une chaîne aléatoire (c'est ce qui est recommandé dans le cas de l'échange de clé par chiffrement RSA de TLS) ;
- utiliser le standard PKCS#1 v2.1 (OAEP, *Optimal Asymmetric Encryption Padding* [JK03]) en remplacement de la version 1.5 obsolète.

Concrètement, seule la dernière solution est réellement fiable. En effet, l'implémentation de la seconde contre-mesure peut être remise en cause dès qu'une différence dans le temps de traitement est observable. De même, comme pour les oracles de *padding* sur le mode CBC, l'attaque de Bleichenbacher a resurgi en 2014 dans l'implémentation JSSE [MSW+14] et en 2015 avec l'attaque DROWN [ASS+16].

On peut attendre trois propriétés d'une application mettant en oeuvre de la cryptographie :

- la sécurité vis-à-vis des attaques cryptographiques connues ;
- la compatibilité avec l'écosystème existant, parfois vieillissant ;
- la modularité du code, au sens de la réutilisabilité et de la maintenabilité.

Les attaques présentées dans cette section ont été découvertes et corrigées, puis redécouvertes et recorrigées plusieurs fois, soit dans la même implémentation, soit dans une nouvelle implémentation. Dans de nombreux cas, la raison était que pour corriger un problème, il fallait remettre en cause une de ces trois propriétés. Avec un peu de recul, il semble qu'un développeur soit en pratique obligé d'en choisir deux parmi les trois :

- combiner la modularité et la compatibilité revient à utiliser les primitives standard sans contre-mesure, au détriment de la sécurité ;
- choisir la sécurité et la compatibilité consiste à réécrire des morceaux entiers du code cryptographique pour ajouter des contre-mesures complexes, au prix de la modularité (et donc de la maintenabilité du code dans le temps) ;
- combiner la sécurité et la modularité s'obtient en faisant évoluer le standard, quitte à ne plus être compatible avec les vieux algorithmes et modes. C'est la seule solution viable dans la durée (et c'est celle qui a été retenue pour TLS 1.3).

La cryptographie est importante en sécurité, et les non-spécialistes du domaine considèrent généralement qu'il s'agit de la partie la plus sûre de l'édifice. Cette vision est généralement vraie, si on s'assure que les algorithmes et constructions obsolètes sont retirés au fur et à mesure.

7.4 Le problème des machines à état complexes

La dernière catégorie de vulnérabilités que nous décrivons concerne les machines à état des piles TLS. Depuis 2014, plusieurs attaques ont été décrites sur le sujet, mais nous ne prendrons que deux exemples de vulnérabilités exploitables par un attaquant réseau actif : *Early Finished* et FREAK [BBD+15].

La première vulnérabilité affecte certains clients qui acceptent des négociations écourtées. L'idée de l'attaque est la suivante : en réponse à un `ClientHello`, l'attaquant répond à la place du serveur. Il envoie les messages suivants : `ServerHello`, `Certificate` (avec le certificat du serveur à usurper) et `ServerFinished`, en omettant le reste de la négociation. Face à une telle série de messages, les implémentation JSSE (Java) et CyaSSL vulnérables considèrent le serveur authentifié, et transmettent les messages `ApplicationData` en clair, sans que cela soit détectable au niveau applicatif.

L'autre attaque, FREAK, a été très médiatisée. Elle repose sur la modification de messages à la volée. Celui-ci force le serveur à négocier un échange de clé faible, RSA-EXPORT, tout en faisant croire au client que l'échange de clé est le chiffrement RSA standard. La suite des messages envoyés par le serveur diffère alors de ceux attendus par le client, mais de nombreuses implémentations s'en accommodent sans lever d'alerte. L'attaquant n'a plus alors qu'à casser la clé RSA faible utilisée dans le cadre du mode export, ce qui est possible en pratique si le serveur réutilise la même clé suffisamment longtemps. Le problème soulevé par FREAK est donc triple :

- certains clients ayant négocié le chiffrement RSA standard acceptent de recevoir un message `ServerKeyExchange` contenant une clé RSA export de 512 bits, pourtant réservé à l'échange de clé RSA-EXPORT (c'est une vulnérabilité de la machine à état) ;

- de (trop) nombreux serveurs TLS acceptent de négocier des suites EXPORT (35 % des serveurs HTTPS étaient vulnérables en mars 2015[9]) ;
- parmi ces serveurs, beaucoup réutilisent la même clé RSA faible pendant toute la durée de vie du serveur, ce qui rend possible la factorisation du module RSA dans le temps imparti.

Ces vulnérabilités, affectant l'ensemble des implémentations TLS à des degrés divers, montrent que la spécification de l'automate d'état du protocole est confuse et mal comprise par les développeurs. Plutôt que de rejeter la faute sur ces derniers, une solution serait de proposer une description plus claire (voire formelle) de la machine à état du protocole TLS. Une implémentation devrait en effet savoir à tout moment quels messages elle peut accepter, au lieu de traiter tous les messages comme ils arrivent.

Dans cette section, nous avons décrit de nombreuses failles de natures différentes. Afin d'améliorer la situation, il est possible d'agir à différents niveaux de manière complémentaire :

- simplifier les spécifications et les rendre les plus limpides possibles. L'idéal serait une description formelle, complète et sans ambiguïté ;
- utiliser des langages offrant par construction des garanties de sécurité, par exemple à l'aide d'un typage statique ou en automatisant la gestion de la mémoire. Quel que soit le langage utilisé, il est important de tirer parti des outils existants (compilateurs, outils d'analyse statique) ;
- implémenter plus de tests, y compris des tests vérifiant des propriétés négatives, et partager ces test entre implémentations.

Conclusion et perspectives

TLS est aujourd'hui une brique essentielle de la sécurité des communications sur Internet. Ce protocole est l'objet de beaucoup d'attentions depuis quelques années, ce qui se traduit d'une part par la multiplication des vulnérabilités, et d'autre part par une activité importante autour de la spécification d'une nouvelle version du protocole.

Dans cette thèse, nous avons étudié le protocole sous trois aspects. Dans un premier temps, nous nous sommes intéressés à l'étude des spécifications, des failles de sécurité, et leur correction. Bien que ces failles soient corrigées dans les implémentations à jour et bien configurées, il est parfois nécessaire de pallier l'absence de mise à jour ; c'est dans cette optique que nous avons proposé des mécanismes de défense en profondeur dans le cas de HTTPS.

Dans un second temps, nous avons observé l'écosystème HTTPS au travers d'expérimentations menées entre 2010 et 2015. Pour cela, nous avons décrit une méthodologie de collecte de données TLS reposant sur l'envoi de stimuli multiples. Nous avons également développé un ensemble d'outils pour permettre une analyse reproductible des campagnes obtenues. D'après les résultats de cette analyse, la situation s'améliore et de plus en plus de serveurs proposent aujourd'hui des configurations de bonne qualité. Cependant, cette amélioration est lente, et on constate toujours une part importante de serveurs vulnérables à des attaques connues depuis des années.

Dans un dernier temps, nous nous sommes intéressés aux difficultés liées à l'implémentation d'une pile TLS. Nous avons particulièrement étudié le problème du *parsing*, une étape nécessaire pour la confection de nos outils d'analyse. Nous avons aussi proposé une analyse des failles d'implémentation des piles TLS, afin de comprendre comment améliorer la situation.

Au-delà de ces travaux, ces trois aspects offrent des perspectives pour des travaux futurs. Concernant les spécifications, la prochaine étape logique est d'étudier TLS 1.3, en poursuivant les travaux en cours de la communauté scientifique. Afin de mieux comprendre l'état des lieux en pratique de l'écosystème, de nouvelles mesures régulières pourraient être conduites, toujours en utilisant des stimuli multiples. Ces mesures pourraient être étendues à d'autres protocoles que HTTPS. Enfin, du point de vue de l'implémentation, les axes de recherche possibles concernent les langages de programmations et leurs outils associés, ainsi que la création d'une base de tests incluant des cas limites et des cas d'erreurs. De plus, ce travail sur les implémentations devrait consister en une boucle de rétroaction pour lever les ambiguïtés et identifier les points durs de la spécification.

[9]Source : `https://freakattack.com/`.

References

[3rd11] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), January 2011.

[ABP+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA*, pages 305–320, August 2013.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA*, pages 526–540, May 2013.

[AP15] Martin R. Albrecht and Kenneth G. Paterson. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. *IACR Cryptology ePrint Archive*, 2015.

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium, Austin, Texas, USA*, August 2016.

[Bar11] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

[BBD+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA*, pages 535–552, May 2015.

[BDF+14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA*, pages 98–113, May 2014.

[Bot12] Carna Botnet. Internet Census 2012: Port scanning /0 using insecure embedded devices. `http://internetcensus2012.bitbucket.org/paper.html`, 2012.

[BS13] Tal Be'ery and Amichai Shulman. A perfect CRIME? TIME will tell. *Black Hat EU*, March 2013.

[BWNH+03] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003. Obsoleted by RFC 4366.

[BWNH+06] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), April 2006. Obsoleted by RFCs 5246, 6066, updated by RFC 5746.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA*, pages 398–412, August 1999.

[Com11] Comodo. Report of Incident - Comodo detected and thwarted an intrusion on 26-MAR-2011. http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html, 2011.

[Deb08] Debian. DSA-1571-1 openssl – predictable random number generator. http://www.debian.org/security/2008/dsa-1571, 2008.

[DKBH13] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain*, pages 291–304, October 2013.

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.

[DR11] Thai Duong and Juliano Rizzo. Here come the XOR ninjas. *Ekoparty Security Conference*, September 2011.

[DWH13] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA*, pages 605–620, August 2013.

[EB10a] Peter Eckerseley and Jesse Burns. An Observatory for the SSLiverse. *Defcon 18*, August 2010.

[EB10b] Peter Eckerseley and Jesse Burns. Is the SSLiverse a safe place? *27. Chaos Communication Congress*, December 2010.

[FI12] Fox-IT. Black Tulip: Report of the investigation into the DigiNotar Certificate Authority breach, August 2012.

[FKK11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[GPvdM15] Christina Garman, Kenneth G. Paterson, and Thyla van der Merwe. Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS. In *24th USENIX Security Symposium, Washington, D.C., USA*, pages 113–128, August 2015.

[Gut14] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), September 2014.

[HBKC11] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements. In *Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference, IMC '11, Berlin, Germany*, pages 427–444, November 2011.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA*, pages 205–220, August 2012.

[Hic95] Kipp E.B. Hickman. The SSL Protocol. http://www.mozilla.org/projects/security/pki/nss/ssl/draft02.html, 1994-1995.

[IOWM13] Takanori Isobe, Toshihiro Ohigashi, Yuhei Watanabe, and Masakatu Morii. Full plaintext recovery attack on broadcast RC4. In *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore*, pages 179–202, March 2013.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[KHF14] Thomas Krenc, Oliver Hohlfeld, and Anja Feldmann. An internet census taken by an illegal botnet: a qualitative assessment of published measurements. *Computer Communication Review*, 44(3):103–111, 2014.

[Mar09] Moxie Marlinspike. More Tricks For Defeating SSL In Practice, July 2009.

[MDK14] B. Möller, T. Duong, and K. Kotowicz. Google Security Advisory: This POODLE Bites - Exploiting The SSL 3.0 Fallback. `http://www.openssl.org/~bodo/ssl-poodle.pdf`, September 2014.

[Möl04] Bodo Möller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. `http://www.openssl.org/~bodo/tls-cbc.txt`, 2002-2004.

[MSW+14] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA*, pages 733–748, August 2014.

[PA12] Kenneth G. Paterson and Nadhem J. AlFardan. Plaintext-recovery attacks against datagram TLS. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA*, February 2012.

[PHG13] Angelo Prado, Neal Harris, and Yoel Gluck. SSL, gone in 30 seconds — a BREACH beyond CRIME. *Black Hat USA*, August 2013.

[Pop15] A. Popov. Prohibiting RC4 Cipher Suites. RFC 7465 (Proposed Standard), February 2015.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece*, pages 142–159, May 2013.

[RD12] Juliano Rizzo and Thai Duong. The CRIME attack. *Ekoparty Security Conference*, September 2012.

[Res02] E. Rescorla. Preventing the Million Message Attack on Cryptographic Message Syntax. RFC 3218 (Informational), January 2002.

[Res16] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. `https://tools.ietf.org/html/draft-ietf-tls-tls13-12`, March 2016.

[Ris09] Ivan Ristic. Qualys Blog: SSL and TLS Authentication Gap vulnerability discovered. `https://blog.qualys.com/ssllabs/2009/11/05/ssl-and-tls-authentication-gap-vulnerability-discovered`, 2009.

[Ris10] Ivan Ristic. Internet SSL Survey. *Black Hat USA*, August 2010.

[Rog95] Phillip Rogaway. Problems with Proposed IP Cryptography. `http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt`, 1995.

[Vas11] Vasco. DigiNotar reports security incident, August 2011.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands*, pages 534–546, May 2002.