A study of the TLS ecosystem

Olivier Levillain

THÈSE DE DOCTORAT DE TELECOM SUDPARIS

Spécialité : **Informatique**

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Olivier LEVILLAIN

Pour obtenir le grade de

DOCTEUR de TELECOM SUDPARIS

Une étude de l'écosystème TLS

Soutenue le 23 septembre 2016 devant le jury composé de :

Rapporteurs :	Karthikeyan Bhargavan	INRIA
	Pascal Lafourcade	Université Clermont Auvergne
Examinateurs :	José Araujo	ANSSI
	Emmanuel Chailloux	Université Pierre et Marie Curie
	Aurélien Francillon	EURECOM
	Kenny Paterson	Royal Holloway, Londres
Directeur de thèse :	Hervé Debar	Télécom SudParis
Co-encadrant :	Benjamin Morin	ANSSI

N° NNT: 2016TELE0014

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.

Remerciements

Au terme de ces six années de thèse, il était plus que temps d'achever ce manuscrit et de soutenir. Avant de laisser place au sujet que j'ai étudié avec passion pendant tout ce temps, je profite de cette page pour remercier les gens qui m'ont accompagné dans cette aventure.

Tout d'abord, je remercie mes encadrants, Hervé Debar et Benjamin Morin, pour leurs conseils et leurs relectures tout au long de ce doctorat-fleuve. Je souhaite également remercier Karthik Bhargavan et Pascal Lafourcade qui m'ont fait l'honneur de rapporter cette thèse, ainsi que l'ensemble des membres du jury.

Ensuite, je voudrais dire un merci tout particulier à Loïc et Éric pour m'avoir fait découvrir le travail de chercheur à mon arrivée à la DCSSI en 2007, et pour avoir entretenu cette curiosité par de nombreuses discussions enrichissantes. Des mauvaises langues pourraient penser que ces discussions sont une des raisons de la longueur de ma thèse, mais j'en assume l'entière responsabilité.

Au fil des ans, j'ai également eu la chance d'interagir avec de nombreuses personnes, que ce soit pour rédiger des articles ou au travers de relectures internes. Parmi ces personnes, je tiens particulièrement à remercier Arno, Vincent et Baptiste.

Au-delà des collègues de travail, de nombreuses personnes ont également été présentes dans ma vie de doctorant, et il serait difficile de les nommer toutes. Je souhaite néanmoins remercier nommément Greg et Nico.

Je remercie également ma famille, qui m'a encouragé dans mes projets depuis toujours, ainsi que ma belle-mère et mon beau-frère pour leur soutien logistique infaillible. Enfin, j'adresse un grand merci à Céline qui m'a soutenu et supporté, surtout pendant cette dernière année de rédaction, malgré mes ronchonnades régulières. Et il ne faut pas oublier le p'tit Lu, qui a su me rappeler les véritables priorités dans la vie.

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.

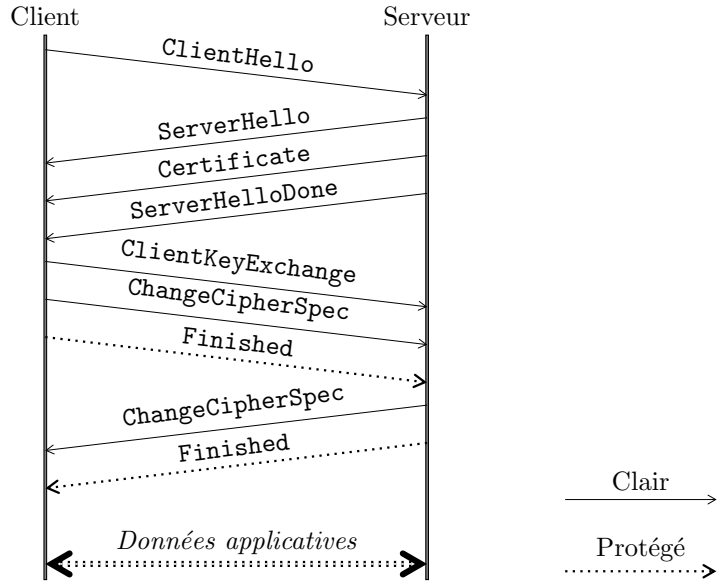


FIGURE 1 : Exemple de négociation TLS avec l'échange de clé par chiffrement RSA.

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* [Bar11b], 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 020202 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.

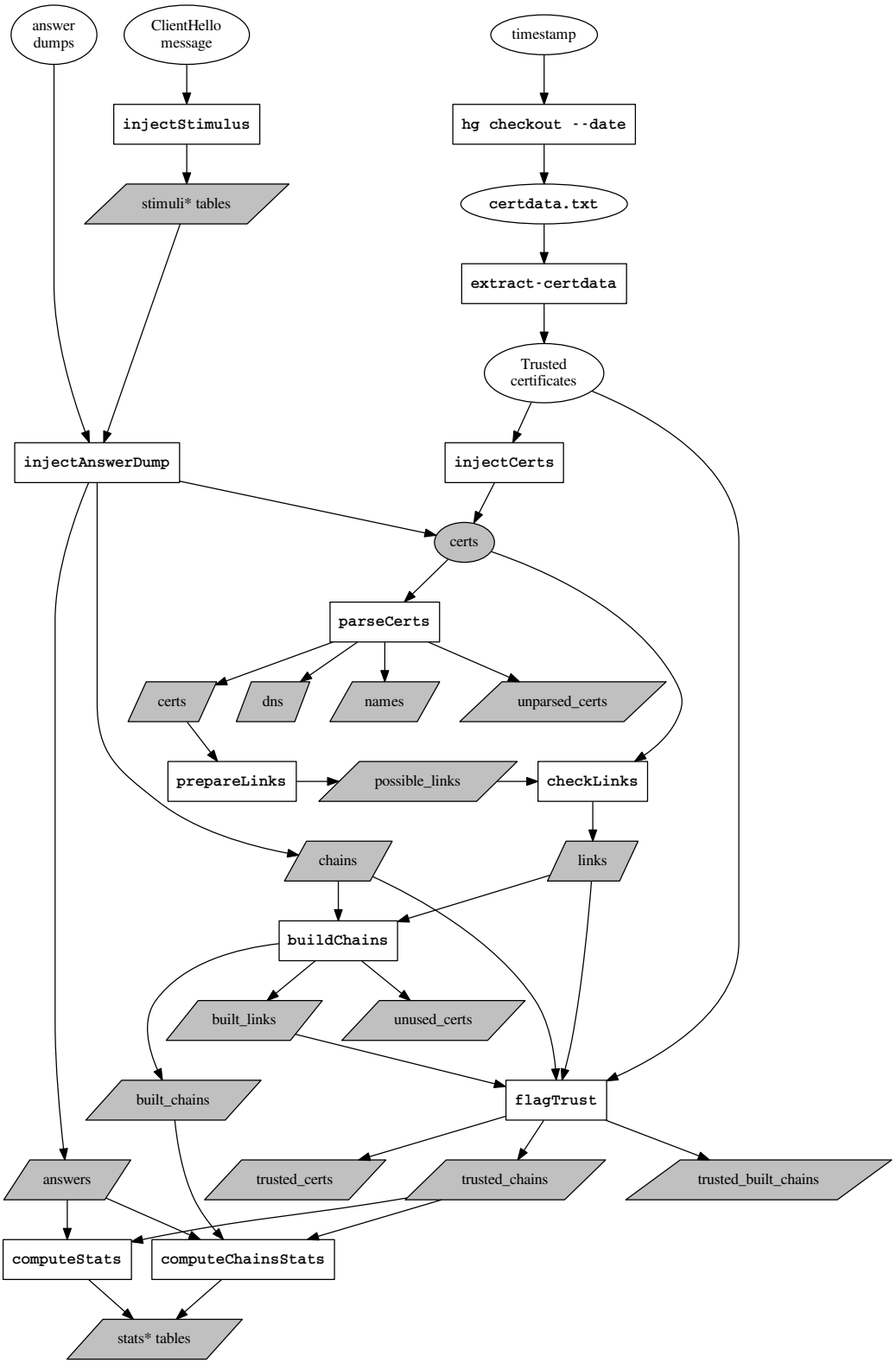


FIGURE 2 : Cinématique complète de `concerto` sur une campagne.

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

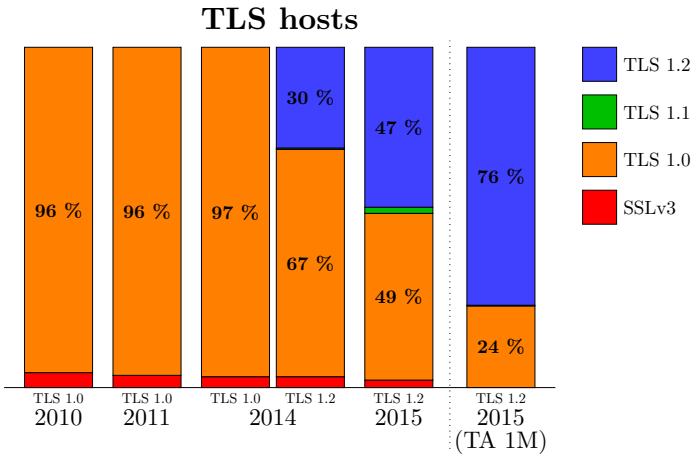


FIGURE 3 : Évolution de la version du protocole choisie par les serveurs entre 2010 et 2015. Les cinq premières colonnes correspondent à des campagnes sur tout l'espace IPv4, alors que la dernière colonne concerne le sous-ensemble constitué des serveurs du Top Alexa 1 Million.

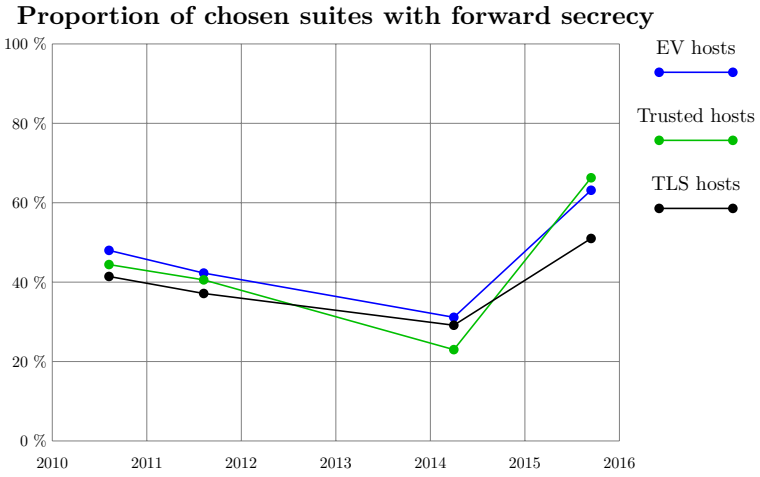


FIGURE 4 : Proportion des suites cryptographiques choisies par les serveurs entre 2010 et 2015 assurant la confidentialité persistante .

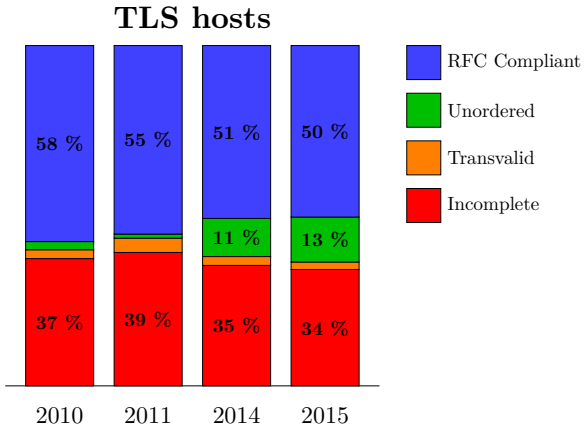


FIGURE 5 : Évolution de la qualité des chaînes de certificats envoyées entre 2010 et 2015.

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 5^{e} 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/`.

Contents

Introduction

SSL (Secure Sockets Layer) is a cryptographic protocol designed by Netscape in 1995 to protect the confidentiality and the integrity of HTTP connections. Since 2001, the protocol has been maintained by the IETF (Internet Engineering Task Force) and renamed TLS (Transport Layer Security). The current version of the protocol is TLS 1.2 [DR08], which was published in 2008. The original objective of SSL/TLS was to secure HTTP online-shopping and banking web sites.

With the deployment of web services using the so-called Web 2.0, its use has broadened drastically. Services provided by actors like Google, Yahoo!, Facebook or Twitter now offer a secure access using TLS, and many of them have switched from HTTP to HTTPS as the default connection mode. Other protocols such as SMTP or IMAP use TLS to protect the traffic. There also exists several VPN (Virtual Private Network) implementations relying on SSL/TLS. Finally, Wifi access infrastructures can use TLS as an authentication protocol (EAP-TLS).

SSL/TLS has thus become a critical building block to secure communications on the Internet. As such, the protocol has attracted the attention of many security researchers since 2010. This activity has been spanning across different aspects of TLS security.

SSL/TLS specifications have been analysed from the cryptographical angle to assess the robustness of the security guarantees advertised by the protocol. These security goals can be summarised as follows:

- an authenticated key exchange between a client and a server. It is worth noting that, in the majority of the use cases, only the server is authenticated;
- a secure channel using symmetric cryptography to assure the confidentiality and the integrity of the exchanged application data.

Theoretically, both these problems have been solved by the cryptographic community a long time ago. However, due to more than 20 years of protocol history, and because of compatibility requirements between implementations, SSL/TLS still relies on old cryptographic constructions and primitives:

- weak algorithms such as RC4, for which statistical biases have been known since 2000 [FM00];
- fragile modes of operation such as CBC (Cipher Block Chaining) using the MAC-then-Encrypt paradigm or PKCS# v1.5 RSA encryption scheme. Both modes have been studied, and proven hard to implement correctly, respectively since 2001 [Kra01, Vau02] and 1998 [Ble98];
- broken high-level constructions that lack expected security properties. Some of them, such as the insecure SSLv2 negotiation protocol was quickly fixed with SSLv3. Other features, like renegotiation or session resumption, were however left vulnerable until 2009 [Ris09] and 2014 [BDF+14].

In addition to these flaws, the original SSL specifications also describe so-called export modes, designed to comply with various national cryptographic export rules. Such modes explicitly weaken the protocol security by reducing the length of the cryptographic keys.

Most of these problems have been taken into account in the latest version of the protocol, TLS 1.2. But this requires TLS stacks to include all the relevant extensions, and more importantly, to be configured to only accept robust configurations. This fact naturally leads to another area of research regarding SSL/TLS: the experimental observation of the ecosystem.

To assess the state of the SSLiverse[10], several studies have been carried out to probe the deployed servers on the Internet. The first public research effort dates back 2010 [EB10a, EB10b]. It focused

[10]This term was invented by the EFF (Electronic Frontier Foundation) to describe their work on the subject.

on the certificate chains presented by HTTPS servers. To this aim, the researchers scanned the whole IPv4 address space to mount SSL/TLS sessions with each host with an open 443/tcp port.

In 2011, new results were published on the HTTPS ecosystem [HBKC11], including an analysis on the impact of the connection source on the gathered certificates. Since 2013, more measurement campaigns have been led [DWH13, DKBH13], including scans targeting electronic mail protocols [DAM+15, MZSH15].

Beyond the efforts to collect SSL/TLS data and analyse the presented certificates, several studies also analysed the quality of other SSL/TLS parameters, such as the selected protocol versions, the chosen cryptographic algorithms or the supported features. Such observations give insight into the real state of the protocol security on the Internet, showing that security extensions are not always deployed on the servers.

Probing actual SSL/TLS servers not only allows researchers to know the real security of the Internet, it can also help developers to understand the proportion of servers that still impose to maintain obsolete features. To be able to safely remove such features from a software (e.g. a browser), concrete data is indeed required to measure how much of the Internet will break.

The third aspect of SSL/TLS that has been studied for several years, alongside the protocol specifications and its concrete deployment, is the quality of SSL/TLS implementations. As early as in 2002, researchers have discovered exploitable bugs related to X.509 certificate parsing [Mar02]. Another typical class of security vulnerabilities affecting SSL/TLS software is memory management errors, such as buffer overflows or use-after-free issues.

Even though security flaws have long been found and published on SSL/TLS stacks, their number has increased drastically since 2014. This year has indeed been a difficult year for SSL/TLS implementations, since every widespread stack was affected by at least one critical vulnerability:

- a buffer overflow, branded Heartbleed, allowed an attacker to retrieve sensitive information from the memory of a process using **OpenSSL**;
- another buffer overflow led to remote code execution in the server-side SSL/TLS implementation from Microsoft, **SChannel**;
- a redundant `goto` statement led to a server authentication bypass in Apple's implementation of the protocol, **SecureTransport**;
- the combination of a parsing bug and an integer overflow allowed for universal signature forgery in Mozilla's **NSS** (as well as other software such as CyaSSL and PolarSSL);
- an error in the certificate parsing code in **GnuTLS** meant an attacker could impersonate any server to a vulnerable client.

In this thesis, we explore these three aspects of TLS security. In the first part, we study the protocol specifications. Chapter 1 describes its history and its features, and we present therein a state of the art of the vulnerabilities and countermeasures. We then focus on the Record Protocol, the layer in charge of data encryption and integrity protection. Chapter 2 analyses the corresponding security flaws. and presents defense-in-depth mechanisms to thwart this class of attacks in the HTTPS context.

The second part consists in a series of experimentations led to better understand the HTTPS ecosystem from field observations. Chapter 3 presents the different methods available to collect data in practice. It then focuses on our methodology, based on the exhaustive enumeration of the IPv4 address space. In chapter 4, we then describe the tools we developed to study TLS datasets. In particular, we insist on the reproducible and open characters of our methodology: our goal is to analyse datasets from different sources and to be able to replay these analyses in the future with different parameters or algorithms. Chapter 5 finally presents the results obtained with our toolsets, both on our campaigns and on public datasets.

The third part of our work regarding TLS deals with SSL/TLS stack implementations. Chapter 6 is dedicated to the parsing problem, applied to binary formats like TLS messages or X.509 certificates. We describe `parsifal`, a generic framework we developed to quickly write robust and efficient parsers. The tools used in the second part have been written with `parsifal`. Beyond the parsing aspects of the implementation, chapter 7 presents an in-depth analysis of TLS implementation flaws. It also discusses ideas to improve durably security of these critical software blocks.

Part I

State of the art of SSL/TLS and focus on the Record Protocol

This part is mostly a state of the art of the SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocol security. Chapter 1 describes the protocol history, how it works. It also lists the known attacks and the associated countermeasures. Chapter 2 then focuses on the Record Protocol. It details recent attacks and proposes generic defense-in-depth countermeasures.

Chapter 1

Presentation of the Transport Layer Security Protocol

This chapter is mostly based on two articles presented at the SSTIC conference; they describe a wide state of the art of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols [Lev12, Lev15].

SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are security protocols whose main security goal is to provide *confidentiality* and *integrity* between two communicating applications. As such, SSL/TLS relies on a reliable transport protocol (usually TCP), and acts itself as a transport protocol for the upper-layer application protocol, e.g. HTTP (see figure 1.1). To protect the application data exchanged between the two endpoints, SSL/TLS needs to authenticate the server[1] and to provide a key exchange mechanism.

The original objective of SSL/TLS was to secure online-shopping and banking web sites. With the deployment of web services using the so-called Web 2.0, its use has broadened drastically. Services provided by actors like Google, Amazon, Facebook or Twitter now all offer a secure access using TLS, and many of them have switched from HTTP to HTTPS as the default (and sometimes only) connection mode. Other protocols such as SMTP or IMAP use TLS to protect the traffic, even offering the ability to negotiate TLS over an existing cleartext connection with STARTTLS [New99]. There also exists several VPN (Virtual Private Network) implementations relying on SSL/TLS. Finally, Wifi access infrastructures may use TLS as an authentication protocol (EAP-TLS).

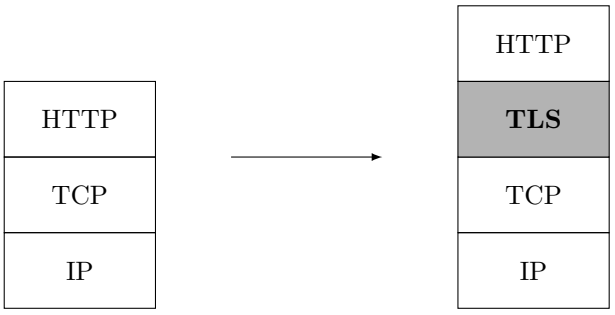


Figure 1.1: SSL/TLS as an intermediary transport layer.

To this aim, an SSL/TLS connection typically consists of two consecutive phases, shown in figure 1.2:

1. the negotiation phase, in which the parties agree on the cryptographic algorithms and keys, and where the server generally authenticates itself to the client using certificates;

[1] Strictly speaking, server authentication is not mandatory in SSL/TLS since several ciphersuites offer so-called *anonymous* Diffie-Hellman key exchanges. A legitimate use for such suites may be *opportunistic encryption*, in case where the fallback strategy is plaintext communication, e.g. between SMTP Mail Transport Agents.

2. the application exchange phase, where application data are sent and received via messages protected in confidentiality and integrity.

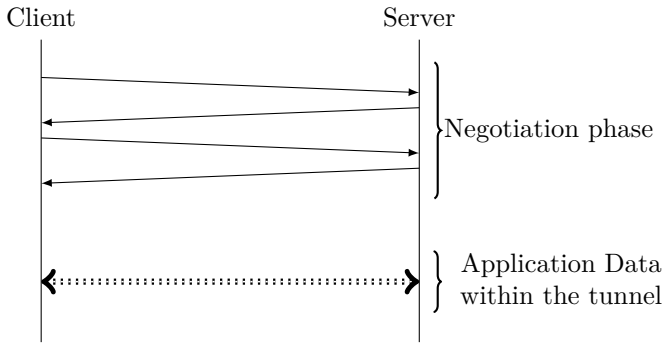


Figure 1.2: A typical SSL/TLS connection.

Apart from the historical section on SSLv2 and the final section on TLS 1.3, this chapter applies to SSLv3, TLS 1.0, TLS 1.1 and TLS 1.2. As described in section 1.2.1, SSLv2 uses a different message format. The TLS 1.3 standard is still under development and should follow a different logic (see section 1.6 for further details). Moreover, most of the chapter applies to DTLS, *Datagram* TLS, a variant of the protocol compatible with an unreliable transport layer (typically UDP), versions 1.0 [RM06] and 1.2 [RM12].

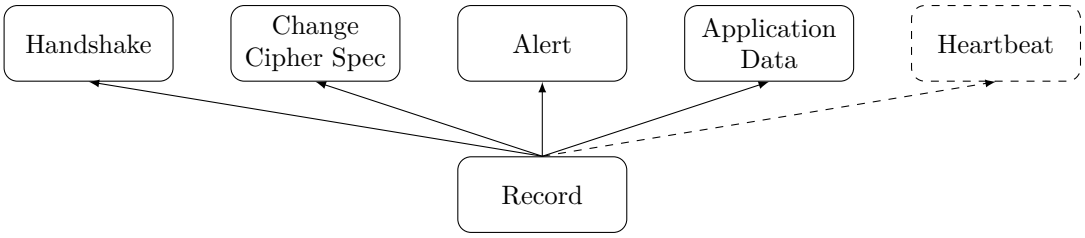


Figure 1.3: SSL/TLS and its sub-protocols.

SSL/TLS relies on several sub-protocols, which are presented on figure 1.3:

- the *Record Protocol*, which segments data in chunks, called records. Initially, records are sent and received in plaintext; then, when cryptographic algorithms and keys have been agreed upon, messages are encrypted and authenticated at this layer. Every SSL/TLS message is encapsulated in records.
- the *Handshake Protocol* is one of the sub-protocol transmitted via records. Its main purpose is to negotiate the cryptographic algorithms and keys, and to authenticate the parties (usually, the server only).
- the *ChangeCipherSpec Protocol* is a trivial protocol with only one message, whose goal is to trigger the use of the negotiated parameters.
- the *Alert Protocol* is intended to signal fatal errors or warnings, at any moment in the connection.
- *Application Data* is a layer containing the application data, which are the purpose of a SSL/TLS connection.
- the *Heartbeat Protocol* is a more recent sub-protocol added on top of the record layer [STW12], which has been brought to light in 2014 with the OpenSSL Heartbleed bug 7.1.4.

It is important to remember that, after a successful handshake, all subsequent messages take advantage of the negotiated security protection, since every sub-protocol (including subsequent handshakes or alerts) is sent via records.

Section 1.1 details how the Handshake Protocol works. Then, section 1.2 presents a brief history of SSL/TLS, from SSLv2 to TLS 1.2 and section 1.3 discusses how TLS interacts with application protocols. Section 1.4 presents the considered attacker models and section 1.5 proposes a structured list of the security flaws affecting SSL/TLS. Section 1.6 is a study of TLS 1.3, as specified in the `draft-12` snapshot [Res16], since the new protocol version is still a work in progress.

1.1 The Handshake Protocol

1.1.1 A typical SSL/TLS session establishment

To establish a secure session between a client and a server, SSL/TLS uses Handshake messages to negotiate its parameters (essentially the protocol version and the cryptographic algorithms) and to establish a shared secret, called the *master secret*, from which symmetric session keys will be derived. Moreover, the server (and optionally the client) authentication is achieved during the negotiation phase.

The algorithms are described by so-called *ciphersuites*, and are registered by the IANA [IAN15]. Theses ciphersuites define how to:

- authenticate the server;
- establish a shared secret used to derive subsequent keys;
- encrypt the application data;
- ensure the integrity of the application data.

Client authentication is also possible with TLS, but the algorithms used to authenticate the client are negotiated independently in the `CertificateRequest` message.

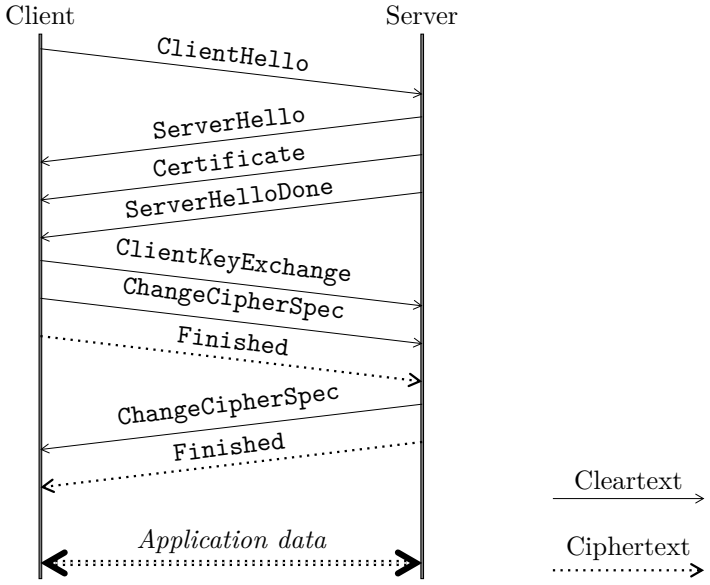


Figure 1.4: A typical SSL/TLS connection: a complete handshake using RSA key exchange, followed by application data exchanges.

Figure 1.4 presents a typical handshake between a client and a server. First, the client contacts the server over TCP and proposes several versions and ciphersuites; this initial message, `ClientHello`, also contains a nonce (`client_random`). If the server finds an acceptable ciphersuite, it responds with several messages: `ServerHello`, containing the selected version and ciphersuite, a fresh nonce

(`server_random`), the `Certificate` message, containing the chain of certificates for the contacted site, and an empty `ServerHelloDone` message ending the server answer. For the sake of simplicity, the negotiation presented here uses RSA encryption as key exchange algorithm (other mechanisms exist and are presented in the following paragraphs). Thus, the chosen ciphersuite could for example be `TLS_RSA_WITH_AES_256_CBC_SHA`[2].

Then, the client checks the certificates received and sends a `ClientKeyExchange` message, carrying a random value (the *pre-master secret*) encrypted with the public key of the server. At this point, the client and the server share this secret value, since the server can decrypt the `ClientKeyExchange` message. Combining both public random `client_random` and `server_random` with the *pre-master secret*, they can derive the *master secret* and the traffic keys, as shown in figure 1.5.

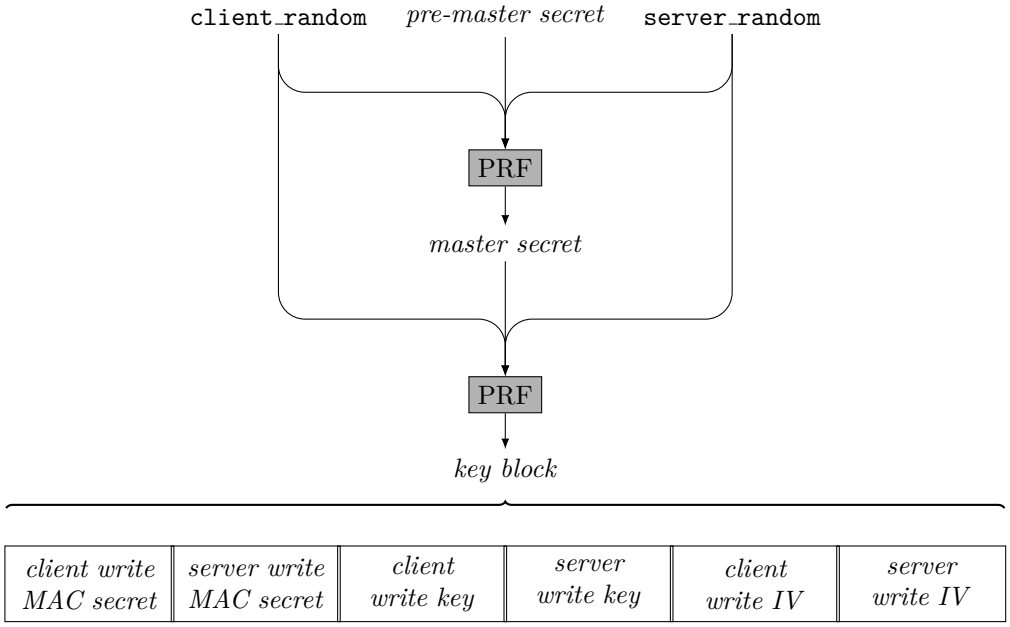


Figure 1.5: SSL/TLS key derivation scheme. The exact pseudo-random function (PRF) and the derived key material are a function of the protocol version and the ciphersuite (e.g. IVs are only derived for CBC mode in SSLv3 and TLS 1.0).

Finally, the `ChangeCipherSpec` messages activate the negotiated suite and keys, and the `Finished` messages ensure the integrity of the handshake *a posteriori*, as they contain a hash of all the Handshake messages previously exchanged (sometimes called the transcript). `Finished` messages are the first ones protected with the algorithms and keys negotiated.

At any moment, an `Alert` message can be sent to signal a problem, for example if no ciphersuite is acceptable, or if the client does not validate the certificate sent by the server.

The certificates used in TLS follow the X.509 standard [CSF+08]. The TLS Public Key Infrastructure is based on several root authorities trusted by default by clients like web browsers.

1.1.2 Alternate message flows

Multiple alternate handshakes are possible. In this section, we describe the most common ones, i.e. those supported by the main SSL/TLS implementations.

Ephemeral Diffie-Hellman and the forward secrecy

One well-known drawback of the handshake previously presented is the use of RSA encryption as a key-exchange protocol: the session secret is protected using the server long-term secret (its private RSA

[2]It is worth noting that OpenSSL, a very common SSL/TLS stack, does not conform to the standardised names and would in particular call this suite `AES256-SHA`.

key). If this long-term secret is compromised in the future, an attacker could decrypt previously recorded connections. To avoid this situation and obtain the *forward secrecy* property, a classical solution is to use an ephemeral Diffie-Hellman key exchange to establish a shared *pre-master secret*.

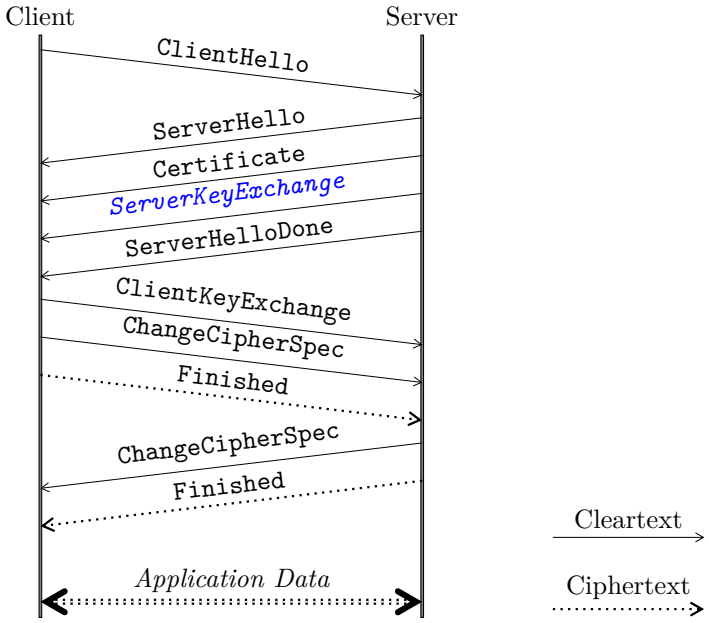


Figure 1.6: A SSL/TLS connection using (EC)DHE key exchange.

Such handshakes can either rely on DHE (Ephemeral Diffie-Hellman on finite fields) or ECDHE (Ephemeral Diffie-Hellman on elliptic curves [BWBG+06]) ciphersuites. Once such a suite has been negotiated, the server must send a new message, called `ServerKeyExchange`, after the `Certificate` message, as shown in figure 1.6. This message contains the Diffie-Hellman parameters, and is signed by the server. The signature covers not only the Diffie-Hellman parameters, but also the `client_random` and `server_random` so an attacker can not replay it[3].

Client authentication

In a typical SSL/TLS session, e.g. using HTTPS or IMAPS, the server is authenticated during the handshake. After the certificate has been presented to the client, the authentication can either be implicit — in the RSA key exchange case, where only the owner of the corresponding private key can successfully terminate the handshake — or explicit — in the (EC)DHE case the private key is used to sign the `ServerKeyExchange`.

On the other side, either the client is anonymous, or they use a login and a password at the application level. However, SSL/TLS also offers a way for the client to authenticate itself using certificates at the SSL/TLS level. The server can request a certificate from the client, via the `CertificateRequest` message, sent just before the `ServerHelloDone`. Then, if the client is able to (and accepts to) do so, it sends a `Certificate` message and a `CertificateVerify` message, respectively before and after the `ClientKeyExchange` message. The first one contains the client certificate whereas the second one contains a hash of all previously exchanged Handshake messages (including in particular the `server_random` field), signed with the client private key. The message flow is represented in the second negotiation of figure 1.8.

Anonymous key exchange

When active attackers are not considered a threat, it is acceptable to establish a session without authenticating the client nor the server: the key exchange then used is an anonymous Diffie-Hellman on a

[3]However, as several flaws in the protocol presented in section 1.5.1 (e.g. cross-protocol attacks [MVVP12] or LogJam [ABD+15]) have shown, this is not always sufficient, and the whole transcript should be signed by the server.

finite field (ADH ciphersuites) or an elliptic curve (AECDH suites). Even if this behaviour is discouraged in most TLS use cases, anonymous key exchange (sometimes called *opportunistic encryption*) is common within SMTP connections, where most servers tolerate to fallback to a plaintext connection in case of errors. Anonymous Diffie-Hellman is similar to the DHE key exchange, where the `Certificate` message has been removed.

Session resumption

Once a SSL/TLS session has been established, it is possible for the client and the server to remember the corresponding *master secret* after the connection is closed. Thus, in a subsequent connection, the client can signal in the `ClientHello` that he remembers the secret; if the server accepts the resumption offer, the connection can be resumed directly without triggering a new key exchange. In this manner, the connection is quickly established, and does not require public key cryptography operations. This scenario is described in figure 1.7.

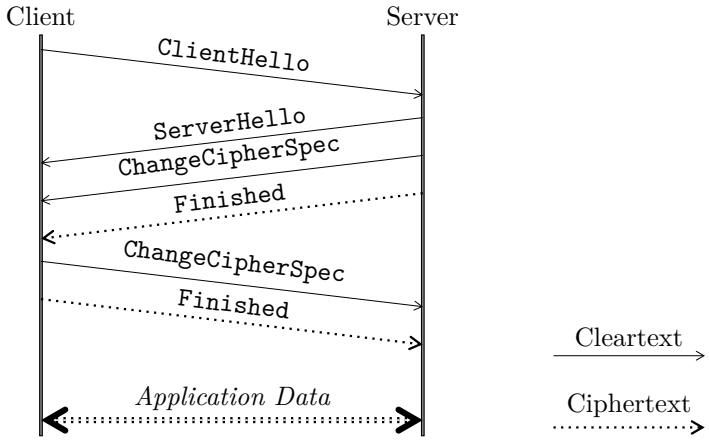


Figure 1.7: Session resumption.

In practice, there are two ways to handle resumption. First, the historical mechanism is a simple *session identifier*, sent by the server inside the `ServerHello` to tell the client to store the session *master secret* along with the identifier. Yet, with this mechanism, the server also requires to remember session *master secrets*. The modern way to handle resumption is via *session tickets*, which do not require server-side state, described in section 1.2.4.

Renegotiation

Another feature of TLS is to allow the renegotiation of session parameters in the middle of a connection. After having established a first session, the client can send a `ClientHello` to trigger a renegotiation: all the Handshake messages corresponding to this second session establishment will be protected using the cryptographic parameters negotiated during the first handshake. The message flow is described in figure 1.8.

Alternatively, the server can trigger the renegotiation by sending a `HelloRequest` message. In either case, the renegotiation can be refused with a warning-level alert. It is possible to combine renegotiation and session resumption, for example to refresh the symmetric keys without requiring asymmetric cryptographic operations.

Two typical legitimate usages of renegotiation are the following:

- key refresh: since the renegotiation introduces new `client_random` and `server_random` values, even if the current session is resumed, the resulting keys will be different;
- late client authentication: after the client has established an anonymous connection to an authenticated server, she requests access to privileged information. The server then needs the client to authenticate, and asks for a renegotiation using a `HelloRequest` message. The new handshake

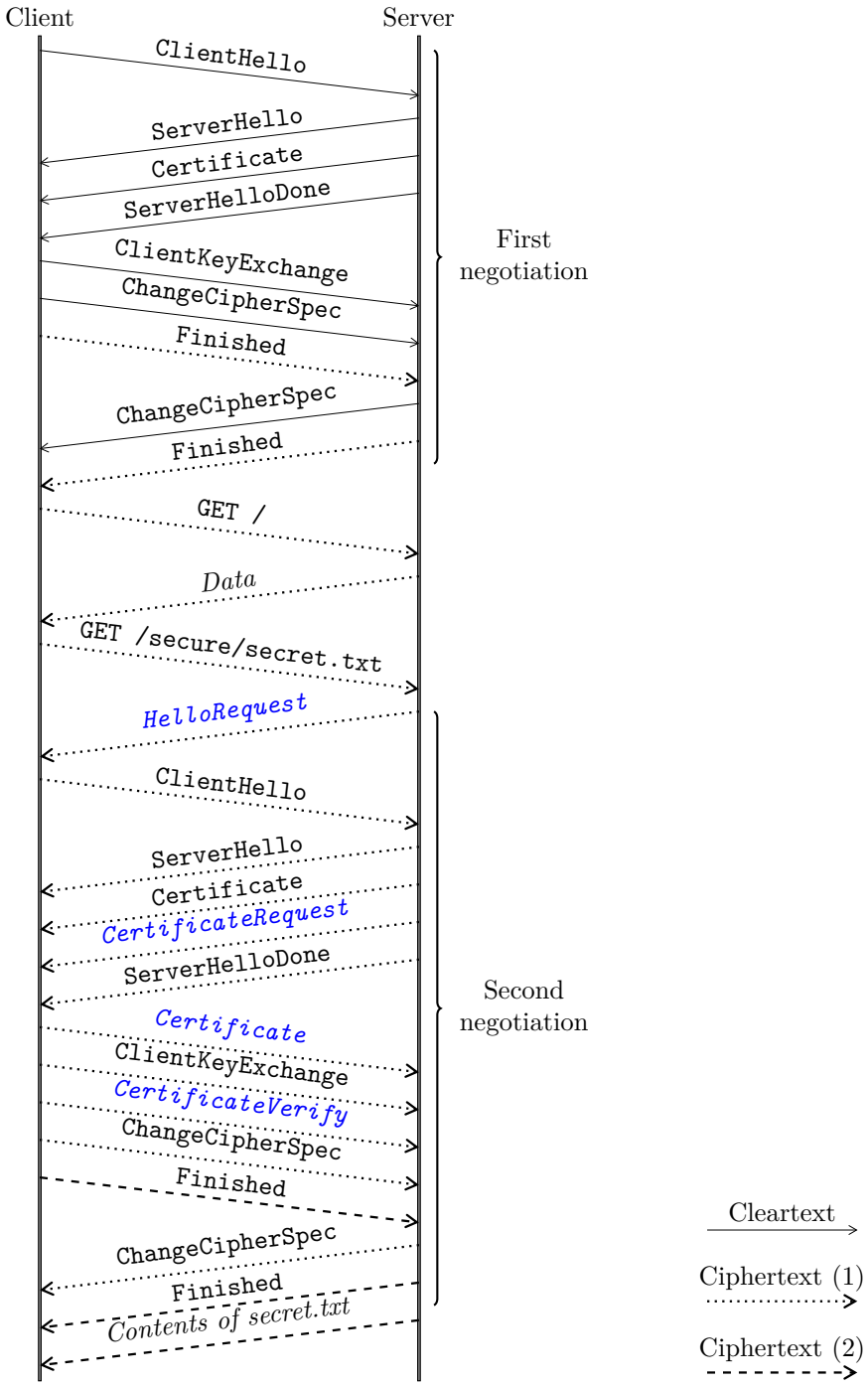


Figure 1.8: Example of a TLS renegotiation. In the first negotiation, the client is unauthenticated and the server accepts to serve public data. The second request concerns a protected document, so the server triggers a renegotiation and requires the client to authenticate herself.

includes a `CertificateRequest` message this time. This mechanism is sometimes used in HTTPS server configuration, where only a fraction of the resources are protected by a certificate client authentication. This is the example illustrated in figure 1.8.

There used to be another use-case of renegotiation, related to American cryptographic export rules in place until the early 2000's: all SSL connections between one peer in the USA and the other outside the USA were required to use export-grade ciphersuites (suites whose key lengths were restricted). To comply with this requirement, all connections with an American peer negotiated export-grade suites; yet, even before these export regulations were lifted, there was an exception allowing the use of strong cryptography for certain financial transactions: the server would signal this ability in its certificate, so the client could trigger a renegotiation to improve the cryptography used. This mechanism was called "International Step-Up" by Netscape and a somewhat similar approach was proposed as "Server Gated Cryptography" (SGC) by Microsoft.

It is worth noting that renegotiations allow to change *all* the parameters negotiated during the handshake, including cryptographic algorithms, compression algorithms, client and server identities, which is rarely intended. The renegotiation mechanism was indeed proven flawed, as shown in section 1.5.1.

Alternate trust models

The standard for TLS peers to authenticate is the use of X.509 certificates, that can be chained up to a trusted root anchor. Yet, other models have been proposed. For example, there exists obsolete ciphersuites that allowed the use of Kerberos (`TLS_KRB5_*` suites [MH99]). Another example is the use of shared symmetric secrets between the client and the server, which can be implemented with `PSK` [ET05] or `SRP` [TWMP07] ciphersuites.

It is also possible to replace the X.509 usual Public Key Infrastructure with other types of certificates: OpenPGP keys [Mav07, MG11], or X.509 certificates (or even raw public keys) signed using DNSSEC records (DNS-Based Authentication of Named Entities, DANE [HS12]).

These models usually rely on `Hello` extensions described in section 1.2.4.

Finally, it is worth noting that in some contexts, client authentication does not rely on a TLS mechanism, but rather uses the server-authenticated channel to transmit an application-level authentication, e.g. a simple login/password scheme as frequently encountered in HTTPS or IMAPS connections.

1.2 A history of versions and features

SSL (Secure Sockets Layer) is a protocol originally developed by Netscape in 1994 to secure HTTP connections using a new scheme, `https://`. The first published version was SSLv2 [Hic95], rapidly followed by SSLv3 [FKK11], which fixed major conceptual flaws. Even if a compatibility mode was specified, SSLv2 and SSLv3 use different message formats.

In 1999, the evolution and the maintenance of the protocol were handed to the IETF (Internet Engineering Task Force) which renamed it TLS (Transport Layer Security). TLS 1.0 [DA99] can be seen as a minor editorial update of SSLv3 (actually, the TLS 1.0 protocol version is encoded as 3.1). TLS 1.1 was published in 2006 [DR06] and TLS 1.2 in 2008 [DR08]. The next version of the protocol, TLS 1.3, is under specification within the IETF TLS working group, and is discussed separately in section 1.6. Today, SSLv2 and SSLv3 should not be used anymore, and TLS versions 1.1 and 1.2 should be preferred.

1.2.1 SSLv2

The initial motivation behind SSLv2 was for Netscape to allow Navigator users to establish secure connections between them and some web sites, to guarantee that the data exchanged remains secret. This was in particular required to establish trust in e-commerce sites. That is why the goal of SSLv2 was to establish a secure channel with an authenticated server (leaving client authentication as optional). The first public version of SSLv2 was released in 1994 [Hic95]. There is no public record of an SSLv1 version.

Even if SSLv2 messages differ from the message flows described earlier, the main concepts were already introduced: ciphersuites[4] describing the algorithms, most of the Handshake messages, the concept of session resumption and the authentication asymmetry (server authentication was mandatory, whereas client authentication was optional). Yet, there are also key differences between SSLv2 and its successors, amongst which the absence of a separate Record layer, which forces Handshake messages to only happen once per connection (at the start), making renegotiation impossible. A typical SSLv2 connection is described in figure 1.9.

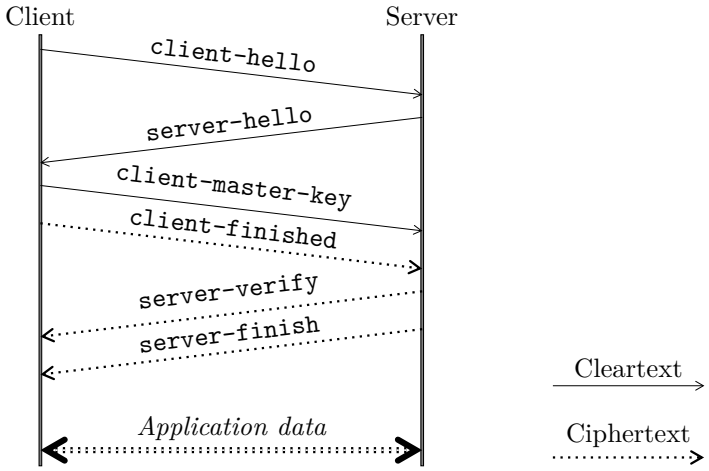


Figure 1.9: Example of an SSLv2 typical connection.

As shown in section 1.5.1, SSLv2 contains many structural flaws, which quickly led to a new version of the protocol, SSLv3. However, SSLv2 has been supported by many implementations for at least a decade after its deprecation, leading to DROWN attack (described in section 7.3.2) in 2016. Finally, it was recently formally declared obsolete by an RFC initially drafted *SSLv2-die-die* [TP11].

1.2.2 SSLv3

Soon after the release of SSLv2, SSLv3 was published to fix the structural flaws of its predecessor. Even if the basic ideas were kept, it was a complete rewrite with new messages (all messages are encapsulated inside records with a specific content type) and new features (renegotiation, compression, anonymous suites, the ability to send a certificate chain instead of just one certificate).

A typical SSLv3 connection was presented in figure 1.4. SSLv3 also introduced alternative key exchange and authentication algorithms: SSLv2 only used RSA encryption, but SSLv3 introduced Diffie-Hellman for key exchange and DSA (Digital Signature Algorithm) to authenticate the server.

As an historical side note, SSLv3 was not the only proposal to replace SSLv2: Microsoft published PCT in 1995 and STLP in 1996, improving respectively SSLv2 and SSLv3. These versions were never widely used besides Microsoft products.

Following the POODLE attack, described in details in section 2.1.6, SSLv3 was officially declared obsolete [BTPL15].

1.2.3 TLS 1.0

When SSL was standardised by the IETF as TLS 1.0 in 1999, changes made to the protocol were essentially minor:

- change the pseudo random function to use an HMAC-based construction instead of an ad-hoc construction;

[4] As a side note, SSLv2 ciphersuites are encoded on 3 bytes, whereas the more recent versions of the protocol use only 2 bytes.

- use a standard CBC padding, which made TLS 1.0 less vulnerable to padding oracle, as POODLE showed (see section 2.1.6);
- an implementation note about Bleichenbacher's attack on PKCS#1 v1.5 encryption scheme (see section 1.5.1).
- to avoid TLS dependence on RSA, a then-patented algorithm, the Network Working Group decided to promote `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`, where key exchange is done using Diffie-Hellman and the server is authenticated with DSA, as the *mandatory-to-implement* ciphersuite[5].

1.2.4 TLS Extensions

In the description of `ClientHello` messages, TLS 1.0 contained a *forward compatibility note* (RFC 2246, section 7.4.1.2) stating that extra data at the end of the message must be ignored. This extra room was indeed used in 2003 to define TLS extensions [BWNH+03][6].

The document first describes how extensions should be added to `ClientHello` and `ServerHello` messages. It then specifies several extensions:

- *Server Name Indication* (SNI), an extension allowing a client to signal the hostname very early, letting the server know which certificate to present;
- *Certificate Status Request*, also known as *OCSP Stapling*, allows the server to provide an OCSP response to the client, without requiring a separate connection;
- four less used extensions, including *Truncated HMAC* that was shown to be detrimental to security [PRS11].

Another popular extension was defined in 2006 to allow session resumption without Server-Side State [SZET06], sometimes simply called *Session Ticket*[7]: when a client announces the support of this mechanism, the server can send the session *master secret* encrypted and integrity-protected by a key known only to the server. This way, when the client wishes to resume the session, the ticket is returned to the server, which can extract the *master secret*.

Published in 2012, *Heartbeat* is one of TLS most famous extensions, since it was subject to a severe security vulnerability, dubbed Heartbleed (see section 7.1.4), in OpenSSL implementation. Heartbeat allows for Path MTU discovery and provides a framework to send *keep-alive* messages in TLS and DTLS.

Furthermore, several extensions have been proposed to add new cryptographic algorithms and modes, e.g. the Encrypt-then-Mac paradigm [Gut14].

1.2.5 TLS 1.1

TLS 1.1 improves the protocol to face known attacks against CBC mode:

- TLS initially used an implicit IV (the last block of the previous record) for the CBC mode. TLS 1.1 changes this behaviour to use an explicit IV for every block. The goal of this change is to counter a then theoretical attack, that was implemented later as BEAST (see section 2.1.1);
- TLS 1.0 could instantiate padding oracles. An implementation note was added in TLS 1.1 to reduce this avenue for attacks. As shown in section 2.1.4, other forms of oracles were however still present in most implementations in 2013.

Besides, RFC 4346 contains mostly clarifications and editorial improvements.

[5] Ironically, the RFC was published in 1999, just before the relevant RSA patents expired.

[6] A minor revision was published after TLS 1.1 in 2006 [BWNH+06]. Since the TLS 1.2 specification incorporated the first part of the TLS extension RFC, it was updated once again in 2011 [3rd11].

[7] The original specification actually contained an error in the length encoding, which led implementations to interpret the RFC differently. Two years later, another RFC [SZET08] fixed the specification to align it with the deployed implementations. The matter is briefly discussed in section 7.2.4.

1.2.6 TLS 1.2

In 2008, a new revision of the protocol was published, TLS 1.2. This version of the protocol introduces significant changes.

Besides various editorial changes and clarifications (including the merge of TLS extensions and ciphersuite definitions), the first series of updates are meant to enable the customisation of more parameters by means of ciphersuites or extensions:

- in previous versions, the pseudo-random function (PRF) used to derive the keys was a combination of MD5 and SHA-1; TLS 1.2 now specifies that the PRF depends on the ciphersuite (and is by default SHA-256);
- TLS signatures in the `ServerKeyExchange` message used to rely on a MD5/SHA-1 combination to hash the data; with TLS 1.2, this is replaced by a single hash, which is negotiated with an extension[8];
- the length of the data in the `Finished` message, that was until then truncated to 12 bytes, can be customised by new ciphersuites.

The major improvement with TLS 1.2 is to support *Authenticated Encryption with Authenticated Data* (AEAD) ciphersuites to protect the record payload, in addition to the existing CBC mode and stream ciphers. Even if TLS 1.2 does not standardise such ciphersuites, the first ones, using AES with Galois Counter Mode (GCM) were specified briefly after [SCM08] or a little later with the Counter with CBC-MAC (CCM) mode of operation [MB12].

Finally, TLS 1.2 defined the first ciphersuites using HMAC SHA-256, and promoted an AES ciphersuite, `TLS_RSA_WITH_AES_128_CBC_SHA` as the mandatory-to-implement one.

1.3 Protocol integration

1.3.1 Implicit versus explicit TLS

Initially, TLS was designed to be an implicit security layer: clients connecting to HTTPS servers would open a TCP connection, and directly start a TLS dialogue. After establishing a TLS session, the client would then speak the application layer protocol. TLS can thus simply be seen as another transport layer, between TCP and HTTP.

There is however another way to integrate TLS into an existing protocol: the STARTTLS (or explicit) way, e.g. in SMTP. Here, the idea is for the client to first send application messages (`EHLO` for SMTP), and only after several exchanges, explicitly propose to use TLS via a specific command (`STARTTLS` in this case). Once the negotiation is successfully completed, the application-layer exchanges can continue.

Even if most protocols allow for both implicit and explicit TLS integration, each protocol is usually strongly associated with one method. Here are some examples:

- With HTTP, in spite of a tentative description of an upgrade method to explicitly switch to TLS [KL00], only implicit TLS on 443/tcp is used in practice;
- POP and IMAP mostly rely on implicit TLS (ports 993 and 995), even if an explicit method is specified for both [New99];
- SMTP is more interesting, since both usages are encountered in practice: implicit TLS using port 465 and explicit STARTTLS method [Hof02]. Yet, the SMTPS port (465) was recently reallocated for another purpose, leaving only explicit TLS as the officially supported method.

The implicit method is conceptually much simpler, since it does not put any burden on the existing application. Moreover, since every connection is implicitly protected, it is harder to bypass. However, there are two main problems with implicit TLS:

[8] There is a recurring argument on the TLS working group mailing list indicating that this leads in practice to a decrease in security, since signatures, that were initially computed using MD5 *and* SHA-1, can sometimes use plain MD5. A recent attack, SLOTH, shows this can actually be exploited (see section 1.5.1).

- originally, there was no way for the client to signal which host it wished to contact in case of virtual hosting; thus, the server would not know which certificate to use, which would require sharing the same certificate for all hosted services. To overcome this limitation, the Server Name Indication extension (presented in section 1.2.4) was specified;

- by construction, plaintext and TLS versions of the protocol can not share the same port, which requires allocating a new service with IANA.

On the other hand, explicit TLS can reuse the existing port, at the cost of a more complex software architecture (see for example CVE-2011-0411, a vulnerability affecting `postfix`, where I/O buffers were not cleaned when switching from cleartext to TLS, allowing a form of plaintext command injection). By construction, STARTTLS is an option and it thus allows for easy downgrade attacks, since an active attacker only has to remove the explicit signals, which has recently been shown to be an efficient attack against SMTP [DAM+15].

1.3.2 Identity issues

Adding TLS to a protocol always raises the question of identity verification: for each application protocol, it is essential to describe how the server identity should be incorporated inside the certificate. For several protocols, the identity is simply the IP or the hostname of the server, but in other cases, such as SMTP servers, it makes sense to signal the domain the server can receive mail for, since the translation from domain name to server name is the result of a (possibly insecure) DNS resolution.

For this reason, a specification describing the different ways to include identity information was published [SAH11]. It is then referenced from application-related specification. For example, in the case of SMTP, there is a document specifying how to check certificate names for SMTP using DANE [DH15].

1.3.3 Additional constraints and dependencies

The application-layer protocol may exhibit dependencies on the underlying TLS layer in unexpected ways. For example, NNTP (Network News Transfer Protocol) over TLS [MVN06] lists compression as a possible advantage of using TLS, yet this feature is now considered dangerous and will disappear in TLS 1.3. Discussions on the TLS working group list led the NNTP community to work on an Internet Draft describing a specific compression layer.

Another example of a cross-layer relation is HTTP/2 [BPT15], which constrains the TLS connection it uses: the protocol version must be at least TLS 1.2, the stack must support *and use* the SNI extension, several features (compression, renegotiation) must be disabled, cryptographic parameters and ciphersuites are restricted to specific guidelines, etc.

1.4 Attacker models

We usually consider two major classes of attackers against SSL/TLS: the passive network attacker and the active network attacker. Since the main use of TLS is to transport HTTP, which comes with a complex and intricate security model, there actually exists another attacker model taking into account the specific case of HTTPS.

1.4.1 Passive network attacker

A passive network attacker is an adversary able to listen to the messages exchanged between legitimate parties. Such an attacker should not learn anything about the application data sent and received by the client and the server. She might however learn metadata about the client and the server (identities, cryptographic capabilities), since they are sent in cleartext Handshake messages[9].

Cryptographically speaking, a passive network attacker can mount known ciphertext attacks: she can observe encrypted messages between legitimate users.

[9]The situation might partially change with TLS 1.3 where most of the Handshake messages are encrypted, in particular to protect the identities against a passive attacker.

1.4.2 Active network attacker

A more powerful attacker is able to actively interfere with the connections between a legitimate client and a legitimate server (she can add, delete or modify any message).

Such an attacker is also able to pose as a client: she can thus connect to real servers. She can also pose as a server, in which case we assume real clients might at some point establish a connection; such *a priori* unsolicited connections are indeed easy to achieve, either by sending an e-mail containing a link (classic social engineering) or by inserting scripts and links inside existing plaintext HTTP connections.

In the cryptographic world, the powers of an active network attacker include partially-known plaintext attacks (she can guess parts of the plaintext messages exchanged between legitimate parties) and chosen ciphertext attacks (she can send messages to be decrypted by a legitimate client or server).

1.4.3 Active HTTPS attacker

Beyond the ability to add, modify and delete messages, an attacker could take advantage of the web ecosystem, where it is easy to have contents from both a sensitive service and an attacker-controlled server in the same browser tab. Examples of such mix are the use of ads inside an e-mail service or the use of plugins to customise a site.

Even if the Same Origin Policy (SOP [Bar11b]), a fundamental security property implemented in web browsers, is intended to prevent scripts loaded from a given site from communicating freely with another site, such communications are only restricted, not banned. An HTTPS attacker could thus leverage this to trigger specific connections towards real servers in an authenticated context (in the HTTP world, such attacks include Cross-Site Request Forgeries, CSRF, first described in 2001 by Percival [Per01]), which can lead to (partially) chosen plaintext attacks[10].

1.5 A history of security flaws

It is generally very hard to categorise security flaws in a relevant way. The proposed approach here is to partition them in four sections: handshake vulnerabilities, record protection flaws, certificate/trust problems and implementation issues.

1.5.1 Handshake vulnerabilities

Negotiation of weak cryptographic parameters

The first reason a particular handshake can lead to a vulnerable TLS connection is when the negotiated parameters are weak. It is obviously the case with EXPORT ciphersuites, which restrict the key length to 40 bits for symmetric encryption and to 512 (or 1024) bits for RSA keys.

Similarly, using short public keys in certificates or small Diffie-Hellman groups may also lead an attacker to break the protection offered by TLS. Moreover, the size of the public key parameters can not really be negotiated: the server presents its certificate and optionally sends its Diffie-Hellman group, and the client either closes the connection with an alert, or chooses to accept them. In the case of DHE ciphersuites with 256- or 512-bit finite field groups, the latter leads to a connection where the cleartext can be recovered by an attacker, as shown by the LogJam attack.

Additionally, not negotiating a suite providing forward secrecy (e.g. a suite using an Ephemeral Diffie-Hellman with an acceptable group) can be seen as a vulnerability: as a matter of fact, the IETF TLS working group is essentially removing non-forward secret ciphersuites from the TLS 1.3 standard.

The special case of weak hash functions

In 2005, MD5 was shown to be vulnerable to collision attacks: instead of requiring the expected complexity of 2^{64} operations, Wang et al. published an efficient method to compute MD5 collisions [WY05][11]. In the following months, several publications improved the algorithm, to the point where one could

[10] The exact range of possibilities is described in concrete attacks in chapter 2.

[11] For the record, this publication also presented concrete collisions for other hash functions such as MD4, HAVAL-128 and RIPEMD. Concerning SHA-1, the researchers only estimated the complexity of their attacks, since they could not compute a collision.

produce two syntactically correct certificates hashing to the same value [LWdW05]. Eventually, a real attack against an existing certification authority was presented in 2009 by Sotirov et al. [SSA+09]: the researchers were able to buy a valid certificate, and produce in parallel another certificate such that both certificates had the same hash. It is also worth noting that a similar technique was used to obtain a valid signature for the FLAME malware [FS15]: a valid signature over an innocuous file was also valid for the malicious program.

Currently, the best attack to compute a SHA-1 collision is estimated to require around 2^{51} calls to the compression function (instead of the expected 2^{80} complexity) and a collision was presented in 2015 against SHA-1 compression function [SKP15]. SHA-1 is thus considered obsolete and should be quickly phased out (its end of life is already hard-coded to 2016 or 2017 in various pieces of software).

Recently, it was shown that allowing the use of raw MD5 in signatures was a mistake that could lead to so-called transcript collision attacks. The attack, called SLOTH [BL16], consists in an attacker mounting two different sessions with a client and a server in such a way that she can obtain two colliding transcripts; then, the genuine signature from one party can be used in another context, leading to possible authentication bypasses. Since TLS 1.2 removed the combined MD5/SHA-1 hash function and added raw MD5 instead, it is thus vulnerable to this kind of attack. For example, client impersonation is possible with TLS 1.2 using RSA-MD5 as the signature algorithm, and requires 2^{39} MD5 hashes. The authors thus advocate for the complete removal of MD5 and SHA-1 in TLS 1.3[12].

Specification flaws

SSLv2, the first public version of the protocol, contains many structural flaws [TP11]. The most significant one is the possible down negotiation during a SSLv2 handshake: since the proposed ciphersuites are not covered by the HMAC in the `Finished` messages, an attacker can easily change the ciphersuites proposed: it is thus possible, as shown in figure 1.10, to change the suite a client and a server will choose. However, both ends must be willing to accept the chosen suites for the attack to work. This was an early warning that merely accepting obsolete and weak cryptography could be turned into real attacks; FREAK [BBD+15] and LogJam [ABD+15] showed that again later.

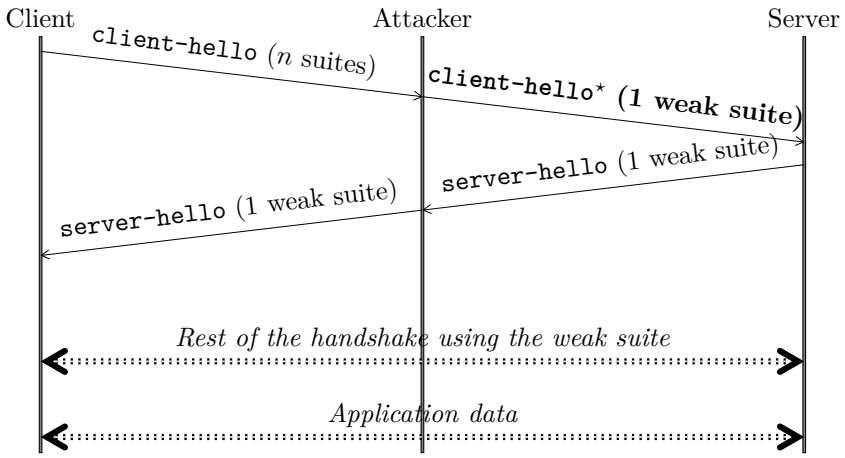


Figure 1.10: SSLv2 downgrade attack.

All current SSL/TLS versions allow RSA encryption as a key-exchange mechanism. TLS 1.1 and TLS 1.2 even include it in their mandatory-to-implement ciphersuite. However, the corresponding encryption scheme, PKCS#1 v1.5, was proven flawed in 1998 by Bleichenbacher [Ble98]. The basic idea, detailed in section 7.3.2, is the following: according to PKCS#1 v1.5, the raw RSA decryption should yield a correctly padded string; since correctly padded plaintexts translate into integers from a given interval ($[2^{n-16}; 3^{n-16}[$), an attacker who is able to distinguish a padding error from the TLS stack behaviour can forge ciphertexts related to a target ciphertext $C^\star$ to learn information about $C^\star$.

[12]The attack actually impacts many implementations, since several SSL/TLS stacks were shown to accept RSA-MD5 signatures, even if they negotiated otherwise.

More recently, two attacks were published that targeted renegotiation and session resumption mechanisms. In 2009, Ray and Dispensa presented an attack where an attacker would act as a man-in-the-middle to inject chosen plaintext between a client and a server *before* letting the client and the server speak freely, and in a protected way [Ris09]. Figure 1.11 shows a transcript of the vulnerability: when the connection is finally established, the client believes it is the initial negotiation while the server believes it is running a renegotiated session. Without any means to detect this discrepancy, several concrete attacks were proposed, essentially against HTTPS, which allow a client to ask for a resource (e.g. a request in the attacker-controlled prefix) before the client sends its credentials (typically via authentication cookies). This flaw was fixed by adding a cryptographic hash of previous negotiation in the `ClientHello` [RRDO10].

In 2014, a similar attack called Triple Handshake was presented [BDF+14]: this more sophisticated attack relied on both renegotiation and session resumption to mount two different TLS sessions resulting in the same *master secret*. It is possible, since *master secret* derivation only depends on the result of the key exchange mixed with publicly sent random values. Once both sessions are initiated, it is possible to use renegotiation in a similar way as in the previously described renegotiation vulnerability.

The first connection in the Triple Handshake attack, described in figure 1.12 aims at establishing two sessions, one between C and A and another one between A and S, sharing the same *master secret*. Moreover, it is associated to the same session identifier by C, A and S. However, C thinks it is connected to A and not to S, and the data contained in the `Finished` messages differ between the sessions, since the transcripts are different. In this figure, we assume the negotiated exchange algorithm is RSA encryption; the attack can be adapted to the DHE case, assuming the attacker chooses a non-prime Diffie-Hellman group.

The second connection is presented in figure 1.13. When the client tries to resume the session with A, A resumes the session with S, using the exact same messages. A can then inject arbitrary plaintext inside the connection on the right, possibly triggering a renegotiation requiring client authentication using certificates. In this case, A lets C reauthenticate himself, and discovering that the server switched from being A to being S. Thus, the request from A can be authenticated a posteriori by C. However, as in the renegotiation vulnerability, once the renegotiation is over, A can not decrypt the data anymore.

The problem comes from the lack of cryptographic binding between the handshake parameters and the *master secret*. An extension including all the handshake transcript during the *master secret* derivation, called *session hash*, was then specified [BDLP+15].

As a side note, it is worth noting that session resumption in itself can be seen as a mechanism breaking the forward secrecy property. To be able to resume a previous session, the server must keep either a *master secret* database (legacy resumption) or a key to decrypt the session ticket, which both lead to the *master secret*, and ultimately to the contents of past communications. This is why several actors have been working on improving the session ticket key protection [HA13] (frequent rotation, keys kept outside long-term storage).

Cross-protocol attacks and bugwards compatibility

The idea of TLS cross-protocol attacks has existed since 1995. It was first described by Schneier and Wagner [WS96], using SSLv3 `ServerKeyExchange` message: since handshake integrity is only guaranteed *a posteriori*, and since the `ServerKeyExchange` does not cover the negotiated ciphersuites, an attacker can change the outcome of the ciphersuite negotiation and reuse the `ServerKeyExchange` for another key exchange algorithm.

Even if the original attack (masquerading an RSA key exchange for a DHE one) could only have affected very liberal TLS parsers (and the conditions were never met in practice), the idea was recently shown to be almost practical for DHE/ECDHE [MVVP12]. Moreover, the generic problem of not authenticating enough information during the handshake[13] was exploited by more recent attacks such as FREAK [BBD+15] (described in section 7.4.2) and LogJam [ABD+15], where parties were misled to use EXPORT ciphersuites: using weak RSA keys in the first case and weak Diffie-Hellman group in the second.

[13] Only the random values are covered by `ServerKeyExchange` signature with DHE/ECDHE key exchanges, which ironically, was the exact same problem that led to a trivial downgrade attack with SSLv2, and nothing is really authenticated before the `Finished` message with RSA.

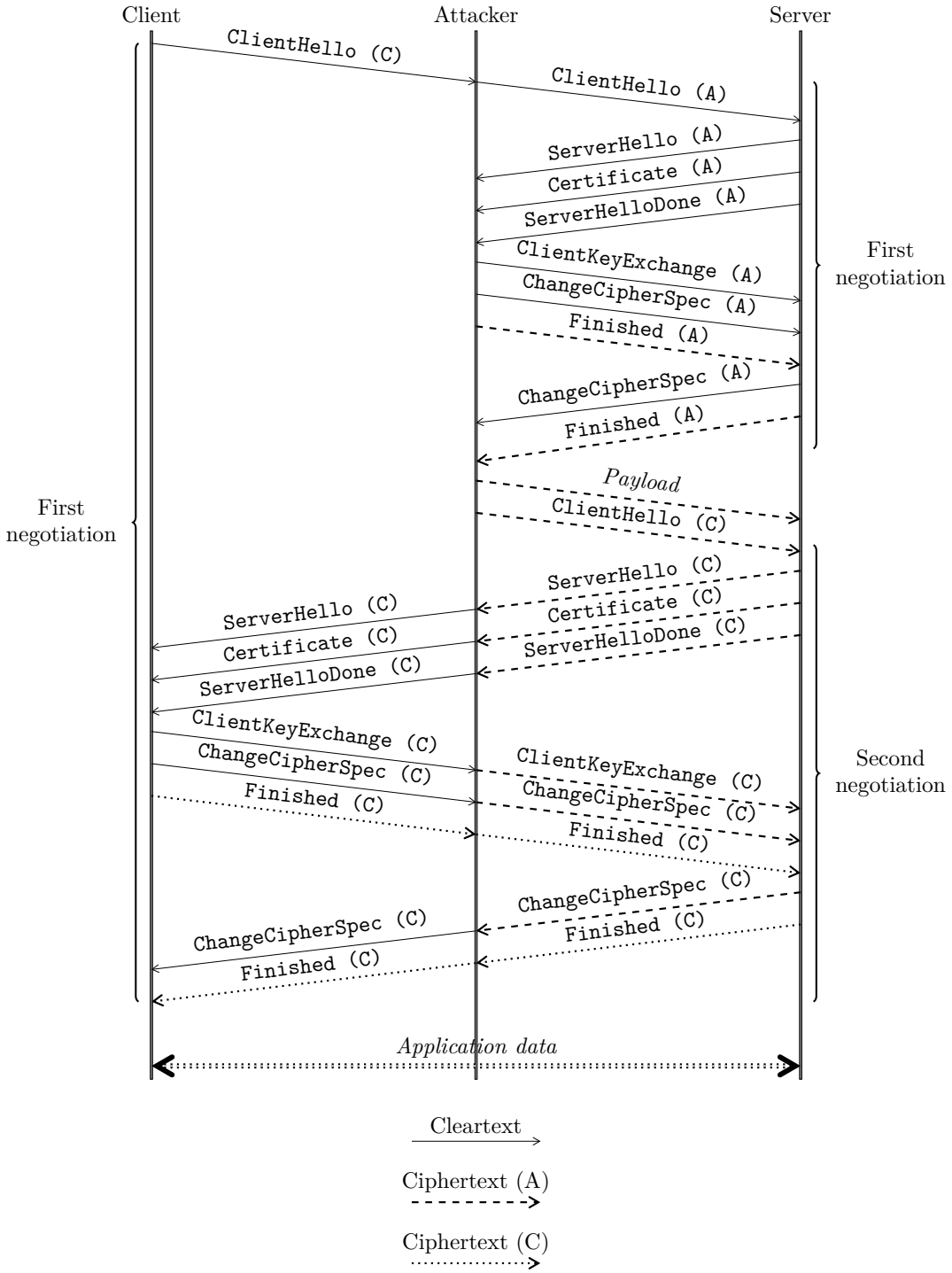


Figure 1.11: Transcript of an attacker exploiting the renegotiation vulnerability.

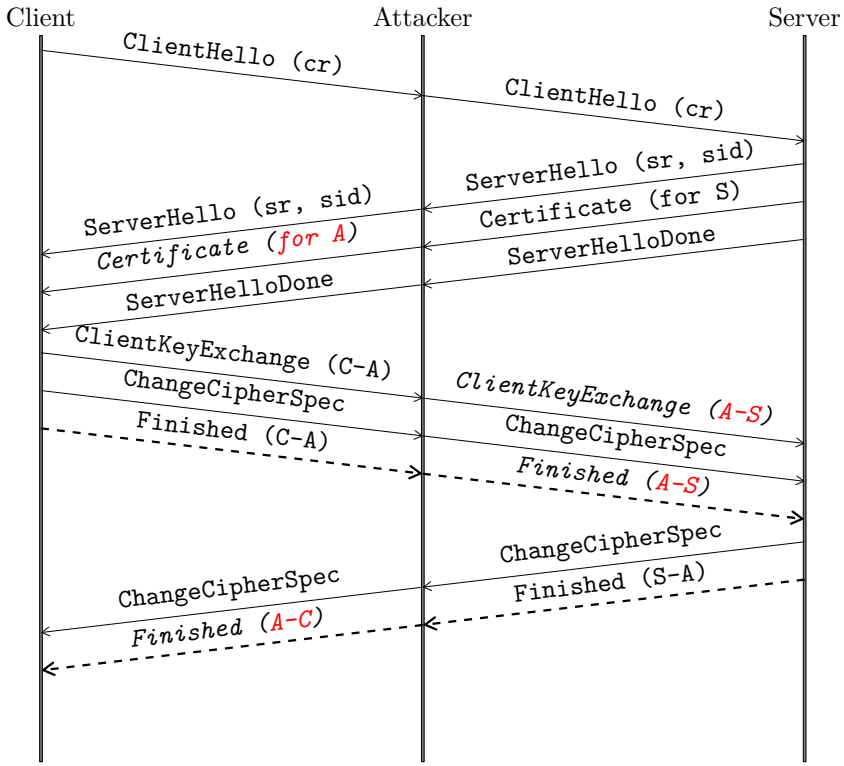


Figure 1.12: First connection in the Triple Handshake attack. `cr` means `client_random`, `sr` for `server_random`, `sid` for `session_id` (used for session resumption). Fields in red / messages in italic are the ones modified by the attacker.

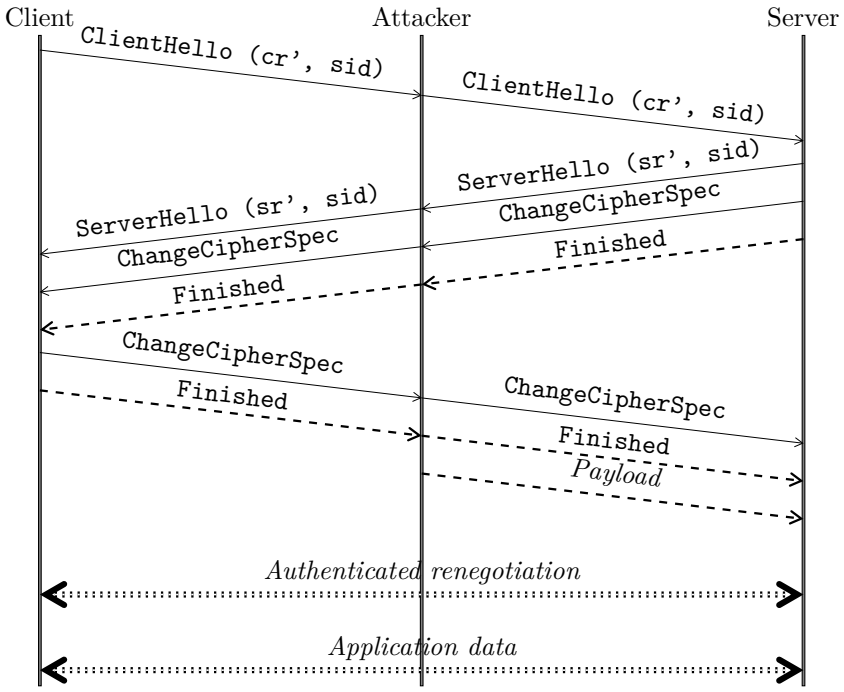


Figure 1.13: Second connection in the Triple Handshake attack. As soon as the client accepts the new server certificate, the end of the attack is similar to the renegotiation vulnerability.

Two proposals were made to overcome this limitation. The first one, presented earlier as a countermeasure against Triple Handshake, is to incorporate all the Handshake messages into the key derivation process. This way, when an attacker tampers with Handshake messages, the client and the server end up with different keys. It is implemented for TLS up until version 1.2 via the *session hash* extension [BDLP+15], and is a mainstream mechanism for TLS 1.3. However, this proved insufficient for LogJam, since leading to different keys is of no use when the attacker has weakened the key exchange to the point that she can recover the keys derived by the client.

This leads us to the second solution, which is to have the server explicitly sign all the Handshake messages in the `ServerKeyExchange`, which is incompatible with RSA key exchange (where no signature takes place). Both measures (using only DHE/ECDHE key exchanges and explicitly signing all the transcript with the server key exchange information) are scheduled for TLS 1.3. Current TLS stacks are at risk as soon as they propose or accept weak cryptographic parameters (such as DHE groups in the LogJam case).

Limiting the proposed features and algorithms is, as often in security, a good approach, especially if this leads to the removal of obsolete versions or algorithm support. However, it is not always possible to only use the latest version and to activate security features, for compatibility's sake. In some cases, clients even retry connecting to the server with weaker parameters, to accommodate intolerant servers, that is servers that refuse to properly negotiate a session in presence of certain extensions or fields (see chapter 5 for more information on the subject); this behaviour is sometimes referred to as the *fallback dance*.

This is why, over the years, several mechanisms have been devised to prevent an attacker from tampering with the outcome of the handshake while still supporting old versions:

- To avoid downgrade attacks leading to SSLv2, which always use RSA encryption as key exchange, SSLv3 specifies that the *pre-master secret* should start with the highest version of the protocol supported by the client. Even if this could be used for other versions as well, several implementations do not check properly this field, which can lead to interoperability issues.

- An extension has been standardised in 2015 to allow a client to signal it is not advertising the correct highest version of the protocol: this allows to continue executing the fallback dance, while allowing a modern server to detect (and reject) the inappropriate fallback. The used signal is a fake ciphersuite, called the *Fallback Signaling Ciphersuite Value* [ML15][14].

- To overcome the rollback from TLS 1.3 to TLS 1.2, even in the case of a powerful attacker capable of tampering with most of the negotiation (as in the LogJam case), the proposed solution consists in encoding a predefined string in the beginning of the `server_random` field when the server receives a TLS 1.2 (or older) `ClientHello`, and only use DHE/ECDHE ciphersuites: this way, the server would sign the value on which the client apparently commits, which makes the inappropriate fallback traceable for the client. This proposal has recently been added to the TLS 1.3 draft.

1.5.2 Record protection flaws

Record protection vulnerabilities are only briefly presented here, since a detailed description is given for each of them in chapter 2.

CBC mode

Theoretical attacks targeting the CBC encryption mode have been known for a long time, starting in 1995 with an adaptive chosen plaintext attack against CBC with a predictable IV [Rog95]. It was later exploited in 2011 by the BEAST attack [DR11].

There is another class of attacks against CBC, at first only from a theoretical point of view: padding oracle attacks. First described by Vaudenay in 2002 [Vau02], they were only shown to be practical in 2012 against DTLS [PA12] and in 2013 against TLS [AP13] (the Lucky 13 attack). Finally, in 2014, a very efficient padding oracle attack called POODLE [MDK14] was presented against SSLv3, which uses a non-standard padding.

[14] A signaling ciphersuite value was specified for the secure renegotiation extension, to avoid the ordeal of extension intolerance for the first negotiation (where the extension value is empty).

As a side note, it is interesting to note that the BEAST and POODLE attacks were already known. They had even been described in 2002 by Möller [Möl04]. The document also contains the first steps towards the Lucky 13 attack. However these attacks were believed to be impractical.

RC4

RC4 is the only standard streamcipher in TLS. Unfortunately, it has been shown vulnerable to statistical biases since 1995. In 2002, researchers used these biases to break the Wifi WEP encryption layer [SIR02].

In 2013, two independent teams presented practical results against RC4 [IOWM13, ABP$^+$13], one of them leading to the recovery of secrets (such as passwords or authentication cookies) by observing many TLS sessions containing the secret data. These attacks were later improved [VP15, GPvdM15].

Compression-related issues

Between 2012 and 2013, several attacks related to compression have been presented against HTTPS: CRIME [RD12], TIME [BS13] and BREACH [PHG13]. The target of these attacks was either a client-sent authentication cookie or a server-sent anti-CSRF token.

The basic idea of the CRIME, TIME and BREACH exploits is the following: even when encrypted, a compressed message may leak information about its content through its length. Thus, compressing a message containing both secret information and attacker-controlled data may allow the attacker to learn information on the secrets by injecting specially-crafted data.

In the case of HTTPS, this adaptive attack can be used to recover secrets repeatedly transmitted using SSL/TLS, as soon as some form of compression is used: TLS-level compression, HTTP compression feature or SPDY[15] header compression.

1.5.3 Certificate/Trust problems

Random generation issues

In cryptography, good random numbers are usually needed to ensure the security properties of algorithms and protocols. In SSL/TLS, such random numbers are used in different situations:

- both the client and the server have to generate a random value in their respective `Hello` messages, which will be used to derive keys in a non-replayable manner;
- during the key exchange, whether it uses RSA encryption or it is a Diffie-Hellman-based scheme, a random number must be generated and kept secret;
- a random number generator is used to produce the long-term private key of the server, which must also be kept secret;
- several symmetric mode of encryption require an explicit random IV (CBC with TLS 1.1 and later) or a nonce (e.g. GCM);
- finally, several signature algorithms, such as DSA or ECDSA, require a random value.

In the last decade, several examples of bad random number generators have been made public concerning the certificate generation case. The most publicised was a bug in the Debian version of OpenSSL, that reduced the effective entropy to only a few dozens of bits from 2006 to 2008 [Deb08]. More recently, Heninger et al. showed that various network devices did not produce enough entropy and reused prime numbers between two RSA generations, allowing moduli to be factored by a simple gcd algorithm [HDWH12].

In both cases, this meant that an attacker could obtain the server private keys. Depending on the used key exchange algorithm, the consequences could be the complete loss of confidentiality of past SSL/TLS communications (in case RSA encryption was used) or the ability for an attacker to mount an active attack and impersonate the server (when Diffie-Hellman was used).

[15] SPDY is a protocol developed by Google to improve the bandwidth usage of HTTPS exchanges. It was the basis for the new HTTP/2 protocol specification [BPT15].

Concerning DSA and ECDSA, it is well known that using a (partially) predictable random value to produce a signature with these algorithms leads to efficient private key recovery attacks. Another common mistake concerning these algorithms is to reuse the same random number across several signatures. Sony made that mistake with the PS3 firmware, where the same nonce was reused to sign different bootloaders, which led attackers to bypass the protections [bms10].

Incidents affecting certification authorities

In many cases, SSL/TLS parties rely on a wide set of certification authorities to authenticate the server they are connecting to. This model is often referred to as the Web-PKI, since the Public Key Infrastructure is usually found in browser and other HTTPS-related software.

Originally, there was no way to treat differently the numerous trusted authorities in this set. This led to problems when these authorities emitted certificates for unauthorised subjects, either by error or following an attack on their systems.

The first documented incident dates from 2001: an attacker contacted VeriSign and used social engineering to convince them to issue a code-signing certificate in the name of Microsoft [Mic01]. Since revocation mechanisms were not really reliable, the fix consisted in blacklisting the certificate in major certificate stores by embedding the corresponding hash.

More recently, several attacks against trusted certification authorities led to the issuance of illicit certificates: in 2011, Comodo [Com11] and Diginotar [Vas11, FI12] were attacked by what seemed to be the same person (or group of people). In the first case, the breach was quickly discovered and the problem fixed, whereas in the second case, compromised certificates were only discovered later, leading to a loss of confidence in Diginotar. In both cases, certificates were revoked, but since revocation did not work any better in 2011 than in 2001, blacklists were also published for major certificate stores[16].

In 2012, Trustwave announced a change in their policy: they would not issue intermediate certificate authorities to their clients [Tru12]. Such certificates indeed allowed said clients to mount SSL/TLS interception attacks. Such interceptions are known to be used in so-called *data-leak prevention* systems, but best practice (and common sense) recommend the interception certification authority *not* to be trusted by public root CA and to be manually installed on monitored computers. More incidents were reported in the following years, including one with the IGC/A, the root certification authority managed by the French Administration and one with the Turkish authority TÜRKTUST; they were the consequence of a human error consisting in feeding a publicly trusted authority inside a data-leak prevention device.

Weak cryptographic parameters

A classical flaw of TLS certificates is the use of weak algorithms or short keys. Certificate signatures rely on hash functions, and we already discussed this matter, and its implications in certificate forgery in section 1.5.1, especially concerning MD5 and SHA-1.

DSA is another algorithm that is usually considered obsolete. Like ECDSA, DSA has the undesirable property to fail dramatically in case signatures are computed using bad random values (for example, should a user reuse a random value between two signatures, an attacker could recover the corresponding private key), but the major problem with DSA is that the early specification required to use 1024-bit keys (which are considered too small nowadays) and the SHA-1 hash algorithms. Even if updated specifications fix these problems, the modern version of DSA is still not widely implemented and leads to compatibility issues.

Finally, for RSA keys, a common problem is the use of small keys. For example it is easy with a standard computer to break a 512-bit RSA key, that is to factor the corresponding modulus. Stories featuring such weak keys are pervasive [Ent11, tic09, Zet12] and the FREAK attack consists in impersonating servers accepting to use 512-bit RSA keys (see section 7.4.2 for more details). In 2010, a 768-bit RSA key was broken by academic research teams [KAF+10], but such a computation is now easier to achieve with recent computers. Since RSA keys can be used for a long period of time, it is recommended to avoid using keys with a modulus shorter than 2048-bit.

[16] For Diginotar, the problem was finally solved by removing their root certificates in most certificate stores, which ultimately led the firm to bankruptcy.

Revocation

In case something goes wrong with a private key (either because it was stolen or computed from the public key), a common practice is to revoke the corresponding certificate, to avoid relying on it in the future. In the SSL/TLS ecosystem, there currently exist two revocation mechanisms: Certificate Revocation Lists (CRL, defined by the X.509 standard [CSF+08]) or Online Certificate Status Protocol (OCSP [SMA+13]), which both have serious limitations.

CRL consist in the regular publication of the list of revoked certificates: for example, when a private key is compromised, the authority that emitted the corresponding certificate should update its CRL. After the previous CRL expires, users downloading the updated CRL are informed of the certificate revocation. Yet, CRL lifetimes are sometimes too long (it is not uncommon to encounter 1-year validity periods). Moreover, CRL delivery is sometimes unreliable, leading to a so-called soft-fail mode: in case the client cannot download an up-to-date CRL, it either falls back to the previous one or ignores the revocation mechanism entirely. It is thus easy for an active network attacker to render CRL useless.

The other revocation mechanism is OCSP, which requires the verifier to establish an another connection to an OCSP server and ask whether a certificate is revoked. This essentially poses three problems: first, requiring a second connection is highly inefficient; second, it is subject to the same soft-fail as CRL; last, OCSP queries are a privacy concern, since they allow a third party (the OCSP server) to learn information about which server a particular client is visiting. To overcome the efficiency and the privacy issues, a TLS extension was designed, *Certificate Status Request*, allowing the server to request (and even cache for a short period of time) an OCSP response and to include it inside the `ServerHello` message.

To overcome the limitations of revocation systems, that were brought to light by the incidents regarding certification authorities since 2011, several solutions have been devised. First, to make CRL checks faster and more reliable (at the expense of a partial coverage), several browsers are now embedding regularly updated CRLs: CRLSets for Chrome [Lan12] and OneCRL for Mozilla products [Goo15]. The revocation information is readily available within the browser, but only for a small selection of certification authorities: CRLSets are limited in size (250 KB at the time of writing), whereas OneCRL only covers CA intermediate certificates. This limited scope is a fundamental flaw of these mechanisms, which makes them mostly unreliable.

Regarding OCSP, a simple way to improve the situation is OCSP Stapling: using a TLS extension, the client asks the server a fresh OCSP response. This solves the performance problem, since the client only needs to make one connection; moreover, the server can cache the response for a certain amount of time. Since only the server directly contacts the OCSP server, there is no privacy concern anymore. To guarantee OCSP Stapling is honoured by a TLS server, the missing piece of the puzzle is to add a specific X.509 extension, dubbed *Must Staple*, to the server certificate [HB15]: a client should not pursue the connection with a server signaling this feature but failing to provide a fresh OCSP response.

Other, complementary, solutions were proposed, for example:

- It is possible to constrain the scope of certification authorities (e.g. on the domain names they can issue certificates for) using X.509 extensions, which limits the consequences of a compromise.
- Certificate Pinning binds a certificate or an authority to a server name, and is standardised in HTTP as HTTP Public Key Pinning, HPKP [EPS15];
- Certificate Transparency [LLK13], which relies on the idea of publishing all relevant certificates in public, append-only, logs.

All-in-all, revocation mechanisms are still complex and not mature enough in the TLS ecosystem (and especially in the web context). Only a combination of them may be reliable and efficient.

1.5.4 Implementation bugs

In this section, we briefly list implementation bugs that affected SSL/TLS stacks over the years. As a matter of fact, several security issues, that were previously identified as related to the Handshake or to the Record protection could also be seen as implementation bugs. This is especially the case for cryptographic flaws, where countermeasures can sometimes be implemented. Chapter 7 proposes a more thorough analysis, including discussions about the real causes of what are usually seen as mere programming bugs.

Memory management flaws

Like any piece of code, an SSL/TLS implementation is bound to contain simple, pervasive bugs, such as memory management flaws. The most famous example of this kind of bug is Heartbleed (CVE-2014-0160), a trivial buffer overflow leading to memory leaks, which could include cleartext messages or private keys.

The same year, a vulnerability was identified in SChannel, that could lead to remote code execution (CVE-2014-6321). The cause was, again, a simple invalid bound checks occurring in the client certificate validation when elliptic curves were used.

Finally, due to the standard complexity, double free and buffer overflow vulnerabilities are very common in SSL/TLS code (usually written in C or C++), even if the consequences are not always as severe as with the previous examples.

Problems in the certificate validation logic

A common problem in SSL/TLS stacks over the years has been the ability for an attacker to bypass certificate validation. Doing so, she is able to impersonate a client or a server.

The first and simplest form of this problem is when no validation is performed at all. This was shown to happen a lot in Android applications, but also in Cloud applications relying on the API provided by Amazon [GIJ+12]. In the latter case for example, the underlying library is `libcurl`, which exports a misleading configuration variable `CURLOPT_SSL_VERIFYHOST`: the developer must set it to 2 to turn on certificate validation (setting it to 1 would not block an unauthenticated connection). Yet, many frameworks (including Amazon EC2 API), set this variable to `true`, i.e. 1.

X.509 is a standard with many extensions, which have not always been correctly interpreted. For instance, the `BasicConstraints` extension is used to distinguish server certificate from authority certificates; in 2002, Marlinspike showed that the distinction was not implemented in Webkit nor CryptoAPI [Mar02]. He showed other vulnerabilities on the major SSL/TLS implementations [Mar09], including the misinterpretation in null character in a certificate subject.

More recently, several implementation errors in different stacks led to the possibility for an attacker to bypass certificate authentication under some circumstances: by simply choosing a DHE/ECDHE ciphersuite (Apple's `goto fail`), by providing an ill-formed certificate (GnuTLS' `goto fail`) or by providing certificates in the wrong order (OpenSSL mis-handling alternative certificate chains, CVE-2015-1793).

Shortcuts in the State Machines

Since 2014, another category of vulnerabilities have been uncovered: protocol state machines interpreting unexpected messages. For example, the EarlyCCS flaw allowed a man-in-the middle attacker between a vulnerable OpenSSL client and a vulnerable OpenSSL server to force the use of a fixed, known, *master secret* [Kik14].

In 2015, several attacks against the state machine automata used in various SSL/TLS implementations were published [BBD+15]. By trying to interact with existing stacks in invalid ways (by skipping or changing messages), they could completely bypass client or server authentication, force the use of fixed keys, or weaken the size of the used keys.

In general, problems arise when libraries interpret the received messages independently of the state they are supposed to be in. Unexpected messages should be detected and rejected early, not parsed and dealt with as is often the case.

1.6 TLS 1.3: the next generation

The first discussions about TLS 1.3 started in fall 2013, when Rescorla proposed "potential new Handshake Flows for TLS 1.3" [Res13]. The main motivation at the time was to improve the latency of TLS session establishment, following Google real-life attempts with TLS extensions such as False Start and Snap Start and with QUIC (Quick UDP Internet Connections [HISW16]), a fully-fledged secure transport protocol aimed at replacing TCP and TLS layers as a combined protocol over UDP.

In 2014 and 2015, the numerous vulnerabilities uncovered have been used as input to improve the TLS 1.3 design and take proper care of these threats, in a somewhat proactive and generic way.

The rest of this section relies on the `draft-12` version of the specification [Res16], which was published in March 2016. As details are not completely specified or still open for debate, part of this description might not be relevant to the final specification.

1.6.1 Goals of the new standard

As described in the TLS Working Group Charter, the main expected goal of TLS 1.3 can be summarised as follows[17]:

- encrypt as much of the handshake as possible;
- reduce handshake latency (1 RTT or even 0 RTT modes) while maintaining current security features;
- revise record payload protection to address known weaknesses (CBC mode, RC4);
- make TLS 1.3 more privacy-friendly;
- minimise gratuitous changes to TLS.

Concerning the second point, the RTT (Round Time Trip) is a common latency unit, describing the number of required exchanges (from client to server and from server to client) between the TCP connection establishment and the first application data emission. Each exchange can contain several messages (e.g. the first server flight is composed of Handshake messages between `ServerHello` and `ServerHelloDone`). As was shown in previous diagrams, current TLS versions require 2 RTTs with a full handshake (see figure 1.4) and 1 RTT for a resumed session (see figure 1.7). When considering that TLS is built on top of TCP, this means that session establishment requires one more RTT in practice.

It is important to understand that these goals are not always compatible with each other; for example reducing latency might impact the security properties of the negotiated connection; also, revising the Record protocol and encrypting more messages will lead to profound changes in the protocol. Thus, the resulting specification will be a trade-off between these goals.

Let us now list the state of the specification in `draft-12`.

1.6.2 New Handshake flows

The major change in TLS 1.3 concerns the Handshake flow, which has been completely revised so that a full handshake is now 1 RTT in a favourable case. The exchanged messages are described in figure 1.14; the following paragraph decompose the required changes to reach this diagram.

First, TLS 1.3 removes RSA encryption as a possible key exchange algorithm, leaving only DHE, ECDHE and PSK key exchanges. Letting aside PSK ciphersuites for a moment, this means that *forward secrecy* is guaranteed by default.

To improve latency, the `ClientHello` must now contain several extensions:

- the *supported curves* extension, renamed *named groups* to include DHE fixed groups, where the client advertises the DHE/ECDHE groups it supports;
- the *signature algorithms* extension, renamed *signature scheme* to present the signature mechanisms (asymmetric algorithm, underlying curve when relevant, and hash function) supported by the client;
- the *key shares* extension that includes the client part of a Diffie-Hellman for several groups from listed in the previous extensions.

It is thus possible for the server to answer with a key share in its first flight, completing the Diffie-Hellman early, and allowing for early protection of Handshake messages, just after the `ServerHello`.

As shown on the figure, in the ideal case, the negotiation can be completed and Application sent after only 1 RTT. Yet, the presented flow can obviously fail if the client only proposes key shares for groups the server does not support: in this case (which is not shown on the figure), the server sends a

[17] The full version is available at `http://tools.ietf.org/wg/tls/charters?item=charter-tls-2015-09-01.txt`.

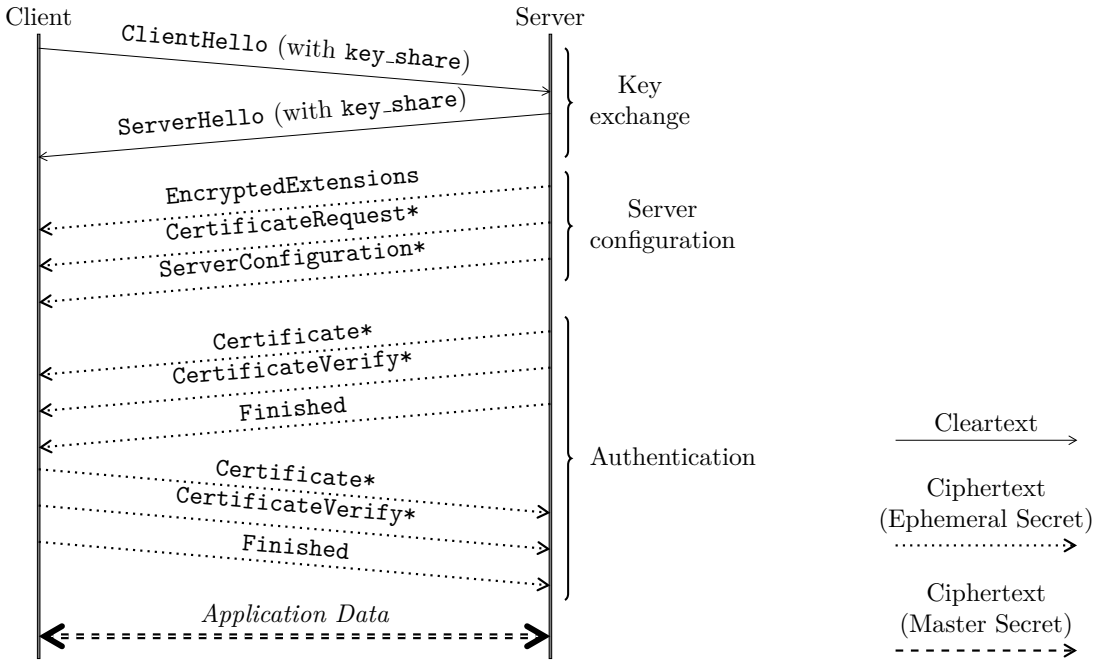


Figure 1.14: TLS 1.3 new Handshake flow. Hello messages contain a new extension, called `KeyShare`, to achieve 1 RTT handshakes. Messages with an asterisk are optional.

`HelloRetryRequest` messages specifying the named group to use, which brings the connection back to a 2 RTT scheme.

Another important change is the way server authentication works: instead of signing the (EC)DHE parameters along with `client_random` and `server_random`, which still allows an attacker to tamper with part of the handshake, TLS 1.3 reuses the `CertificateVerify` message (only used for client authentication in previous versions) for the server: the signature now covers all previously exchanged Handshake messages.

During the specification of TLS 1.3, the `ChangeCipherSpec` messages have been removed. They indeed represent a useless explicit signal, that is redundant with the implicit message flow described in the specification. Moreover, it is an unauthenticated message, which led to attacks against several implementations (see section 7.4).

This new flow leads to a clean separation of the three stages of the negotiation: the key exchange, the transmission of server configuration, and the authentication phase. It is interesting to note that the new scheme is more similar to the way IKEv2 and SSHv2 work.

Learning from recently uncovered vulnerabilities, TLS 1.3 incorporates the *session hash* mechanism as a mainstream feature. The old PRF (which was essentially an HMAC) has been replaced with a new primitive, called HKDF [Kra10], which simplifies the proofs of the protocol. Moreover, the whole derivation scheme has been further modified to produce different independent secrets: one derived to protect Handshake messages coming after the Diffie-Hellman has been completed (`xES`, the extracted ephemeral secret), one derived to protect the Application Data (`traffic_secret_0`), and one to allow for resumption (`resumption_secret`)[18]. This new scheme is presented in figure 1.15. Separating the resumption secret from the *master secret* allows to extend *forward secrecy* to resumption, since the resumption secret can not be used anymore to recover the keys used to protect the previous sessions.

In the new standard, there still exists an alternative to DHE/ECDHE key exchange: PSK (Pre-Shared Key). As in previous specifications, it can be used in combination with DHE/ECDHE to guarantee *forward secrecy*. The novelty is to use PSK as the resumption mechanism: to resume a previous session, the client and the server can use the PSK key exchange using the resumption secret as the pre-shared secret.

[18] Another secret, `exporter_secret`, is also derived, and can be used to provide channel binding at the upper layer.

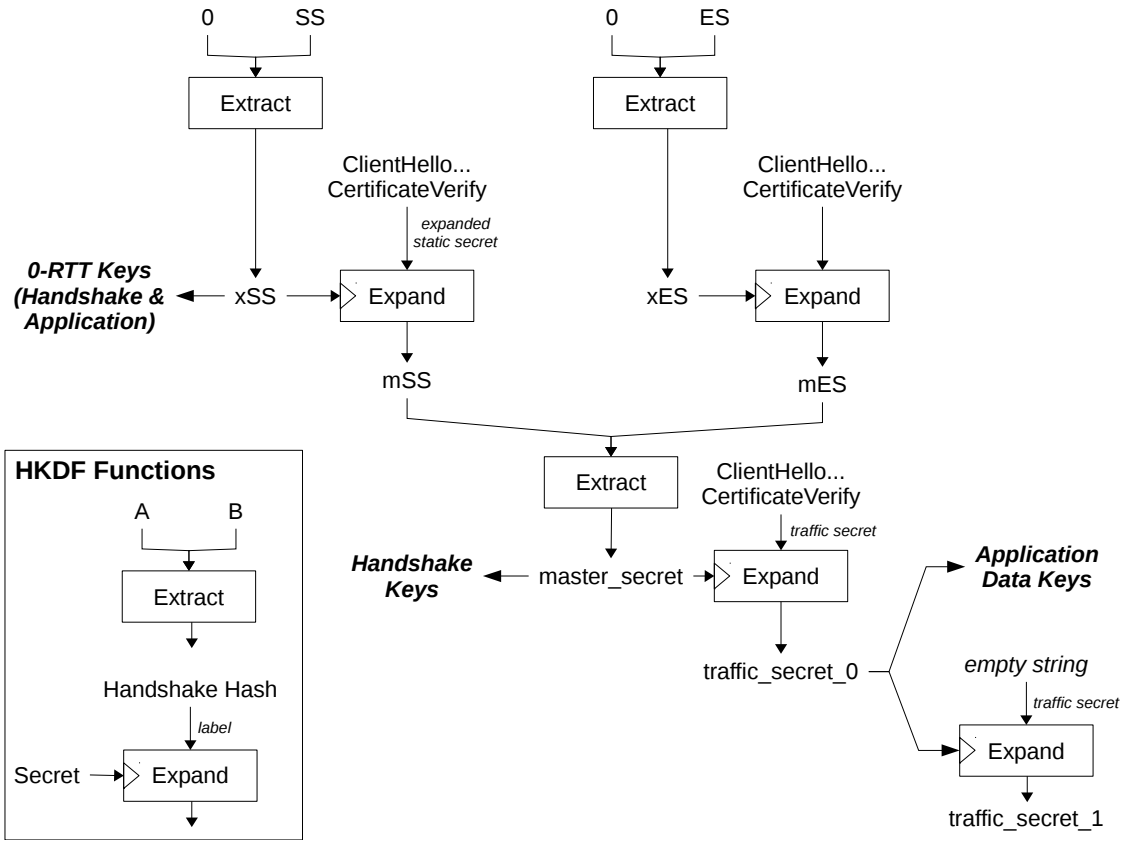


Figure 1.15: TLS 1.3 key derivation scheme. The derivation relies on the HKDF primitive, which is composed of two functions: `Extract(A, B)` and `Expand(Secret, Label + Handshake Hash)`. Keys are derived using the HKDF Expand functions. The *resumption secret* and the *exporter secret* are derived from the *master secret* using the Expand function. To provide forward secrecy, `*SS` and `*ES` values should be securely erased as soon as they are not useful anymore; similarly, once a key update has been successfully completed, the previous *traffic secret* should be securely erased.

1.6.3 A major cleanup

The new standard has also been an opportunity to remove obsolete features and algorithms. We have already seen that, contrary to previous versions, TLS 1.3 does not allow the server to propose arbitrary finite field groups for the Diffie-Hellman exchange anymore. As in IKE and SSH, groups have been generated for the purpose [Gil16][19]. The previous situation was far from acceptable: when DHE was negotiated, the server would choose the finite field group and send it to the client, that could only accept it as is, or shut down the connection. Moreover, testing for the quality of a finite field group is very hard, especially since TLS did not have the server send additional helpful information (such as the group order). Finally, several client stacks being intolerant to large DHE groups (for a long time, Java implementations have been limited to 1024-bit groups), servers could not easily use strong Diffie-Hellman groups. For all these reasons, defining well-known named finite-field groups was a necessary improvement to keep DHE in TLS.

Another obvious improvement is the removal of obsolete cryptographic constructions: broken or weak hash functions (MD5, SHA-1, SHA-224), broken ciphers (RC4, EXPORT suites in general), and the CBC mode. In practice, only the AEAD modes (which corresponds to AES using GCM and CCM modes) have been kept.

Concerning asymmetric cryptographic primitives, even if PKCS #1 v1.5 signature is still accepted

[19] Actually this RFC, which is referenced by the TLS 1.3 draft, defines the groups and how to use them in previous TLS versions.

in certificates for compatibility reasons, it is replaced by the 2.1 version (PSS, Probabilistic Signature Scheme [JK03]) for the signatures produced within the protocol (in `CertificateVerify` messages)[20]. To improve further the signature scheme, all performed signatures must now cover a *context string*, e.g. "TLS 1.3, server CertificateVerify", prepended to the actual data: the idea is to bind the signature to a protocol version and a given Handshake message.

1.6.4 Additional changes

On top of the already described major changes, here are additional noteworthy changes:

- with the new Handshake flows, renegotiation disappears;
- to remove obsolete features while keeping the `ClientHello` compatible with older protocol versions, the corresponding fields are now fixed: the compression list is hardcoded to the *null* singleton and the session id (used for legacy renegotiation) must be an empty string;
- similarly, the Record protocol version is now fixed to 3.1 (the bytes encoding TLS 1.0);
- the `random` fields used to contain a 4-byte timestamp, which might be used to fingerprint clients and servers. To avoid this, `client_random` and `server_random` is now only composed of 32 random bytes;
- as described in section 1.5.1, the 8 first bytes of the `server_random` are reused to encode a version signal (`DONWGRD` followed by one byte set to 0 or 1) in case the server receives an old (TLS 1.2 or earlier) `ClientHello`. Since this field is covered by the signature embedded in `ServerKeyExchange`, a compatible client choosing an adequate ciphersuite will detect a version roll-back. This change is part of a global effort to avoid attacks relying on obsolete cryptography/protocols [Bha16];
- finally, the contents of an encrypted record have been revised, on the one hand to include variable padding, allowing TLS stacks to implement traffic analysis countermeasures, and on the other hand to include the Content Type (the upper-layer protocol, e.g. Alert, Handshake or Application Data) inside the encrypted payload, fixing the external cleartext Content Type field in the header to the Application Data value.

1.6.5 Still in motion

The TLS 1.3 specification is still a work in progress, and important chunks of the document have been changing a lot while this manuscript was written. Let us look at three particular features that were much debated over the last months.

Key update

With the removal of the dangerously flexible renegotiation feature, several people argued for a key refresh mechanism. The proposed mechanism relies on a single, empty Handshake message called `KeyUpdate`. It can be sent by either party at any time, once the initial handshake has been completed.

The meaning of the message is to signal that any subsequent messages in the same direction will be protected using the next generation of the traffic keys. On reception, the other party must update its incoming keys, and if the outgoing keys still correspond to the earlier generation, respond with a `KeyUpdate`.

The mechanism is very simple, and does not require a TLS stack to keep a complex state: the current traffic keys and the key generation number are sufficient. The actual secret derivation is explained in figure 1.15: `traffic_secret_n+1` is derived from `traffic_secret_n` using the HKDF Expand function.

[20] As described earlier, PKCS #1 v1.5 *encryption* (used in RSA key exchange) has also been removed.

Late client authentication

Another point that resulted from the removal of the renegotiation feature is the question of the post-handshake client certificate authentication, where a client can present a certificate after a TLS session has been negotiated. Currently, initial client authentication resembles the TLS 1.2 process, while post-handshake corresponds to the following flow: the server sends a `CertificateRequest` and the client answers with the relevant messages: `Certificate`, `CertificateVerify` and `Finished`. If the client declines the request, it must send an empty `Certificate` message and the corresponding `Finished` message.

0 RTT handshakes

Finally, TLS 1.3 should include a 0 RTT feature, which has been the subject of many discussions. The initial idea was that the client could learn static parameters from the server (either during a previous connection, or via an out-of-bound method), which can be used to compute a semi-static Diffie-Hellman within its first flight of messages. Another way of doing 0 RTT, which is on the verge of supplanting the Diffie-Hellman-based mechanism, is to rely on the PSK-based resumption to reuse the previous connection secrets for the first flights of data. It is then possible to exchange fresh, ephemeral Diffie-Hellman parameters for further data exchanges, to provide forward secrecy.

For now, 0 RTT is still a work in progress, and its security is being analysed. In particular, the interaction between 0 RTT and late client authentication was proven insecure in the TRON workshop[21], held in February 2016 [BKB16]

Moreover, another generic problem is that, by its very nature, the security properties guaranteed for the 0 RTT encrypted data are not as tight as those concerning traditional TLS sessions (it might be replayed or not reliably transmitted), which should require a specific API to avoid confusion between both security *qualities* in the eye of a developer using TLS.

For all these reasons, 0 RTT will not be described in more details in this section, since only the 1 RTT specification seems to be stable enough at the time of writing.

1.7 Concluding thoughts on the SSL/TLS presentation

Since its introduction as SSLv2, the SSL and TLS protocols have become a generic solution to provide confidentiality and integrity protection to almost any application protocol. In parallel, these protocols have been subject to many attacks, ranging from specification issues to cryptographic vulnerabilities to implementation flaws. Each of them have been fixed, by specifying a new extension or protocol version, by obsoleting broken primitives, or by patching the affected code.

Recent TLS 1.2 stacks with strict configurations take essentially into account all the known issues. The next version of the protocol, TLS 1.3, goes even further by removing from the specification most of the problematic primitives. However, since up-to-date SSL/TLS stacks must continue interoperating to older ones, it is not sufficient to specify a robust and state-of-the-art version: the transition must be dealt with.

[21] TRON stands for "TLS 1.3, Ready Or Not".

Chapter 2

TLS Record Protocol security: analysis and defense-in-depth countermeasures

This chapter is based on results presented at the ASIA-CCS conference [LGD15].

Since 2011, several researchers have presented attacks against the Record Protocol. Each time, to prove the applicability of their findings, they implemented attacks against HTTPS to recover small parts of the plaintext. Typical HTTP secrets include cookies and anti-CSRF tokens. If stolen, they enable further attacks such as message replay or session hijacking or web site compromise. Fundamental to state maintenance in web applications, these secrets are usually transmitted several times both within sessions and across different sessions. Each proof of concept relies on this very repetition of secret elements inside TLS connections.

This chapter presents a comprehensive state of the art of recent attacks affecting the Record Protocol, and the associated proposed countermeasures. In parallel to the community efforts to find reliable long-term solutions, we propose masking mechanisms to avoid the repetition of sensitive elements inside the encrypted channel, either at the transport level or at the application level. Thus, secrets are no longer repeated, which blocks the described attacks. We also assess the feasibility and efficiency of such defense-in-depth mechanisms.

It is worth noting that the masking mechanisms we propose were devised before the POODLE vulnerability was made public, and that they were nevertheless efficient to thwart this then-unknown attack, confirming the relevance of our defense-in-depth proposal.

In the remainder of this chapter, we only consider attacks on TLS Record Protocol, setting aside Handshake-related subtleties such as session resumption, renegotiation or client authentication.

Depending on the Handshake outcome described in section 1.1, the records can be protected using one of the three following cryptographic schemes:

- *MAC-then-encrypt with a streamcipher*, available since SSL inception (as of today, only RC4 is standardised[1]);
- *MAC-then-encrypt with a blockcipher using CBC mode*, available since SSL inception (several blockciphers have been standardised, and the most common are AES, 3DES and Camellia);
- *AEAD* (Authenticated Encryption with Additional Data), available from TLS 1.2 only (e.g. AES using GCM mode). As a matter of fact, TLS 1.3 will only support AEAD protection.

Optionally, the plaintext can be compressed before cryptographic transformations occurs. Figure 2.1 describes the three possible workflows.

[1] In reality, TLS specifies also defines the NULL cipher, present in integrity-only ciphersuites, as a streamcipher. Since we consider attacks recovering confidentiality-protected secrets, the NULL cipher is obviously out of our scope.

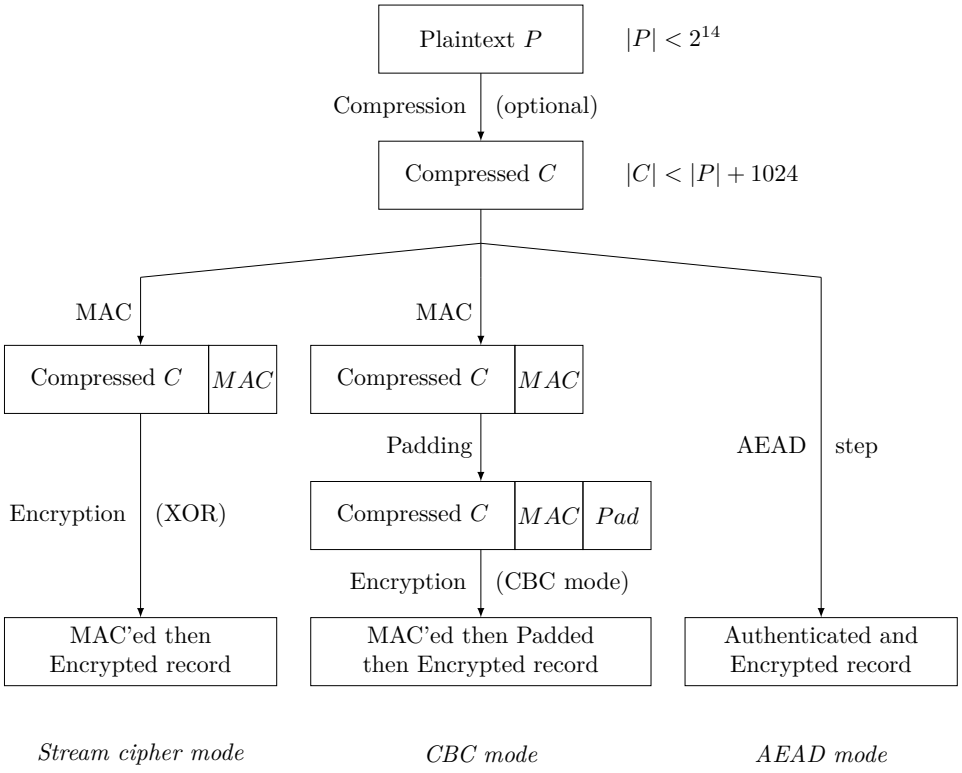


Figure 2.1: TLS Record Protocol possible workflows (streamcipher mode, CBC mode and AEAD mode).

Section 2.1 describes several recent attacks, and the proposed countermeasures. Section 2.2 presents our attacker model and the masking principle. Section 2.3 describes two proofs of concept to assess the applicability and efficiency of our proposals. Section 2.4 analyses the security benefit of the proposed mechanisms, as well as their impact on performance. Section 2.5 proposes a comparative analysis of attacks and countermeasures, including ours.

2.1 Attacks on the Record Protocol

This section describes five recent attacks published between 2011 and 2014, targeting either a session cookie sent by the client, or an anti-CSRF token sent by the server.

2.1.1 BEAST: implicit IV in CBC mode

In 1995, Rogaway described an adaptive chosen plaintext attack against the CBC encryption mode with a predictable IV [Rog95]. In 2002, it was noticed that TLS used predictable IVs, since the last encrypted block of a record is used as the next record IV [Möl04]. However, this attack was deemed impractical at the time, or at least challenging [Bar04, Bar06], since it was an adaptive chosen plaintext attack. The situation changed in 2011 when Duong and Rizzo presented BEAST (Browser Exploit Against SSL/TLS) [DR11], a proof of concept of the vulnerability.

Figure 2.2 details the "Encryption (CBC)" step presented in figure 2.1, when implicit IV is used: after the first record is sent, C_2 is used as the "IV" for the second record. Similarly, after the second record is sent, the attacker knows that C_5 will be used to encrypt the next record.

Let us now consider an attacker trying to recover P_1. Using the same notations, there is a way for the attacker to "guess" that the value of P_1 is equal to some $P^\star$ and check the validity of her guess. Indeed, after the two records have been sent, she knows C_5 will be the next record IV. Thus, she sends $P_6 = P^\star \oplus C_5 \oplus C_0$ as the next plaintext block, and observes $C_6 = E(P_6 \oplus C_5) = E(P^\star \oplus C_0)$. If the guess was correct (i.e. P_1 is $P^\star$), $C_6 = C_1$, which is observable. The corresponding encryption steps are shown in figure 2.3.

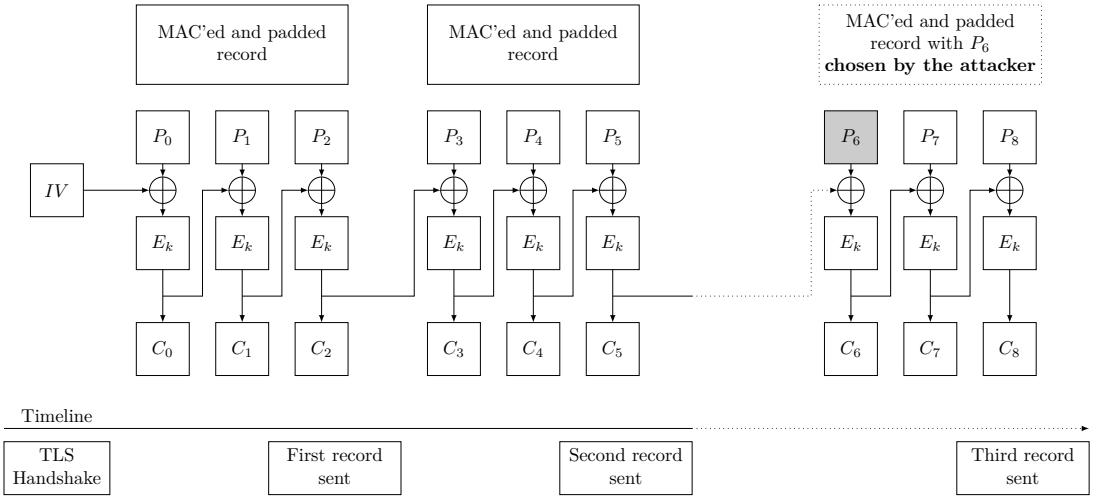


Figure 2.2: CBC with implicit IV in SSL/TLS before TLS 1.1: IV is generated during the Handshake, then all records are encrypted as a continuous CBC flow. After two legitimate records, the attacker chooses a plaintext record to encrypt starting with a chosen block (in gray) P_6.

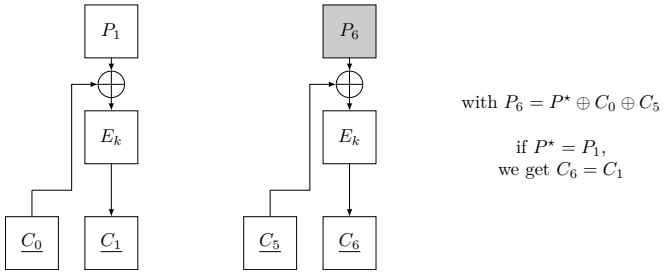


Figure 2.3: Knowing the ciphertext blocks (underlined in the schema), it is easy to check whether $P_1 = P^\star$. If this is the case, the two applications of E_k result get the same input and produce the same output, that is the attacker can observe that $C_6 = C_1$.

Furthermore, to avoid having to guess a whole block, the BEAST proof of concept cleverly aligns the blocks so that the block to guess only contains one unknown byte. For instance, if the attacker knows P_1 is "`:SESSION_TOKEN=?`" where `?` is unknown, she only needs at most 256 attempts (128 on average) to recover the byte, or even less, if the targeted byte belongs to a constrained charset.

Finally, to send the chosen plaintext data in the first block, the Same Origin Policy (SOP) [Bar11b] must be bypassed. Due to the complexity of the web ecosystem, vulnerabilities allowing SOP violations are regularly found on standard browsers and web applications.

Hypotheses and prerequisites

CBC. The connection uses CBC mode.
Implicit IV. CBC uses an implicit IV.

Passive network attacker. The attacker is able to observe encrypted packets.

Plaintext injection. The plaintext is partially controlled, adaptively.

Secret repetition. Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Use TLS 1.1. This is the long term solution, since TLS 1.1 introduces an explicit (and unpredictable) IV for each record. Yet, TLS 1.0 is still the most recent version supported by servers, as

shown in chapter 5[2]. What is worse, to accommodate broken implementations, up-to-date browsers use a fallback strategy when a TLS connection fails, and retry using older versions (TLS 1.0 or SSLv3)

Use TLS 1.2 AEAD suites. AES-GCM or AES-CCM are not vulnerable to this attack, but they are only available with TLS 1.2. Even if the proportion of TLS 1.2-compatible servers has recently grown, this protocol version only represents around half of the deployed servers in 2015, and may still be subject to a downgrade attack.

TLS_FALLBACK_SCSV. To block downgrade attacks when a browser tries to fall back to an earlier version, there is a mechanism using a fake signaling ciphersuite [ML15]. Yet, legacy stacks would still be at risk.

Use RC4. This is an efficient way to counter *this* attack, and it can be deployed easily and reliably (but section 2.1.5 shows this is not an overall acceptable solution).

Split the records. By splitting each record to send into two records, the first one containing the first byte of the original record, and the second one the remaining data, the attack still works, but only on the first byte. The second record is indeed encrypted with a *randomised* IV. This so-called "$1/n-1$ split" is efficient and implemented in major browsers[3].

Fix SOP violations and XSS bugs. To mount the attack, forged requests must be sent to the target, either by bypassing the SOP or by exploiting a Cross Site Scripting (XSS) vulnerability. It is obviously desirable to fix all these bugs, but the ever-evolving web ecosystem makes this goal difficult to reach. Moreover, TLS security should not depend on it.

2.1.2 CRIME and TIME: client-side compression

In 2012, Rizzo and Duong published another attack against TLS named CRIME (Compression Ratio Info-leak Made Easy) [RD12]. Again, their objective was to recover a secret cookie. The attack is based on the compression step in the Record Protocol and assumes the attacker is able to choose part of the cleartext, e.g. the URL path; furthermore, the attacker is able to observe the ciphertext length by capturing encrypted records. The following year, another research team presented the TIME (Timing Info-leak Made Easy) attack [BS13], a variant of the CRIME attack, relying on a different feedback method.

To explain these attacks, let's assume the secret cookie, `SESSION_ID`, is an hexadecimal string, and that the attacker can trigger successive HTTPS connections while controlling part of the cleartext (typically the URL path). This does not violate the Same Origin Policy, and the resulting HTTP requests will contain both the forged URL (`www.target.com/SESSION_ID=X` in our example, `X` being an hexadecimal character chosen by the attacker) and the cookies corresponding to the target. When these requests are compressed, the redundancy will be maximum when the attacker has guessed the secret correctly, which should result in a better compression. In some cases, better compression will be observable, since the compressed record will get smaller[4].

CRIME and TIME propose two different methods to observe the impact of compression on the plaintext. CRIME simply relies on the encrypted packet sizes, assuming the attacker is able to capture the victim's traffic. TIME uses a different feedback method: the variation of transmission time between a correct guess (where compression is more efficient) and an incorrect one. To amplify the effect of a compressed plaintext being one byte shorter, the idea behind TIME is to forge a plaintext such that the encrypted packet just crosses the TCP window size and requires a TCP ACK from the server before sending the remaining byte. This way, when the attacker guesses the correct character, the compression kicks in and the encrypted data contains one byte less, which does not require to wait a Round Trip Time for the ACK. This difference in timing is observable from the client-side script launching the requests, which does not require the attacker to observe encrypted network packets.

Hypotheses and prerequisites

TLS compression. TLS compression is activated.

[2]This statement is hopefully becoming less true eveyday.

[3]Initially, a "$0/n$ split" had been implemented in OpenSSL, but it proved to break several implementations, despite empty ApplicationData records being licit.

[4]To this aim, several parameters need to be tuned, like the block alignment (with CBC encryption) and a way to reset the compression dictionary (the main compression algorithm in TLS, Deflate [Deu96], is stateful).

Passive network attacker. The attacker is able to observe encrypted packets.

Packet length observation. Alternatively, the attacker is able to infer the ciphertext length, e.g. via timing leaks.

Plaintext injection. The plaintext is partially controlled, adaptively.

Secret repetition. Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Disable TLS compression. This blocks the attack completely, and has no significant impact on performance.

Randomise the packet length. Several proposals were made to add random-length padding to application messages (adding random bytes or slicing the HTTP content in chunks). Unless the added data is significant enough (which is equivalent in practice to disabling compression), these proposals essentially force the attacker to collect more data, but do not fundamentally invalidate the attack.

Restrictions on cross-site requests. If cross-site requests were forbidden, the attacker would first need to exploit an XSS to mount this attack, but many web applications would also break. Another approach would be to exclude sensitive information like authentication cookies from cross-site requests; such a mechanism, called First-Party Cookies, is currently studied at the IETF [WG16].

2.1.3 TIME and BREACH: server-side compression

In the article describing the TIME attack, in addition to targeting client-side compression, the researchers proposed to target server-side secret information repeatedly sent on the wire. A typical example of such secrets is anti-CSRF tokens.

Cross-Site Request Forgery (CSRF) is a well-known class of HTTP attacks, where we assume the victim is browsing both a legitimate site S and a malicious site M. We also assume the victim has an authentication cookie for server S. Then, a CSRF is a request triggered from site M towards S. Because of the Same Origin Policy, there will be limitations on the type of requests M can send (and the attacker will never have access to the data retrieved); yet the attack consists in a one-shot *authenticated* request to server S. Depending on the web application hosted by S, such a request may lead to an authenticated action server-side.

Such attacks are usually blocked by having the server insert a token in forms, which is later checked on form submission. Only server-side actions associated with a valid token will be performed. These anti-CSRF tokens are usually associated to a session, which means that they are reused for a given user during a certain amount of time.

Server-side messages can be compressed using two different mechanisms. In addition to the aforementioned TLS compression, the server can also use HTTP compression. Both forms of compression can be targeted, as soon as the attacker can inject attacker-controlled data in the server answer (preferably close to the targeted token in the payload). The attack on TLS compression was presented in the TIME attack, whereas the HTTP compression is exploited by the BREACH attack [PHG13][5]. Even if the original BREACH attack only targeted RC4, it can be extended to TLS connections using blockciphers as well, which was shown in a recent presentation, along with other improvements [ZK16]. All these attacks aim at retrieving the anti-CSRF token sent by the server.

Hypotheses and prerequisites

TLS compression. TLS compression is activated.

HTTP compression. Alternatively, HTTP compression is activated.

Passive network attacker. The attacker is able to observe encrypted packets.

Packet length observation. Alternatively, the attacker is able to infer the ciphertext length, e.g. via timing leaks.

Plaintext injection. The plaintext is partially controlled, adaptively.

Secret repetition. Multiple connections containing the same secret can be triggered.

[5] It is however worth noting that both forms of server-side compression attacks had been devised in the original CRIME presentation.

Proposed countermeasures

Disable TLS and HTTP compression. This measure blocks the attack. TLS compression can be (and has been) disabled, but HTTP compression is essential to reduce bandwidth and disabling it would drastically increase the size of HTTP responses (for example, disabling HTTP compression on Wikipedia articles leads to answers up to 6 times larger).

Use different compression contexts for sensitive and attacker-controlled data. In the new HTTP/2 standard [BPT15], a new compression algorithm, HPACK [PR15], has been specified, which partially tackles the issues by using different contexts for headers and body. Another possibility would be to use different servers/origins to handle secrets and attacker-controlled content. Such countermeasures rely on significant changes in the application layer.

Selectively disable HTTP compression. To mount the server-side compression attacks, the attacker has to trigger cross-site requests, which will lead to a `Referrer` header corresponding to an origin different from the accessed site. Disabling compression for such requests would block the attack, while keeping performance at its best most of the time (most of the requests are expected to come from the same origin).

Randomise the packet length. Several proposals were made to add random-length padding to application messages (adding random bytes or slicing the HTTP content in chunks). Unless the added data is significant enough (which is equivalent in practice to disabling compression), these proposals essentially force the attacker to collect more data, but do not fundamentally invalidate the attack.

Single-use tokens. Changing the token value for each request would block the attack but it also requires an important change in the way anti-CSRF tokens usually work.

2.1.4 Lucky 13: CBC padding oracle

In TLS, when a blockcipher is used with CBC, the plaintext is MAC'ed then padded and encrypted, which allows for attacks exploiting padding oracles, first introduced by Vaudenay in 2002 [Vau02]. As soon as an attacker can distinguish between a MAC error and a CBC padding error, be it through an out-of-band message or a timing difference, she can gain information about the plaintext.

When decrypting a CBC-encrypted record, the recovered plaintext should end with a valid padding: p bytes all valued $p-1$ (for example, blocks ending with 00, 01 01, 02 02 02, etc. are correctly padded). Let $P = p_0 p_1 \cdots p_{n-1}$ be a plaintext block, and $C = c_0 c_1 \cdots c_{n-1}$ be the corresponding ciphertext (see figure 2.4, on the left side). To guess the value of p_{n-1}, the attacker can send a fake ciphertext containing two blocks: $C^\star C$, with C being the ciphertext block to recover and $C^\star$ a random block, ending with $c^\star_{n-1} = g$ (the decryption of the second block is described in figure 2.4, on the right side).

If the guessed byte g is equal to $p_{n-1} \oplus IV_{n-1}$, the output of E_k^{-1} will end with a null byte, the padding will be correct, and this will lead to a MAC error (since the attacker can not create a valid MAC). Otherwise, if the guess is incorrect, the padding will very probably be incorrect[6].

If the attacker can distinguish between MAC errors and CBC padding errors, she can use the resulting padding oracle to guess the content of a block, one byte at a time. Indeed, once the attacker has identified the last byte p_{n-1}, she can forge a new $C^\star$ block ending this time with $(g \oplus 01)|(p_{n-1} \oplus 01)$. In case of a MAC error (instead of a padding error), she will know that the decrypted plaintext ends with 01 01. She can thus infer that $g = p_{n-2} \oplus IV_{n-2}$. The attack can be led further to recover the remaining bytes with 128 requests on average for each byte.

The initial specifications of SSL/TLS states that both error cases (invalid MAC and padding error) should lead to different alert messages. However, this is not directly useful from the attacker point of view, since these alert are always encrypted. Another way to differentiate the two error cases is to measure the time needed to reject the invalid packet. When no MAC is performed, the answer is returned faster. That is why TLS 1.1 [DR06] contains a note stating that *implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct.* Moreover, such attacks were initially considered impractical against TLS since modified records would eventually trigger a MAC error, be rejected, and cause the whole session to close.

In 2012, Paterson et al. studied the applicability of this attack to DTLS [PA12]. Datagram TLS (DTLS) [RM12] is a cryptographic protocol similar to TLS relying on UDP instead of TCP; since UDP

[6] To be precise, there is a 2^{-16} probability to get a plaintext ending with 01 01, a 2^{-24} to get 02 02 02, and so on. To eliminate those false positives, the attacker can simply repeat the operation with another $C^\star$ block where $c^\star_{n-2}$ differs.

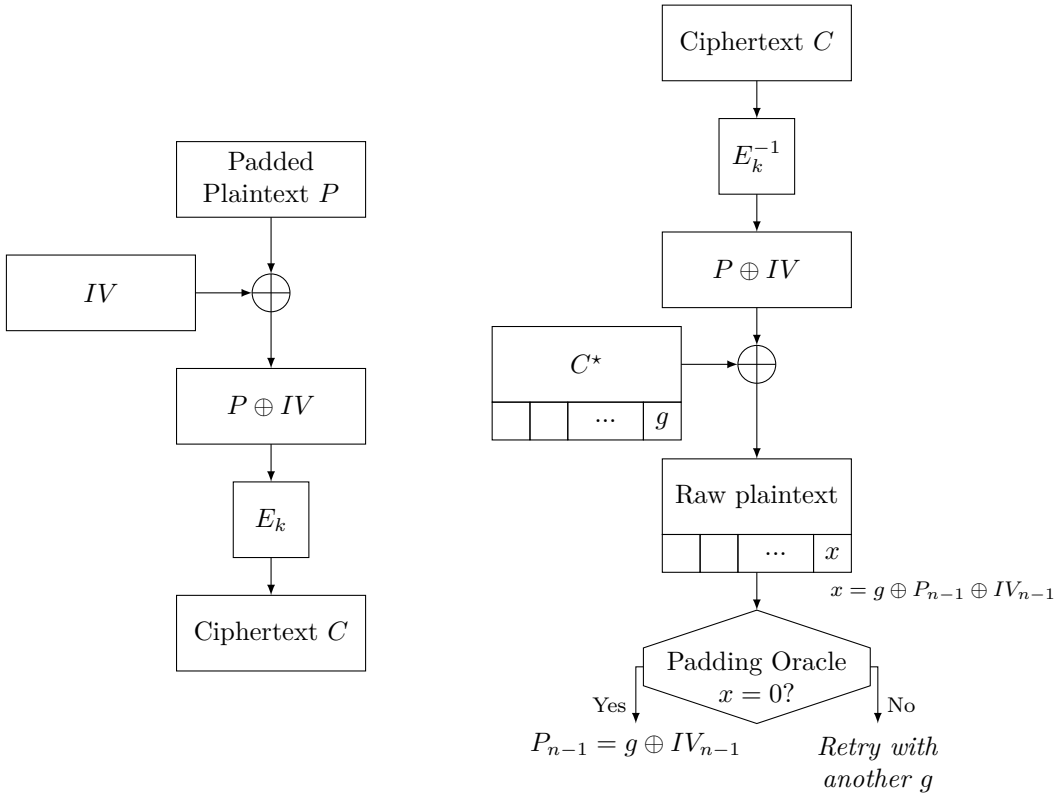


Figure 2.4: CBC encryption and decryption, in the light of a padding oracle exploitation. C, $C^\star$ (and in particular its last byte g) and IV are known. In presence of a padding oracle, the attacker can discriminate a valid padding (where $x = 0$) from an invalid one, and she can recover the plaintext, one byte at a time.

is not a reliable transport layer, datagrams may be lost or corrupted. Furthermore, DTLS does not close a session when a MAC error is encountered (nor does it emit a warning). The authors identified a timing attack in OpenSSL implementation that made it possible to distinguish a padding error from a MAC error (to amplify the timing info-leak, several identical consecutive packets had to be sent on the wire): they had found a padding oracle, which allowed them to recover arbitrary plaintext.

The next step was to adapt the technique to TLS. In case of a padding error, the standard implementation of TLS CBC decryption assumes a fixed-length pad, which, according to the implementation note quoted earlier, *leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable.* It was established in 2013 by the same team that this small info-leak was in fact exploitable in TLS to obtain a padding oracle [AP13]. Moreover, when the target is a constant secret string repeatedly sent in the encrypted channel, it does not matter that the TLS session is closed and that the keying material changes across sessions. The proof of concept was named Lucky 13, after the size of the pseudo-header MAC'ed with the message.

Hypotheses and prerequisites

CBC. The connection uses CBC mode.

Padding oracle. CBC decryption leads to a padding oracle, e.g. a timing info-leak.

Active network attacker. The attacker is able to intercept and modify packets.

Secret repetition. Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Add random delays to CBC-mode decryption. This would only increase the complexity of the attack (requiring more samples to compute the mean value).

Implement constant-time MAC-then-Encrypt. Such implementations counter the attack, but the code needed to obtain efficient record processing in effectively constant time is complex (the OpenSSL patch is almost 300 lines long).

Use RC4. This is an efficient way to counter *this* attack, and it can be deployed easily and reliably (but section 2.1.5 shows this is not an overall acceptable solution).

Use TLS 1.2 AEAD suites. AES-GCM or AES-CCM are not vulnerable to this attack, but they are only available with TLS 1.2. Even if the proportion of TLS 1.2-compatible servers has recently grown, this protocol version only represents around half of the deployed servers in 2015, and may still be subject to a downgrade attack.

Switch to Encrypt-then-MAC. This would solve the problem, but the specification [Gut14] is still young. Moreover, it relies on an extension, which would lead to the same kind of deployment and reliability issues as TLS 1.2.

2.1.5 RC4 biases

RC4 is a stream cipher designed by Rivest in 1987. It is very simple to implement and has very good performance in software. It has thus been widely adopted in protocols (Wifi encryption protocols WEP and WPA, or TLS for example). Since 1995, several statistical biases have been identified in the first bytes of an RC4 keystream. These flaws eventually led to very efficient attacks against WEP [SIR02].

As these attacks rely on initial biases of the keystream, it was proposed to drop the first n bytes of the keystream, but later findings show the existence of additional statistical biases, even after the initial bytes [FM00]. In 2013, two research teams presented practical attacks against the encryption of the same fixed sequence of plaintext using large numbers of different keys [IOWM13, ABP$^+$13], which apply to HTTPS cookies. More recent publications have been improving these results [Man15, VP15, GPvdM15, BMPvdM16], leading to attacks with realistic complexities.

Here is a short description of such an attack on RC4, presented in the article by AlFardan et al. [ABP$^+$13]. Their single-byte bias attack relies on the fact that the first 256 bytes of the keystream are strongly biased. The researchers generate a lot of RC4 keystreams to observe the actual distribution of each of the 256 first bytes. Using an empirical reference of 2^{45} keystreams, it is possible to recover the first 256 bytes of a plaintext, as soon as it is encrypted a sufficient number of times; this number varies from 2^{24} to 2^{32} as a function of the byte position in the 256 bytes keystream. Using the reference distribution and the encrypted distribution, the idea is to find the most probable byte value, by measuring the distance between the reference distribution and each of the candidate keystreams using a likelihood function.

Figure 2.5 shows an example of how to recover a particular byte using this method, knowing that the 50th plaintext byte is either "`A`", "`X`" or "`.`"[7]. The first graph represents the reference distribution of one particular keystream byte (Z50). Gathering 2^{30} encrypted values of `A` (`0x41`) in position 50, the small graphs reconstruct what the keystream distribution would have been, assuming the plaintext was "`A`" (`0x41`), "`X`" (`0x58`) or "`.`" (`0x2e`). As the first one is closer to the reference distribution (peaks are aligned), it is the best candidate.

However, the attack is hard to implement, since it requires a lot of different TLS connections, and only works for data sent in the first few bytes. Moreover, the very first encrypted bytes correspond to the `Finished` message, pushing the interesting application data at least beyond the 36th byte. To overcome these limitations, the researchers also use long-term biases described by Fluhrer and McGrew [FM00] on consecutive bytes to perform a practical attack requiring more keystream, but which could work in a pipelined HTTPS stream (i.e. using different messages within the same TLS connection). This double-byte bias attack is more practical than the single-byte bias one, and a proof of concept was developed to recover an HTTP cookie. It has even been improved to attacks recovering a secure cookie with a success rate of 94 % using around 2^{30} ciphertexts, obtained in 75 hours [VP15].

[7]The attack does not need such a restricted charset to work. This simple case was chosen only to present a visual example.

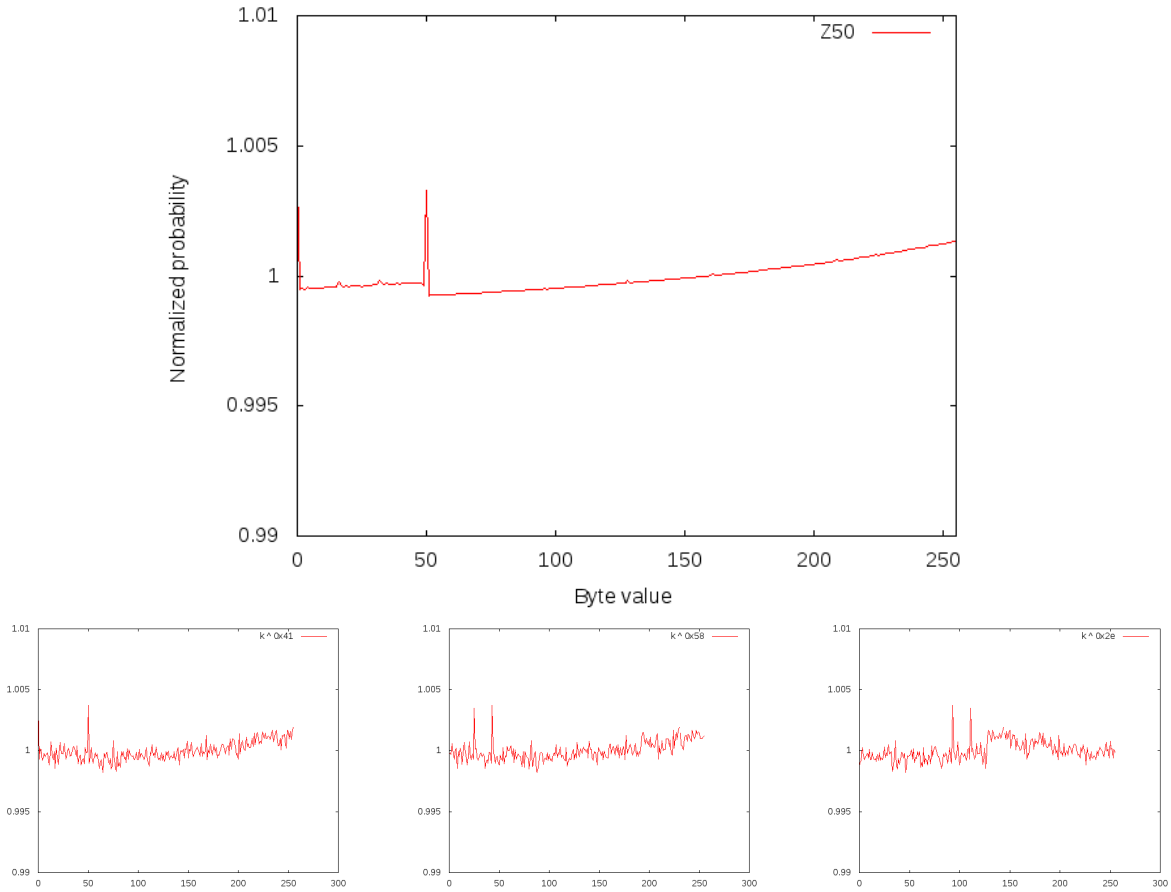


Figure 2.5: [Top] Statistical distribution of Z50, the 50th byte of the keystream. [Bottom] Statistical reconstitution of Z50 using 2^{30} ciphertexts, assuming the plaintext is either "`A`", "`X`" or "`.`".

Hypotheses and prerequisites

RC4. The connection uses RC4.

Passive network attacker. The attacker is able to observe encrypted packets.

Secret repetition. Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Use CBC mode. This is an efficient way to counter *this* attack, and it can be deployed easily and reliably. Obviously, this recommendation is in contradiction with the *Use RC4* recommendation (from section 2.1.1 and 2.1.4).

Use TLS 1.2 AEAD suites. AES-GCM or AES-CCM are not vulnerable to this attack, but they are only available with TLS 1.2. Even if the proportion of TLS 1.2-compatible servers has recently grown, this protocol version only represents around half of the deployed servers in 2015, and may still be subject to a downgrade attack.

Use another streamcipher. This could work in practice (since ciphersuite negotiation is well supported), but no alternative streamcipher has been specified as such in TLS yet. ChaCha20 [Ber08] was recently added to TLS [LCM+16] as an alternative streamcipher, but it was standardised as an AEAD suite (in combination with the MAC algorithm Poly1035).

Throw away the first bytes of the keystream. This behaviour could be specified in TLS (with a new ciphersuite or extension) or HTTP (by padding the beginning of messages), but we know exploitable long-term RC4 biases exist.

Randomise the packet length. This would at best increase the attack complexity for position-dependent biases and force the attacker to collect more data, but not fundamentally invalidate the attack.

2.1.6 POODLE: another padding oracle

In October 2014, Möller, Duong and Kotowicz presented POODLE (Padding Oracle On Downgraded Legacy Encryption) [MDK14], another padding oracle targeting SSLv3 CBC mode. The old SSL version indeed handles CBC padding in a specific way: when n bytes are needed to pad a plaintext, the last byte is set to $n-1$ (as in TLS, described in section 2.1.4), but the other bytes can take any arbitrary value. An attacker can use this liberty to get a padding oracle.

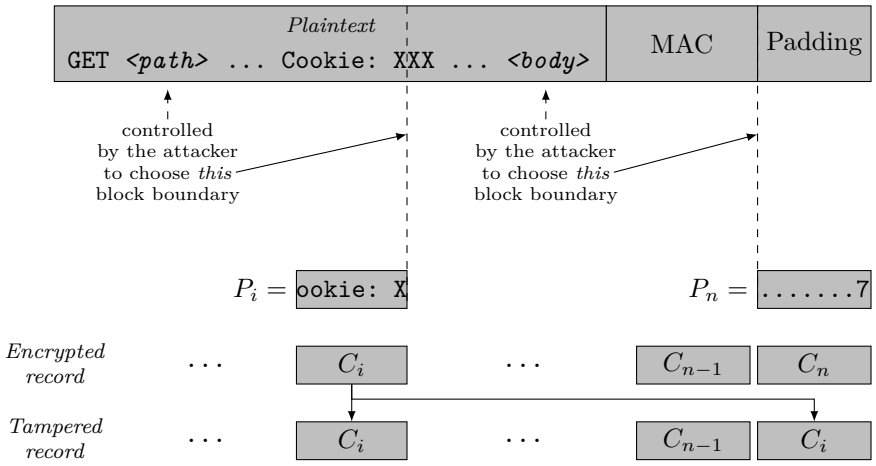


Figure 2.6: POODLE attack exploiting SSLv3 CBC Padding, assuming an 8-byte blockcipher.

Let us assume an active attacker can trigger requests to the vulnerable site using SSLv3 and CBC mode. Since she may alter the path fragment in the URL, she may prepare the request in such a way that the cookie begins just before a block boundary. Moreover, she may include a request body of arbitrary length after the headers, which allows her to craft a plaintext message (once the MAC is appended) whose length is a multiple of the block size (see figure 2.6). This way, a whole block would be added in the padding phase. Such a block has only one constraint: its last byte must be $n-1$ where n is the block length.

Once the request is sent by the browser, the attacker needs to modify the record on the wire. She must replace the all-padding block by the block where the last byte is to be guessed, as shown in the figure. If the decryption of the last blocks leads to the correct value $(n-1)$, the rest of the padding block is ignored, the original MAC is correctly compared to the computed value, and the record is accepted by the server. On the contrary, if the padding does not end with $n-1$, the decryption will lead to a MAC error and to the end of the connection.

So, in the absence of error, the attacker will learn that the last byte of $E_k^{-1}(C_i) \oplus C_{n-1}$ is $n-1$, and that the last byte of P_i is $C_{n-1} \oplus C_{i-1} \oplus (n-1)$. If the attacker retries, another key will be used and C_{n-1} will be randomised. Thus, each byte can be guessed with a 2^{-8} probability, which results in 256 requests needed on average to recover each byte of the secret value[8].

As a side note, it is interesting that this attack relies on the browser using a fallback strategy, and on the server accepting the obsolete SSLv3 version of the protocol.

Hypotheses and prerequisites

CBC. The connection uses CBC mode.
SSLv3 padding. CBC decryption checks the padding as specified in SSLv3.

[8]Contrary to BEAST and Lucky 13, the attacker can not choose the value she wants to test, since C_{n-1} is randomised and cannot be controlled. This is why the probabilities are different.

Plaintext injection. The plaintext is partially controlled, adaptively.

Active network attacker. The attacker is able to intercept and modify packets.

Secret repetition. Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Use TLS 1.0. Since this powerful padding oracle is only present in SSLv3, forbidding this deprecated version is an efficient countermeasure. Moreover, only a small portion of the internet still relies on this version of the protocol, which makes this measure also practical.

Use RC4. This is an efficient way to counter *this* attack, and it can be deployed easily and reliably (but section 2.1.5 shows this is not an overall acceptable solution).

Use TLS 1.2 AEAD suites. AES-GCM or AES-CCM are not vulnerable to this attack, but they are only available with TLS 1.2. Even if the proportion of TLS 1.2-compatible servers has recently grown, this protocol version only represents around half of the deployed servers in 2015, and may still be subject to a downgrade attack.

Switch to Encrypt-then-MAC. This would solve the problem, but the specification [Gut14] is still young. Moreover, it relies on an extension, which would lead to the same kind of deployment and reliability issues as TLS 1.2.

TLS_FALLBACK_SCSV. To block downgrade attacks when a browser tries to fall back to an earlier version, there is a mechanism using a fake signaling ciphersuite [ML15]. Yet, legacy stacks would still be at risk.

Split the records. Opera and Google proposed to split SSLv3 CBC records to counter the POODLE attack. The proposed method is supposed to avoid whole blocks of padding. Yet, POODLE paved the way for new SSLv3 padding oracle attacks, which may not easily be blocked this way.

It is worth noting that TLS 1.0 implementations were shown to be vulnerable to POODLE: they did not correctly check the padding as specified in the RFC. The matter is further discussed in section 7.3

2.1.7 Attack summary

Table 2.1 summarises the recent attacks against the Record Protocol described in this section. The first part shows the hypotheses and prerequisites to mount the attack, while the second part presents the proposed countermeasures and their applicability to thwart each attack. Concerning plaintext injection, it is worth noting that the attacker needs to control the first block of a record to mount the BEAST attack. This requires in practice to bypass the Same Origin Policy, whereas she only needs to inject headers or elements in the URL for Lucky 13 or compression-related attacks. This difference is highlighted by a double check mark for BEAST.

2.2 Attacker model and the masking principle

The legitimate actors we consider are: the user agent (e.g. Firefox or Chrome), the HTTP(S) server (e.g. Apache) and the web application (e.g. a program written in PHP or Python). The web application may rely on a framework designed to abstract the inherent complexity of web development (e.g. Django or Zend). The attacker we consider is an active network attacker, able to read, modify or delete packets between the client and the server. Figure 2.7 illustrates these actors. We also assume, as for each of the attacks presented in section 2.1, that a secret cookie is repeated across different TLS messages.

Given a TLS session, we assume the attacker is able to retrieve some information about κ consecutive bytes of the corresponding plaintext. Typically, $\kappa = 1$, and the attacker is able to check whether a cleartext byte is equal to a guessed value. Thus, by repeating the attack on constant plaintext bytes, she can recover this part of the plaintext.

To draw a parallel with side-channel attacks [CJRR99, PR13], such attacks may be called first order attacks. A typical countermeasure is to mask the secret value: each time a secret s of κ bytes must be transmitted, a random value m (the mask) of the same length is drawn and the pair $(m, m \oplus s)$ is sent instead of s. The value can trivially be recomputed by the other party, but the representation on the wire is different for every message. If the secret s to mask is longer than κ, s can be split in κ-byte

HYPOTHESES AND PREREQUISITES

	Beast	Lucky 13	Poodle	RC4	Crime	Time	Breach
TLS features							
CBC	✓	✓	✓				
Implicit IV	✓						
Padding oracle		✓					
SSLv3 padding			✓				
RC4				✓			
TLS compression					✓	✓	
HTTP compression							✓
Attacker model							
Passive network attacker	✓			✓	✓	✓	✓
Packet length observation					(or ✓)	(or ✓)	(or ✓)
Active network attacker		✓	✓				
Plaintext injection	✓		✓		✓	✓	✓
Environment							
Secret repetition	✓	✓	✓	✓	✓	✓	✓

PROPOSED COUNTERMEASURES

	Beast	Lucky 13	Poodle	RC4	Crime	Time	Breach
TLS features							
Use TLS 1.0			✓				
Use TLS 1.1	✓		✓				
Use TLS 1.2 AEAD suites	✓	✓	✓	✓			
TLS_FALLBACK_SCSV	detect version downgrades						
Use CBC mode				✓			
Use RC4	✓	✓	✓				
Use another streamcipher				✓			
Switch to Encrypt-then-MAC		✓	✓				
Disable/adapt TLS compression					✓	✓	
Disable HTTP compression							✓
Low-level implementation tricks							
Throw away part of the keystream				✓			
Split the records	✓		✓				
Constant-time MAC-then-Encrypt		✓					
Noise addition							
Randomise the packet length				increased complexity			
Randomise CBC decryption time		more complexity					
HTTP-level measures							
Single-use anti-CSRF tokens							✓
Fix SOP violations and XSS bugs	✓						
Restrictions on cross-site requests					✓		

Table 2.1: Summary of the recent attacks against the Record Protocol.

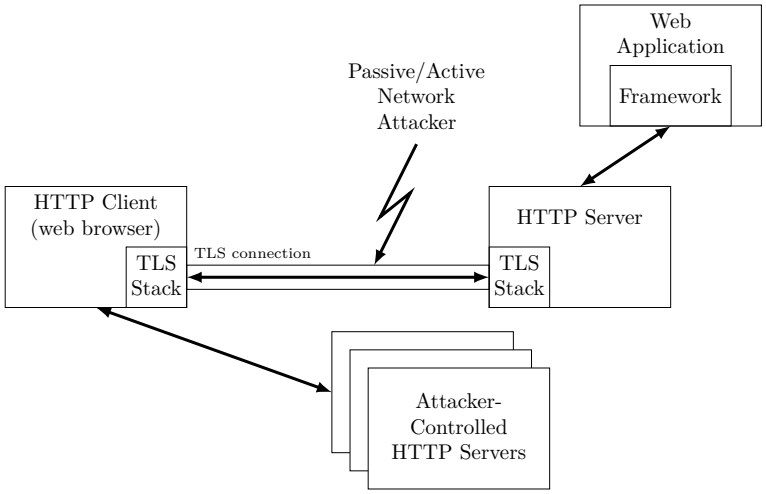


Figure 2.7: The different entities involved in our model.

words, masked by the same mask. Alternatively, it is possible to choose a longer mask to cover the secret entirely.

Masking all secrets using a fresh mask would force the attacker to mount a second order attack, that is to find a way to simultaneously retrieve information about the mask and the masked value, to learn something about the secret.

Most of the attacks described in section 2.1 are designed to recover the plaintext one byte at a time, which makes them first order attacks with $\kappa = 1$. Other attacks against RC4 also exploit statistical biases on two consecutive bytes, which also corresponds to our model (for $\kappa = 2$).

In practice, the BEAST, CRIME, TIME, BREACH and Lucky 13 attacks could easily be extended to guess κ consecutive bytes at once (for example, for BEAST, this would mean aligning the boundary of the block to guess differently). However, the complexity to recover κ bytes at once would be proportional to $2^{8\kappa}$ (instead of $\kappa \cdot 2^8$), which limits κ to small values in practice. To be conservative, we consider the maximum number of recoverable successive bytes κ to be 8.

Overall, the recent attacks against the TLS Record Protocol can all be considered as first order attacks (with $\kappa \leq 8$). So, masking secret values using unique 8-byte random strings will mitigate these attacks. In the following sections we present two implementations of this concept applied to the transport and the application level.

Concerning RC4, it can be noted that other known biases exist and are related to distinct distant groups of keystream bytes. Thus, second order attacks against RC4 might be possible by exploiting such biases. We briefly discuss this case at the end of this chapter.

2.3 Proposed mechanisms

This section presents generic mechanisms to mitigate the impact of TLS security flaws, by leveraging the masking principle. The first one acts at the transport (TLS) layer, while the second and third one work at the application (HTTP) level.

2.3.1 TLS Scramble: Masking at the TLS level

The idea of masking application data at the transport level is not new in TLS. During the specification of WebSockets [FM11], a recent HTML5 feature, a randomisation step was added to avoid confusion between WebSocket traffic and other protocols, that could be leveraged by an attacker. WebSocket randomises client-to-server traffic using 4-byte long masks. An interesting side effect of this change was to block the early version of the BEAST attack, forcing Duong and Rizzo to rewrite their exploit using Java instead of WebSockets. As shown in figure 2.8, only the TLS stacks need to be modified to implement TLS Scramble.

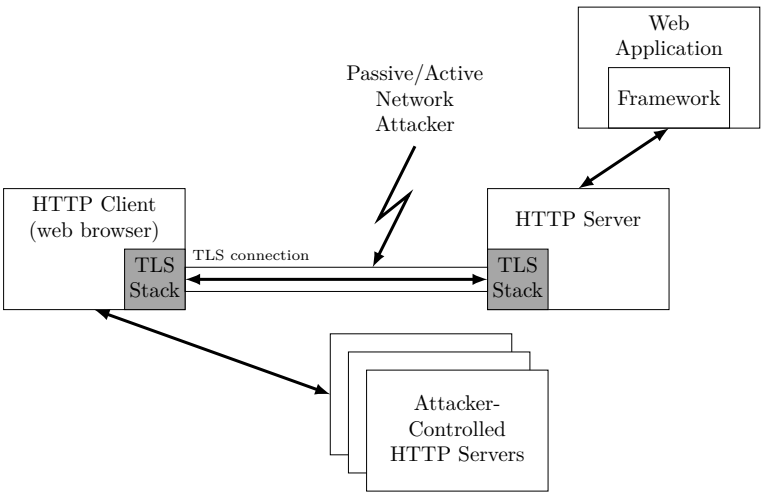


Figure 2.8: Changes needed to implement TLS Scramble in our model.

A fake compression method: Scramble

As shown in figure 2.1, record processing may optionally compress the plaintext before the cryptographic transformations. This step takes a plaintext record of at most 2^{14} bytes, and produces a compressed record that can be at most 1024 bytes longer than the plaintext. Only three compression methods have been specified in TLS: NULL (no compression), DEFLATE (the popular compression algorithm also used by `gzip`) and LZS.

To generalise the idea behind WebSockets masking, we define a *fake* compression algorithm, Scramble. Given a κ parameter (the mask length) and a plaintext P, the way the `scramble_record` function *compresses* P is the following:

- a κ-byte random string m is generated;
- m is repeated, and possibly truncated, to be as long as P. The result is a masking string M;
- the *compressed* record is $m|P \oplus M$, which is exactly κ-byte longer than P.

The `unscramble_record` operation is straightforward:

- on receiving a *compressed* string c, which should contain at least κ bytes, the first κ bytes of c are extracted: they represent the mask m. We call X the remaining string;
- m is expanded to be as long as X to obtain M as before;
- the *uncompressed* value is $M \oplus X$.

Implementation in OpenSSL

To check the feasibility of this idea, we implemented the Scramble compression method in OpenSSL (v1.0.1) with 8-byte masks. The patch affects the `crypto/comp` directory. It adds `c_scramble.c`, a 75-line file describing the method, as well as trivial changes to `comp.h` and to the corresponding `Makefile`.

The core of the Scramble compression method is the `scramble_record` function (and its counterpart `unscramble_record`), which is implemented as follows:

```
static int scramble_record(COMP_CTX *ctx,
    unsigned char *out, unsigned int olen,
    unsigned char *in, unsigned int ilen)
{
  int i;
  unsigned char mask[MSIZE];

  if (olen < (ilen + MSIZE)) return -1;

  if (RAND()->bytes(mask.bytes, MSIZE) < 0)
    return -1;
  memcpy (out, &mask, MSIZE);

  out = out + MSIZE;
  for (i = 0; i < ilen; i++)
    *(out++) = *(in++) ^ mask.bytes[i % MSIZE];

  return (ilen + MSIZE);
}
```

To test the method with real connections, we also patched `apps/s_client.c` and `apps/s_server.c` to exchange data over the scrambled channel, simply using `openssl s_client` and `openssl s_server`.

2.3.2 MCookies: Masking at the application level

Another way to tackle the problem is to mask secret values at the application level, which requires less bandwidth (only relevant elements would need to be masked) and avoids modifying TLS stacks. In this section, we propose a method to mask cookies at the HTTP level. Figure 2.9 shows the changes required to the different entities.

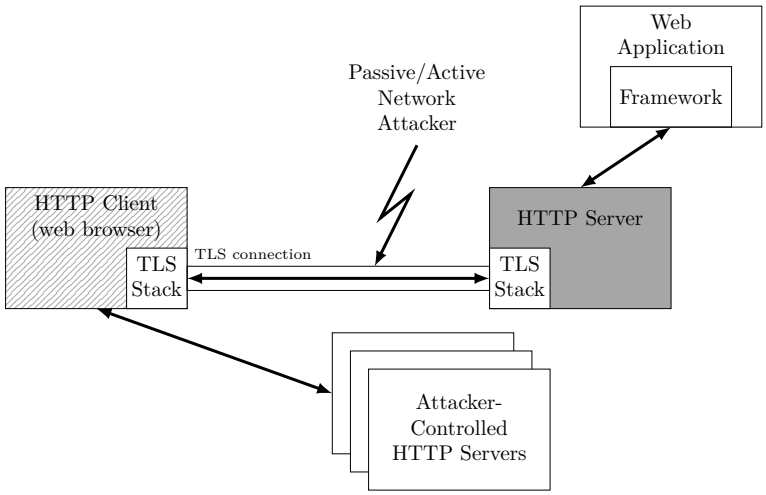


Figure 2.9: Changes needed to implement MCookies in our model. Depending on the implemented version, client support may be required.

MCookies principle

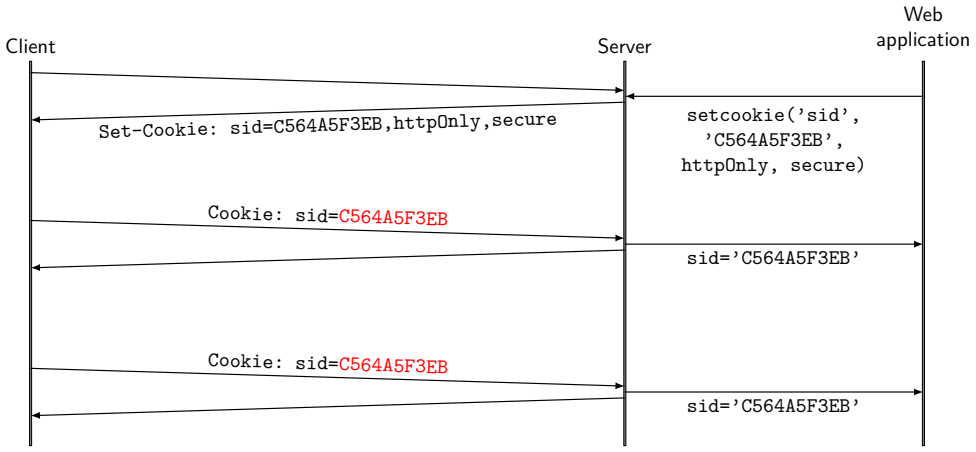


Figure 2.10: HTTP cookies: definition and restitution.

Usually, HTTP cookies work as specified in figure 2.10: a server can define cookies to be stored by its client, then each time this client sends a request to the server, the cookies are added to the headers [Bar11a]. Sometimes, they may also be read and modified by client-side scripts.

If we only consider cookies that are never read nor modified by client-side scripts, there is a simple way to break this repetition while modifying only the HTTP server, which is described in figure 2.11:

- **Cookie definition:** when the web application defines such a cookie (e.g. by calling the function `set-cookie(SESSID, V)`), the HTTP layer generates a fresh mask M and rewrites the

`Set-cookie` header to send[9] `SESSID=`$M:M \oplus V$ instead of `SESSID=`V.

- **Cookie restitution (and redefinition):** for each request containing a `SESSID=`$X:Y$ cookie, the HTTP server transmits `SESSID=`$X \oplus Y$ (the unmasked cookie) to the web application. Then, three cases may arise:
  - the web application updates the cookie, which is covered by the Cookie definition step;
  - the web application can erase the cookie by setting an outdated expiration time, in which case the HTTP layer simply transmits the header as is;
  - otherwise (the cookie is left unchanged by the application), the HTTP server sets a new version of the cookie, M':$M' \oplus V$, that is the same intended value, masked using a fresh random mask.

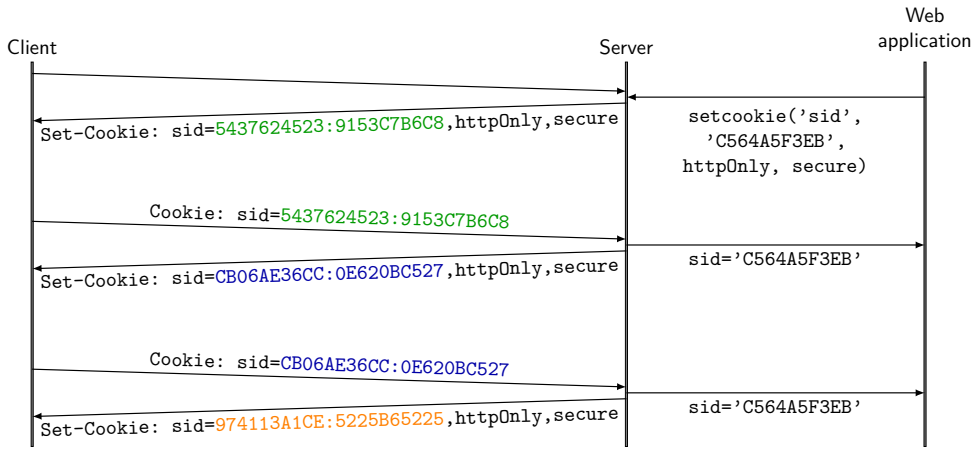


Figure 2.11: Definition and restitution of MCookies.

Discussion of MCookies feasibility

Since MCookies modify the concrete value of the exchanged cookies, we must be careful to only select sensitive cookies that can be modified at the HTTP layer. To this aim, a simple way would be to define a static list of cookie names. But a natural heuristic is to protect every cookie flagged both `httpOnly` and `secure`. Only considering `httpOnly` cookies guarantees that client-side script have no access to the cookie value, leaving the HTTP server free to change the cookie representation at will. However, from the web application point of view, the cookie value sent and received remains the same. Moreover, protecting a non-`secure` cookie is pointless as this one can be easily stolen with our attacker model in general.

Rewriting the cookies on every request has negative consequences. First, it adds extra-bandwidth to the server messages. Second, the original cookie attributes (`Expires`, `Max-Age`, `Domain`, `Path`) are lost in the process. These attributes would need to be specified at each redefinition to keep application cookies consistent. To fix this problem, the attributes from the original `Set-cookie` header are encoded inside the masked cookie: a sensitive cookie V with attributes A would be transmitted to the client as $M:M \oplus V:A$. It would thus be possible to remember the correct attributes for each request[10].

Optionally, it may be useful to add a MAC, covering M, $M \oplus V$ and A. This way, cookie forging could be detected at the HTTP layer. Consequently, the MCookie associated to a cookie value is: `base64(Mask:MaskedValue:Domain:Path:Expire:MAC)`.

[9] Since M and $M \oplus V$ are binary strings, Base64 encoding is used, but we omit this detail in the description for the sake of simplicity.

[10] In practice, the "Max-Age" attribute cannot be transmitted as is since it is relative to the fetching time. It is thus easier to convert it to an "Expires" attribute based on the web server system clock.

Yet, by fixing the attribute problem, we amplify the bandwidth overhead. To reduce this overhead, we can use smaller representations for the attributes. For the `Domain` attribute, the server can rely on the browser to send the cookie appropriately: to distinguish a cookie belonging to the `.example.com` domain from another cookie associated to `sub.example.com` within a request sent with the `Host: www.sub.example.com` header, it is sufficient for the server to remember the number of subdomains in the domain and the presence of a starting dot. For the given example, the attached attributes would thus only become respectively `.2` and `3`. Therefore, we only need a single byte to encode domains, using the sign bit to store the presence of the starting dot, leaving 7 bits for the node count. A similar transformation can be applied to `Path` attributes using the query path. Finally, the expiration attributes can be converted into a 8-byte timestamp.

MCookies have another drawback: they cannot prevent active network attacks (such as Lucky 13). When the record packets are modified by the attacker, they are seen as corrupted records by the server TLS stack, and discarded. Since the HTTP server never receives the corresponding request, it cannot answer with a freshly masked cookie. To counter active attacks, the browser could monitor failed successive HTTPS connections for each origin and, after a given number of broken connections, erase the cookies associated to this domain. Drawbacks of this countermeasure are twofold:

- the client has to be modified to maintain this counter;
- setting a correct threshold is hard: the trigger should be effective against real attacks, but a low threshold would easily break HTTPS sessions on poor quality network connections.

To overcome MCookies limitations, we extend MCookies with a new HTTP header: `Masked-Cookie`. This extension requires a change in the browser that is now put in charge of masking cookies.

Masked-Cookie headers

In addition to the MCookies mechanism, we introduce a new `Set-Cookie` attribute, `masked` to signal the presence of masking to the client. Then, a compliant client would, for each request, send this cookie in a new header, `Masked-Cookie`, with the value here masked by the client : `Masked-Cookie: SESSID=`$M' : M' \oplus V$. Figure 2.12 describes the protocol use of this new header.

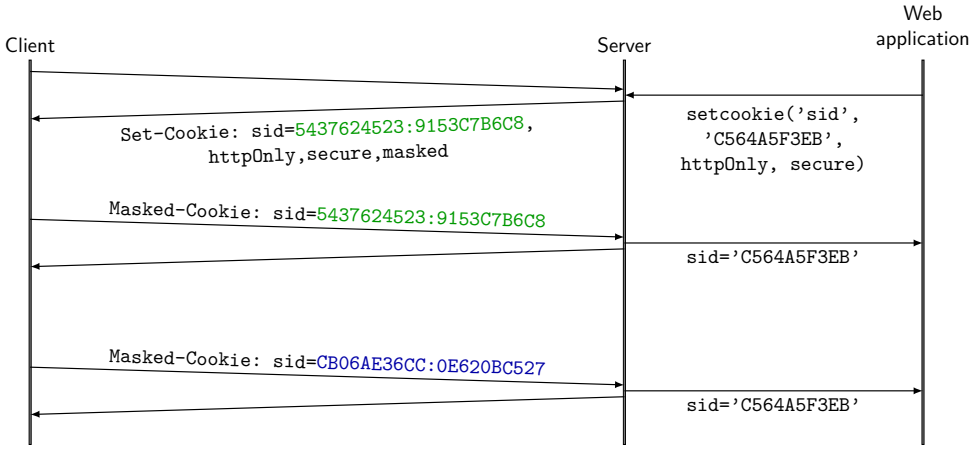


Figure 2.12: Use of `Masked-Cookie` headers: masking is done by the client.

Moving the cookie masking process to the client side renders active attacks ineffective. Faced with a compliant client, a server only needs to define MCookies once, so the extra bandwidth cost is essentially removed. On the other hand, a standard client will ignore the `masked` attribute, and the server will gracefully fall back on the previous behaviour.

Apache Implementation

Implementing MCookies is easily done as a filter module for HTTP servers. We chose Apache for our proof of concept, since it is open-source, and currently the most deployed HTTP server.

Apache exposes a powerful module system with hooks allowing to interact with the request and response processing. In order to mask the cookies sent by the server during the emission of the `Set-Cookie` header, we hook the response process using an `output_filter`, defined by `mod_filter`. To this aim, we call the `ap_register_output_filter` and `ap_add_output_filter_handle` functions. Then, to unmask the cookies received via `Cookies` headers from the client, we hook the request handling. The `input_filter` hook happening too late in the process, we use an earlier control point, when headers are parsed, using `ap_hook_header_parser`. The last step is to send a new representation of the cookie parsed in the request, along with the response; this step is easy to implement since request cookies are available from the `output_filter` hooks.

The overall code to implement the MCookies (including the Base64 code to encode the masked value safely) is around 500 lines of C. It handles both the MCookie rewriting, without compression, and the `Masked-Cookie` extension.

Masked-Cookies for Chromium

Chromium is a popular, open-source and modular web browser. We thus decided to patch it to prove the feasibility of `Masked-Cookies` headers in a real-world context. The overall C++ patch for Chromium (version 31) only counts 241 lines. It adds the `masked` attribute to the internal cookie representation, the `CanonicalCookie` class. Masking and unmasking are implemented in the `CanonicalCookie::Create` and `CookieMonster::BuildCookieLine` methods.

2.3.3 MTokens: Masking anti-CSRF tokens

To mitigate server-side attacks, the CSRF tokens (and similar objects) are easily protected using a technique close to MCookies, i.e. by masking the token with a different value for each message. The *intended* value of the token would remain the same, avoiding out-of-sync problems, while randomising the data that is sent over the network. As shown in figure 2.13, implementing this requires very small changes to web applications (or even no change at all if web frameworks are modified): it would simply amount to replacing every call to the function producing the token (which we will call `write_csrftoken()`) with `mask(write_csrftoken())`, and each call to the function getting the token from the client form (`read_csrftoken()`) with `unmask(read_csrftoken())`.

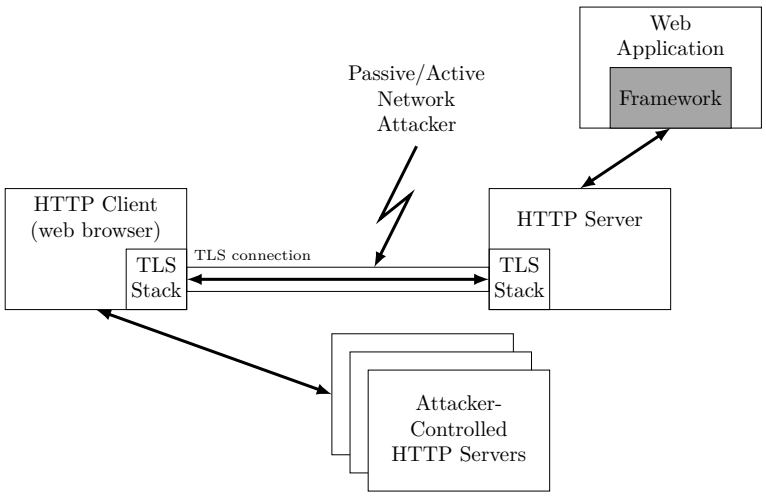


Figure 2.13: Changes needed to implement MTokens in our model.

Here is a simple implementation of how MTokens could be written in Python:

```
def mask(v):
 m = os.urandom(len (v))
 masked_v = ""
 for i in range(len(v)):
```

```
    tmp = ord (v[i]) ^ ord (m[i])
    masked_v += chr(tmp)
  m_b64 = m.encode("base64")[:-1]
  masked_v_b64 = masked_v.encode("base64")[:-1]
  res = "%s:%s" % (m_b64, masked_v_b64)
  return res

def unmask(s):
  try:
    m, masked_v = s.split(":")
    m = m.decode("base64")
    masked_v = masked_v.decode("base64")
    v = ""
    for i in range(len(masked_v)):
      tmp = ord (masked_v[i]) ^ ord (m[i])
      v += chr (tmp)
    return v
  except:
    return ""
```

It is essential to consider a robust source of random bytes. In the example, `urandom` is chosen over the standard library `random` module, which "should not be used for security purpose", as stated in the documentation.

2.4 Analysis of masking mechanisms

2.4.1 Security analysis

We now assess the efficiency and reliability of the proposed mechanisms against known attacks.

With regards to the countermeasures presented in section 2.1, the masking mechanisms described (TLS Scramble method, MCookies, `Masked-Cookies` headers and MTokens) are discussed in terms of efficiency, deployability, reliability and compatibility with HTTP (see the last rows of table 2.4).

TLS Scramble method

Specified as a new TLS compression method, TLS Scramble would require deployment efforts. Even if compression method negotiation were as reliable as ciphersuite negotiation (which would have made Scramble a reliable mechanism between compatible TLS stacks), the idea of compression in TLS is considered obsolete since the CRIME, TIME and BREACH attacks.

TLS Scramble disables real TLS compression, trivially defeating CRIME and TIME attacks. This technique is also efficient against BEAST, Lucky 13, RC4-biases and POODLE attacks, since they only recover the secret one byte at a time, and because Scramble makes this byte a moving target.

However, TLS Scrambling masks the *entire* messages, and not only the secret values, which means this method does not meet the principle described in section 2.2 *per se*. In particular, the attacks relying on mixing secret values and attacker's guess might still work, e.g. compression attacks in the application layer like BREACH.

All in all, relying on such a fake compression step in TLS does not really seem realistic from a security or a deployment point of view.

MCookies

MCookies randomise only the cookie values sent by the client, which fits exactly our masking principle. As long as secrets are masked, first-order attacks will be defeated by MCookies: this is the case for *passive* network attacks like BEAST, client- and server-side compression attacks and RC4-biases.

To be effective, MCookies require HTTP requests to reach the server and the corresponding answers (containing a freshly masked cookie) to get back to the client. Active network attackers may be able to block such answers. So MCookies are inefficient against such attackers.

All in all, apart from thwarting most of the attacks, MCookies have the advantage of requiring only a small modification of the HTTP server, leaving the browser and the web application untouched, which makes it a reliable solution to deploy.

MCookies with `Masked-Cookies` headers

As MCookies, this extended mechanism fits the masking principle. Moreover, since sensitive cookies are masked by the browser, every request will be masked differently, even in the presence of an active attacker: all the studied attacks are covered by the countermeasure.

This mechanism is even backward compatible: it is possible with the same HTTPS server to handle old and new clients, taking advantage of the new `Masked-Cookies` headers when available, but still defeating passive network attacks with older clients.

A remark about HTTP Basic/Digest Auth

Concerning client-side attacks, we only discuss cookie protection. Yet, other methods exist to authenticate the client at the HTTP layer: Basic and Digest Authentication [FHBH+99]. In practice, they are rarely used, as the user interfaces do not allow for a clean integration with modern web applications (e.g. no logout feature).

Both headers are as vulnerable as cookies with regards to the described attacks. For Basic Authentication, TLS scrambling would be as efficient as for cookies. However, it is not clear how the header could be randomised at the application layer: there can be no equivalent to MCookies, unless the mechanism is significantly modified.

For Digest Authentication however, the standard already allows for possible randomisation resembling MCookies, producing a new server nonce value for each new request. Such policy would defeat all passive network attacks, since the client would produce a different header for each request.

MTokens

As for MCookies with `Masked-Cookies` headers, MTokens are effective against server-side first order attacks against anti-CSRF tokens in general, since they literally and systematically apply our masking principle. In particular, the attacker can not force the server to replay an answer with the same mask, which also makes active attacks inefficient.

2.4.2 Performance analysis

TLS Scrambling overhead analysis

Masking plaintext data at the TLS level is easy to implement, and is completely transparent from the application layer point of view. Yet, each and every record has to be κ-byte longer, even for short messages.

Performance-wise, the CPU overhead is negligible, and, considering typical HTTPS traffic, the network bandwidth overhead is less than 1 % in bytes for $\kappa = 8$.

MCookies network overhead analysis

At application level, masking cookie values increases the HTTP requests and responses size by adding the mask and the attributes to the initial cookie value and by sending new headers. In order to quantify this overhead, we built a script that simulates MCookies on a real internet navigation traffic. It computes the overheads according to whether the web browser used supports `Masked-Cookies` headers or not.

We instrumented a web browser (Chromium) to analyse HTTP requests and responses from HTTPS secured traffic obtained during the following one day scenario. The user logs in Google, Facebook, Twitter, Dropbox, an RSS aggregator, and other popular web sites; he keeps tabs opened on multiple pages for each services, uses them during the day and browses other websites as well. 8,185 HTTP requests (122 MB) were retrieved. Among these, 4,823 requests (24 MB) were actually setting or sending sensitives cookies.

With this HTTP trace, we ran a tool to sequentially process each request and to simulate the overheads induced by installing the MCookie module on every reached host. We identify every sensitive

Traffic type	Raw traffic volume	Extra bandwidth		
		w/o UA support		with UA support
		naive	compr.	
Sensitive	24 MB	+20.1 %	+14.9 %	+10.8 %
Overall	122 MB	+4.1 %	+3.0 %	+2.2 %

Table 2.2: Network overhead evaluation.

cookie by looking at the `httpOnly` and `secure` flags. Table 2.2 describes the extra bandwidth in different situations: without User-Agent support (that is using MCookies), either with naïve or compressed encoding, and with User-Agent support (the `Masked-Cookies` headers). The results show a significant overhead on requests containing sensitive cookies. However, when looking at the big picture, the overall HTTPS traffic, this overhead turns out to be rather small. Finally, compared to the entire web communications, both HTTP and HTTPS, the cost induced by cookie masking proves negligible.

Apache module system overhead analysis

To evaluate the efficiency of MCookies and assess their scalability, we performed a web server benchmark with and without the MCookies module enabled. In order to evaluate the module overhead, we ran the benchmarking tool on a single HTML page. Furthermore, we also benchmarked the module on a Wordpress website for a more realistic scenario. Each request embedded two different sensitive cookies.

The host used for this evaluation was an Intel Xeon X5650 with 6 GB of RAM running a Debian system with an Apache (v2.4.7) web server, a MySQL (v5.5) database server and hosting a Wordpress (v3.8.1) web site.

	Vanilla server	MCookies enabled	
		w/o UA support	with UA support
Static page	384	318 (-17 %)	382
Wordpress page	221	212 (-4 %)	220

Table 2.3: Performance results (transactions/second).

We used Siege[11] in benchmark mode to assess the number of transactions the web server is capable to process per second with the three scenarios. The results, as described in Table 2.3, show a small decrease of 4 % of the Wordpress web server capacity when dealing with User-Agents without the support of this mechanism, whereas the overhead is negligible otherwise. However, for a static page served to a User-Agent with no `Masked-Cookie` header support, performance are much more degraded, but this is a worst-case scenario, since static web sites rarely produce sensitive cookies.

MTokens

MTokens require small modifications in web applications (or in frameworks), so they are easy to deploy, reliable and compatible with web applications, especially if the changes are made in the framework.

2.5 Comparative analysis of attacks and countermeasures

2.5.1 Related work

To avoid repeating the same cookie across different TLS messages, a natural idea would be to change its value for every new connection, or at least to limit the cookie lifetime. In fact, PHP proposes a way to handle session identifiers this way with the `session_regenerate_id` function, usually called after a user logs in, to decorrelate the old and the new sessions and avoid session fixation attacks [Kol02].

[11] `http://www.joedog.org/siege-home/`

Short-lived cookies (which could even be pushed to single-use cookies) should thwart all passive attacks, but choosing the right lifetime is not easy. In fact, mitigating attacks would require a very short lifetime, which could easily lead to out-of-sync cookies when dealing with parallel HTTP connections. Modern web sites heavily use JavaScript asynchronous requests, and session regeneration is known to provoke request concurrency errors. Actually, our 1-day HTTPS capture, presented in the previous section, contained such multiple concurrent requests, that would have been broken by short-lived cookies. This is why MCookies are designed to always have the same *intended* value.

One-Time Cookies [DCAT12] are another solution to protect plaintext HTTP cookies against replay, by having the client bind the cookie with the sent request. This mechanism uses cryptographic mechanisms (symmetric encryption and HMAC), borrows the idea of Kerberos tickets and proposes an elegant solution requiring no server-side state. Applied to HTTPS and our attacker model, One-Time Cookies (OTC) do not counter attacks, since repeating the same exact request would lead to the same OTC; however, the value retrieved could only be replayed for the request in question, which would limit the scope of the attacks. Furthermore, as OTC rely on specific HTTP headers, their implementation requires browser and server/web application modifications. With regard to the Record Protocol attacks exposed here, `Masked-Cookie` headers are much simpler to implement (no cryptographic primitives are needed).

Both solutions (single-use cookies and One-Time Cookies) are described in Table 2.4. Other alternative cookie protocols have been proposed, such as [FSSF01, LKHG05], but they share OTC advantage (unique cookies bound to the request data) and drawbacks (limited protection, heavy changes needed on both end points).

2.5.2 Summary

Countermeasures	Dep.	Rel.	HTTP	Beast	L 13	RC4	Compression			Poodle
							C	S1	S2	
Structural changes to TLS										
Use TLS 1.0	+	+	+							+
Use TLS 1.1	–	–*	+	+						+
Encrypt-then-MAC	– –	–	+		+					
Changes related to TLS ciphersuites or compression methods										
Use CBC mode	+	+	+			+				
Use RC4	+	+	+	+	+					+
Use a new stream cipher	–	+	+	+	+	+				+
Use AEAD (TLS 1.2)	–	–*	+	+	+	+				+
No TLS compression	+	+	+				+	+		
No HTTP compression	+	+	–						+	
Changes related to TLS implementations										
$1/n-1$ split	–	+	+	+						
Constant-time CBC	–	+	+		+					
Anti-Poodle split	–	+	+							+
Other countermeasures in related work										
Single-use cookie	– –	+	–	+		+	+			+
One-Time Cookies [DCAT12]	– –	+	+	partial						partial
Countermeasures presented in this chapter										
TLS scrambling	– –	+	+	+	+	+	+	+		+
MCookies (server)	–	+	+	+^{p}		+^{p}	+^{p}			+^{p}
MCookies (client/server)	– –	+	+	+	+	+	+			+
MTokens	+	+	+					+	+	

Notes:

- With the recent publication and deployment of `TLS_FALLBACK_SCSV`, the deployment of recent TLS versions is becoming more reliable, which explains the * mark in the **Reliability** column for several rows.
- The "partial" indication for One-Time Cookies columns means that the measure only partially blocks the attacks.
- MCookies without client support only counters *passive* attacks, which is emphasised by the ^{p} mark.

Table 2.4: Summary of the proposed countermeasures.

To mitigate the threats against the Record Protocol, many countermeasures have been proposed. Our analysis in section 2.1 shows that some of them have no real effect on the attacks: throwing the

first bytes of RC4 keystream, randomising the packet length or adding random delays to CBC-mode decryption. Others require significant changes to the architecture of web applications and would be hard to enforce: restricting cross-site requests or fixing all SOP/XSS bugs. The remaining countermeasures are listed in Table 2.4, and compared using different criteria:

- **Dep.** relates to the ease of **deployment** of the proposed solution. In particular, the data presented in section 3 shed light on the problem of the capabilities of servers at large.
- **Reliability** (**Rel.**) corresponds to the assurance we have the countermeasure will not be easily bypassed between a client and a server both implementing the solution. The idea is to capture the possible down-negotiation and fallback strategies (for example issues related to TLS version negotiation).
- **HTTP** assesses the compatibility of the measure with HTTP use-case. Current web applications have to e.g. accommodate with multi-tab browsing. Countermeasures should not break or limit such features.
- A set of columns state whether the countermeasure is efficient against each attack (**Beast**, **L13**, **RC4**, **Compression** for compression attacks (**C** for client-side attacks like CRIME or TIME, **S1** for TLS server-side attacks like TIME and **S2** for HTTP server-side attacks like BREACH), and **Poodle**).

2.5.3 Concluding thoughts on the Record Protocol security analysis

We have studied recent attacks on TLS Record Protocol, and thoroughly analysed the proposed countermeasures. In practice, the countermeasures implemented in most of the software are specific to each attack: $1/n-1$ split for BEAST, constant-time CBC decryption for Lucky 13, deprecation of RC4, disabling TLS compression for CRIME and TIME, and deprecation of SSLv3 for POODLE.

In parallel, we showed that all the attacks relied on the common assumption that a secret would be repeatedly sent in different TLS sessions. We suggested a common model to describe these attacks. We also proposed to reuse the concept of masking, borrowed from the side-channel community, to mitigate the attacks. Such a technique can be implemented as a complementary measure, a defense-in-depth strategy. We described different ways this countermeasure could be implemented, and wrote two proofs of concept to check its feasibility to protect cookies.

At the TLS level, our Scramble compression method builds on the idea of WebSockets masking, which actually blocked the BEAST attack. At the HTTP level, our MCookies extend the idea of single-use cookies, without requiring complex changes in web protocols and applications. Masking allows for a defense-in-depth strategy, giving developers and integrators more time to solve the crisis. It would have been effective against the presented attacks, and might be against yet unknown ones. The POODLE attack did in fact meet all the criteria, and was published after we implemented our proposals. The fact it would have been blocked by our proofs of concept actually validates our work.

We would however make it clear that masking is designed to be a defense-in-depth measure *in addition* to specific countermeasures, not instead of them. When a cryptographic algorithm or scheme shows significant weaknesses, they should be phased out and correctly patched. In the particular case of RC4, we now know a lot of statistical biases, some of which can lead to efficient first order attacks, but realistic second order attacks could be the next attack against TLS Record Protocol. All the masking proposals could be easily extended to use two (or three) masks instead of one. Yet we consider RC4 is a good example of a streamcipher that should have been phased out a long time ago, since many RC4 practical and theoretical flaws have been known for a decade.

Part II

Observation and analysis of the HTTPS ecosystem

The second part of this document presents experimental results regarding the HTTPS ecosystem. Chapter 3 first focuses on the data acquisition process. Chapter 4 presents the analysis framework developed to handle our SSL/TLS campaigns. Chapter 5 describes results obtained using this framework on various datasets.

Chapter 3

HTTPS measurement campaigns: data acquisition and validation

This chapter presents the methodology used in 2010, 2011 and 2014 to collect HTTPS data. 2010 and 2011 campaigns were analysed and the results were published in 2012 [LÉMD12].

In chapter 1, we have shown that SSL/TLS has become an essential part of the security of the Internet infrastructure and services. As such, we expect it to offer robust and state-of-the-art confidentiality and integrity properties, in particular for HTTPS, its primary application. Theoretically, the protocol allows for a trade-off between secure algorithms and decent performance. In practice however, servers do not always support the latest version of the protocol, nor do they all enforce strong cryptographic algorithms.

It is thus legitimate to assess the security of the SSL/TLS ecosystem. Since HTTPS still represents most of the daily TLS usage, we designed experiments to get a clear view of what browsers face on a daily basis, and whether this view is satisfying or not.

To evaluate the quality of HTTPS servers in the wild, we enumerated HTTPS servers on the Internet in July 2010, in July 2011 and in March 2014. Since our first campaigns in 2010 and 2011, many independent campaigns have been conducted by different research teams. What differs in our campaigns is that we sent *several stimuli* (different `ClientHello` messages) to the contacted servers, enabling us to gather detailed information about the features effectively supported by the servers.

First, section 3.1 presents the different methodologies available to enumerate HTTPS hosts and focuses the method we chose. Section 3.2 then explains which `ClientHello` messages we chose to send to get relevant information about the HTTPS servers, while section 3.3 describes the different kinds of answers we received. Section 3.4 offers some elements to check the consistency of our datasets.

3.1 Scanning methodology

3.1.1 Enumerating HTTPS hosts

Gathering data about what a browser faces on a daily basis can be done in several ways:

- enumerating every routable address in the IPv4 space to find open HTTPS ports (443/tcp) and establish SSL/TLS sessions with them;
- contacting HTTPS hosts based on a list of hostnames;
- collecting real HTTPS traffic from consenting users.

The first method is *a priori* the most exhaustive, because it tests every IP in the world. It is thus possible to communicate with many different TLS implementations to broaden our knowledge of the ecosystem. However, it leads to contacting many non-HTTPS hosts. Also, it does not take into account the popularity of Internet sites: it does not discriminate sites like `www.google.com` from

`randomhost.dyndns.org` or even from an unnamed host (e.g. an ADSL modem or a so-called smart power outlet).

The second option is more restrictive, but better represents user needs, and the proportion of HTTPS servers among the hosts to contact is higher. Besides, this method is compliant with the TLS SNI (Server Name Indication) extension [3rd11], which allows a client to contact different virtual hosts at the same address. Both the first two methods allow to send multiple `ClientHello` messages (stimuli).

Finally, the last one is completely passive and is really centered on users' habits. In this case it is important to have access to the traffic of many different consenting users to get relevant data that would be comparable to other studies. On the one hand, such measures are truly representative of real traffic, but on the other hand, the stimuli used to test the server are not controlled by the experimenter.

	Full IPv4	**Hostname-based**	**Passive analysis**
HTTPS servers coverage	Nearly 100 %	Subset of named sites	Subset of visited sites
TLS features coverage (typical number of stimuli)	Limited (10)	Large (100)	Uncontrolled
User representativeness	No	Mostly	Fully
Network invasiveness	Significant	Moderate	None
SNI-awareness	No	Yes	Partial
Examples of campaign	[EB10a] [DWH13]	[Lab15b] [Kar15]	[HBKC11]

Table 3.1: Comparison of the different methods to collect HTTPS data. The examples of campaigns are detailed in section 5.1.

Table 3.1 summarises the differences between these methods. We chose the first method to acquire a broad vision of the HTTPS world. This method also enabled us to get consistent answers to multiple stimuli for each given host.

3.1.2 Description of the IPv4 campaigns

In July 2010, in July 2011 and in March 2014, we launched several campaigns to enumerate HTTPS hosts present in the IPv4 address space. We used different *stimuli* (different `ClientHello`) to grasp the behaviour of the different TLS stacks encountered.

Phase 1: finding the HTTPS hosts

The first task was to find out which hosts were accepting connections on TCP port 443. Using BGP (Border Gateway Protocol) Internet routing tables, we reduced the search space from 4 billion IPv4 addresses (2^{32}) to 2 billion routable addresses.

Then, instead of using existing tools such as `nmap` to enumerate open 443 ports, we developed home-made probes to randomise the set of routable addresses globally. For each host, the test consisted simply in a SYN-probe to determine whether the port was open or not.

To prevent this first phase from being too intrusive, we bounded our upstream rate at 100 kB/s, allowing us to explore the 2 billion addresses in about two weeks.

Phase 2: TLS sessions

Once a host offering a service on port 443 was discovered, we tried to communicate with it using one (or several) TLS `ClientHello`. In this second phase, we used a full TCP handshake followed by several packets, but only with the fraction of servers listening on port 443 (about 1 percent in 2011 and 1.5 percent in 2014). The second phase could thus be run in parallel with the first one. As a matter of fact, running it in parallel was required when dealing with dynamic IP address ranges to improve our chances of contacting the IP before the server behind it moved.

To limit the computational impact on servers, we only recorded the first server answer (messages between `ServerHello` and `ServerHelloDone`) before ending the connection. This way, we collected the protocol and ciphersuite chosen by the server, as well as the sent certificate chain.

In the 2010 campaign, we sent only one `ClientHello` message. On the contrary, as we were interested in server behaviour, in the July 2011 and March 2014 campaigns, we sent several `ClientHello` messages containing different protocol versions, ciphersuites and TLS extensions.

3.1.3 Issues encountered

Our July 2010, July 2011 and March 2014 campaigns each took two to three weeks to complete. As explained earlier, we chose to do so to avoid link saturation during the host enumeration phase. However, spanning our measures over several weeks has an impact on the picture of the Internet we are seeing. In fact, while probing the different hosts, two factors need to be balanced:

- the time spent acquiring the data. Like the exposure time in photography, it should ideally be as short as possible, to get consistent data. In particular, IPs belonging to dynamically-assigned address blocks can change every day or so;
- the network load induced; sending too many packets can result data loss at either end of the connection.

Considering the network bandwidth at our disposal and the way IPs were globally randomised, we are confident we did not overload links during the first phase of the campaigns. We believe that it is as close as we could get to a time-coherent snapshot[1]. Sections 3.4.3 and 3.4.4 answer questions about the impact of time on IP address stability.

Even with this in mind, randomisation can be insufficient and our SYN packets may be interpreted as an attack, and our IP filtered out. A solution could have been to use several source IPs. Yet, if these addresses were not located in the same neighbourhood, we might have ended up measuring different inconsistent views of the Internet, as shown in [HBKC11].

Finally, one aspect of gathering such data we did not anticipate was data storage. By definition, full IPv4 campaigns target the entire IPv4 space, theoretically containing 4 billion IP addresses. In practice, the number of addresses actually routed is around 2.5 billion. Among those, between 1 and 2 % of the contacted hosts have the 443/tcp port open. One easy way to store the data would be to use one file per active IP (between 25 and 40 million), but this rapidly fills up inode/block tables, while the answers to one stimulus only take 20 GB in total. We ended up grouping answers by /8 IP ranges and developing tools to work on such files.

3.2 The stimulus choice

Once we chose the scanning method (actively contact all routable IPv4 address), we needed to select the relevant stimuli to send, since we retained an active method. The stimulus (the `ClientHello` message) is composed of various parameters. Our goal was to understand the different possibilities while keeping the number of stimuli low.

3.2.1 Protocol version

The first relevant parameter in a TLS connection is the protocol version negotiated. A `ClientHello` message actually includes two version fields: the external version used for the transport of this particular message (present in the Record protocol since SSLv3) and the maximum version supported by the client, v_{max}. The standard indicates that the server should choose the maximum version it supports, up to v_{max}. If no such version exists, it should terminate the handshake with an alert.

A popular belief is that the Record protocol version is the minimum version supported by the server. However, TLS 1.2 specifies in an appendix that a compliant server must accept any Record protocol version starting by `03`.

[1] Chapter 5 describes related work, including campaigns made within a much shorter time frame, and discusses further the shortcomings of both approaches

The combination of an ill-conceived Record protocol version field and of useless extra checks made by several TLS stacks regarding v_{max} led to a phenomenon often referred to as *version intolerance*: instead of correctly negotiating a commonly supported version, several servers shut down the connection (either with a TLS alert or by tearing down the TCP connection).

Finally, another interesting variation of the SSL/TLS version parameter relies on the message format itself. Since SSLv3 messages, that introduced a proper Record protocol, are fundamentally different, there are essentially two sorts of `ClientHello` messages:

- SSLv2-compatible messages, using the old format while advertising modern protocol versions. Such messages are limited by construction to the SSLv2 features, which means in particular that TLS extensions can not be carried by such stimuli;
- SSLv3/TLS `Client Hello`, using the Record protocol.

Gathering all this, it first seemed interesting to consider SSLv2-compatible and more modern `ClientHello` messages, advertising standard TLS 1.0 as v_{max}. For the first kind, we even chose to use a pure SSLv2 stimulus, that is a message announcing SSLv2 as the only supported version. Finally, we were also interested in the server reaction to a TLS 1.2 `Client Hello` (in the first campaign, the Record protocol version was set to TLS 1.2, then we added a second TLS 1.2 stimulus where the Record protocol version was TLS 1.0).

Since we did not want to use too many stimuli, we did not investigate the following parameters:

- we never sent SSLv3 or TLS 1.1 as v_{max};
- we did not advertise unspecified future versions, like TLS 1.3 (`0x0304`) or `0x03ff`;
- we did not study discrepancies between Record protocol version and v_{max}.

3.2.2 Ciphersuite

Another security-related parameter conveyed by the `ClientHello` is obviously the list of proposed ciphersuites. The range of existing ciphersuites is officially defined by a IANA registry[2] and contains 319 entries, including two *signaling cipher suite values* (SCSV), used to signal a particular features: `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` to advertise secure renegotiation without using extensions, and `TLS_FALLBACK_SCSV` to avoid unnecessary and potentially dangerous version downgrades (see section 1.5.1). On top of these, we can also consider the 7 SSLv2 suites and the 4 SSLv3 FIPS suites which are now obsolete or redundant. Among those suites, a typical client offers between 10 and 30 suites.

For our study, we first eliminated a lot of suites that contain seldom used algorithms: fixed Diffie-Hellman, PSK (Pre-Shared Key), SRP (Secure Remote Password), etc: such suites were never proposed in our stimuli. We then chose several ciphersuite subsets: standard ones, DHE-only, EC-only. The idea was to detect possible ciphersuite intolerances among servers.

For the SSLv2-compatible stimuli, we also used the SSLv2-specific ciphersuites (which were the only ones present in the pure SSLv2 `ClientHello`). As a side note, SSLv2 ciphersuites are expressed as 24-bit values, whereas SSLv3/TLS suites are encoded as 16-bit values: inside an SSLv2 message, the convention is to prepend modern ciphersuite values with a null byte.

Again, if it had been possible to send more stimuli, further combinations could have been used:

- single-suite stimuli, to test every possible suite;
- representative lists of `ClientHello` messages sent by standard browsers;
- very long list of suites, including not yet specified ones.

Such tests are actually performed by several tools like `sslscan`, or by the online SSL Server Test tool published by Qualys (see section 5.1.4).

[2] `http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4`

3.2.3 TLS Extensions

As studied in an RFC draft [Lan11], several servers do not support TLS extensions and other implementations simply reject `ClientHello` containing extensions. This is unfortunate since this mechanism is necessary for the extensibility of the protocol:

- security fixes like secure renegotiation [RRDO10];
- support for new ciphersuites using elliptic curves [BWBG+06];
- new features like session tickets [SZET06].

Moreover, several extensions are expected to be mandatory in TLS 1.3, which makes extension intolerance a real concern. From a security standpoint, when a client proposes security features via extensions, the server should support it.

We only focused on three extensions in our study:

- the renegotiation extension (including its SCSV form to comply with the SSLv2 `ClientHello` message), both to evaluate the extension intolerance and to assess the proportion of servers supporting it;
- the EC-related extensions (supported elliptic curves and point compression).

Further work might consider other extensions:

- the Heartbeat extension, that led to the OpenSSL Heartbleed vulnerability (see section 7.1.4);
- new security features like the standardised DHE groups or the *session hash* (presented in section 1.5.1);
- hacks to work around broken implementations such as the padding extension.

Regarding the latter, several TLS stacks confuse TLS `ClientHello` messages with SSLv2 ones, when the record length is a value between 256 and 511 (the length is of the form `0x01XX`): the corresponding `0x01` byte is interpreted as the SSLv2 message type. To avoid having such messages, the padding extension was proposed to pad the `ClientHello` to 512 bytes at least. It is interesting to note that such behaviour can explain various forms of protocol version or extensions intolerance in the wild.

3.2.4 Unexplored parameters

There are other fields we did not study at all: other SSLv2-specific parameters (e.g. the variable-length challenge field), the session identifier, the list of compression methods. Finally, interactions between the Record protocol and the Handshake layer could be checked by splitting the `ClientHello` across different records in several ways (record splitting was indeed shown to trigger a version downgrade in OpenSSL, see section 7.2.2).

Adding these stimuli would need to send many more `ClientHello` messages, which does not seem compatible with a full IPv4 scan.

3.2.5 Actual stimuli

During our campaigns, we used several different stimuli: 1 in 2010, 7 in 2011 and 10 in 2014. They are described in table 3.2. The first part of the table contains historical, SSLv2-compatible, `ClientHello`; the second part is made of standard stimuli (in terms of versions, suites and extensions); finally to test for specific features (Ephemeral Diffie-Hellman, elliptic curves), we used specific messages with a restricted set of ciphersuites and extensions.

Id	SSLv2	Max version	Ciphersuites	Extensions
Historical (SSLv2-compatible) stimuli				
SSL2	yes	SSLv2	SSLv2 suites only	None
SSL2-TLS1.0	yes	TLS 1.0	SSLv2 + TLS 1.0 suites	Reneg (SCSV)
Standard stimuli				
TLS1.0-NoExt	no	TLS 1.0	Standard Firefox suites	None
TLS1.0	no	TLS 1.2	Standard Firefox suites	EC, Reneg, Ticket
TLS1.2	no	TLS 1.2	Standard Firefox suites	EC, Reneg, Ticket
TLS1.2-modern	no	TLS 1.2	mostly TLS 1.2 suites	EC, Reneg, Ticket
Feature-oriented stimuli				
DHE	no	TLS 1.0	DHE suites only	None
EC	no	TLS 1.0	EC suites only	EC (only *named* curves)
EC-explicit	no	TLS 1.0	EC suites only	EC (only *explicit* curves)

Table 3.2: Different `ClientHello` messages sent during the 2010-2014 campaigns. The "SSLv2" column indicates that a SSLv2-compatible `ClientHello` was sent. DHE means Ephemeral Diffie Hellman; EC means Elliptic Curves; Reneg stands for the renegotiation extension [RRDO10], and Ticket for the Session Ticket one [SZET06].

3.3 Answer description

In the second phase of our campaigns, we opened a TCP connection with the hosts with an open 443/tcp port and sent one of the selected stimulus. We then recorded their answers. When this answer could be parsed as valid SSL/TLS messages, we would read messages until we read an alert or the end of the flight of expected Handshake messages (`ServerHello` through `ServerHelloDone` messages). In case we were unable to parse such answer, we waited 5 seconds before tearing down the connection. As a result, we were able to sort server answers in different categories: non-TLS answers, alerts and Handshake messages. These three answer types are described in figure 3.1.

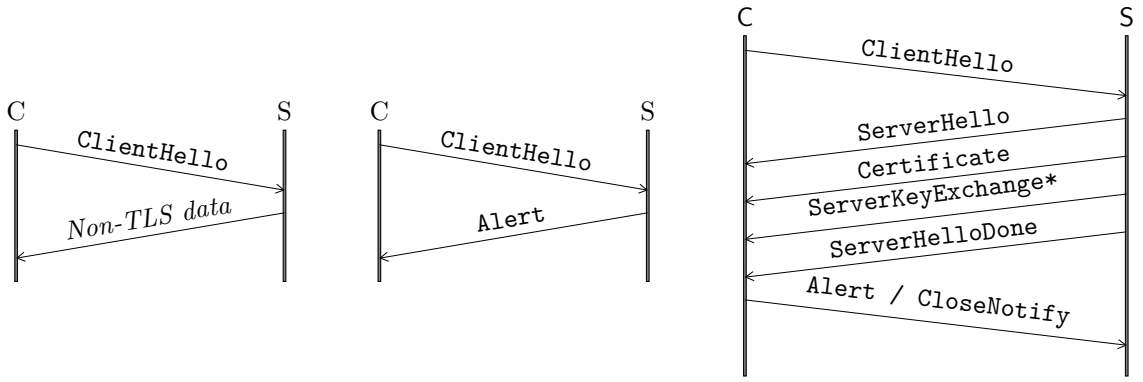


Figure 3.1: The three different types of received answers, from left to right: non-TLS data, alerts and Handshake messages. When sending SSLv2 stimuli, we also received SSLv2 alerts and messages.

3.3.1 Non-TLS answers

About half the collected answers were not valid SSL/TLS messages. This can easily be explained by the fact that 443/tcp is often used to host various services beyond HTTPS, since the port is usually not filtered and let as is by outbound firewalls or proxies. Analysing the recorded data, we could easily confirm this hypothesis. Here are the typical results we obtained from non-TLS servers.

Empty answers Sometimes, the server simply did not send anything back, which usually meant that the contacted service was not an HTTPS server: as the hosted service did not recognise the `ClientHello`

as belonging to the expected protocol, it shut down the connection or waited until we did. It is worth noting that this behaviour was also observed with intolerant TLS servers facing unacceptable proposals: instead of sending an alert describing why the TLS connection could not be pursued, they just ended the connection.

HTTP A large proportion of non-TLS servers were in fact HTTP servers interpreting the TLS `ClientHello` as a broken HTTP request. They thus answered with an error HTML page with or without HTTP headers (the latter corresponds to an early version of the HTTP protocol, sometimes dubbed 0.9). Such answers typically began with `HTTP/1.0` or `<!DOCTYPE`

SSH Another type of answers regularly observed started with `SSH-1` or `SSH-2`, which is representative of an SSH server presenting its banner.

SMTP Some servers seemed to answer with an SMTP status code (`220 ... ESMTP`), which tends to prove that 443/tcp is used to host a wide variety of services across servers.

Beyond identifying the obviously recognisable protocols, we did not try and look for information in these non-TLS answers.

3.3.2 SSL/TLS alerts

Since we explored several corner-case configurations in our stimuli, we expected that some servers would answer with an SSL/TLS alert explaining why they could not accept the TLS connection. The main reason for this is the absence of a common ciphersuite between the proposition present in the `ClientHello` and the server configuration (which should raise a `HandshakeFailure` alert), but we encountered other errors such as `ProtocolVersion`, `InsufficientSecurity`, `IllegalParameter`, which conveyed either an incompatibility between the client proposal and the server parameters, or some form of intolerance.

The sent alert can either be an SSLv2 alert, using a specific message format, or an SSLv3/TLS alert, sent over the Record protocol.

3.3.3 SSL/TLS Handshake

Finally, an important proportion of the contacted hosts answered our stimuli with valid SSL/TLS Handshake messages: *a priori*, these servers were engaging in a TLS connection and had chosen the parameters based on the client's proposal.

As for alerts, this kind of answer could be transported using an SSLv2 old-style `ServerHello` message, or via Handshake messages above the Record protocol (for SSLv3 or TLS). Even if these cases differ, the answer always conveys at least the following parameters: the selected protocol version, the chosen ciphersuite and the server certificate.

By answering with a `ServerHello`, the server actually acknowledges that the proposed parameters are somewhat acceptable, and the parameters present in its answer should be compatible with the received `ClientHello`. However, we observed several inconsistencies, where the server chooses a protocol version or a ciphersuite *that was not proposed by the client*. Such behaviour does not conform to the specifications, and would lead to a client-sent alert in a real TLS connection.

As a side note, we witnessed a particular bug that could lead to the ciphersuite inconsistencies. Ciphersuites are encoded as a 16-bit value, but for a long time, the most significant byte of all the specified ciphersuites was null. This led some developers to only compare the least significant byte of the ciphersuite values. In 2006, elliptic curves were introduced in TLS [BWBG+06], with `0xc0XX` ciphersuite values, which could lead to confusions in broken implementations. Indeed, in our EC-only stimulus, we only proposed `0xc0XX` suites, and obtained several answers selecting the `0x0005` ciphersuite (`TLS_RSA_WITH_RC4_128_SHA`) in response to our proposal, including the `0xc005` ciphersuite (`TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA`).

3.3.4 Examples

Considering a typical stimulus (TLS 1.0, standard ciphersuites), the following snippet is an example of the extracted answer types:

```
108.XXX.XXX.XX  E
--
108.XXX.X.XX    J   HTTP
108.XX.XX.XXX   J   SSH-2
108.XXX.XXX.XXX J   10c22e31e0d6ea6e0847a30706d99ec6 (...1...n.G......)
--
108.XXX.XX.XX   A   Fatal     HandshakeFailure
--
108.XXX.XXX.XX  H   TLSv1.0   TLS_RSA_WITH_AES_256_CBC_SHA
108.XX.XXX.XX   H   TLSv1.0   TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
```

where `E` means Empty and `J` represents junk data (with a possible hint about its nature, be it HTTP or SSH for example). SSL/TLS alerts are tagged with an A, followed by the nature of the alert, whereas SSL/TLS handshake messages are signaled by an `H`, with the remaining fields containing the selected protocol version and ciphersuite.

For this particular campaign in 2014, the proportions of the answer types are given in table 3.3.

Answer type	Proportion
Non-TLS answers	**27.5 %**
Empty	17.1 %
HTTP	6.1 %
SSH	0.1 %
SSL/TLS alerts	**0.3 %**
SSL/TLS handshake	**72.2 %**

Table 3.3: Proportion of answer types for a typical campaign launched in 2014 (TLS 1.0 with standard ciphersuites). All percentages are computed with regards to the overall number of host with an open 443/tcp port.

3.4 Dataset consistency

To check the consistency of our scanning methodology, we compared several statistics concerning our dataset to similar statistics computed by other research teams.

Moreover, since we tried to compare the answer types for a given IP against different stimuli, we tried to assess when such a comparison was relevant and when it was not. In particular, we observed that comparing results for a given IP in data collected months apart was completely irrelevant. We thus tried to validate the internal consistency of our multi-stimuli datasets, to ensure the servers contacted at the same IP address in a short period of time (within an hour) with different stimuli were actually the same. To this aim, we defined a stability indicator for 2011 and 2014 campaigns.

In this section, we consider different subsets of the contacted servers: the TLS hosts, which answer with valid records; the Trusted hosts, which present a trusted certificate chain; and the EV hosts, which present a valid EV server certificate. Chapter 4 will explain in details how these subsets are defined and built.

3.4.1 Validation of independent results

The first method we used to validate our datasets was to compare big picture figures with other datasets. In 2010, the EFF presented their work about a campaign launched in August 2010. We compared some of their results with the most similar campaign we had: the campaign corresponding to an SSLv2-compatible `ClientHello` also advertising TLS 1.0 ciphersuites. It was sent in July 2011.

The EFF presented several relevant figures at the CCC in 2010 [EB10b]. One of the first figure was that 11.3 million hosts answered with Handshake messages. After classifying the answers for the appropriate stimulus, we indeed found the same figure.

Among these, they found 4.3 million hosts with valid cert chains. In July 2011, we only found 4.0 million such hosts, but the EFF included both Microsoft and Mozilla root trust stores, whereas we only considered the Mozilla trust store, which is embedded in the open-source NSS library.

3.4.2 When comparison really makes sense

We compared the list of IPs from the TLS subset between our measure of July 2010 and the EFF measure of August 2010: 7.5 million IPs are present in both sets but 2 million were only seen by our trace, and nearly 4 million were only present in the EFF trace. So a vast proportion of TLS hosts do not have a stable IP over a month or a year.

This fact is even more visible when comparing our measures in 2010 and 2011 (using the standard TLS 1.0 stimulus without extensions): 5.5 million IPs correspond to TLS hosts in both cases, but about 4 million (resp. 6 million) IPs are present only in the 2010 (resp. 2011) trace. It is thus clear we can only do per-IP comparisons between measures that were made at the same time.

This basic comparison sheds light on the surprising figures obtained in the second EFF campaign: between the EFF experiments conducted in August and December, 7.5 million IPs represent TLS hosts both times, about 4 million are only present in August whereas only 60,000 IPs are new in December. This can be explained by the fact that the EFF did not launch the first phase (enumerating IP with open 443/tcp port) again in December and reused the list of IPs from August 2010.

3.4.3 IP Stability across July 2011 campaigns

We now focus on the seven measures of July 2011 that were conducted simultaneously[3] on the same address pool to understand server answers against different stimuli. We need to check that the servers we contacted were the same during all the communications. Let's first compare the IPs corresponding to a TLS answer between two similar campaigns: the TLS 1.0 standard stimuli (with and without extensions). Over 99.6 % of IPs corresponding to a TLS host in one measure also do in the other. The correlation is even better if we focus on trusted or EV hosts.

To confirm that, we also compare the server certificates returned by the servers. Considering the set of IPs that answered at least once with a server certificate, we count for each IP the number of different server certificates received over the seven communications. More than 99.6 % of them presented the same server certificate each time they answered with a valid `ServerHello`. If we compute the same statistics using the Trusted subset or the EV subset, the stability is even better, since 99.9 % served only one certificate. The hosts that do not consistently send the same server certificate (0.4 %) define our precision margin; when analysing the server behaviour based on the comparison of answers for different stimuli, all statistics smaller than this margin should be discarded.

Another indicator variant would be the subject DN of the received server certificate, which leads to better results: 99.9 % of consistent hosts. This shows that a small proportion of hosts do load balancing with servers using different certificates. The figures are presented in table 3.4.

	TLS	**Trusted**	**EV**
Same chain	81.1 %	80.0 %	90.9 %
Same server certificate	99.60 %	99.92 %	99.86 %
Same subject DN	99.97 %	99.95 %	99.87 %

Table 3.4: Proportion of the servers presenting the same chain (resp. the same server certificate and the same subject distinguished name) each time they answer a valid `Handshake` answer, for each considered subset in the 2011 campaigns.

As a side note, the table also presents the proportion of servers presenting the same certificate chain for each stimulus, which is significantly smaller. It can easily be explained by the use of SSLv2: the

[3]For a given IP, all `ClientHello` messages were sent within a minute.

older version of the protocol indeed does *not* allow the server to send a certificate chain, but only its server certificate.

3.4.4 IP Stability for July 2014 campaigns

For the 2014 campaigns, we ran similar comparisons and obtained similar results: 99.7 % of TLS hosts are consistent across different stimuli, regarding the presented certificates.

Moreover, we chose in 2014 to send an identical stimulus twice (the DHE), and among the 40+ million of servers with an open 443/tcp port, only 0.55 % of them had a different behaviour between both answers. The remaining servers either answered with TLS messages containing the same protocol version, ciphersuite and certificate chain, or rejected the connection in a similar fashion both time.

This allows to set the precision margin to around 0.55 %.

3.5 Concluding thoughts on the data collection

To better understand the HTTPS ecosystem, we chose to contact every HTTPS server world-wide. Such measurement campaigns require a trade-off between the time spent to acquire the data and the network load induced. The former can lead to an imprecise snapshot of the ecosystem while the second one can cause data loss. For our campaign, we chose to span the data collection over three weeks.

In 2011 and 2014, we sent multiple stimuli to probe the HTTPS servers we found. As shown in chapter 5, this additional information helps us better understand how HTTPS servers behave in practice.

The next chapter presents the analysis framework developed to handle our SSL/TLS campaigns, called `concerto`, and chapter 5 describes the experimental results obtained using this framework on various datasets, including those described in this chapter.

Chapter 4

`concerto`: a methodology towards reproducible analyses

This chapter describes our analysis methodology for TLS data. This methodology was implemented inside `concerto`*, a set of simple tools to conduct reproducible analysis on possibly heterogeneous datasets. At first developed to study the chain quality on very small datasets,* `concerto` *was then extended to scale and to support full IPv4 campaigns. The source code was publicly released in 2016 on GitHub*[1]

In the previous chapter, we have presented existing TLS campaigns and some preliminary analyses on various datasets. One problem we faced in 2014 was to propose similar analyses to those produced for the data previously collected in 2010 and 2011. This led us to consider a methodology to obtain reproducible analyses.

Our approach is based on a set of tools, `concerto`, an open source software, which allows to reproduce and compare indicators to draw trends across different datasets, including available third party campaigns. The parsing steps rely on parsers written in Parsifal, a framework to write binary parsers described in chapter 6.

In a first time, section 4.1 describes how TLS `Certificate` message are specified and interpreted in practice, which was the origin of `concerto`. Section 4.2 explains the issues to face to handle full IPv4 datasets. Next, section 4.3 presents the different tools that compose the `concerto` framework and section 4.4 discusses some of our implementation and algorithmic choices. Finally, section 4.5 shows some examples of `concerto` usage and section 4.6 concludes the chapter.

4.1 X.509 certificates in TLS: the origin of `concerto`

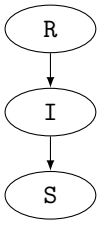
Initially, our goal with `concerto` was to study the quality of certificate chains. Indeed, many TLS errors come from a wrongly configured certificate chain. Let us first dive into X.509 and the related issues in TLS.

4.1.1 `Certificate` messages: the good, the bad and the ugly

The certificates used in TLS follow the X.509 standard. The TLS Public Key Infrastructure is based on several root authorities trusted by default by web browsers. The server certificate is sent in the `Certificate` message, along with its certificate *chain*, that is the ordered set of certificates needed to build, link by link, a certificate path to a trusted root. Figure 4.1 presents a typical chain, where `S` is the server certificate, `I` is the certificate of an intermediate authority, and `R` is the certificate of a trusted root; it also describes three different `Certificate` messages representing this chain. A conforming TLS implementation must send `S`, `I` and `R`, *in that order* (the first `Certificate` message of the figure). A server may omit the root certificate since the client needs to know (and trust) `R` to validate the chain[2]. In theory, all other `Certificate` messages should be discarded.

[1] `https://github.com/ANSSI-FR/concerto`.

[2] This assumption, which is true with the classic Public Key Infrastructure, does not necessary hold with new trust models like DANE [HS12] where the client may only know *the hash* of the expected trusted root.



Different types of `Certificate` messages	Position in message			
	0	1	2	3
Compliant message	S	I	R	
Unordered message	S	X	R	I
Incomplete message	S	R		

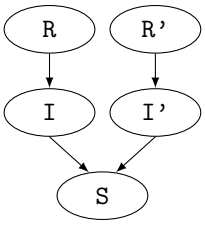
Figure 4.1: A typical certificate chain (left), and different `Certificate` messages representing this chain. The columns indicate the position of each certificate inside the `Certificate` message.

In practice, a lot of TLS servers provide an unordered certificate chain (e.g. the second message of figure 4.1). It is relatively easy to accommodate such messages; moreover, unordered chains are sometimes justified in case there exists two valid certificate chains, leading to different root certificates: it is thus impossible to produce a correct `Certificate` message for the chain to be accepted by clients trusting only one of the root certificates. This matter has been discussed within the TLS working group, and the order constraints should be relaxed in TLS 1.3. Section 4.1.2 describes such a problematic case.

Another problem with chains is that a small proportion of servers even omit intermediate certificates (as in the third message of figure 4.1). To be able to build a chain in this case, the client has to grab missing certificates using out-of-band mechanisms. This can either be done by relying on the `Authority Information Access` X.509 extension (thus initiating connections based on a still-unauthenticated piece of data), or by maintaining a cache of previously seen certificates.

Several implementations, like the Java TLS stack, do not try and repair broken `Certificate` messages, which leads to errors when a client uses such a stack to visit a non-conformant server. On the contrary, repairing broken chains may be complex to implement and can lead to security flaws, as was recently shown in OpenSSL (CVE-2015-1793): the stack could be misled to accept untrusted certificate chains by abusing the code trying to build alternate certificate chains from the `Certificate` message.

4.1.2 An example of multiple certificate chains



Different `Certificate` messages	Position in message				
	0	1	2	3	4
Compliant messages	S	I	R		
	S	I'	R'		
Non-standard message	S	I	R	I'	R'

Figure 4.2: Multiple chains and the encountered `Certificate` messages. On the one hand, the first two messages are only valid for clients trusting `R` (respectively `R'`). The third one, on the other hand, requires the client to be liberal while parsing the `Certificate` message.

Sometimes, a server may need to present different certificate chains to different clients. Assume the server certificate `S` has been signed by two intermediate authorities `I` and `I'`, each being signed by a corresponding trust anchor (`R` and `R'`). Figure 4.2 describes such a situation. It is worth noting that `I` and `I'` are different certificates containing the same subject and the same public key. This situation may appear when the corresponding certification authority is certified by several trusted roots.

If this server wants to conform to the RFC, it must send a unique and ordered certificate chain. Yet, certain clients will only trust `R` in their certificate trust store, while others will only trust `R'`[3]. So a server must choose between RFC compliance and maximum compatibility. Usually, a server will send `S`, `I` and `I'` (possibly along with root certificates), which will be understood by most clients, while violating the TLS 1.2 specification.

To avoid this problem, TLS 1.3 should relax this constraint: it would then be officially allowed to send unordered chains, which are already supported by the majority of TLS stacks.

[3]There exists an extension called `TrustedAuthorities` the client can use to signal the certificate authorities it trusts, but it is not used in practice (and to the best of our knowledge no server-side library acts on this extension).

4.1.3 Presentation of the first tool

The origin of `concerto` was to propose a simple tool to present, for each known server, the presented certificate chain, along with the chains we could build from it, and then assess their quality. Moreover, `concerto` was designed to iterate analyses at different points in time.

To illustrate how this first version of `concerto` works, figure 4.3 shows a screenshot of `concerto` analysing the Top Alexa 1000 HTTP servers: 953 servers answered with valid Handshake messages, 1 sent an alert (the precise type is `Unrecognized Name`), and 46 did not seem to speak TLS. For each answer, the table contains information about the contacted hosts (the hostname, its IP address and the TCP port), the time when the request was made, the answer type and information about the certificate chain: a link to the page describing the chain and the best grade for this chain, considering the *trusted* certificate store (the NSS certificate store at the time of the campaign in the example).

Sometimes, the certificate chains are simple and straightforward, but there may also be alternative built chains. Let us look at a particular example from the Top Alexa 1000 dataset in figure 4.4. The server presents a chain of 5 certificates, properly ordered, ending with a trusted root. However, since several authorities in this chain cross-certified themselves, `concerto` was able to build 29 different chains (the algorithm is described in section 4.3.3).

Figure 4.5 represents the graph of all the chains `concerto` could build, using both the provided certificates and known certification authorities gathered elsewhere. The graph clearly contains different certificates corresponding to the same certification authority (and the same public key).

Another interesting feature of the first version of `concerto` was to analyse a given certificate. Figure 4.6 shows the description of a certification authority, including the list of server answers containing this particular certificate in the campaign.

4.2 The challenge of full TLS datasets

Three years after our first campaigns, we launched new scans, with 10 stimuli (including the previous ones). The challenge was then to reproduce similar analyses in a reliable way, using the new data. Yet, our answer dump format had changed a little and our custom tools had been considerably rewritten, leading us to choose between using old tools or losing insurance that the results would be comparable.

We chose a third way: clearly split our analysis process in simple tasks, implement them and make the whole process repeatable. The implementation was built upon the existing code of `concerto`. Instead of comparing old results on old data to new results on new data, the idea was to recompute the results each time the tools change, first to ensure the non-regression, then to allow for a meaningful comparison.

This section describes miscellaneous challenges we faced to extend our simple certificate chain analyser working on small datasets (typically a 20 MB answer dump concerning about 2 500 hosts) to a toolset extracting all sort of TLS features from full IPv4 datasets (the typical size is 60 GB for one stimulus).

4.2.1 Scalability

The first version of `concerto` was a monolithic program extracting the certificates presented by different servers, analysing them to build all possible chains and finally grading the chains and the servers accordingly.

The first challenge was to be able to handle datasets more than one thousand times bigger. To this aim, we rewrote `concerto` as a small set of programs whose tasks were clearly defined. Some of them could then be run in parallel, while others required to consider the whole picture. By splitting the overall process into small tasks, `concerto` was quickly able to manage full-IPv4 datasets.

4.2.2 Dataset quality assessment

To be able to do reproducible analyses and to confront the state of the TLS ecosystem between different points in time, the data collection must however follow certain rules, that were discussed briefly in section 3.1.3.

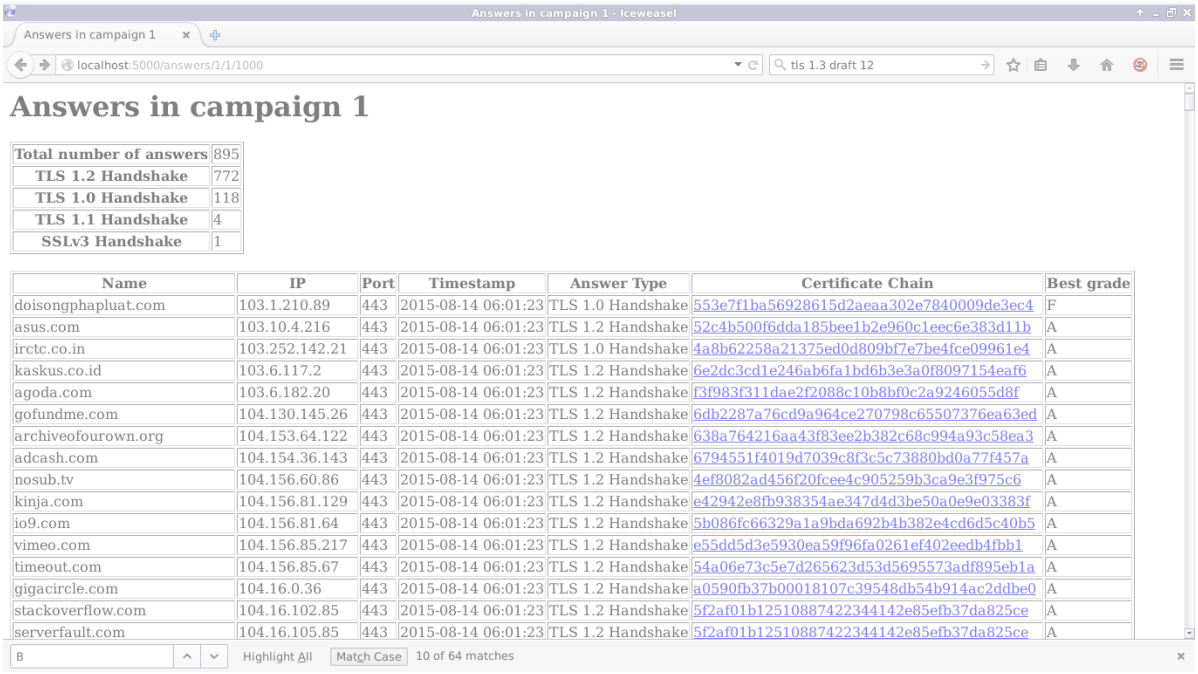


Figure 4.3: `concerto` screenshot: description of the answers collected using Top Alexa 1000 data from `scans.io` (August 14th, 2015).

Figure 4.4: `concerto` screenshot: concrete example of a chain.

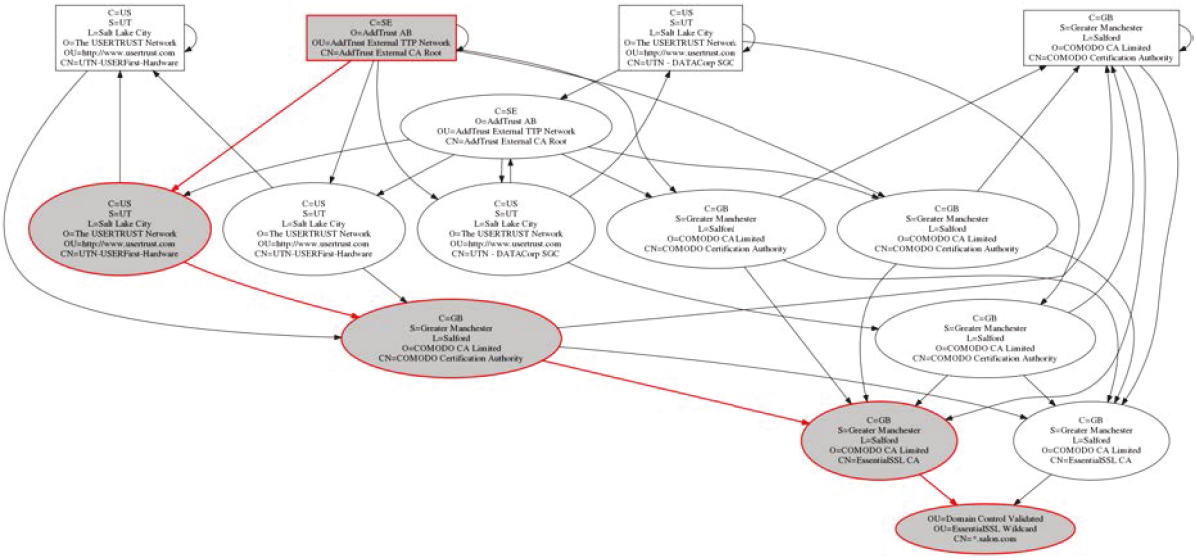


Figure 4.5: Results of `concerto` built chains: grey nodes were sent by the server, and square nodes represent trusted root certificates.

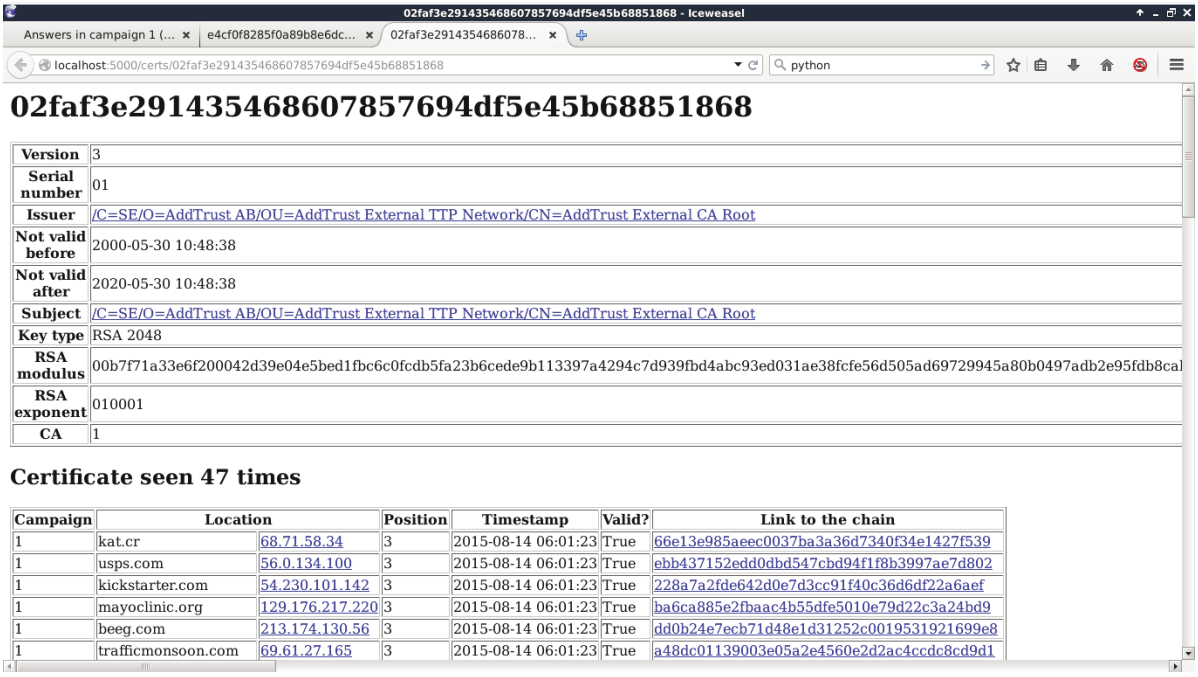


Figure 4.6: `concerto` screenshot: description of one particular certificate.

First, as shown in [HBKC11], the connections must be issued from a unique source: sending the same stimulus from different sources to the same IP sometimes leads to different answers, and in different certificates in particular.

Another necessary rule is to use a well-defined and stable stimulus for the whole campaign: for most of the studied parameters, it is useless to compare data obtained using different `ClientHello`: how to interpret the server-selected ciphersuite when the proposed list of ciphersuites varies from one host to the other? This is why we find it hard to analyse data collected in a passive way, where we do not control precisely the sent stimuli, at least to infer information about servers' behaviour.

For entire IPv4 measures, it is also important to find an acceptable trade-off between the duration of the campaign and the packet loss. As the time spent acquiring the data (the exposure time) increases, more IP addresses can appear or disappear during the campaign: we would prefer this time to be as short as possible. On the contrary, sending too many packets can overload network links or trigger alarms, leading to dropped packets. As described in the previous chapter, early campaigns such as ours usually spanned their measures across two to three weeks, but more recent datasets tend to choose shorter exposure time (around 1 day). In this case, to accommodate packet loss during the SYN-scanning phase, it is not sufficient to randomise the IPs, but it is also possible to send multiple SYN packets or to remember past results in the case of recurrent measures.

4.2.3 The need for metadata

Finally, with such guarantees on the collection phase, it is possible to work on the data (the messages sent by the servers). Usually, to improve our analysis, we also need the following metadata:

- the `ClientHello` message used as stimulus for the campaign, to check the conformance and the quality of the server answer;
- the timestamp of each answer, to validate the certificate at retrieval time;
- one (or several) certificate store(s) to check the validity of the certificate chain against a given configuration.

4.3 Towards reproducible analyses of TLS datasets

Provided with different acceptable datasets, we define a modular methodology to allow for reproducible analysis. Our goal is to feed a database with the data and the relevant metadata to produce statistics and to allow for finer-grain requests.

The different phases of our methodology are the following:

- **Context preparation.** Depending on the available information, we can inject the contents of the `ClientHello` messages sent and the certificate stores used to validate certificate chains;
- **Answer injection.** This step consists in parsing the results to record the contents received for each contacted server;
- **Certificate analysis.** The previous steps respectively feed the database with trusted certificates and with all certificates from the campaigns. It is thus possible to parse certificates, find all possible links and build chains;
- **Statistics.** The final step consists in producing statistics on different criteria, possibly with the application of filters (e.g. only consider the hosts that are trusted with regards to a given trust store).

All parsing operations, from the stimulus analysis to the answer extraction and the certificate parsing, rely on tools written using `parsifal`. `parsifal` is a framework allowing to write robust and efficient binary parsers. It will be discussed in details in chapter 6. The analyses proposed by `concerto` (answer compatibility, certificate chain quality, etc.) result from our knowledge of the TLS and X.509 specifications and from our experience with real-world implementations.

In the following figures, the database tables are represented by grey parallelograms, programs by rectangles and binary files by circles (grey ones correspond to binary data inside the database). The detailed database schema is given in appendix A.

4.3.1 Context preparation

Two simple tools are used to make the database aware of the campaign context: `injectStimulus` and `injectCerts`. The first one injects the versions, ciphersuites and extensions proposed by the client during the campaign inside the database. Such information then allows us to spot anomalies as those described in section 3.3.3, i.e. when a server chooses a ciphersuite that was not proposed.

`injectCerts` simply adds given certificates into the list of certificates to parse later. It is logically the first step allowing to flag certificates, chains and ultimately hosts with a certificate store. Before checking that a given host is trusted according to a certificate store, we indeed need to make sure the corresponding certificates are loaded.

In some cases, it may be useful to compare campaigns using the same certificate store at different times. For example, Mozilla products use the certificate store from the NSS library. We wrote scripts to extract the trusted certificates and the Extended Validation root certificates from NSS source. It is thus easy to check out the source code from NSS at the time of a given campaign and extract the corresponding certificate store.

The certificate store generation is represented in figure 4.7, whereas the injection tools are described in figure 4.8.

4.3.2 Answer injection

Here, the goal is to process the bulk data to extract the relevant data for each host (which can be identified by a domain name or an IP address). First, we need to characterise the answer type (non-TLS contents, SSL/TLS alerts or SSL/TLS Handshake messages). Depending on this type, we may gather more information: the protocol version, the alert type, the chosen ciphersuite, and the presented certificate chain.

Of course, this step will be different for each dataset. For the EFF datasets and for ours, we directly parsed the raw TCP stream sent by each server in response to the stimulus, with the `injectAnswerDump` tool. On the contrary, for ZGrab results obtained from `scans.io`, we had to write a different tool, `injectZGrabResults` to process the compressed custom JSON format (where the messages had already been parsed).

Finally, this step produces an `answers` table containing one line per host, a `chains` table with unique certificate chains and the set of all unique certificates collected for the given campaign in raw format. If the context information about the `ClientHello` used for the considered campaign is present, `injectAnswerDump` can also use it to enrich `answers` and flag inconsistencies.

Figure 4.9 describes how raw data are parsed and injected into the first tables and the global certificate store.

4.3.3 Certificate analysis

In this step, we first parse all certificates with `parseCerts` to extract relevant information such as public keys, distinguished names and some extensions.

Then, we build and check all the existing links, i.e. the list of (s, i) couples where s is a certificate signed by i. This operation, led by `prepareLinks` and `checkLinks`, is the most expensive operation, since it considers all the available certificates, with no regard to their origin. The reason we use two different tools is that the second step, checking cryptographic signatures, can be run in parallel, whereas the first program only quickly enumerates all possible links based on distinguished names and extensions.

A link is confirmed when the following conditions are checked: the distinguished names match, the key identifier extensions (if present) match, the issuer is a CA (i.e. it contains a true `cA` boolean in the `basicConstraints` extension, and finally the signature is correct.

Next, we build all possible chains from the certificate chains presented by the servers in their TLS `Certificate` message. The idea is to start from the server certificate, and to try every possible certificate path using the previously computed links. As a matter of fact, with real-world data, the number of possible chains can be huge because of cross-certification between authorities. That is why we introduced a parameter, `max-transvalid` in our algorithm, to limit the number of out-of-sent-chain certificates that can be used while building a chain[4]; there is an exception: a missing root certificate

[4]The term *transvalid* was first used by the EFF to describe such chains.

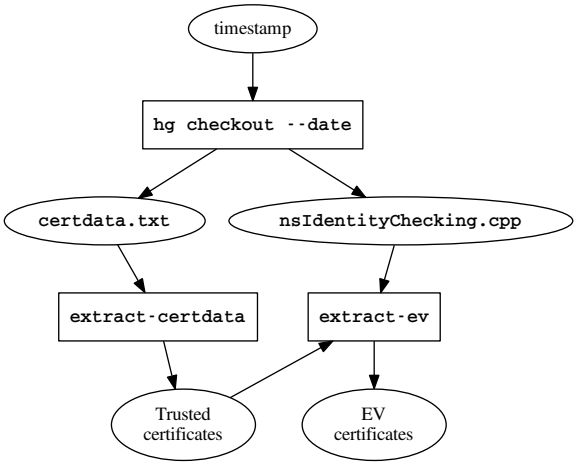


Figure 4.7: Preparation steps: trust store generation.

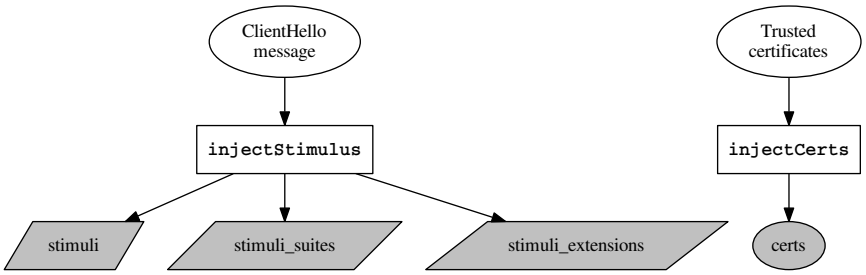


Figure 4.8: Preparation steps: initial metadata injections.

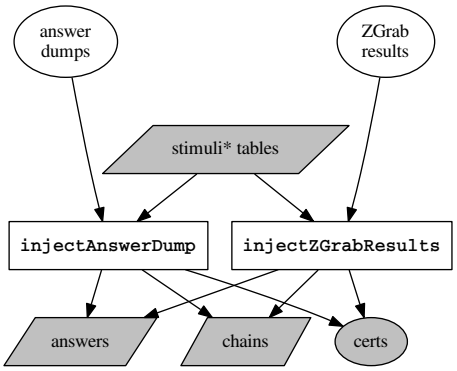


Figure 4.9: Data injection from different sources.

does *not* count as a transvalid certificate, since it may legitimately be omitted. It is worth noting that our tools does not follow certificate loops; this was not the case of all the TLS stacks we studied, since an early version of NSS could actually loop until a maximum chain length (20 certificates) was reached.

Once the `buildChains` program has produced the set of valid chains, the `flagTrust` tool takes trusted certificates as its arguments and performs three actions. First, it recursively flags the signed certificates as trusted by following the computed links, starting from the trusted certificates. Then, it marks as trusted all sent chains with a trusted server certificate (from the previous step). Finally, the program browses the built chains to flag those containing a trust root as trusted. We thus obtain three tables: `trusted_certs`, `trusted_chains` and `trusted_built_chains`. It is possible to call `flagTrust` several times with different trust stores.

Since the `max-transvalid` parameter restricts the number of built chains, the number of trusted chains in the last two tables might differ. This is further discussed in section 4.4.2.

Finally, `rateChains` takes into account several parameters into account to grade the built chains. For example, an A chain is a trusted RFC-compliant chain whereas unordered chains will be capped at C grade. The complete set of rules concerning certificate chains is given in table 4.1. It first consists in checking whether the chain ends with a self-signed certificate (is it complete?), and then checks whether the chain is trusted, whether it relies on external certificates (the transvalid character), whether it is ordered and whether it contains extra unused certificates. It is however important to understand that this grading system is not absolute, and only helps quickly categorise the certificate chains (A and B chains are essentially good, while E and F are clearly problematic). Furthermore, it is worth noting that these grades are always computed with regards to a given trusted certificate store.

Complete	**Trusted**	**Transvalid**	**Ordered**	**With extra certs**	**Grade**
Yes	Yes	No	Yes	No	**A**
				Yes	**B**
			No	-	**C**
		Yes	-	-	**D**
	No	No	Yes	No	**C**
				Yes	**D**
			No	-	**D**
		Yes	-	-	**E**
No	-	-	-	-	**F**

Table 4.1: Rules used to grade certificate chains.

Self-signed server certificates are commonly found on the Internet. `concerto` does not treat them in any special way. Such *chains* are considered complete if and only if the server certificate is also a valid certification authority (i.e. it contains the required `BasicConstraints` extension). Moreover, it will only be trusted if the self-signed server certificate itself is trusted, which is very improbable with a standard certificate store.

Figure 4.10 shows how our tools work to parse certificates, build and rate chains, according to the described procedure.

4.3.4 Overall statistics

With all the computed tables, we are finally able to produce statistics on whole campaigns, such as the preferred ciphersuites for a given `ClientHello`, the proportion of servers supporting TLS 1.2, or the quality of certificate chains sent by servers. The advantage of our toolset is that these statistics can easily be computed in a reproducible way on diverse datasets, which is obviously interesting with recurring campaigns such as those published on the `scans.io` website. Moreover, the flexibility provided by our trust flags allows for finer-grained statistics on restricted subsets, e.g. hosts presenting an EV certificate or hosts presenting an RFC-compliant certificate chain.

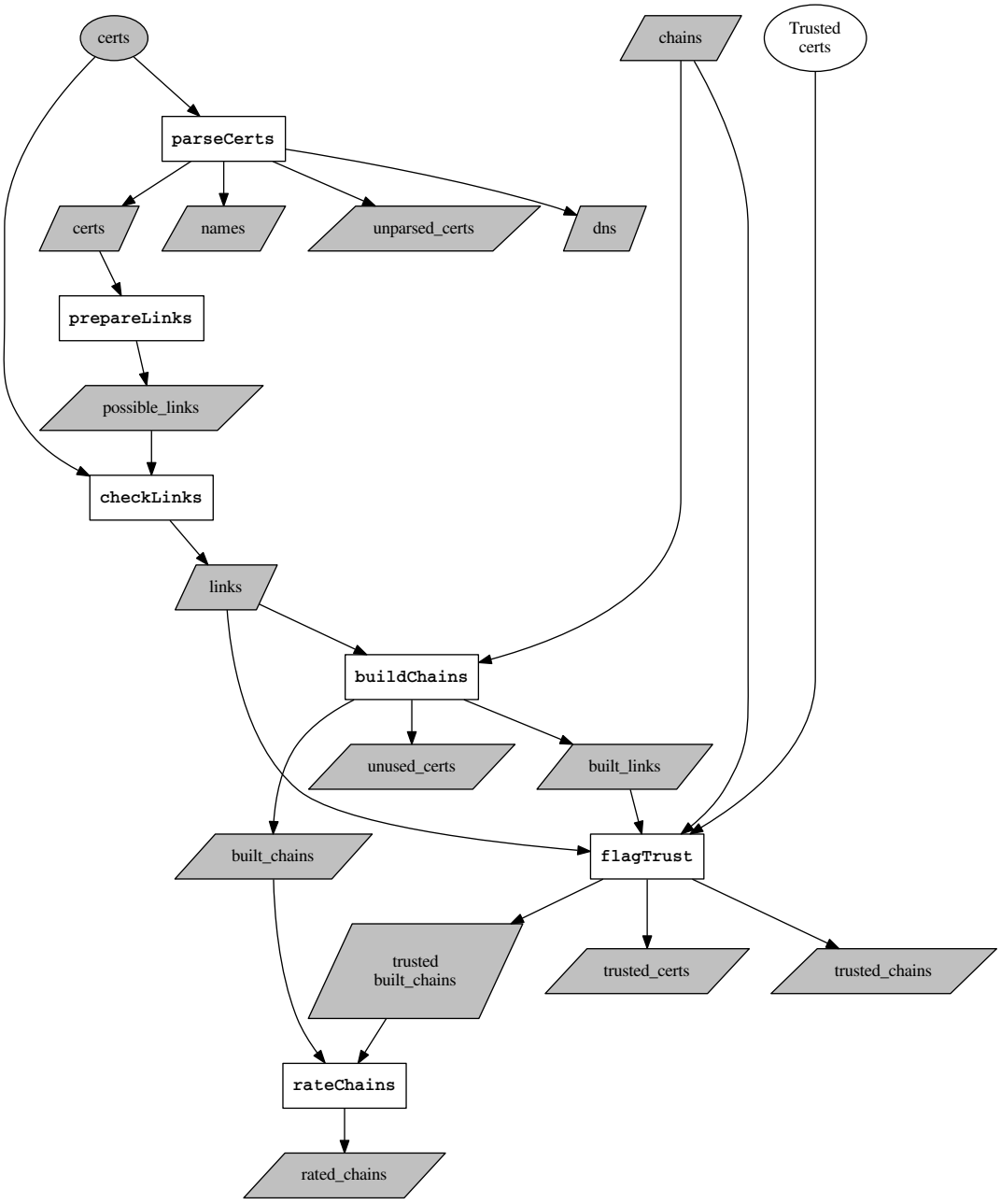


Figure 4.10: Programs used to analyse certificates and chains.

4.3.5 Tool cinematics

Finally, figure 4.11 presents the overall process to inject data, analyse them and produce statistics, assuming that we use the NSS trust store at validation time and that we know the `ClientHello` message sent. This was the standard process we applied to the studied campaigns.

4.4 Discussion

In this section, we analyse the backend we chose, essentially based on CSV flat files. Then we discuss `max-transvalid`, a customisable parameter of the process, and the issue with X.509v1 certificates. Finally, we take a step back to analyse our first results published in 2012 with a critical eye.

4.4.1 Some words on the implementation

Our current toolset is made of small tools written in OCaml, Python or Shell, that work on a shared directory containing the *database*. In practice, the processed data is stored into simple (but sometimes huge) CSV files. To produce the presented tables, we only have two ways of processing data in our tools: we either need to load the whole table to build a graph (e.g. to enumerate all possible links), or we read them as a stream, one row at a time (e.g. to check possible links). In both cases, CSV files suit our needs, and can still be inspected with standard tools. Figure 4.12 contains typical figures for a single-stimulus full IPv4 campaign.

We also tried to feed these tables into two SQL engines: `sqlite` and PostgreSQL. Such database engines appropriate to write fine-grain requests, whereas CSV is more adapted to bulk computation. The web application in `concerto`, shown in figures 4.3 and 4.4, relies on this SQL backend to allow the user to browse the different answers or certificates in the injected datasets. After computing the relevant indices, we are able to process complex requests, but PostgreSQL seemed to be more efficient, especially since its proposed data types are much richer, and would allow for easy optimisation.

On a side note, to help us quickly investigate subsets of answers, and reduce the size of the relevant data, we also developed a tool, `filterDataDir`, which can extract a subset of a campaign: it works on CSV tables and produces new, smaller CSV tables, containing only the relevant data. The extraction can be based on different criteria, such as a specific trust flag associated to a given trust store, a list of IP addresses, or the domain name in the server certificate.

Finally, it might be worth considering the use of a graph-oriented database to inject the `links` table. This would allow us to reuse existing tools, instead of relying on specific tools for this part of `concerto`.

4.4.2 Tuning the `max-transvalid` parameter

Building all the possible certificate chains from the received `Certificate` messages and the whole set of available certification authority in the dataset can be a daunting task.

Figure 4.13 presents a simpler version of the real-world example presented in section 4.1.3. Grey nodes were sent by the considered servers, whereas white ones were collected from other servers or from the certificate trust store. Depending on the chosen path, `concerto` will use a different number of white nodes. The total number of such nodes (excluding the last one) is the number of transvalid certificates in the built chain.

During our study, we observed that the `max-transvalid` parameter (the maximum number of transvalid certificates to use when building possible chains) could lead to very large `built_chains` and `built_links` tables. On the contrary, using a small value misses several built chains. In practice, the tables depending on the precise built chains are `trusted_built_chains` and `rated_chains`, so we might rate a host with a different grade, depending on the `max-transvalid` parameter.

The only statistics programs relying on the built chains is `computeChainsStats`, a tool computing statistics on chain quality. All the other tools use the `trusted_chains` table, which is populated using a breadth-first search in the global certificate graphs from the trusted roots, as explained earlier.

Let us look again at the example in figure 4.13. We assume that only the `B root` certificate is present in the trust store (and not the `A root` certificate). Thus only the lines ending with B_R will lead to a trusted built chain. Since these chains contain 1 or 3 transvalid certificates, a `concerto` run using 0 for `max-transvalid` would not build any trusted chain in this situation.

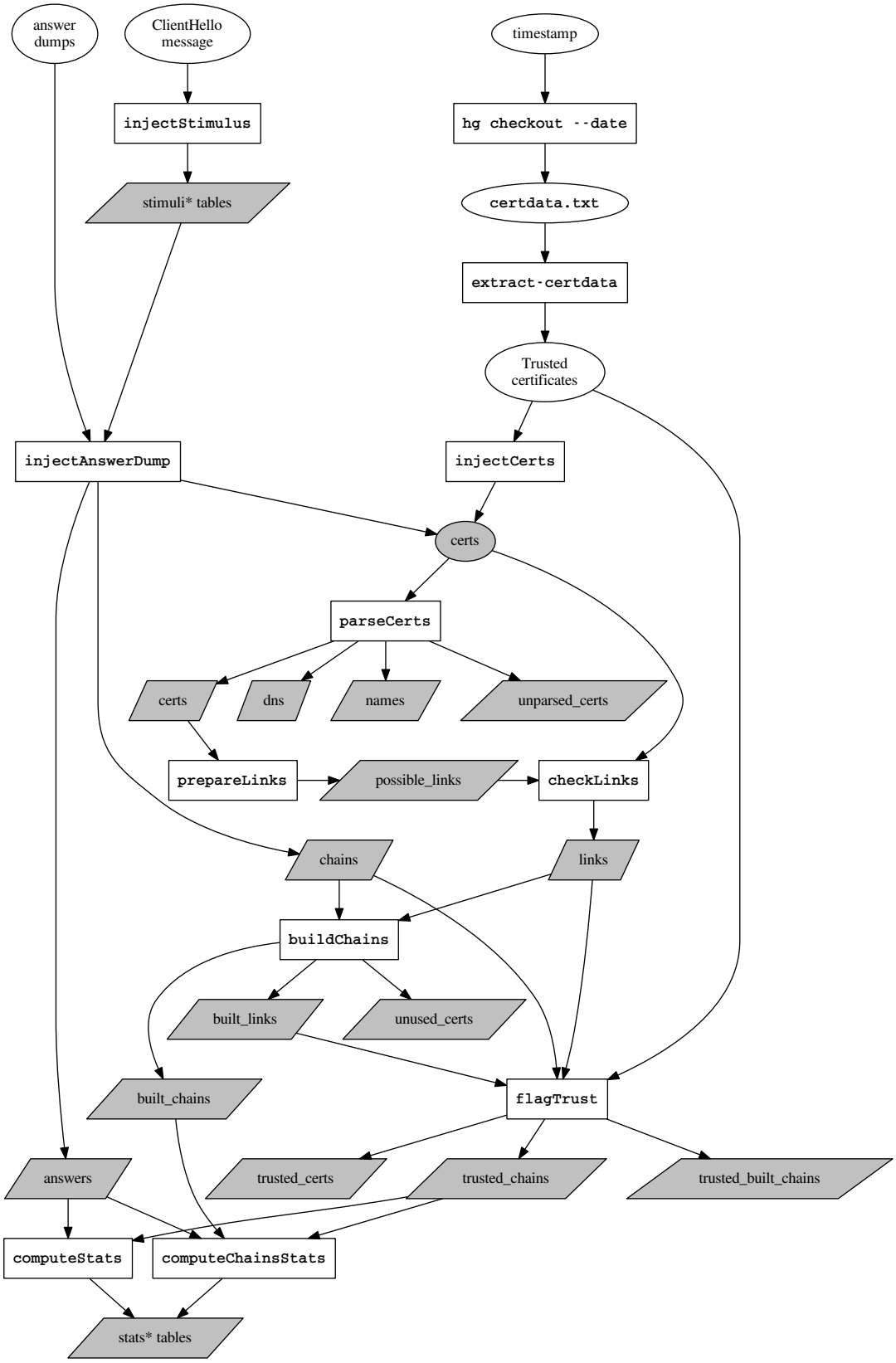
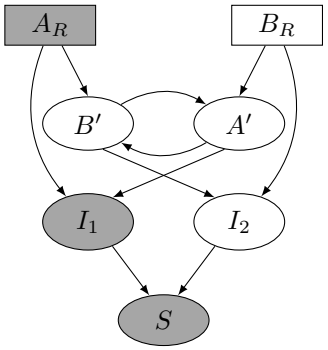


Figure 4.11: Overall process for a typical campaign.

Table	N rows	Size
`answers.csv`	40 M	4 GB
`chains.csv`	20 M	2 GB
`parsed_certs.csv`	10 M	6 GB
`links.csv`	14 M	1 GB
`built_chains.csv`	120 M	12 GB
`trusted_certs.csv`	6 M	300 MB
`trusted_chains.csv`	9 M	450 MB

Binary contents	N	Size
raw certificates	10 M	10 GB

Figure 4.12: Typical figures regarding a full IPv4 campaign.



$$
\begin{array}{lcll}
S \leftarrow I_1 \leftarrow A_R & \Rightarrow & \text{Transvalid certs:} & 0 \\
S \leftarrow I_1 \leftarrow A' \leftarrow B_R & \Rightarrow & \text{Transvalid certs:} & 1 \\
S \leftarrow I_1 \leftarrow A' \leftarrow B' \leftarrow A_R & \Rightarrow & \text{Transvalid certs:} & 2 \\
S \leftarrow I_2 \leftarrow B_R & \Rightarrow & \text{Transvalid certs:} & 1 \\
S \leftarrow I_2 \leftarrow B' \leftarrow A_R & \Rightarrow & \text{Transvalid certs:} & 2 \\
S \leftarrow I_2 \leftarrow B' \leftarrow A' \leftarrow B_R & \Rightarrow & \text{Transvalid certs:} & 3
\end{array}
$$

Figure 4.13: Example of a complex chain, possibly leading to 6 different built chains. The grey certificates are sent by the server.

To assess the impact of the parameter on chain quality statistics, we experimented with values from 0 to 6 for the `max-transvalid` parameter, on a full campaign. We then confronted the number of trusted built chains identified from the `trusted_built_chains` table with the total number of trusted chains (using the `trusted_chains` table). We thus only considered distinct certificate chains, with no regards to the number of times they were observed.

Another way to assess the loss of precision of a small `max-transvalid` parameter is to consider the proportion of trusted hosts for which a trusted chain can be built, taking this time into account the repetition of a given chain among different hosts. The complete results are given in figures 4.14 (for chains) and 4.15 (for hosts).

`max-transvalid` value	Number of built trusted chain		Proportion of all trusted chains
0	896,309		21.71 %
1	3,681,379	(+ 2,785,070)	89.15 %
2	4,065,397	(+ 384,018)	98.45 %
3	4,128,439	(+ 63,042)	99.98 %
4	4,128,832	(+ 393)	99.99 %
5	4,129,018	(+ 186)	99.99 %
6	4,129,195	(+ 177)	100.00 %

Figure 4.14: Number of trusted chains that can be built, with regards to different values of the `max-transvalid` parameter.

For values ranging from 0 to 3, the number of trusted chains (respectively hosts) for which a trusted chain can be built progressively grows up to reach almost the total number of trusted chains (resp. hosts). The total is actually reached when `max-transvalid` is equal to 6, but the loss of precision between values 3 and 6 is negligible. To avoid producing too much data while avoiding missing too many built chains, we chose to use 3 as our `max-transvalid` parameter for all further analyses concerning chain quality.

max-transvalid value	Number of hosts with a built trusted chain		Proportion of all trusted hosts
0	3,339,641		20.03 %
1	15,836,582	(+ 12,496,941)	94.96 %
2	16,542,076	(+ 705,494)	99.19 %
3	16,675,667	(+ 133,591)	99.99 %
4	16,676,160	(+ 493)	> 99.99 %
5	16,676,530	(+ 370)	> 99.99 %
6	16,676,723	(+ 193)	100.00 %

Figure 4.15: Evolution of the number of trusted hosts for which a trusted chain can be built, with regards to different values of the `max-transvalid` parameter.

4.4.3 Handling X.509v1 certificates

The initial version of the X.509 standard did not include extensions. Since extensions are now used to indicate the fact that a given certificate is a certificate authority (CA), X.509v1 certificate used to be implicitly trusted as CAs.

Nowadays, trusted roots do not emit such certificates anymore, but trusted roots themselves sometimes still use the old format. We would thus be inclined to accept X.509v1 certificates as CAs, since other, non-root, X.509v1 certificates are never linked to trusted root in practice.

In an early version of `concerto`, we considered all X.509v1 certificates as certificate authorities. Yet, this led to a huge combinatorial explosion with several classes of certificates: some appliances or virtual machine management engines produce unique X.509v1 certificates for each equipment or instance, with the same subject and the same issuer. Though, with X.509v1 certificates, as we can not rely on extensions such as *SubjectKeyIdentifier* and *AuthorityKeyIdentifier* to remove irrelevant possible links to check, the mere presence of n of such appliances would trigger n^2 certificate pairs to check.

For example, a full IPv4 campaign contains around 1.2 million X.509v1 distinct certificates. Among those, we observe many certificate clusters, where the subject and issuer fields are the same for all certificates. The biggest one contain more than 140,000 distinct certificates (associated to the "CN=Parallels Panel" common name), and there are 122 other clusters containing more than 100 certificates each.

To avoid having to deal with such an amount of bogus links to check, we decided to only consider as CA the X.509v1 certificates included in the trusted roots (which only represent a dozen certificates). This reproduces the behaviour of modern TLS stacks.

4.4.4 A critical eye on the previous analyses

When we chose to reproduce our analyses on our previous datasets, we had to be prepared to get inconsistent results between the results published in 2012 [LÉMD12] and the newly obtained results.

However, most of the results were identical, since we uncovered no major bugs between 2011 and 2015 in our parsing tools. However, one part of the results we could not reproduce was the statistics concerning EV certificates. Even if the global results are confirmed, the number of hosts flagged as EV hosts was drastically smaller in our first study. Determining that a host relates to an EV root was essentially done looking at a list of certificates obtained from the Opera website, whereas we improved the EV certificate collection in `concerto`, by extracting the metadata from NSS source code. This new method allows to confidently compare EV hosts at different times, simply by extracting the certificate store from the code repository at the given time.

There is another limitation we spotted while working on more recent data: the growing use of SHA384 among HTTPS servers. Since the cryptographic library used in `concerto`, namely `CryptoKit`, did not support SHA384/SHA512 in its early version, we were not able to properly check the corresponding signature, which led to an error in the chain validation (`Unknown algorithm`). This has since been fixed, but it shows the need to compare our results and to improve our tools to detect anomalies.

More generally, several criteria have evolved since 2012, e.g. the ciphersuites strength classification. Such differences are discussed in the following chapter.

4.5 Concrete examples

In this section, we describe several interesting features of `concerto` on small datasets (Top Alexa 1 Million and Top Alexa 1000). The results on larger datasets are presented in the next chapter.

4.5.1 Simple requests

First, we look at the repartition of the answers by type or by ciphersuite. `concerto` allows to get the repartition of selected ciphersuites for a given campaign, or the proportion of servers selecting a ciphersuite with forward secrecy. Moreover, since our goal with `concerto` was to propose a way to compare results across campaigns, we automated the retrieval and analysis of TLS data concerning Top Alexa 1 Million campaigns available on the `scans.io` website, between August 2015 and April 2016[5].

The results of this simple experiment are given in figure 4.16 and figure 4.17. The first figure shows an important increase of the proportion of servers speaking TLS among Top Alexa 1 Million hosts, from 35 % to almost 60 %. Only very few servers (around 0.1 %) answered with an alert for each campaign. In the second figure, we can see that ciphersuites with forward secrecy are preferred by a (still growing) majority of servers; it must however be noted that this result depends on the list of ciphersuites proposed by the client, which does not necessarily correspond to a given browser[6]. This illustrates the powerful aspect of having a simple and reproducible analysis methodology.

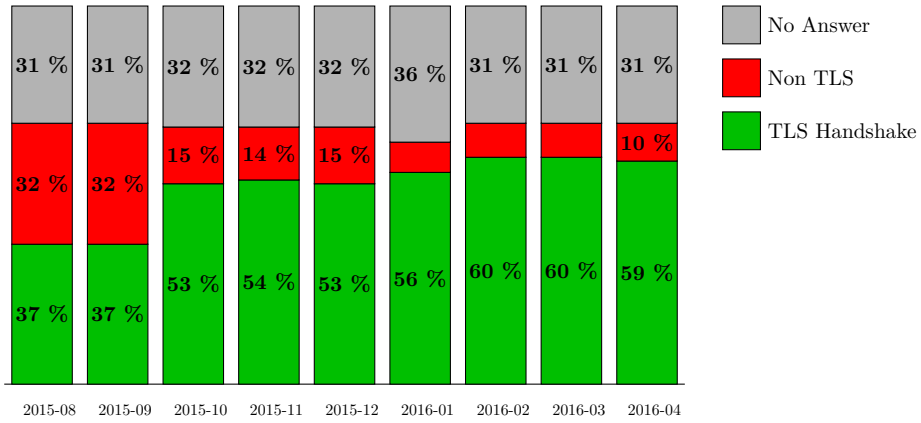


Figure 4.16: Answer types on 443/tcp port from Top Alexa 1 Million, between August 2015 and April 2016. Source: daily scans from `scans.io`, from the 1st of each month.

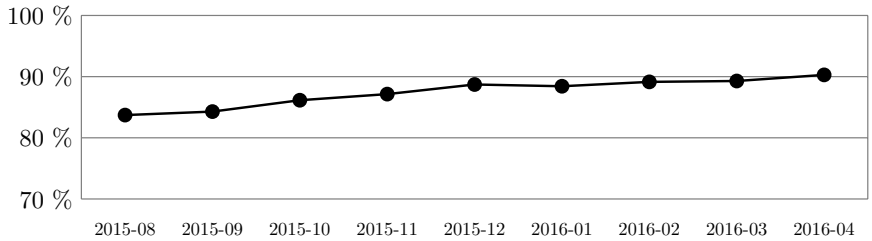


Figure 4.17: Proportion of TLS answers on 443/tcp port from Top Alexa 1 Million selecting a ciphersuite with Forward Secrecy, between August 2015 and April 2016. Source: daily scans from `scans.io`, from the 1st of each month.

[5] Before August 2015, the JSON schema was different, so they could not be parsed *as is* by `injectZGrabResults`.

[6] On a side note, a few servers (less than 100 in August 2015, up to 240 in April 2016) answered with an unknown ciphersuite. In theory, we thus could not properly classify the ciphersuite. Yet, this phenomenon can easily be identified as the early use of ciphersuites using algorithms such as Chacha20-Poly1305 [LCM+16], that had not yet been standardised in TLS.

4.5.2 Long chains

One aspect of looking at TLS campaigns is that it becomes possible to look for anomalies, such as very long `Certificate` messages. For example, in the simple Top Alexa 1000 campaign we studied earlier, we found a message containing 9 certificates. The best chain we could build from it is represented in figure 4.18. It is worth noting that the sent chain actually only contains 4 *distinct* certificates: even if the built chain seems to be perfect, it contains duplicate and out of order certificates. To get the full picture, we thus have to look at the tables given in figure 4.19.

The same request can easily be replayed on another campaign. For example, on a Top Alexa 1 Million campaign from April 2016, the longest `Certificate` message we observed was 30-certificate long. Among those, only 14 certificates were unique. Figure 4.20 presents the best certificate chain that could be built. The graph shows that more than half the sent certificates (in grey) do not even relate to the server certificate!

We also looked at the longest built chains on different samples. For the Top Alexa 1000 dataset, we obtained the chain described earlier in figure 4.5. However, contrary to the chain described in red in this figure, the longest built chain is made of 7 certificates. The Top Alexa 1 Million gave essentially the same results, since the longest built chains ended with the same CAs.

4.5.3 Some figures about hostnames

We can also use the database to get interesting figures concerning hostnames: for example, on a recent Top Alexa 1M campaign, we found nine certificates containing more than 500 DNS names in their `SubjectAltName` extension. It is not unusual to find long lists of domain name for Content Delivery Networks (CDN) certificates.

Another way to share the same certificate across several domain names is to use a wildcard certificates, where the hostname in the certificate starts with a `*`. For a recent Top Alexa 1M campaign, among the 467,618 hosts that presented a trusted certificate, more than a third (138,794) presented a server certificate containing a wildcard).

This kind of tests are not yet implemented in the GUI but it would be interesting to implement them in the future. For example, it is useful to check that the server certificate corresponds to the contacted host, or to check that `NameConstraints` extensions correctly apply along a given built chain.

4.6 Concluding thoughts on `concerto`

Since 2011, a lot of researchers have conducted SSL/TLS campaigns to better assess the actual state of TLS deployment. Today, such datasets are pervasive, easily available, and there exists high-quality sources. Yet, it is not that easy to study them in a reproducible and uniform way. With `concerto`, our goal was to be able to reproduce previous analyses, and to provide a sound methodology to assess the quality of TLS deployment in the wild.

One of the major advantages of using many small tools is the ability to easily consider subsets of the overall campaign. This can be done by extracting data with `filterDataDir`, or by flagging trusted hosts and certificates with `flagTrust`. Both can be run after the data has been inserted and the links have been computed.

We believe `concerto` could also be used to produce efficient differential analysis regarding several indicators. The following chapter presents several datasets we were able to obtain, study, and tried our approach on. We could indeed easily reproduce several indicators across them.

Future work concerning `concerto` will be to improve backend efficiency and to explore further fine-grain SQL requests, beyond our first experiments. Another useful improvement would be to add revocation information to the database. Finally, it would be interesting to implement more checks and to add the possibility to provide global reports that would help an administrator to identify anomalies and to fix configuration errors.

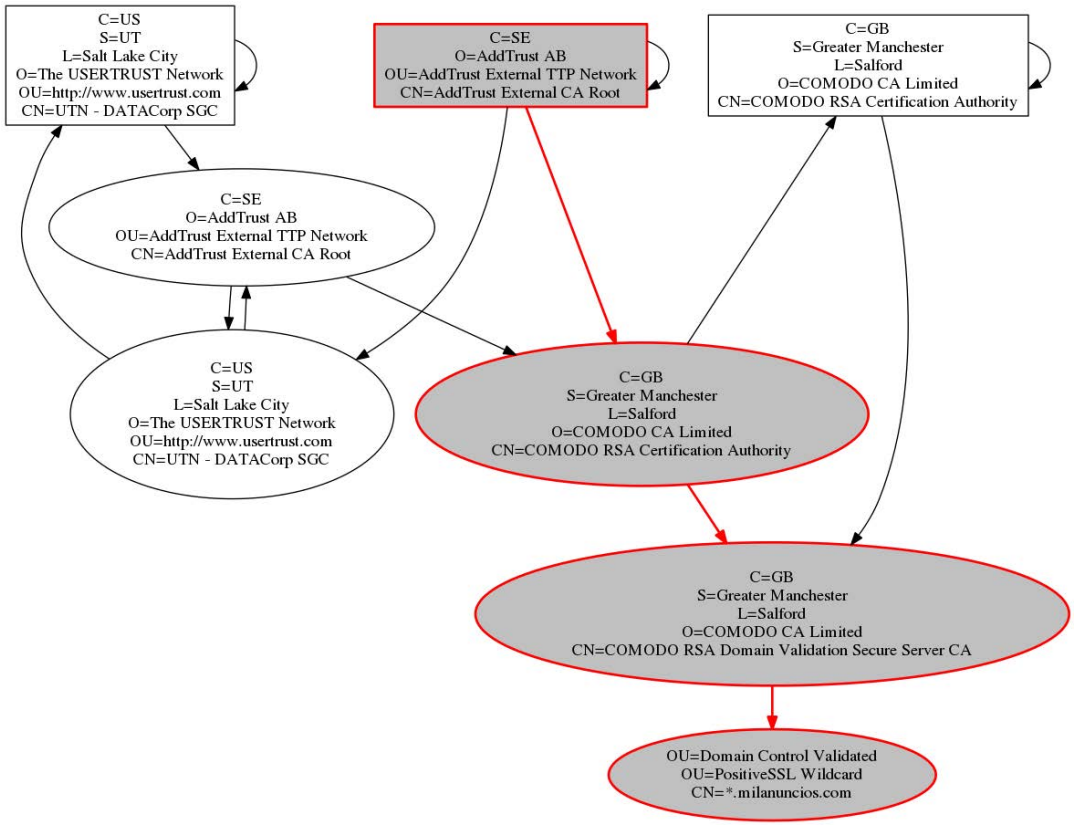


Figure 4.18: Graph presenting the best built chain (in red) for a given `Certificate` message. Even if the resulting chain seems to be perfect, the sent chain is out of order, and even contains useless duplicate certificates (see figure 4.19).

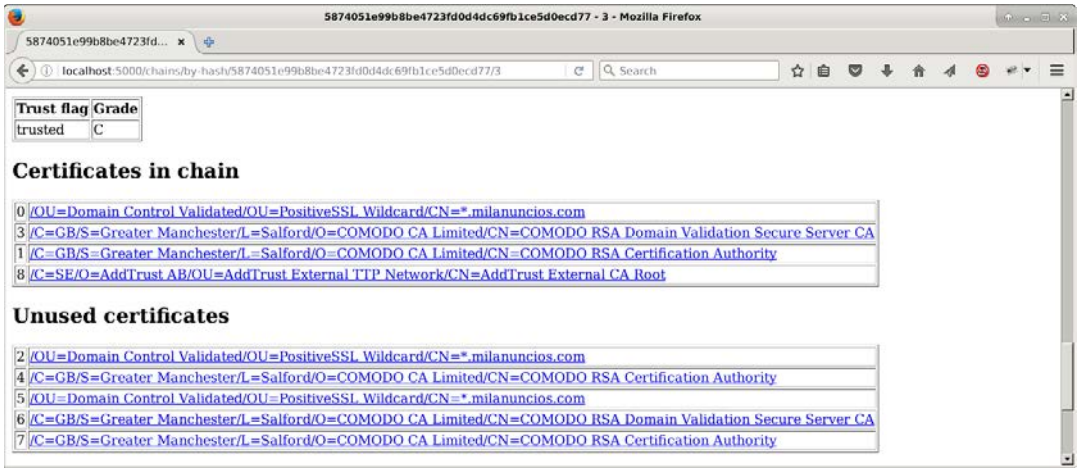


Figure 4.19: The raw data describing the built chain represented in figure 4.18. The tables show that the chain is unordered and that several certificates are duplicated.

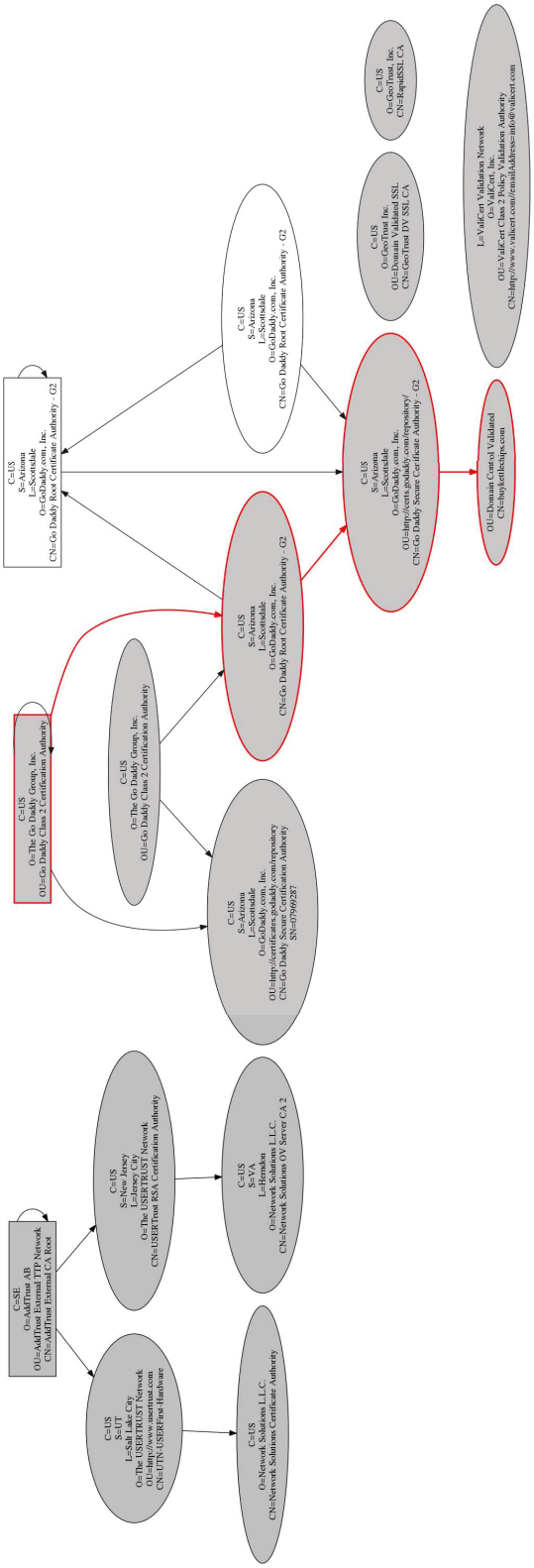


Figure 4.20: Longest chain found in a Top Alexa 1M campaign from April 2016. Several certificates were sent by the servers, despite the complete absence of relation to the server certificate.

Chapter 5

A comparative analysis of the HTTPS ecosystem between 2010 and 2015

Building on `concerto`*, the framework described in the previous chapter, this chapter presents results using various datasets to understand the HTTPS ecosystem and its evolution between 2010 and 2015. It extends results published in 2012 about 2010 and 2011 SSL/TLS campaigns [LÉMD12].*

Several results differ from those published in 2012, for several reasons. First, the certificate store used to select trusted and EV certificates were different. Then, the methodologies used to assess several parameters have changed: the classification of ciphersuites by strength was revised, we chose not to consider X.509v1 certificates as certification authorities by default, and we modified how to study the server behaviour (as explained in the corresponding section).

Using `concerto`, we can compare the HTTPS ecosystem at different times, using various datasets. Some of them are campaigns led with Telecom SudParis in 2011 and 2014, while others are public datasets like the EFF ones from 2010 or the more recent Internet-Wide Scan Data Repository hosted by the ZMap Team at the University of Michigan.

We analyse several parameters of the collected data and their evolution. Several studied criteria are related to the protocol (version, ciphersuite, extensions) and others are about certificate chains (chain order, signature algorithms, key lengths, validity periods). For our multi-stimulus campaigns, we also used a differential analysis on the answers, to infer information about server behaviour.

In addition to overall analyses, we also focus on two subsets of TLS hosts within our measure: the trusted hosts (i.e. hosts sending a valid certificate chain at the time of the probing) and the EV hosts (presenting a trusted, so-called *Extended Validation* certificate).

Section 5.1 lists the available datasets, and explains which ones we chose to use. For the campaigns we studied, section 5.2 presents global statistics: number of contacted hosts, proportion of TLS servers... The following sections contain detailed results on these campaigns: section 5.3 addresses TLS parameters such as the protocol version or the supported ciphersuites; section 5.4 studies the certificate chains; section 5.5 compares the answers of each server for different stimuli to learn more about server behaviour.

5.1 Available datasets

Beyond our campaigns, several projects aiming at understanding the SSL landscape have performed similar measures on the internet. In this section, we enumerate several datasets and assess their relevance for our approach.

5.1.1 2010: the first public campaigns

The EFF (Electronic Frontier Foundation) performed two campaigns to explore the *SSLiverse* in August and December 2010. They presented their results about the gathered certificates [EB10a, EB10b]. These

campaigns follow the first methodology described in section 3.1, i.e. a full IPv4 space scan.

One main difference with our approach is the stimulus used: the EFF sent an SSLv2 `ClientHello` whereas we tried to obtain more information by sending different stimuli. Moreover, contrary to their study, our work did not focus solely on the certificate chains sent by trusted hosts; we also broadened the criteria to assess the quality of servers' answers and to show trends by focusing on different subsets of hosts.

The EFF published some of their tools, as well as the data, both processed (the certificate database) and unprocessed (the raw answers). This allowed us to understand their collection methodology and to analyse their data ourselves.

To enumerate 443/tcp open ports on the IPv4 space, they used `nmap`. Then, to grab the certificates, they used a custom Python script sending an SSLv2 `ClientHello`. Finally, they analysed the collected certificates with OpenSSL. Moreover, the set of IPs initially probed is described in the scripts, as well as the certificate stores used to check chain validity. Yet, in their second measure (December 2010), it seems they did not enumerate the 443/tcp hosts again and used the August 2010 list instead, which explains the reduced number of hosts in this campaign (see section 5.2 for more details).

5.1.2 2011: analysis of the impact of different points of view

In 2011, Holz et al. from the University of München gathered and studied different data sets: active probing of popular sites, passive monitoring on a 10 GB link and the EFF campaigns [HBKC11]. They provided a thorough analysis of the certificates received for each of the campaigns studied. In particular they compared the certificates received by clients from different source addresses and obtained differences in these sets. Using real world traffic is a very interesting source of information and it is complementary to full IPv4 host enumerations.

Since the researchers' datasets were either already available (EFF) or only partial (they only provided the collected server certificates for popular sites from different locations, not the whole TLS answers), we did not investigate this source further.

5.1.3 2010-2011 and 2014: multiple stimuli to grasp the server behaviour

In 2012, we presented a study of the HTTPS ecosystem, based on several full IPv4 scans: the EFF datasets and the campaigns described in chapter 3 [LÉMD12]. Data were collected using the same methodology in 2014.

We used custom Python scripts to scan open 443/tcp ports and to retrieve server answers to different `ClientHello` messages. For the article, the data was then analysed using Parsifal (presented in chapter 6) and ad-hoc scripts.

5.1.4 since 2012: SSL Pulse

Since 2009, Qualys proposes an online tool called SSL Server Test[1]. In 2010, Ristic presented an SSL survey, based on these tools, focusing on a DNS enumeration of HTTPS hosts [Ris10]. His goal was to assess the quality of TLS answers from servers reachable via a DNS hostname: for example, at the time, SSLv2 was still widely supported, while TLS 1.1 and TLS 1.2 were virtually nonexistent. Since April 2012, the study has been extended into SSL Pulse [Lab15b], a dashboard aiming at measuring various parameters related to SSL on a set of 200,000 popular servers.

Even if these tools give an insight into the HTTPS ecosystem, they do not represent a usable dataset since the raw results are not available.

5.1.5 2012: the Internet Census

In March 2013, an independent researcher published results on an Internet-wide scan on multiple ports led in 2012 [Bot12]. To complete this so-called Internet Census, the researcher took control of open embedded devices across the Internet[2] and used them as a botnet to probe the Internet. The collected data (600 GB of compressed data, 9 TB uncompressed) were made public via a BitTorrent link.

[1] `https://www.ssllabs.com/ssltest/`

[2] Many online devices expose administration interfaces with default passwords, and the attacker/researcher could easily take control of them.

Concerning TLS data, we observed that the answers were truncated well before the end of the first server flight, making it impossible to get relevant data in an homogeneous way. Moreover, beyond the obvious illegal and unethical aspects of the approach, it was shown that the data, though apparently authentic, "suffered from qualitative problems such as methodological flaws or lack of meta data" [KHF14]. This is why this dataset does not constitute a valid dataset.

5.1.6 2013-2015: ZMap and ZGrab

Since 2013, several tools have been published to lead Internet-scale scans, like ZMap [DWH13], an asynchronous scanner relying on SYN-cookies to track answers to given probes. Such tools allow for scanning the Internet very quickly (in a matter of minutes or hours), but at the expense to an increased data loss; the usual countermeasures are to randomise the address space and to reemit packets.

In particular, Durumeric et al. used ZMap and other tools to analyse the HTTPS certificate ecosystem in 2013 [DKBH13]. They estimated the packet loss in the SYN-scanning phase at 2 %. Since they launched 110 campaigns over a period of 14 months, they compensated by reusing the list of IP addresses acquired in the first phase across campaigns. Their data has been made public on the `scans.io` website. For our study, we considered two of the recurring datasets provided by Durumeric et al.: the Top Alexa 1 Million scans and the full IPv4 campaigns, both on the 443/tcp port using a TLS 1.2 `ClientHello`.

There were also recent studies exploring the use of TLS in electronic mail protocols such as POP, IMAP or SMTP [DAM+15, MZSH15, HAM+15], but they were not taken into account in this thesis.

5.1.7 Summary

Table 5.1 summarises the studied campaigns and their properties. We finally retained several suitable datasets, following several criteria, including those described in section 4.2:

- the data must be collected using a rigorous and documented protocol, and must originate from a unique source IP;
- when possible, the exact `ClientHello` message used as stimulus must be available, to check the conformance and the quality of the server answer;
- the timestamp of each answer must be available to validate the certificate at retrieval time.

	Campaign type	**Date**	**Available**	**Retained**
EFF [EB10a]	1	2010	yes	**yes**
Holz et al. [HBKC11]	1 + 2 + 3	2011	partially	no
Chapter 3	1	2010, 2011 and 2014	yes	**yes**
SSLPulse [Lab15b]	2	recurring since 2012	no	no
Internet Census [Bot12]	?	2012	yes	no
Durumeric et al. [DKBH13]	1 + 2	recurring since 2013	yes	**yes**

Table 5.1: Campaign description. Campaign types are 1 for Full IPv4 campaigns, 2 for Hostname-based scans and 3 for Passive analyses; the last column indicates the studied datasets.

5.2 Global statistics on the campaigns

To test our approach, we focused on the following datasets:

- both EFF campaigns from 2010;
- our 2010, 2011 and 2014 datasets, respectively corresponding to 1, 7 and 11 stimuli;
- a Full IPv4 ZGrab campaign from August 2015;
- a Top Alexa 1M ZGrab campaign from August 2015.

The complete set of stimuli for each campaign is described in table 5.2. For multi-stimuli campaigns, we sent standard stimuli, but also historical, SSLv2-compatible, `ClientHello` messages, and feature-oriented stimuli, to test the support for specific features. Table 5.3 describes the number of hosts with an open 443/tcp port, and the proportion of TLS answer types in each campaign (for multi-stimuli campaign, the values are computed for a standard TLS 1.0 stimulus).

Date	Source	Sent stimuli
2010/07	Chapter 3	TLS 1.0
2010/08	EFF	EFF
2010/12	EFF	EFF
2011/07	Chapter 3	TLS1.0-NoExt, TLS1.0, TLS1.2-modern *Historical stimuli:* SSL2, SSL2-TLS1.0 *Feature-oriented stimuli:* DHE, EC
2014/03	Chapter 3	TLS1.0-NoExt, TLS1.0 (twice), TLS1.2, TLS1.2-modern *Historical stimuli:* SSL2, SSL2-TLS1.0 *Feature-oriented stimuli:* DHE (twice), EC, EC-explicit
2015/08	`scans.io`	TLS 1.2

Table 5.2: Stimuli used for each studied campaign. The stimuli were described in table 3.2 page 94. Several stimuli were sent twice in 2014 to measure the IP stability, as described in section 3.4.4.

Campaign			N hosts with	TLS answers	
Date	Source	N stimuli	open port		
2010/07	Chapter 3	1	21,342,205	9,683,050	(**45** %)
2010/08	EFF	1	15,665,414	11,459,645	(**73** %)
2010/12	EFF	1	7,777,511	7,705,488	(**99** %)
2011/07	Chapter 3	7	26,218,647	11,313,083	(**43** %)
2014/03	Chapter 3	11	40,126,225	28,957,611	(**72** %)
2015/08	`scans.io`	1	46,231,300	34,083,332	(**74** %)
2015/08	`scans.io` (TA)	1	684,365	355,912	(**52** %)

Table 5.3: Proportion of TLS answers for each campaign (TA is for Top Alexa 1M).

There seems to be significant differences between the EFF campaigns and the other datasets. This may be explained by a long delay between the SYN scan completion and the second phase where TLS sessions were established: in our campaigns, as well as those retrieved from `scans.io`, both phases were run along. It does not seem to be the case for EFF campaigns; the problem is that within a three-week period, dynamic IP addresses often change. Moreover, it seems the SYN scan was not repeated in December by the EFF, which explains why there seem to be an important proportion of servers missing. Finally, the results show that the data were post-processed before publishing the results, to eliminate most of non-TLS hosts.

For each campaign, we used the NSS trust store, an easily accessible and representative certificate store. As described in chapter 4, the trust store, as well as the EV root certificates, can be retrieved from the project source code for the considered periods of times.

To enrich our findings regarding the criteria studied in the following sections, we chose to define subsets of the overall set of contacted hosts.

TLS hosts. TLS hosts are those that simply answered with a TLS handshake.

Trusted hosts. These servers presented a server certificate for which we could build a valid chain up to a root certificate in the NSS trust store. To build the chain up to a root certificate, the missing links could be gathered from other certificate chains or in the root certificate store. Our method to build chains is described in details in the previous chapter.

EV hosts. EV certificates are a novelty in the internet PKI officially introduced in 2007, which aims at improving the quality of certificates. The EV guidelines [For15] describe how the certificates should be issued, the audit procedures needed for the certificate authorities and the cryptographic algorithms that should be banned. An EV certificate must be issued by an EV authority (recognised by the browsers) and contain a certificate policy (a specific value in the `CertificatePolicies` X.509 extension) matching the EV authority. Once a certificate is validated and recognised as EV, the browsers typically use green address bars to indicate to the user that the site is EV-trusted. The EV subset consists of hosts that sent valid EV chains. By construction, EV hosts are a subset of the trusted hosts.

Among the TLS servers, table 5.4 describes the number of trusted and EV hosts, with regards to the NSS certificate store at the time of the campaign. There again, EFF campaigns seem to have a bias. If, as we suspected earlier, the SYN scan was not led in parallel of the TLS session establishment phase, a lot of dynamic IP addresses would have been missed; since they are seldom trusted hosts, it would lead to an over-representation of trusted and EV hosts in these campaigns. Other campaigns present consistent views for 2010/2011 on one hand, and 2014/2015 on the other hand.

Campaign		Trusted	EV
Date	**Source**	**hosts**	**hosts**
2010/07	Chapter 3	18 %	8 %
2010/08	EFF	28 %	13 %
2010/12	EFF	46 %	21 %
2011/07	Chapter 3	17 %	10 %
2014/03	Chapter 3	32 %	15 %
2015/08	`scans.io`	36 %	17 %
	(Top Alexa 1M)	38 %	28 %

Table 5.4: Proportion of trusted/EV hosts among TLS-answering servers.

If we look at non-EFF datasets, we observe that the number of TLS-speaking hosts on port 443 is clearly growing. In particular, between 2011 and 2014, the number of hosts with a 443/tcp open port has grown drastically (from 26 million to 40 million), and the proportion of TLS-speaking servers behind these ports has also increased (from 43 % to 72 %). We thus observe a significant increase in HTTPS support world-wide, which may in part be seen as a response to revelations concerning pervasive network monitoring [FT14]. This evolution is presented in figure 5.1.

5.3 Analysis of TLS parameters

In this section, we study protocol-related properties to assess the quality of TLS hosts in different campaigns: protocol version, ciphersuites and extension support.

5.3.1 Protocol version

We consider the version chosen by the servers in the different campaigns. Version negotiation in TLS is not very well defined but all implementations agree on the following propositions:

- a server can only answer with an SSLv2 `ServerHello` when the `ClientHello` used the SSLv2 format;
- a server can not choose a version greater than the one received inside the `ClientHello`.

The minimum version supported by the client is however not defined; in particular, the version used in the Record protocol by the client has essentially no meaning. A good server should choose the maximum protocol version available. As explained in section 1.2, we also consider that a good server should ban SSLv2. Since 2014 and the POODLE attack (see section 2.1.6), it has also become clear that SSLv3 should be phased out, and a server should only accept this obsolete version when TLS is not proposed.

In this section, we only consider standard TLS 1.0 stimuli for 2010, 2011 and 2014 campaigns, and TLS 1.2 stimuli for 2014 and 2015 campaigns. Indeed, for the 2010 campaigns, the maximum version advertised was TLS 1.0. Moreover, we could not properly test TLS 1.2 support in 2011, since the TLS1.2-modern stimulus also proposed a restricted set of ciphersuites: we had two parameters varying at the same time in our experiment. This led to a massive rejection of our TLS 1.2 stimulus, but we can not reliably establish the reason thereof.

Figure 5.2 presents the evolution of the protocol version chosen by the contacted servers. Table B.3 in the appendix shows the distribution of the versions chosen by servers for each campaign and each subset.

For standard `ClientHello` messages limited to the TLS 1.0 version, we observe the same results over time: TLS 1.0 is the preferred version of the protocol. Around 96 % of the TLS hosts answer with TLS 1.0 and 4 % use SSLv3 between 2010 and 2014. If we consider the same stimuli but focus on trusted or EV hosts, the proportion becomes more than 99 % for TLS 1.0 and 1 % for SSLv3. Using the data in appendix (table B.5), it is worth noting that the results do not change significantly when extensions are sent, or when an SSLv2 compatibility `ClientHello` is used.

Even if we have no hard figures in our datasets, TLS 1.2 was hardly deployed in 2010 and 2011. In 2014, the proportion of TLS servers accepting TLS 1.2 is 30 %, and it also improved significantly during the next year. TLS 1.2 has become the preferred version of the protocol for trusted hosts in 2014 and for EV hosts in 2015. For TLS hosts, the trend is similar, with TLS 1.0 and TLS 1.2 almost tied at around 50 % in 2015.

Another interesting fact is that very few servers answered with TLS 1.1 to TLS 1.2 stimuli, which seems to indicate that most of TLS 1.1-compatible servers also support TLS 1.2.

Finally, even if TLS is largely preferred over SSLv3, there are still many TLS servers that choose the obsolete version of the protocol. The situation is better with trusted and EV hosts.

Beyond standard TLS 1.0 and TLS 1.2 stimuli, the other results are more difficult to interpret. Indeed, as table 5.3 shows, the results obtained for several stimuli led to significantly fewer TLS answers. It shows that some servers refuse to negotiate certain protocol versions or ciphersuites. Among these servers, some will emit an alert, in compliance with the standards. Others will not answer or return inconsistent TLS messages: we call this intolerance to particular versions/suites. Section 5.5 discusses this matter further.

5.3.2 Ciphersuites

Currently, the ciphersuite registry contains more than 300 values. Among those suites, a typical client offers between 10 and 30 suites. In their answer, servers will either select one of them to build the `ServerHello` message, or send an alert

To study this parameter, we first discard a lot of suites that contain seldom used algorithms: fixed or unauthenticated Diffie-Hellman, PSK (Pre-Shared Key), SRP (Secure Remote Password), Kerberos, and IDEA. In particular, such suites were never proposed in the studied campaigns.

Next, we classify the ciphersuites in different groups:

- **Completely broken.** SSLv2 and export ciphersuites, and suites using DES or NULL encryption; these suites were only proposed with SSLv2-compatible stimuli;
- **Weak.** Suites using MD5, RC4 or DSS (see sections 1.5.1 and 1.5.3 for the corresponding explanations);
- **Strong.** Suites using ECDHE as the key exchange algorithm and an AEAD algorithm to protect application data. Such suites can only be negotiated with TLS 1.2;
- **OK.** The remaining suites.

We thus obtain 22 broken suites, 22 weak suites, 66 OK suites and 16 strong suites. The exact list of ciphersuites is given in the appendices, in section B.2. Naturally, we would like the servers to select only strong suites.

We prefer looking at the categories, instead of looking at the exact ciphersuites the servers chose, since the stimuli sent across different campaigns were not identical. It would thus make results difficult

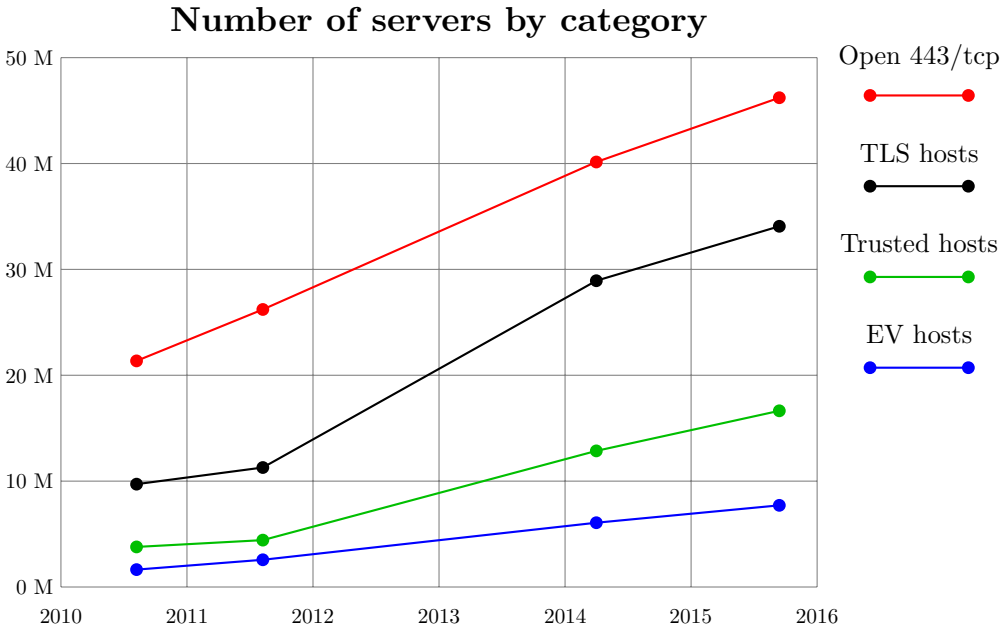


Figure 5.1: Evolution of the proportion of TLS answers between 2010 and 2015.

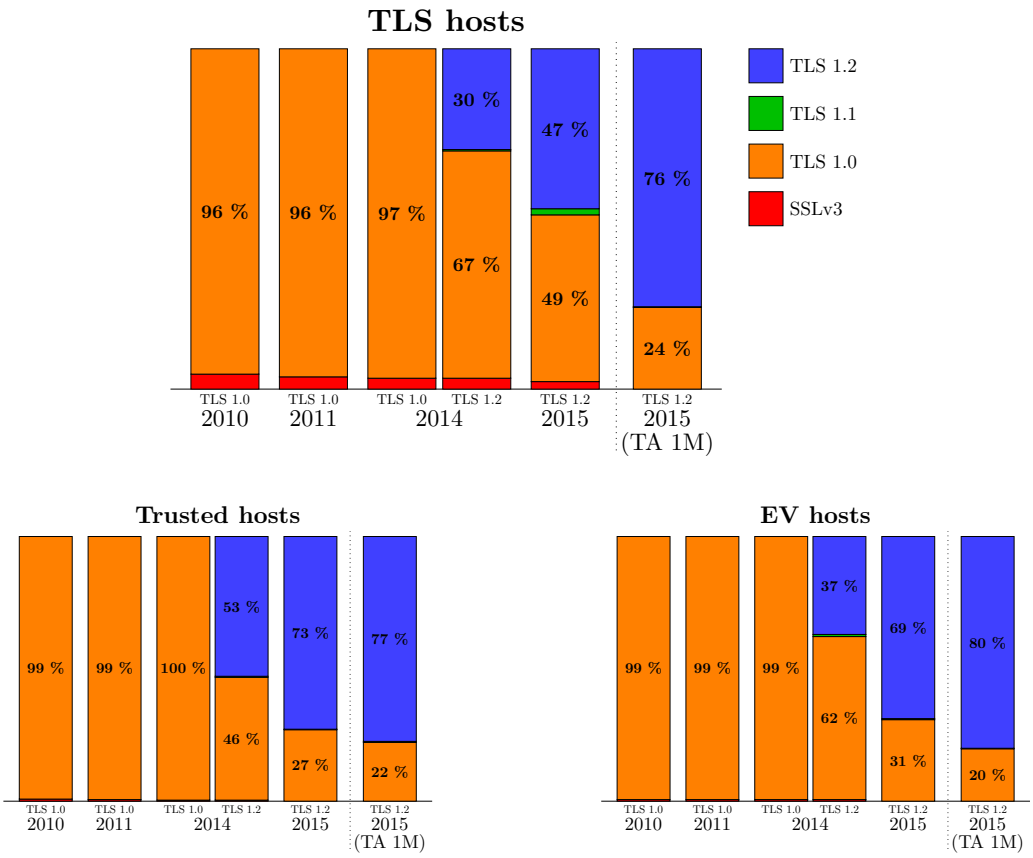


Figure 5.2: Evolution of the protocol versions chosen by servers between 2010 and 2015. For each subset (TLS, Trusted, EV), the five first columns correspond to full-IPV4 campaigns, while the last column is the result on the 2015 Top Alexa 1 Million campaign.

to compare. Figure 5.3 present how the repartition between categories evolved between 2010 and 2015, by using the same standard stimuli as in the previous section. The proportion of broken or unsolicited ciphersuites is not represented since they are negligible in the considered campaigns.

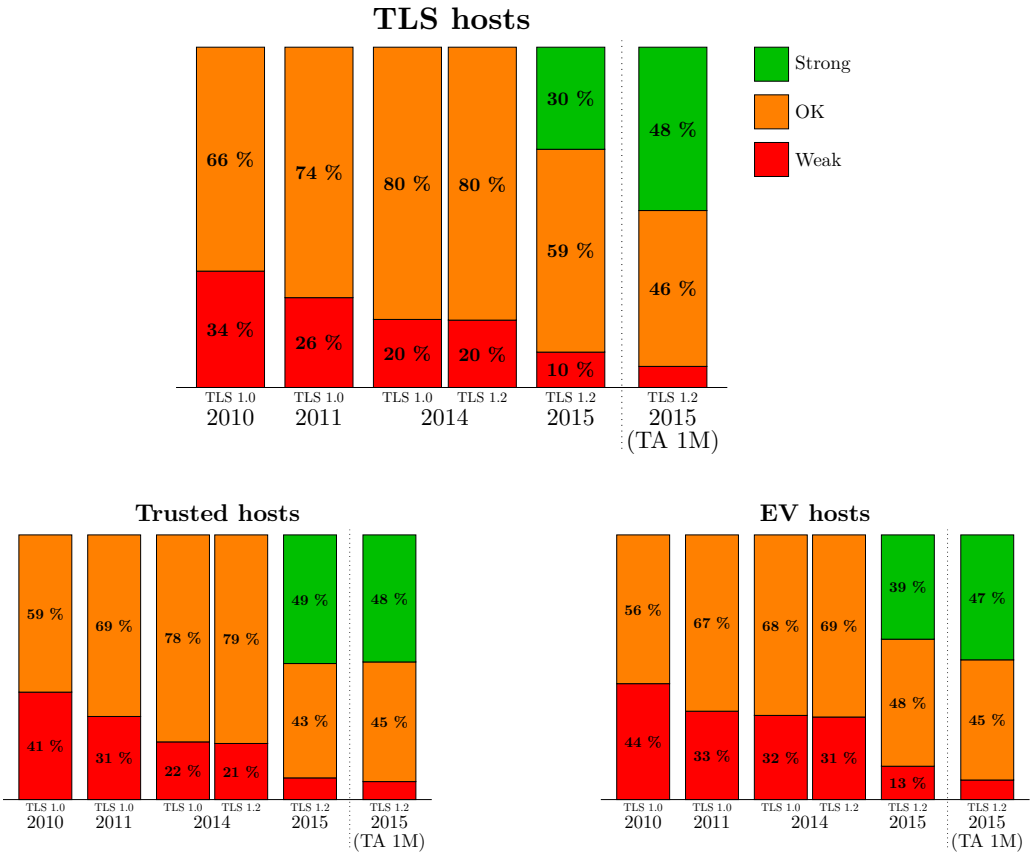


Figure 5.3: Evolution of the ciphersuites chosen by servers (by kind) between 2010 and 2015. For each subset (TLS, Trusted, EV), the five first columns correspond to full-IPV4 campaigns, while the last column is the result on the 2015 Top Alexa 1 Million campaign. The proportion of broken or unsolicited ciphersuites is negligible in the considered campaigns.

In 2010, one third of the answers contained a weak ciphersuite. For trusted or EV subsets, this proportion was even greater (more than 40 %). Between 2010 and 2014, the situation improved and the proportion of acceptable ciphersuites reached 80 % for TLS and trusted hosts (the proportion only reached 70 % for EV hosts).

In 2014, even if some servers chose TLS 1.2, there are almost no strong ciphersuites in the statistics. The situation eventually improved in 2015 with almost one third of the TLS hosts choosing a strong suite; the results are even better for trusted and EV hosts.

The seemingly bad results are due to the fact that our standard stimuli proposed the very popular `TLS_RSA_WITH_RC4_128_SHA` ciphersuite. This suite was indeed often preferred by high-traffic sites, since RC4 has been the most efficient ciphersuite for a long time[3]. As EV hosts usually correspond to high-traffic sites, it is logical to find more performance-oriented ciphersuites chosen by these servers.

Since 2013, RC4 has been shown to be vulnerable to practical attacks, and it has been progressively deprecated by major actors. In the mean time, many servers started preferring ECDHE ciphersuites for performance reasons. Both theses trends are presented on figure 5.4.

While analysing the results, we did not expect servers to answer with ciphersuites *not* proposed in `ClientHello`, as it is not compliant with the specifications. This phenomenon is significant in the

[3]With dedicated hardware, AES-GCM is nowadays more efficient. New ciphersuites combining Chacha20 and Poly1035 should also be faster than RC4/HMAC-SHA1.

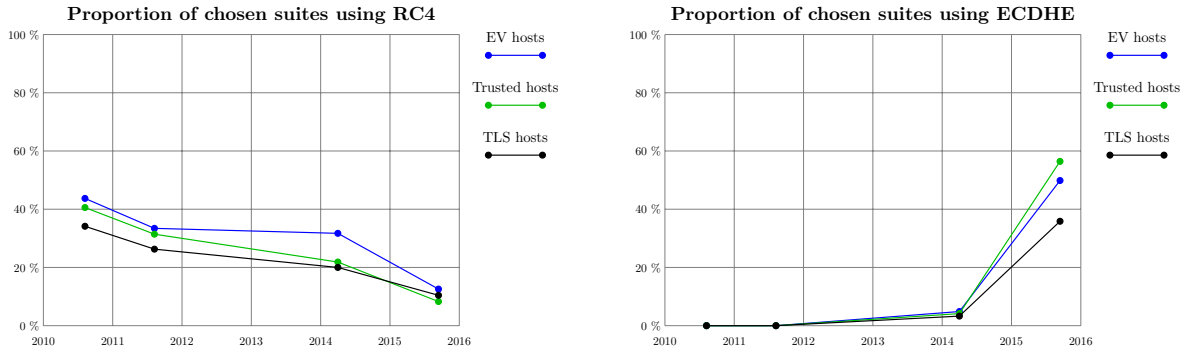


Figure 5.4: Server preferences concerning RC4 and ECDHE between 2010 and 2015.

DHE and EC campaigns, when the servers faced a limited choice. We also witness this behaviour with TLS12-modern stimulus, essentially because we chose not to propose the popular `RC4_MD5` ciphersuite.

This can be seen as a manifestation of server intolerance to DHE/EC/TLS1.2. This problem has been discussed on the TLS Working Group mailing list, and the conclusion was that several implementations are broken in a very strange way: they only consider the least significant byte of the 2-byte value representing the ciphersuite. Until RFC 4492 [BWBG+06], defining the first elliptic curves ciphersuites, was published, all the registered ciphersuites indeed included a null most significant byte. Those implementations could thus answer a stimulus proposing `TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA` (`0xc005`) with `TLS_RSA_WITH_RC4_128_SHA` (`0x0005`). In both cases (server intolerance or stripped ciphersuites), the answers correspond to broken implementations.

Another way to analyse the ciphersuites chosen by servers is to compute the proportion of them that provide forward secrecy (FS), to prevent past sessions to be decrypted if the server private key is compromised. As explained in section 1.1, with TLS ciphersuites, this property is obtained with ephemeral Diffie-Hellman key exchanges, as opposed to the RSA encryption key exchange. Results are presented in figure 5.5.

Figure 5.5: Proportion of ciphersuites chosen by servers enforcing forward secrecy between 2010 and 2015. All the corresponding stimuli proposed ECDHE and DHE ciphersuites.

For all the sets of hosts, the evolution is the same: between 2010 and 2014, the proportion of servers choosing forward secrecy went down from 40-50 % to 20-30 %, but in 2015, the situation improved significantly. As for the proportion of strong ciphersuites, it can be explained by the massive adoption of ECDHE by TLS hosts.

5.3.3 Secure renegotiation

From a security standpoint, when a client proposes the secure renegotiation extension specified in RFC 5746, the server should support it. This extension was proposed in several 2011 stimuli, in several 2014 stimuli, and in the ZGrab stimulus, either as a proper TLS extension, or via the dedicated signaling ciphersuite (`RENEGOTIATION_SCSV`).

Figure 5.6 shows the evolution of the proportion of servers answering with the secure renegotiation extension to a stimulus advertising the security feature. The trend is positive, and trusted/EV hosts seem to support significantly more the extension than the general TLS servers population. However, it is not satisfying at all to get less than 90 % for the trusted hosts in August 2015, more than five years after the publication of RFC.

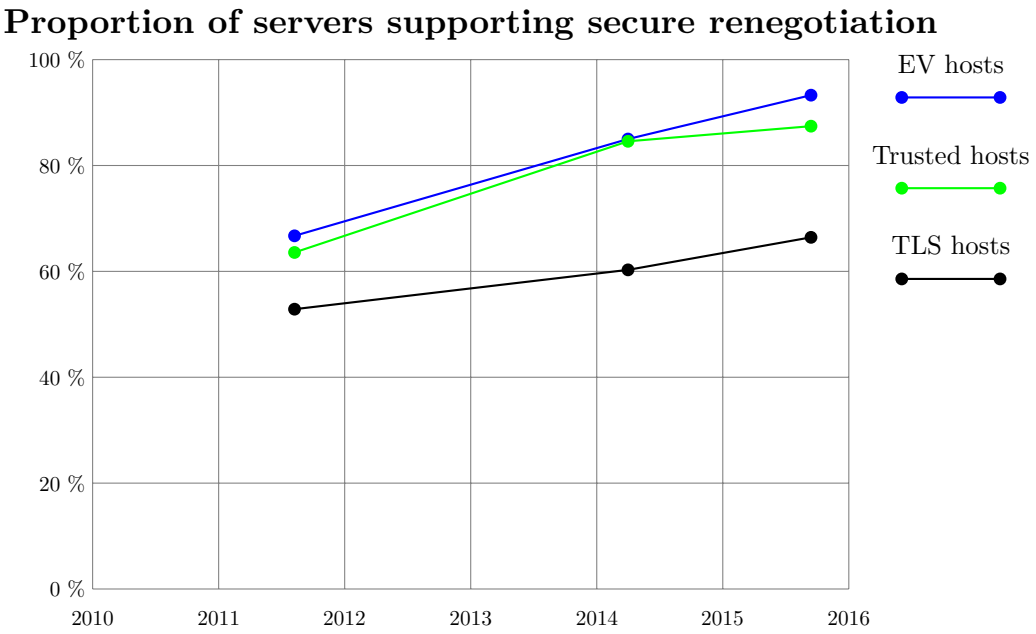


Figure 5.6: Proportion of servers supporting secure renegotiation between 2010 and 2015.

It is worth noting that servers are only vulnerable if they do not support the extension *and* if they accept to renegotiate. As we did not pursue the connection, nor tried to renegotiate, we can not reliably tell how many servers were indeed vulnerable. Yet, from the client's perspective, the only way to be sure that the server is not vulnerable is if it supports the secure renegotiation extension, since triggering a renegotiation to test the server could lead to the connection being shut down with a fatal alert.

5.4 Analysis of certificate chain quality

As for the TLS parameters, we only consider standard TLS 1.0 stimuli for 2010, 2011 and 2014 campaigns, and TLS 1.2 stimuli for 2015 campaigns; TLS 1.2 results were the same as TLS 1.0 ones for the 2014 campaigns, with regards to the criteria related to certificates. For each subset and each campaign, we compute several statistics on the best certificate chains — according to `rateChains` — we could build using `concerto`.

Generally, servers send 2 or 3 certificates and the certificate chains we build also contain the same number of certificates. However, several servers send more certificates. The longer `Certificate` message

we observed was 150 certificate long, with only one duplicate certificate; it was found in the second EFF dataset and corresponded to a trusted server.

5.4.1 Certificate chain order

As presented in chapter 4, the `Certificate` message contains a list of certificates used to establish the identity of the server. This list should be strictly ordered, but this is not always the case. Using the chains built by `concerto`, we classify the certificate chains in four groups:

- *RFC-compliant* chains do contain only the useful certificates in the correct order;
- *unordered* (or self-contained) chains contain useless or unordered certificates, but include all the information needed to build a complete chain;
- *transvalid* chains lack *intermediate* authority to build a complete chain. To reach the root certificate, we thus need to use certificates from other chains;
- *incomplete* chains could not be built up to a self-signed certificate, even by using external certificates.

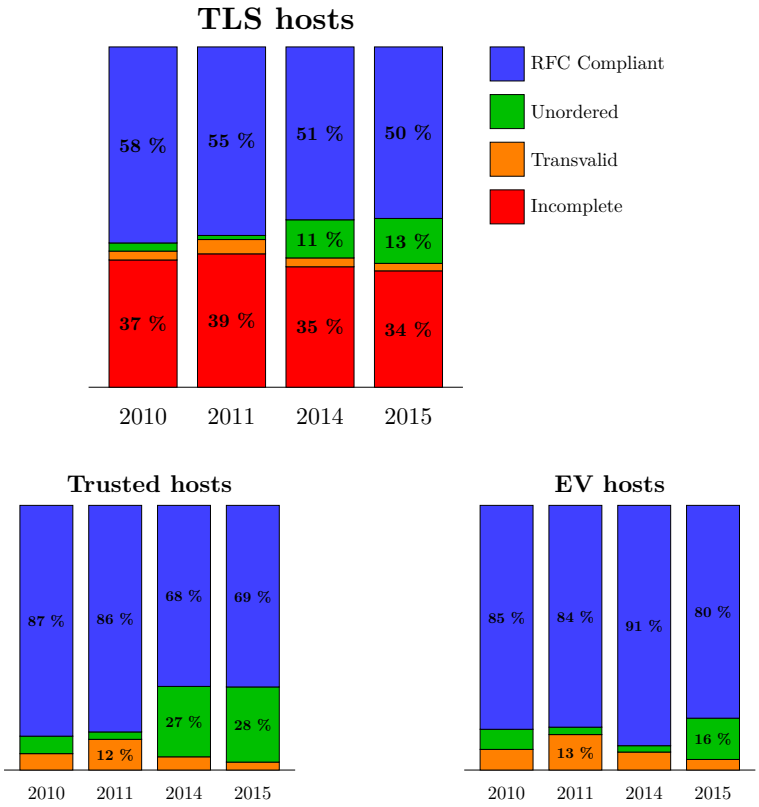


Figure 5.7: Evolution of the quality of certificate chains between 2010 and 2015.

The partition of certificate chains built along these categories are given in figure 5.7. Leaving aside the incomplete certificate chains, the trends for TLS and trusted hosts seem to be the same: there is a majority of RFC-compliant servers, but the proportion of unordered or transvalid is far from negligible. Actually, the proportion of unordered chains has grown significantly in 2014 for trusted hosts. This may be explained by the need to present different certificate chain in one message, as described in section 4.1.2.

EV hosts data is mostly the same for 2010 and 2011, but the proportions of unordered chains dropped in 2014, before a rise in 2015. However, the proportion of transvalid chains is always slightly more important for EV hosts than for trusted hosts.

Another way to look at this data is to merge the RFC-compliant and the unordered chains, since they are already accepted by most TLS stacks, and that the order constraint should be relaxed in TLS 1.3. With this modification, trusted and EV essentially behave similarly, with slightly better results for trusted hosts. In both cases, we would expect the servers to present at least self-contained chains. This can lead to situations where one can only visit a site after having browsed another site embedding the missing certificates.

5.4.2 Chain robustness

For all the full IPv4 campaigns we considered, the best certificate chains we computed overwhelmingly use RSA for each certificate. The only campaign where ECDSA certificates are non negligible is the recent Top Alexa 1 Million dataset, which is not represented in this section.

Thus, we focused on the RSA-only certificate chains for this section, and we considered the following criterion: the minimum length of all the moduli in the certificate chain. We call this the chain robustness. Figure 5.8 represent the repartition of said robustness for the studied campaigns.

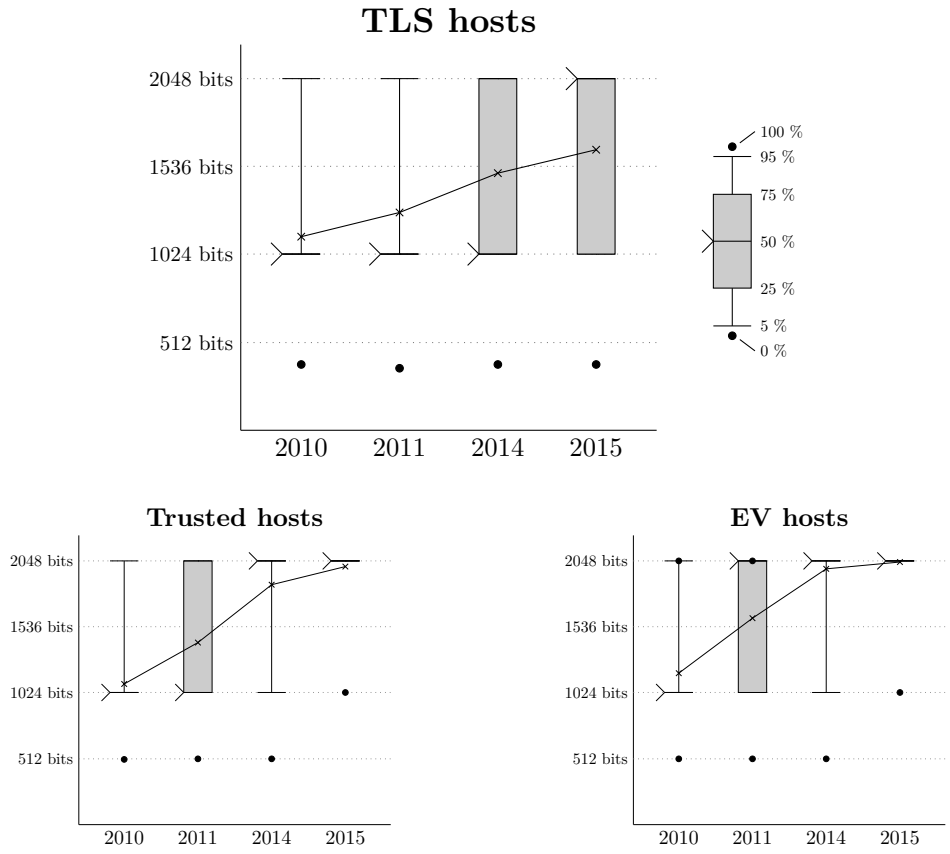


Figure 5.8: Evolution of the RSA key robustness (in bits) in certificate chains between 2010 and 2015. The box plot shows different percentiles for the value repartition, and the line represents the mean value for each campaign. In some cases, the maximum value is off-chart and is not represented.

For each subset, we observe the same trend: the median robustness went from 1024 bit to 2048. For trusted and EV hosts, it is also true for the first quartile. This means that 2048 bit has become the new standard, whereas it was 1024 bit in the past. The change was a little quicker for trusted and EV hosts, probably because EV certificates were required to be at least 2048 bit long since December 2010.

For all the considered campaigns, the maximum key robustness was 16384 bit for TLS hosts and 4096 bit for trusted hosts. For EV hosts, the longer robustness observed was 2048 in 2010 and 2011 (which appears on the figure), and 4096 in 2014 and 2015.

It is however worrying that the minimum key robustness was 512 until 2014 for trusted and EV hosts (it was 384 until 2014 for TLS hosts). Even if the situation improved in 2015, where the minimum was 1024 bit, such weak RSA keys should never be used today.

5.4.3 Validity periods

Finally, the last parameter we studied was the validity period of the chain (i.e. the intersection of the validity periods of the certificates in the chain). The repartition of this parameter across the different campaigns is given in figure 5.9.

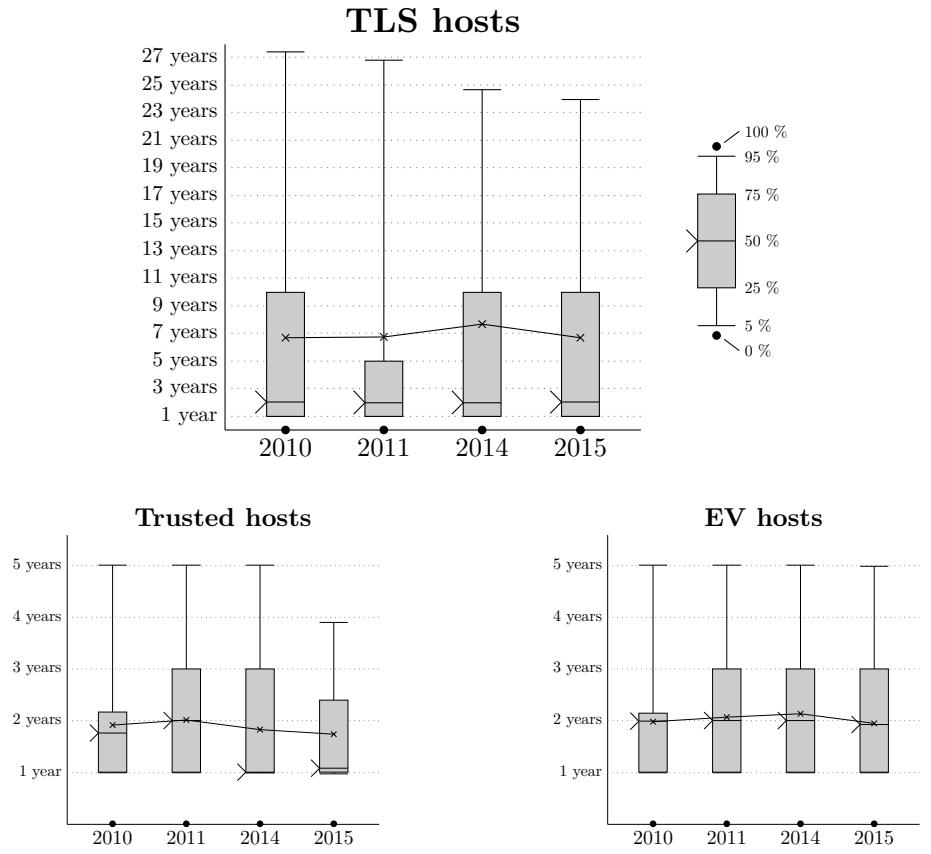


Figure 5.9: Evolution of the chain validity period (in years) between 2010 and 2015. The box plot shows different percentiles for the value repartition, and the line represents the mean value for each campaign.

Even if this parameter corresponds to the validity period of the *chain*, we expect the certification authorities to emit their certificates in a consistent way: a one-year certificate should not be signed by an authority whose certificate is due to expire next week. We however observed four EV hosts with a chain that was valid less than a week (which explains the minimum values in the figures).

As expected, trusted and EV validity periods are sensible: mostly between one and two years in 2011, between one and three years since 2011. Considering the maximum values, they are very surprising for the overall TLS hosts: 25 % of the chains were valid for at least 10 years, several chains were valid for more than 1000 years. The situation is cleaner for trusted and EV hosts, since 95 % of the hosts present chains valid at most 5 years. Yet, we still observe chains signed by an EV root, that were valid more than 20 years.

Overall, the chain validity period and the key robustness are parameters that have improved over time and that mostly correspond to sensible values for trusted and EV hosts: 1- to 3-year chains with a 2048-bit RSA robustness in 2015. This is fortunate, as trusted authorities, and EV authorities in particular, are expected to enforce sensible policies regarding these parameters.

5.5 Analysis of server behaviour

In July 2011 and in March 2014, as we sent servers multiple `ClientHello`s, we gained an interesting insight into the behaviour of servers: their reaction to restricted ciphersuite choices, their intolerance to versions and extensions. We could indeed correlate answers to standard or specific stimuli.

For TLS `ClientHello`s (that is, excluding SSLv2-compatible `ClientHello` messages), we would expect the sum of hosts answering with alerts and handshake messages to be essentially the same across stimuli. Yet, we observed a well known fact in the community: under some circumstances, such as feature-oriented stimuli or TLS 1.2 `ClientHello`s, several servers seem to panic and either abort the connection abruptly, or answer with an incompatible `ServerHello` (e.g. an answer containing a ciphersuite that was not proposed). This behaviour can partly be attributed to known TLS stacks (e.g. Microsoft SChannel) that shut down the TCP connection when they face unacceptable propositions, but all these intolerant servers do nevertheless not conform to the TLS specification. Sadly, studying these intolerance situations is a necessary step to assess the deployability of a new version, ciphersuite or extension.

In this section, we explore these intolerances by comparing the answers obtained for different stimuli for each IP in the considered subset (TLS hosts, trusted hosts and EV hosts). Our baseline is a standard TLS 1.0 stimulus. For each IP answering with valid Handshake messages to this reference stimulus, we look at the answer from the same IP to a different stimulus.

As a side note, the methodology is slightly different from the one used in 2012, where we considered the baseline to be the set of all servers answering with a Handshake message for at least one stimulus for the TLS subset, and the set of all servers answering with a trusted chain (respectively an EV chain) for the trusted subset (resp. the EV subset). We believe that the methodology described in this section is simpler and that the corresponding results are easier to interpret.

As shown in section 3.4.2, around 0.4 % of the contacted IP addresses seem not to be stable during the campaign in 2011 (and 0.55 % in 2014), so all statistics smaller than these margins should be discarded.

5.5.1 Intolerance to the DHE / EC / TLS12-modern stimulus

Tables 5.5, 5.6, 5.7 present the answers of this sample group, which answered with TLS Handshake messages to a TLS 1.0 stimulus, to the DHE, EC and TLS12-modern stimuli. The Alert line shows the proportion of servers that refuse to negotiate and assert this choice properly. This should happen if the proposed parameters are unacceptable from the server's point of view. Theoretically, this should not happen with a protocol version, since the servers of the sample groups accepted TLS 1.0 and could have answered with this version. Yet, our TLS12-modern stimulus did not contain all the suites proposed in the reference `ClientHello`s, so it is legitimate for a server to accept the TLS 1.0 stimulus but send an Alert to the TLS12-modern message.

The last lines of each table represent servers that did not respond correctly to the stimulus. As they correctly answered a TLS 1.0 stimulus, we would expect an Alert message to signal the negotiation failure, instead of a non-TLS answer or an incompatible `ServerHello` message. We call such servers DHE-, EC- or TLS 1.2-intolerant, and their behaviour does not conform to the standards.

	2011			2014		
	TLS	**Trusted**	**EV**	**TLS**	**Trusted**	**EV**
Compatible Handshake	38.9 %	43.4 %	44.7 %	30.4 %	24.2 %	33.3 %
Alert	37.9 %	26.5 %	20.9 %	55.6 %	59.3 %	40.6 %
Intolerant servers	**23.1 %**	**30.1 %**	**34.3 %**	**13.9 %**	**16.6 %**	**26.1 %**
Non-TLS answer	22.2 %	30.0 %	34.3 %	13.2 %	16.6 %	26.0 %
Incompatible Handshake	1.0 %	0.0 %	0.0 %	0.7 %	0.0 %	0.0 %

Table 5.5: Answers to the DHE stimulus in 2011 and 2014.

For each case (DHE, EC, TLS12-modern), the proportion of intolerant servers was very important in 2011: about 20 % globally, and more than 30 % for trusted and EV servers facing the DHE or TLS12-modern stimuli. In 2014, the situation has improved drastically for TLS 1.2 and significantly

	2011			2014		
	TLS	**Trusted**	**EV**	**TLS**	**Trusted**	**EV**
Compatible Handshake	6.2 %	9.2 %	11.6 %	10.8 %	15.4 %	23.0 %
Alert	75.9 %	68.0 %	63.8 %	82.2 %	78.2 %	67.3 %
Intolerant servers	**17.9 %**	**22.8 %**	**24.6 %**	**7.1 %**	**6.4 %**	**9.7 %**
Non-TLS answer	16.9 %	22.7 %	24.5 %	6.2 %	6.4 %	9.6 %
Incompatible Handshake	1.0 %	0.1 %	0.1 %	0.9 %	0.0 %	0.0 %

Table 5.6: Answers to the EC stimulus in 2011 and 2014.

	2011			2014		
	TLS	**Trusted**	**EV**	**TLS**	**Trusted**	**EV**
Compatible Handshake	38.9 %	43.4 %	44.7 %	86.3 %	90.2 %	85.6 %
Alert	37.9 %	26.5 %	20.9 %	7.3 %	4.0 %	5.6 %
Intolerant servers	**23.1 %**	**30.1 %**	**34.3 %**	**6.4 %**	**5.8 %**	**8.8 %**
Non-TLS answer	22.2 %	30.0 %	34.3 %	5.7 %	5.8 %	8.8 %
Incompatible Handshake	1.0 %	0.0 %	0.0 %	0.7 %	0.0 %	0.0 %

Table 5.7: Answers to the TLS12-modern stimulus in 2011 and 2014.

for the EC stimulus. Yet, the proportion of intolerant servers is still high (between 6 and 10 % for EC and TLS12-modern stimuli, and between 14 and 26 % for the DHE stimulus).

Since we also sent a TLS 1.2 stimulus resembling the TLS 1.0 baseline `ClientHello` (where only the version has been changed) in 2014, we can study the TLS 1.2 intolerance further for this year. As can be seen in table 5.8, almost all the servers seemed to accept TLS 1.2 (the proportion of intolerant servers is around 0.7 %, which is very close to the precision margin computed in chapter 3). This is good news for the TLS 1.2 deployment, as long as TLS 1.2-compatible clients continue to propose legacy ciphersuites.

	TLS	**Trusted**	**EV**
Compatible Handshake	99.2 %	99.4 %	99.2 %
Alert	0.1 %	0.1 %	0.1 %
Intolerant servers	**0.7 %**	**0.5 %**	**0.7 %**
Non-TLS answer	0.7 %	0.5 %	0.7 %
Incompatible Handshake	0.0 %	0.0 %	0.0 %

Table 5.8: Answers to the TLS12 stimulus in 2014.

5.5.2 Extension intolerance

We used the same methodology to study the intolerance to extensions. To this aim, we compared the results for a standard TLS 1.0 stimulus with different TLS extensions (TLS1.0), using the same stimulus without any extensions (TLS1.0-NoExt) as our baseline.

The results, presented in table 5.9, show a similar result in 2011 and in 2014: the proportion of servers rejecting a `ClientHello` only because it contains extensions is small (and even smaller than the precision margin most of the time). This does not mean that extension intolerance does not exist, but that it may happen in very subtle ways. For example, extensions make the `ClientHello` longer, which might trigger a confusion for SSLv2-compatible appliances (see section 3.2.3); another example was briefly discussed on the TLS working group mailing list in March 2016: several servers shut down a connection when the last extension of the `ClientHello` is empty.

	2011			2014		
	TLS	**Trusted**	**EV**	**TLS**	**Trusted**	**EV**
Compatible Handshake	99.3 %	99.5 %	99.5 %	99.0 %	99.6 %	99.5 %
Alert	0.3 %	0.3 %	0.3 %	0.2 %	0.1 %	0.1 %
Intolerant servers	**0.3 %**	**0.2 %**	**0.2 %**	**0.8 %**	**0.3 %**	**0.4 %**
Non-TLS answer	0.3 %	0.2 %	0.2 %	0.8 %	0.3 %	0.4 %
Incompatible Handshake	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

Table 5.9: Answers to a stimulus containing extensions, using a stimulus with no extension as the baseline stimulus, in 2011 and 2014.

5.5.3 Reaction to the SSL2 stimulus

Finally, let's consider the proportion of the sample groups answering correctly to an SSLv2 `ServerHello` when the stimulus is a pure SSLv2 `ClientHello` (SSL2 stimulus). Table 5.10 shows the answers received in 2011 and 2014. We did not expect TLS servers to behave correctly, since SSLv2 uses different messages and has now long been deprecated. In fact, we would have expected fewer servers to accept negotiating a SSLv2 session.

	2011			2014		
	TLS	**Trusted**	**EV**	**TLS**	**Trusted**	**EV**
Compatible Handshake	**48.2 %**	**42.1 %**	**32.3 %**	**40.4 %**	**34.1 %**	**40.8 %**
Alert	3.4 %	3.5 %	3.7 %	4.6 %	4.0 %	6.4 %
Non-TLS answer	48.4 %	54.4 %	64.0 %	55.0 %	61.9 %	52.9 %
Incompatible Handshake	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

Table 5.10: Answers to the SSL2 stimulus in 2011 and 2014.

Many TLS servers are still fully compatible with SSLv2, whereas they should not negotiate the obsolete version of the protocol. The situation is even more worrying if we consider the recent DROWN attack, which uses an SSLv2 server as an oracle to decrypt TLS `ClientKeyExchange` messages sent to a server using the same RSA private key (the underlying issue is described in section 7.3.2). In practice, in 2011, 99.3 % of the servers answering the SSLv2 only stimulus reused the same server certificate in the TLS handshake. These servers represent 48 % of the TLS hosts.

The proportion of servers vulnerable to DROWN is actually more important. After flagging all certificates presented by SSLv2 servers, we count the number of TLS servers reusing such a certificate (on the same IP address or not). We thus get 55 % of vulnerable servers (57 % among trusted servers) in 2011 and 46 % of vulnerable servers (40 % among trusted servers) in 2014.

It is however important to notice that these figures, however alarming, are still an understatement for two reasons: first, we did not look at shared RSA *public keys* across different certificates (which should have a minor impact); last but not least, we did not take into account other secure protocols such as SMTPS, where RSA certificates are often reused, and where SSLv2 is still very common.

5.6 Concluding thoughts on the HTTPS ecosystem and its analysis

We found that several parameters, like RSA key robustness or chain validity periods, are already close to the desired situation (at least for trusted hosts). Other parameters are improving, e.g. the support of TLS 1.2 or the strength of the chosen ciphersuites. However, there is still an important margin of progression: a lot of servers are still intolerant to elliptic curves ciphersuites and others still support SSLv2.

In 2012, we observed that EV hosts did not behave significantly better than trusted or TLS hosts, which is normal since Extended Validation only deals with the format of the certificate issued to servers. To take into account other parameters, we would need another form of quality label representing the

overall quality of TLS sessions (server configuration, implementation and cryptographic parameters). In a way, some effort from the community, such as the Qualys SSL Server Test have been made to fill the gap.

Our study also proposes an overall methodology to assess the overall quality of SSL/TLS servers: how to collect data and how to anlayse them. First, data must be collected in an homogeneous way, using documented parameters (the used stimuli) and recording metadata (e.g. the timestamp for each answer). Then, the analyses must be precisely described, including all the relevant parameters (the certificate stores, the `max-transvalid` parameter). This allows others to reproduce the experiment.

However, looking back at our previous work, this might not be sufficient. We indeed had to change several scales or algorithms, to take into account the evolution of the state of the art. It was thus important for us to be able to replay new tools with different analyses on the raw data. Since 2012, here are examples of the differences we implemented in our tools:

- X.509v1 are not considered certification authorities anymore, unless they are trusted roots;
- we systematically used the NSS trust store present in the source code at the time of the campaigns;
- we changed the categories for the ciphersuites;
- we simplified the methodology to analyse server behaviour.

All these changes were possible since we had access to the raw data. Moreover, the incremental way `concerto` works means that such changes do not always require to run the complete set of tools. For example, the last two changes only required to re-compute the statistics on the already processed data.

Finally, another lesson learned from these analyses is that using multiple stimuli to test the server behaviour is useful, and could easily be extended to regularly analyse smaller server subsets such as the Top Alexa 1 Million.

Part III

Implementation aspects and focus on the parsing problem

In this third part, we study implementation aspects of TLS stacks. Chapter 6 first addresses the parsing problem. To run the analyses described in the previous part, we indeed needed reliable parsers. After presenting the different options available, this chapter describes `parsifal`, our solution. Chapter 7 proposes an in-depth anaylsis of TLS implementation flaws, and proposes some thoughts to improve the situation.

Chapter 6

`parsifal`: a pragmatic solution to the binary parsing problem

To analyse the collected data from our campaigns, we decided to write our own tools to parse and interpret TLS records and X.509 certificates. After several attempts, `parsifal` *was written and proved useful for numerous use cases, even beyond its initial goal.* `parsifal` *was presented several times during the last years [LDM13, Lev13, Lev14b, Lev14c]. The source code was publicly released in 2013 on GitHub*[1]*.*

Parsers are pervasive software basic blocks: as soon as a program needs to communicate with another program or to read a file, a parser is involved. However, writing robust parsers can be difficult, as is revealed by the amount of bugs and vulnerabilities related to programming errors in parsers. It is especially true for network analysis tools, which have to deal with a wide variety of inputs, including malformed and malicious ones. Even though CVE counting is a biased criterion, it is worth noting, as an example, that the Wireshark project, which contains a wide variety of network protocol parsers, has been affected by 82 CVE in 2013, 29 in 2014 and 33 in 2015.

In our work, we distinguish binary formats, which rely on structured low-level constructions, from textual formats, such as HTTP, SMTP or XML-based formats. Binary protocols and formats should naturally be simpler to parse, but they are less standardised than textual formats which can benefit from well-known grammars (e.g. using the Backus-Naur Form) and tools (such as `lex` and `yacc`)[2].

Studying network protocols requires parsers (sometimes also called dissectors) to analyse the exchanged binary messages. Experience has taught us that we needed trusted, robust tools to really understand how a protocol works, particularly to detect and characterise anomalies. Indeed, available parsers are not usually suited for this need. This led us to write custom tools. Our goal was to *quickly* develop *robust* and *efficient* parsers. To this aim, we tried to use several programming languages (mostly Python, C++ and OCaml). The latest tool was `parsifal`, a generic framework to describe parsers in OCaml, which gave us some insight into binary formats and parsers.

In the end, our contributions to software security with `parsifal` are the following:

- it offers a simple way to describe complex formats concisely;
- it produces robust and rather efficient tools, allowing for analysis of large amounts of data, e.g. with SSL/TLS data or BGP routing information;
- writing parsers taught us which data structures are preferable and which parsing features are complex to manage.

`parsifal` is a building block of the `concerto` framework presented in chapter 4, since it provides the parsing primitives for TLS messages and X.509 certificates.

Section 6.1 presents the context for our parsers: initially aimed at SSL/TLS messages and X.509 certificates, they were extended later to other network protocols and file formats. Then, section 6.2

[1] `https://github.com/ANSSI-FR/parsifal`

[2] Even though methods and tools exist for text-based formats, there are still very complex and ambiguous specifications, such as PDF, which we studied as a side project [ELM16].

describes several ways to parse binary messages, illustrating them with TLS alert messages. Our implementation, `parsifal`, is succinctly presented in section 6.3, with examples of several features applied to TLS. Section 6.4 contains examples from several use cases to show `parsifal` features and section 6.5 presents several performance feedbacks. Finally, section 6.6 concludes the chapter with several learned lessons. Appendix C contains tutorials explaining how to develop parsers for TAR archives and PNG images with `parsifal`, as well as the grammar handled by the preprocessor.

6.1 Binary parsers: motivation and presentation

The TLS data presented in chapter 3 is composed of an important set of answers from servers on the 443/tcp port worldwide. This significant amount of data contained legitimate TLS messages, as well as invalid messages or even messages related to other protocols (HTTP, SSH). To face this challenge and extract relevant information, we needed robust tools on which we could depend. Yet, existing TLS stacks did not fit our needs: they can be limited (valid options are rejected), liberal (they silently accept wrong parameters) or fragile (they crash on unexpected values, even licit ones). Thus, we decided to write our own parsers.

To parse binary protocol messages, such as TLS messages or X.509 certificates, the expected properties of our tools were:

- their ability to handle *complex structures*;
- their *robustness*;
- the *ease* to write new parsers;
- their *performance*.

Our first attempt to write a TLS parser was using the Python language; it was quickly written and allowed us to extract some information. However, this implementation was unacceptedly slow and led to unexpected crashes. The second parser was written in C++, using templates and object-oriented programming; its goal was to be flexible and fast. Yet, the code was hard to debug (memory leaks, segmentation faults on flawed inputs), and lacked extensibility since every evolution of the parsers required many lines of code.

So a new version has been written, in OCaml: it uses a DSL (Domain Specific Language) close to Python to describe the structures to be studied. This third parser is as fast as the previous one, less error-prone, but still requires a lot of lines to code simple features. That is why we finally decided to use a preprocessor to do most of the tedious work, letting the programmer deal only with what is important, description of the message structure, while avoiding usual mistakes in low-level memory management. This last implementation, `parsifal`, has all the properties we expect: efficient and robust parsers, written using few lines of code.

Our work originally covers X.509 certificates and TLS messages, but we soon experimented `parsifal` on another important dataset. Since 2012, ANSSI has been studying the structure of the Internet in France; this so-called "Observatoire de la résilience de l'internet français" [ANS13] analysed vast amounts of BGP announces collected by the Routing Information Service (RIS, a project developed by the RIPE NCC). As a challenge, we applied `parsifal` to this dataset (around 2 GB of daily data).

In both cases (TLS and BGP), the analysis leads to several issues. First the amount of data is significant. Then, the relevant information we would like to extract is contained within complex protocol messages. Finally, we are handling raw unfiltered data, whose quality (or even their harmlessness) is not guaranteed.

On a related note, `parsifal` was also used to parse other network protocols (DNS, TCP/IP stack, Kerberos, NTP) and several file formats (TAR, PE, PCAP, PNG). Some of these parsers are still an early stage, but one of the strength of `parsifal` is that it is easy to describe part of a protocol, and focus only on what really needs to be dissected.

Looking at so many different binary formats (network protocols, image formats, etc.), we learnt a lot about software requirement for binary parsers: the wide variety of integer representation[3], variable-size

[3]Big- or little-endian integer are usually well known, but ASN.1 DER encoding use three different representations, and TAR files rely on a character string representing the octal value!

fields, context-dependant fields, non-linear parsing (that is requiring to follow pointers within a message or a file).

6.2 Different methods to parse binary formats

The grammar describing TLS messages is complex. Without dwelling on the details (the exact message representation depends on the protocol version, messages can be split across several records), there exist variable-length fields and context-sensitive fields. Moreover, TLS uses X.509 certificates, which rely on DER encoding, a syntax different from the rest of the messages. Before looking at more complex structures, we restrict ourselves to a very simple subset of TLS messages: alerts.

To illustrate different possible implementations of TLS parsers, we apply them on TLS alerts. As explained previously, the Alert Protocol is a sub-protocol of TLS that is encapsulated inside a record. Tables 6.1 and 6.2 describe the corresponding binary structures.

Offset	Field	Size	Type
0	Content type	1	Enumeration (5 possible values) from 0x14 to 0x18
1	Protocol version	2	Two integers (major and minor versions)
3	Content length `sz`	2	Big-endian integer
5	Record content	`sz`	Content-type-dependent

Table 6.1: Description of a TLS record.

Offset	Field	Size	Type
0	Alert level	1	Enumeration (2 possible values)
1	Alert type	1	Enumeration (many possible values)

Table 6.2: Description of the record content when its content type is `0x15` (`Alert`).

6.2.1 Direct conversion of the binary stream using low-level structures (C casting)

For very simple formats, the easiest way to parse a format is to cast it directly into the expected structure. This is mostly a C approach, which could be done in the following way for a TLS Alert:

```
typedef struct __attribute__((packed)) {
  uint8_t alert_level;
  uint8_t alert_type;
} tls_alert;

typedef union __attribute__((packed)) {
  tls_alert alert;
  // Other message types...
} record_content;

typedef struct __attribute__((packed)) {
  uint8_t        content_type;
  uint16_t       tls_version;
  uint16_t       record_length;
  record_content record_content;
} tls_record;

tls_record r;
read (f, &r, 5);
read (f, &r.record_content, r.record_length);
```

This solution has the advantage of being very simple to write, and very efficient. However, it is completely intractable when dealing with complex formats such as TLS, since the parsing must be split at each length field, and every time a message depends on a discriminating value.

Moreover, we have no guarantee that the filled fields have a licit value (for example, only several values are acceptable for the `protocol_version` or `content_type` fields). Finally, in terms of robustness, many programming errors are possible, that could lead, among others, to buffer overflows.

This method is clearly not suited for the analysis of complex protocols such as TLS, so we did not really use it in our study.

6.2.2 Extended pattern-matching (OCaml `bitstring`)

A method extending the pattern-matching construction of OCaml was proposed in the `bitstring` syntax extension [Jon12]. The idea is to incorporate the possibility to analyse a binary payload directly in the patterns, as in the Erlang programming language. In a way, it resembles the previous case since the interpretation does not need any copy. Parsing TLS alerts with `bitstring` would look like the following snippet:

```
let parse_record_content content_type bits =
  bitmatch bits with
  | { alert_level : 8;
      alert_type  : 8 }
    when content_type = 0x15 -> Alert (alert_level, alert_type)
  | ...
  | { raw : -1 : string } -> Unparsed raw

let parse_record bits =
  bitmatch bits with
  | { content_type : 8;
      tls_version : 16 : bigendian;
      record_length : 16 : bigendian;
      record_content : record_length*8 : bitstring } ->
    content_type, tls_version, (parse_record_content content_type
   record_content)
  | { _ } -> failwith "Invalid␣Record"
```

This approach allows to extend the OCaml language to extract bit fields from a binary stream, but the offered expressiveness was limited. Even though several complex structures could theoretically be expressed (variable-length fields, discriminating values using `when` clauses), `bitstring` does not allow to easily add field constraints or to support non-linear formats (which is not the case of TLS however); in both cases, more OCaml code is needed to glue together `bitmatch` blocks, making the code difficult to follow. Finally, the `bitmatch` construction can be very inefficient in practice (especially when adding `when` clauses).

For our study, this method did not seem adapted to the analysed formats, which are full of ad-hoc constraints. This led us to consider other approaches and not to implement this one.

6.2.3 Manually write types and functions

If we want to preserve expressiveness in the formats supported by our tools, a solution is to manually write all the types and the parsing functions. To this aim, we would need library functions to eliminate repetitive tasks (e.g. read an unsigned integer on 16 bits, read a character string, or read and validate an enumerated value). Here would be the relevant code needed to define the TLS alert type and the corresponding parsing function in OCaml:

```
type tls_content_type =
  | CT_ChangeCipherSpec    | CT_Alert
  | CT_Handshake           | CT_ApplicationData

type tls_version = SSLv2 | SSLv3 | TLSv1 | TLSv1_1 | TLSv1_2
```

```
type alert_level = AL_Warning | AL_Fatal
type alert_type = ... (* 30 alert types *)

type tls_alert = {
  alert_level : alert_level;
  alert_type : alert_type;
}

type record_content =
  | Alert of tls_alert
  | ...  (* other possible content types *)

type tls_record = {
  content_type : tls_content_type;
  record_version : tls_version;
  record_content : record_content;
}

let parse_tls_version input =
  match (parse_uint16 input) with
  | 0x0002 -> SSLv2
  | 0x0300 -> SSLv3
  | 0x0301 -> TLSv1
  | 0x0302 -> TLSv1_1
  | 0x0303 -> TLSv1_2
  | _ -> raise ...

let parse_tls_content_type input = ... (* 3 other functions   *)
let parse_alert_level input = ...      (* to parse enumerated *)
let paree_alert_type input = ...       (* values              *)

let parse_tls_alert input =
  let l = parse_alert_level input in
  let t = parse_alert_type input in
  { alert_level = l; alert_type = t }

let parse_record_content content_type input =
  match content_type with
  | CT_Alert -> Alert (parse_tls_alert input)
  | ...  (* other cases to handle *)

let parse_tls_record context input =
  let content_type = parse_tls_content_type input in
  let record_version = parse_tls_version input in
  let record_content_string = parse_varlen_string parse_uint16 input
   in
  let record_content = parse_record_content content_type
   record_content_string in
  in {
    content_type = content_type;
    record_version = record_version;
    record_content = record_content;
  }
```

Obviously, the major downside of this approach is that, even if we rely on various helpers from libraries, it requires to write a lot of code. On the positive side, we could write the parsers with all the relevant details, including value validation and complementary checks, while keeping the code efficient.

We implemented this method in Python, C++ and OCaml, to ensure we could express the parsers we needed and handle all the details such as ill-specified or wrongly implemented messages. Moreover, several sanity checks about the low-level input parsing can be done in the basic helpers (`parse_uint16` should abort properly if the input does not contain at least two bytes). Yet, this method does not scale, and this process of writing a large number of repetitive lines of code is actually very error-prone (we regularly encountered implementation bugs such as typos).

6.2.4 Use of a Domain-Specific Language (Scapy)

To avoid having to write a lot of repetitive tedious code, it is possible to rely on a Domain Specific Language (DSL) to describe the structures. Such a language could then be interpreted to parse and access the structures. This is the approach provided by Scapy [BtSc16], a toolbox frequently used to analyse and forge network protocols. Using a Scapy-like syntax, TLS alerts could be described as follows:

```
_tls_type = { 20: "Change␣Cipher␣Spec", 21: "Alert",
              22: "Handshake",          23: "Application␣data" }

_tls_version = { 0x0002: "SSLv2",   0x0300: "SSLv3",
                 0x0301: "TLSv1",   0x0302: "TLSv1.1",
                 0x0303: "TLSv1.2" }

_tls_alert_level = { ... }   # Two other
_tls_alert_type = { ... }    # enumerations

class TLSPlaintext():
    name = "TLS␣Plaintext"
    fields_desc = [ ByteEnumField("type", None, _tls_type),
                    ShortEnumField("version", None, _tls_version),
                    FieldLenField("len", None, length_of="fragment",
   fmt="!H"),
                    StrLenField("fragment", "", length_from=lambda
   pkt: pkt.length) ]

class TLSAlert():
    name = "TLS␣Alert"
    firlds_desc = [ ByteEnumField("alert_level", None,
   _tls_alert_level),
                    ByteEnumField("alert_type", None, _tls_alert_type
   ) ]

bind_layers (TLSPlaintext, TLSAlert, type=21)
```

This elegant and concise solution allows to quickly describe formats featuring variable-length fields, context-dependent fields. However, the interpretation step leads to a significant penalty on the runtime performance (especially in a language such as Python).

During our study, we implemented partial parsers in Python with this approach in mind. This language, which allows introspection, makes it easy to mix the DSL to the language in a concise way. Yet, the resulting programs were too slow, and more importantly, did not provide strong guarantees on data types, which is not in accordance with our robustness goal.

Beyond Scapy, there exists another library to describe parsers using a DSL: Hachoir [Sti15], which is mostly focused on binary file formats.

It is worth noting that in parallel, efforts were made at ANSSI, outside this thesis, to implement a fully functional TLS stack with Scapy, which could be used to interact with existing stacks.

6.2.5 Use a preprocessor (OCaml and `parsifal`)

Since the previous method was a clear step forward, we specified a DSL in a strongly-typed language, OCaml. We went back to the manual method and tried to automate all the steps using a preprocessor. While writing parsers for different TLS messages, several patterns emerged. We thus envisioned a way to generate types and parsing functions from simple descriptions (enumerations, structures, unions, ASN.1 DER types). Using `parsifal` syntax, detailed in the following section, the TLS alert example would become:

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

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

enum tls_alert_level (8, Exception) = ...
enum tls_alert_type (8, UnknownVal AT_Unknown) = ...

struct tls_alert = {
  alert_level : tls_alert_level;
  alert_type : tls_alert_type
}

union record_content (Unparsed_Record) =
  | CT_Alert -> Alert of tls_alert
  | ...

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

Like the previous method, the developer only needs to write a short description, but the goal is to expand this description into the code that would have been manually written. So, at the expense of a short description, this solution brings the advantages that usually go with manually writing the functions (performance, robustness if the language permits it). It is however worth noting that every network construction or file format can not be described by simple descriptions. Using a preprocessor expands simple constructions while the developer still can write proper types and functions when the DSL expressiveness is not enough.

In an ideal world, we would like all the formats to fit within a small subset of patterns, but protocol and file format designers are usually very innovative to invent ad-hoc structures and constraints, which might not be easy to include in a description language. `parsifal` represents a trade-off between the DSL approach to describe the easy parts of the analysed format and the manual approach for the tricky parts.

The following sections further describe `parsifal` and its constructions. Since its inception, our framework has indeed been enriched by various features. Yet, we believe it is not desirable to tweak our DSL to natively deal with every existing specific representations.

6.2.6 Method comparison

Table 6.3 summarises the presented solutions, with regards to the expected properties.

	cast	***match***	**Manual**	**Interpreted DSL**	Preprocessor
Representative language	C	OCaml	all	Python	all
Language used in our tools	-	-	C++ Python	Python OCaml	OCaml (2)
Adequateness to complex formats	--	-	++	+	++
Ease/speed of writing	+	+	--	++	++
Performance	++	-	+	-	+
Robustness and ability to implement constraint checks	-	+	++ (theory) - (practice)	+	++

Table 6.3: Summary of the studied parsing methods. For the Manual column, the gap between theory and practice comes from the fact that, even if it is possible to control every detail of the format, this approach needs to write a lot of code, leading to more bugs in practice.

6.2.7 Related work

`parsifal` is not the first generic framework for writing binary parsers. As already discussed earlier, popular alternatives are Scapy [BtSc16] and Hachoir [Sti15], two Python projects aiming respectively at dissecting network protocols and file formats. As a side note, it is worth noting that Scapy's goal was not only to parse network packets, but also to forge such packets, even invalid ones. Since our goal was to build robust and concise parsers, instead of fuzzing tools, we chose a different path while developing `parsifal`.

Another solution similar to `parsifal` is the `binpac` language [PPSP06], developed within the Bro project. The idea is to describe network layers in a DSL, which is then compiled to produce C code. Here again, the C language does not offer the same guarantees as OCaml. Furthermore, several parsers we later wrote confirmed that parts of a complex parsers are easier to manually write. Mixing manually written and preprocessed parts can prove hard in practice with an external preprocessor as the `binpac` one, especially from the debugging point of view.

Another promissing functional language we considered is Haskell. Using typeclasses seems to be a good way to automate code generation for parsing functions. As a matter of fact, there already exists a popular Haskell library, Parsec [LM15]. This library seems essentially used to parse LL(1) grammars, which does not always fit binary formats. Furthermore, the automating process we really need is not easily available in Haskell: there is no standard way to automatically *derive* a function in Haskell for an arbitrary typeclass, which is exactly what `parsifal` does.

Finally, in 2014, a research team presented Nail [BZ14a, BZ14b], another tool to parse and generate data formats. The proposed approach is close to the one followed in `parsifal`: a describing language (the protocol grammar) allowing to generate C code to parse and dump data formats, while letting the developer write ad-hoc code to handle specific constructions. The examples illustrating Nails are the ZIP file format and the DNS network protocol.

6.3 `parsifal`: a generic framework to write binary parsers

`parsifal` is the result of a process taking into account both our needs and the gathered experience about binary parsing. It is a generic framework relying on a `camlp4` preprocessor and a library containing many helper functions. The preprocessor allows us to use new keywords to automate the generation of large, boring chunks of code, while letting the developer manually write the complex parts.

6.3.1 $\mathcal{PT}$ypes

`parsifal` relies on the idea that tedious code should not be written by humans since it can be generated. The basic blocks of a `parsifal` parser are $\mathcal{PT}$ypes, that is OCaml types augmented by the presence of several manipulation functions: a $\mathcal{PT}$ype `t` is composed of:

- the corresponding OCaml type `t`;
- a `parse_t` function, to transform a binary representation of an object into the type `t`;
- a `dump_t` function, that does the reverse operation, that is dumping a binary representation out of a constructed type `t`;
- a `value_of_t` function, to translate a constructed type `t` into an abstract representation, which can then be printed, exported as JSON, or analysed using generic functions.

There are essentially three kinds of $\mathcal{PT}$ypes:

- basic $\mathcal{PT}$ypes, provided by the standard library, like integers, strings, lists or ASN.1 DER basic objects;
- keyword-assisted $\mathcal{PT}$ypes, like records, are descriptions using a pseudo-DSL integrated to the language thanks to a preprocessor (some of the offered constructions are presented in the remaining of the section);
- custom $\mathcal{PT}$ypes: when the corresponding structure is too complex to simply describe, you can always fall back to manually writing the type and the functions.

As an example of the conciseness of `parsifal` code, the remaining of the section briefly describes how to write a TLS parser. Some subtleties, such as record fragmentation, are intentionally left out.

6.3.2 Basic TLS record structured

To write a TLS parser, the first step is to describe some basic blocks and the record structure, as described earlier in table 6.1. To implement enumerated fields such as the protocol version or the record content type, we use the `enum` keyword:

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

enum tls_content_type (8, Exception) =
  | 0x14 -> CT_ChangeCipherSpec
  | 0x15 -> CT_Alert
  | 0x16 -> CT_Handshake
  | 0x17 -> CT_ApplicationData
  | 0x18 -> CT_Heartbeat
```

Its meaning is essentially self-explanatory: it creates a concordance between the integer value and their names. The `enum` keyword also takes two arguments, the number of bits used to represent the integer value and the parser behaviour in case the parsed value does not exist: `Exception` to raise an exception, `UnknownVal` to add a fallback constructor.

A TLS record is composed of four fields: a content type enumeration, a protocol version, the content length on 16 bits and the record content itself, which depends on the content type. Such a structure is easy to code with the `struct` keyword, by enumerating the different fields:

```
struct tls_record = {
  content_type : tls_content_type;
  record_version : tls_version;
  content_length : uint16;
  record_content : binstring[content_length];
}
```

Since we can not precisely describe the record content yet, we use a generic variable length binary string (using the `content_length` field to define its length).

As for `enum`s, the `struct` definition will tell `parsifal` to automatically generate the associated `parse`, `dump` and `value_of` functions. We can thus test our trivial implementation with the following lines:

```
let input = string_input_of_filename Sys.argv.(1) in
let r = parse_tls_record input in
print_endline (print_value (value_of_tls_record r))
```

Here is the result of the compiled programs on a TLS alert:

```
value {
  content_type: CT_Alert (21 = 0x15)
  record_version: TLSv1 (769 = 0x301)
  content_length: 2 (0x2)
  record_content: 0228 (2 bytes)
}
```

Of course, this is only the beginning, and one would likely want to improve the description by enriching the record content. This is actually a strength of our methodology: it is generally easy to describe the big picture and then to refine the parser to progressively take into account more details.

6.3.3 $\mathcal{PC}$ontainers: a useful concept

As we began using `parsifal` to describe various file formats and network protocols, it dawned on us that it might be useful to create another concept that could be reused: $\mathcal{PC}$ontainers. Initially, the only existing containers were lists, but the notion of containers can be broader: the abstraction of a container containing a $\mathcal{PT}$ype makes it possible to automate various kinds of transformations that must be applied at parsing and/or dumping time. Here are several examples:

- encoding: hexadecimal, base64;
- compression: DEFLATE, zLib or gzip containers;
- safe parsing: some containers provide a fallback strategy when the contained $\mathcal{PT}$ype can not be parsed;
- miscellaneous checks: CRC containers are useful to check a CRC at parsing time and to generate the CRC value at dumping time.

There again, the idea is to reuse code to reduce the time spent writing the same code time and again. For our simple record description, we can simplify its representation by using the variable length container to replace both the `content_length` field and the `record_content`:

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

An assumed design choice in `parsifal` is to simplify the manipulation (parsing and dumping) of valid files when possible, even if it makes other use cases (like fuzzing) less accessible. By using an automatic container, we indeed loose the ability to easily forge malformed records. Such records can nevertheless be useful to tests some attacks such as the Heartbleed vulnerability.

6.3.4 Union and progressive enrichment

To illustrate how to enrich the `record_content` field, let us start with the alert sub-protocol, corresponding to the `CT_Alert` content type. As explained earlier in table 6.2, an alert payload is composed of two bytes (the alert level and the alert type). Similarly to the record and its related enumerations, the alert content can be described with the following code:

```
enum alert_level (8, UnknownVal AL_Unknown) =
  | 1 -> AL_Warning
  | 2 -> AL_Fatal

enum alert_type (8, UnknownVal AT_Unknown) =
  | 0 -> AT_CloseNotify
  ...
  | 110 -> AT_UnknownExtension

struct alert = {
  alert_level : alert_level;
  alert_type : alert_type
}
```

To use this new structure when dealing with the `CT_Alert` content type, we have to create a `union` $\mathcal{PT}$ype, depending on the content type:

```
union record_content [enrich] (UnparsedRecord) =
  | CT_Alert -> Alert of alert
```

Finally, we have to rewrite the `record_content` field in the `tls_record` structure:

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

Enriching other content types should be clear from now: write the $\mathcal{PT}$ype corresponding to the associated payload, then add a line in the `record_content union`. When facing an unknown content type, `parse_record_content` will produce an `UnparsedRecord` value containing the raw unparsed payload. Now that we have partially improved the description of records, we can recompile our program printing records and use it to parse a TLS alert and a `ClientHello` message, to show both cases:

```
 $ ./tlsparser alert.bin
value {
  content_type: CT_Alert (21 = 0x15)
  record_version: TLSv1 (769 = 0x301)
  record_content {
    Alert {
      alert_level: AL_Fatal (2 = 0x2)
      alert_type: AT_HandshakeFailure (40 = 0x28)
    }
  }
}

$ ./tlsparser clienthello.bin
value {
  content_type: CT_Handshake (22 = 0x16)
  record_version: TLSv1 (769 = 0x301)
  [Unparsed]_record_content: 0100008b030100000000... (143 bytes)
}
```

There is obviously some more work to handle real-life TLS messages, in particular all the Handshake messages. More examples of `parsifal` constructions can be found in appendix C, which contains step-by-step tutorials to handle TAR archives and PNG images.

6.4 Case studies

Initially, `parsifal` has been written to develop robust parsers able to analyse vast amounts of data, and more importantly to understand how several protocols or file formats really work. The first parsers covered TLS messages and X.509 certificates, but `parsifal` proved useful to describe more formats, sometimes to study new areas, sometimes as a challenge to test `parsifal`'s expressiveness.

At the beginning, these challenges required changes in `parsifal` preprocessor or in its standard library, but such modifications have become rare lately, which means we have reached a balance between language expressiveness and implementation complexity. Here are several examples of encountered difficulties, as well as how we handled them in `parsifal`.

6.4.1 ASN.1 RSA key

ASN.1 is usually considered complex to parse. However, it is important to remember that ASN.1 is an *abstract* syntax notation, that can be encoded using different rules. The Basic Encoding Rules (BER) allow variable length data structures, whereas the Distinguished Encoding Rules (DER) are more restrictive and require data are in canonical forms. For our needs, we only considered the latter, which uses a relatively simple TLV (Tag/Length/Value) header.

To simplify the development of ASN.1 DER types, header and scalar value parsing is automated by `parsifal`, through basic DER types implemented in `parsifal` standard library, and thanks to new helper keywords to build ASN.1 sequences. For example, PKCS #1 [JK03] describes an ASN.1 structure for RSA private keys. The following code implements this type and is a complete program to parse a PEM[4] file named `key.pem` containing an RSA private key and print the corresponding modulus:

```
asn1_struct rsa_key = {
  version : der_smallint;
  modulus : der_integer;
  publicExponent : der_integer;
  privateExponent : der_integer;
  prime1 : der_integer;
  prime2 : der_integer;
  exponent1 : der_integer;
  exponent2 : der_integer;
  coefficient : der_integer
}

let input = string_input_of_filename "key.pem" in
let key = parse_base64_container parse_rsa_key input in
print_endline (hexdump key.modulus)
```

6.4.2 DNS

At first, analysing DNS messages was a challenge to better understand the notion of *parsing context*. DNS resource records can indeed be *compressed* by referencing to previously read domain names. This form of compression requires a context retaining the domain names encountered during parsing. DNS parsing thus relies on the following data structure:

```
type dns_pcontext = {
  base_offset : int;
  direct_resolver : (int, domain) Hashtbl.t;
}
```

[4]PEM (Privacy-enhanced Electronic Mail) is the name given to Base64-encoded DER data.

The hash table is updated every time a new domain is parsed in the message, and can be used when encountering compressed domains. Offsets are computed relatively to the beginning of the message. The same effort must be done when dumping a message: record the offsets of the names produced and reuse them to compress the result.

Another challenge in DNS message handling was to offer a simple way to handle bit fields. As in many network protocols, boolean fields are encoded on a single bit, and small integers are sometimes encoded on less than 8 bits. Handling these values could be left to the developer, but since they are so pervasive, we added support for bit fields in `parsifal`, extending the semantics of the `enum` keyword and adding several $\mathcal{PT}$ypes. In the specific case of of DNS, the following snippet describes how flags can be described in `parsifal`:

```
enum opcode (4, UnknownVal UnknownOpcode) =
  | 0 -> StandardQuery
  | 1 -> InverseQuery
  | 2 -> ServerStatusRequest

struct dns_flags = {  (* 16-bit bit field     *)
  qr : bit_bool;      (*  - 1-bit boolean     *)
  opcode : opcode;    (*  - 4-bit enumeration *)
  (* ... *)
  rcode : bit_int[4]; (*  - 4-bit integer     *)
}
```

As a side note, it might be worth noting that, in the same spirit integers can be big- or little-endian, bit integers can be read from left-to-right (the somewhat *intuitive* way) or from right to left, leading us to improve bit fields with a `rtol_` (right-to-left) prefix. The latter convention can be found in the DEFLATE compression format [Deu96][5].

6.4.3 DVI: an old format

We also briefly analyse the DVI file format, which proves an interesting anti-pattern, as far as binary formats are concerned. Indeed, a file can simply be described by the following structures:

```
struct dvi_command = {
  opcode : opcode;
  command : dvi_command_detail (opcode);
}

alias dvi_file = list of dvi_command
```

This means that a DVI file is a list of *commands*. Yet, command details (and in particular the command *length*) depend on the opcode. It is thus necessary, to parse a given file, to at least know the lengths of the DVI commands it contains. This clearly is *not* a desirable property, since it limits its format extensibility: adding new commands would probably break command alignment in older implementations, even if these commands are optional.

6.4.4 PE

To better understand UEFI, several co-workers at ANSSI had to study Portable Executable programs; the format of Windows executables (or a variant thereof) is indeed used in UEFI. To this aim, they tried `parsifal`, and had to deal with what strikes us as a bad idea: non-linear parsing. To analyse a PE, you have to walk through the file using offsets, forwards and sometimes backwards.

We made it possible in `parsifal`, but as a result of this back-and-forth walk in the file, it becomes very hard to check several properties on such file formats: how to check that parts of the files are not used several times? what about unused chunks, especially when digital signatures are computed against a recomposed version of the file?

[5] Actually, to uncompress DEFLATE blocks, you need right-to-left bit fields for the headers, but the actual Huffman compression uses left-to-right bitfields...

The use of arbitrary pointers within a file format, which we call non-linear parsing, was also found in Exif, the Exchangeable image file format used by TIFF and JPEG images to store metadata.

6.5 Performance comparison

This section proposes preliminary comparisons between parser implementations. The criteria used are the number of lines needed to code the parser, and the time needed to parse a typical input.

6.5.1 Extracting certificates from TLS answers

First, let us compare several of our TLS parsers described in the introduction: the C++ parser, the OCaml one (using a DSL) and the `parsifal` one. One typical tasks for those parsers is to extract the server certificates from an answer, when possible. The results on a typical input (containing both valid TLS messages and other random responses) are presented in the following table:

	C++	OCaml	`parsifal`
LOC	8,500	4,000	1,000
Processing time	100 s	40 s	8 s

The poor results for the C++ implementation most certainly comes from a lack of skill in the complex features of the language that we used. Yet, the difference in the number of lines of code speaks for itself.

6.5.2 PNG parsers: the intern's choice

In 2013, an intern studied two image file formats. PNG was the first one to be developed. We instructed our intern to write two PNG parsers: one in C (a language he already knew) and one in OCaml using `parsifal`. The following table gives several elements to compare both implementations, tested against a set of 63,000 PNG images.

	C	`parsifal`
LOC	4,000	350
Processing time	1,5 h	4 h

There again, the code written in `parsifal` is very short, by comparison to the C code. As a side note, when the intern had to implement the JPEG format, he did not have enough time for two developments, and he chose `parsifal` over C, even if writing in C could have led to better performance.

6.5.3 MRT/BGP: the need for an efficient robust parser

To better understand the structure of internet and its resilience, we studied the BGP protocol (Border Gateway Protocol [RLH06]), through which networks are connected to form the internet. The RIPE[6] provides daily archives of BGP messages collected at different points of the internet, the collectors. This data, contained in MRT archives (Multi-Threaded Routing [BKL11]), amounts to about 2 GB per day.

At first, we used existing tools to extract the BGP route declarations. `libbgpdump`[7], a program written in C was the right tool, except when the program crashed due to complex memory bugs. Since we could not easily understand the reasons of the errors, we quickly rewrote a tool to parse MRT files and match the output of `libbgpdump`. This was done using `parsifal` in three days. The following table compares different implementations: the `libbgpdump` C version, an independent OCaml implementation, and the `parsifal`-powered tool.

	`libbgpdump`	OCaml	`parsifal`
LOC	4,000	1,200	550
Processing time	23 s	180 s	35 s

[6] The RIPE is one of five Region Internet Registries providing Internet resource allocations that support the operation of the Internet globally.

[7] `http://www.ris.ripe.net/source/bgpdump`

Both TLS and PNG cases compare different versions of the same code, written by the same person using different development methods. On the contrary, the MRT case shows results for implementations written by different people. Obviously the results only cover few use cases, but on these examples, `parsifal` parsers are much shorter than the other ones. It is even harder to conclude anything about the relative speed of the compared tools; however, from our point of view, it is sometimes acceptable to pay the price for robustness, as in the MRT case.

6.6 Lessons learned

`parsifal` has been developed at ANSSI for more than 5 years and its interface has become rather stable. In this section, we discuss several lessons we learned while writing binary parsers.

6.6.1 On formats

There exists such thing as a good or a bad format. Formats relying on simple *TLV* (Tag/Length/Value) structures are very easy to parse. Moreover, they allow for partial parsing (for example when considering unknown extensions). A concrete example of such a good format, according to our experience, is PNG: chunks respect the TLV logic and the corresponding structures are rather simple.

On the contrary, several properties lead to error-prone parsers and should be avoided. For example, *non-linear parsing* makes it unnatural to check whether the data we parse are in a licit location; this is the case in PE and JFIF formats, the latter actually being similar to a filesystem with directories and entries. One other form of non-linearism is the use of trailers in files or network messages, as in the ZIP format, but we did not investigate such cases with `parsifal` yet.

Another undesirable property is when parsers are required to know every option to correctly interpret the file, such as the DVI format.

6.6.2 On the language

`parsifal` was written in OCaml, a strongly-typed functional language. Here are the advantages of this language, regarding our goal to write parsers:

- the language is naturally expressive, which helps to write concise code;
- higher order functions allows to write containers easily (e.g. `parse_base64_container` naturally takes as an argument a `parse` function to handle the encapsulated $\mathcal{PT}$ype);
- as the memory is handled automatically by the garbage collector, several classes of attacks (double frees, buffer overflows) are impossible[8];
- pattern matching exhaustiveness check is a very helpful feature when dealing with complex structures or algorithms, since the compiler will remind you of unhandled cases.

It is interesting to note that these last two points were also highlighted by another team writing a TLS stack in OCaml [KMMMS15]. Another benefit of OCaml (and functional languages in general) is the ability to write in a pure functional style, as described in several projects [KMMMS15, BFK+13, BDK+15].

More general thoughts on programming languages can be found in appendix D.

6.6.3 On the process

In the end, our choice to automatically generate the tedious parts of the code paid of: `parsifal` allows to quickly write binary parsers, even for complex formats. What's more, the description process turned out to be accessible, even for persons with no initial OCaml (or functional programming) background. In particular, the ability to progressively enrich the format definitions was a very popular feature.

Moreover, interactions with `parsifal` users confirmed that we should not try to add everything in the DSL. Parts of a parser are so complicated that they should remain manually written. That is the reason why we kept the possibility to fall back to manual $\mathcal{PT}$ypes when needed.

[8] However, this alone is far from perfect, since we may avoid arbitrary execution but not a fatal exception.

6.6.4 Concluding thoughts on `parsifal`

`parsifal` is a generic framework to describe parsers in OCaml which has been used to describe several file formats and network protocols. From our point of view, this tool has all the expected properties: expressive language, code conciseness, efficient and robust programs. Moreover, `parsifal` allows for an incremental description, which is useful to progressively learn the internals of a new format. The contribution of `parsifal` to security is twofold. First it can help provide sound tools to analyse complex file formats or network protocols. Secondly we can implement robust detection/sanitisation systems.

However, our tool has has several limits. First, due to design choices, `parsifal` is not meant to write fuzzers, but to simplify the parsing of *valid* inputs. Moreover, we currently only use `parsifal` to analyse data and to build standalone sanitising tools. One logical next step would be to use it in more real-world contexts, possibly with other programming languages, e.g. IDS software including our robust parsers. Future work also includes writing libraries to completely handle file formats and network protocols, beyond the mere parsing step, e.g. a reference TLS stack or PNG tools.

Even if `parsifal` now handles a lot of use-cases in an efficient way, several ideas from related projects might be useful to include, as the *combinator* concept from Nails, an elegant way to handle ad-hoc transformations, the ability to create recursive $\mathcal{PT}$ypes (they appear for example in EXIF, DEX or PKCS#7), or an enhanced way to manipulate parsed data. These improvements would however require a major rewriting of the preprocessor engine.

Finally, `parsifal` is not suited for text-based file format, where `lex` and `yacc` would be more adapted. In 2015, we studied PDF (Portable Document Format [ISO08]), which uses both text-based structure elements and binary elements. There again, `parsifal` did not fit our needs and we had to write an ad-hoc parser, `caradoc` [ELM16]. Unifying these different types of parsers would also be very interesting.

Chapter 7

Challenges in TLS implementations

This chapter describes the implementation challenges one has to face when writing a TLS stack. These difficulties cover standard programming issues, parsing bugs, cryptography-related problems and errors in the state machine automata. This chapter, that was partially presented in [Lev15], presents known attacks and discuss possible approaches to tackle their root causes. However, these thoughts are mostly perspectives and the corresponding publications, [JL14] (which is the basis of appendix D) and [MRLG15], only partially cover the subject.

After writing SSL/TLS parsers using `parsifal`, the next step was to write a TLS stack. A rudimentary implementation built on top of `parsifal` led us to grasp the real complexity of the protocol and of a possible implementation. In parallel, many flaws have been discovered; some of them have already been discussed in chapters 1 and 2 but we consider them here from the implementation point of view. This chapter is a state of the art of TLS implementation flaws.

We first analyse standard and simple programming issues in section 7.1. Then we move on to parsing bugs in section 7.2. Section 7.3 presents thoughts about the impact of cryptographic specifications on implementations. Finally, section 7.4 discusses security flaws resulting from errors in the state machine automata. In each section we try to understand how TLS implementations could be improved with regards to the corresponding flaws, and we study whether TLS 1.3 could play a role in these improvements.

7.1 Classic programming errors

Let us first investigate common programming mistakes that were recently made in SSL/TLS stacks.

7.1.1 CVE-2014-1266: Apple's `goto fail`

In February 2014, Apple released a security advisory, indicating that an attacker could completely bypass the server authentication mechanism in its operating systems.

The flaw was quickly identified: in the `SSLVerifySignedServerKeyExchange` function, a `goto fail` instruction was wrongly duplicated (see listing 7.1), which led a portion of the code to be unreachable. So, when a client negotiated DHE or ECDHE ciphersuites, the server signature over the Diffie-Hellman parameters was *not* checked.

An attacker could thus simply impersonate a TLS server by forcing the use of a vulnerable suite in the `ServerHello` message, then present the legitimate certificates, and finally send an arbitrary `ServerKeyExchange` message, since its authenticity would not be checked: from the client's point of view, the server certificate was however correctly validated, leading to an authentication bypass.

The problem was fixed in the week following the bug publication, but it showed that the corresponding code had never been checked using simple static analysis, since such a simple case of dead code should have been detected. As the relevant code was open-source, it also showed, once again, that it is not because people can review code that it is achieved.

After this bug, people started to advocate for better compilers, analysis tools, or even for the use of safer alternative languages instead of the C language. Narrow-minded commentators explained that

```
SSLVerifySignedServerKeyExchange( ... )
static OSStatus
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Listing 7.1: Vulnerable code from Apple's `goto fail`.

this bug would have been avoided, had the developers used curly brackets for their `if` statements. Yet, without tools enforcing such practice, we would still have no guarantee; and if we were using tools, it would be safer to have them identify the root cause (dead code).

7.1.2 CVE-2014-0092: GnuTLS' `goto fail`

Several days after Apple's `goto fail`, GnuTLS released an advisory concerning a vulnerability in the code checking signatures. This time, the issue was (a little) subtler and was about the `check_if_ca` function (which was supposed to return whether a certificate was a certification authority) and the `_gnutls_verify_certificate2` function (supposed to return the result of a signature check). Even though the documentation stated these functions returned a boolean value (that is 0 or 1), they could actually return something else. There could indeed be *three* different outcomes:

- 1 meant the signature was correct;
- 0 meant the signature was wrong;
- in some parsing error cases, the result was a negative int (see listing 7.2 for example).

So, when critical functions, such as `gnutls_x509_crt_check_issuer`, called a flawed function and treated the result as a boolean, they would accept valid certificates, but also malformed certificates. As can be seen in listing 7.3, only the `ret == 0` would indeed lead to reject the certificate.

GnuTLS developers fixed the corresponding logic in a defensive way, on the one hand by insuring the called functions indeed returned only `0` or `1`, as specified, and on the other hand by having the calling functions check the result in a stricter way.

It is worth noting that a very similar bug was found six years earlier, in 2008, in OpenSSL (CVE-2008-5077). There again, the question of the used tools and languages was raised.

7.1.3 More certificate validation security flaws

TLS has been subject in its history to other certificate validation flaws. Such vulnerabilities are usually critical, since they defeat one of the main security guarantee of the protocol: server authentication. In this section, we discuss two other security bugs related to validation security flaws.

```
/*
* Returns only 0 or 1. If 1 it means that the certificate
* was successfuly verified. [...]
*/
static int _gnutls_verify_certificate2( ... )
{
    ...
    result = _gnutls_x509_get_signed_data( ... );
    if (result < 0) {
        gnutls_assert();
        goto cleanup;
    }

    ...

cleanup:
    if (result >= 0 && func)
        func(cert, issuer, NULL, out);
    _gnutls_free_datum(&cert_signed_data);
    _gnutls_free_datum(&cert_signature);

    return result;
}
```

Listing 7.2: Several functions (here, `_gnutls_verify_certificate2`) returned negative values when the data could not be parsed, which was contrary to the function comments.

```
    ret = _gnutls_verify_certificate2( ... );
    if (ret == 0) {
        /* if the last certificate in the certificate
         * list is invalid, then the certificate is not
         * trusted.
         */
        gnutls_assert();
        status |= output;
        status |= GNUTLS_CERT_INVALID;
        return status;
    }
```

Listing 7.3: An example call site of the vulnerable `_gnutls_verify_certificate2` function from the `gnutls_x509_crt_check_issuer` function: if the returned value was `-1`, the execution continues as if the certificate was valid.

CVE-2002-0862 and CVE-2011-0228: `BasicConstraints` checks

To distinguish a CA certificate from a server certificate, the X.509 standard requires relying parties to check the `BasicConstraints` extension. This X.509 extension contains a boolean, `cA`, which should only be true when the certificate can be considered to be a certification authority. As extensions are also signed, their integrity is guaranteed.

In 2002, Marlinspike showed that Internet Explorer did not actually check this boolean [Mar02], allowing an attacker to reuse any certificate she possesses (even a simple SSL server one) to sign new arbitrary certificates.

What is more interesting is that the same bug reappeared in 2011 in a different, independent, TLS stack: Apple iOS [Mar11].

CVE-2015-1793: The problem with alternative chains

To accommodate misconfigured servers sending out-of order or incomplete certificate chains (as described in section 4.1), OpenSSL implemented a mechanism to build certificate chains and to validate more server `Certificate` messages.

However, in July 2015, an OpenSSL security advisory was published, describing a flaw in this recently added code: an attacker could craft a specific `Certificate` message to get OpenSSL to accept an untrusted certificate, when the alternative chains option was activated.

7.1.4 CVE-2014-0160: Heartbleed

On April 8th 2014, a new, devastating, vulnerability regarding TLS was presented. It came with a website, a logo and a brand: Heartbleed. It relied on a buffer overflow in the Heartbeat extension implementation.

Heartbeat is a TLS extension allowing peers to exchange records of a new type. When it has been negotiated, the client or the server can, *at any time*, send a Heartbeat records containing data, and the other end has to echo the data back. Such a mechanism can theoretically be used for two purposes:

- Path MTU Discovery: by using variable-size payloads, it is possible to discover the maximum acceptable packet size for the used channel;
- Keep Alive: by regularly emitting Heartbeat messages, which do not carry meaningful application data, it is possible to keep an unused connection open.

In practice, both goals are mostly relevant to DTLS, the datagram version of TLS, which can be used over UDP. However, the Heartbeat extension was integrated to OpenSSL for both DTLS *and* TLS on December 31st 2011[1], and activated by default.

The vulnerability is very simple: when OpenSSL receives a `HeartbeatRequest` message advertising a content longer than the sent payload, a vulnerable implementation fills the `HeartbeatResponse` message with the received content, and then add whatever was present afterwards in memory.

Initially, it was hard to understand the exact impact of the flaw, but it was later shown to be critical: Heartbleed could allow anyone to spy on communications between *other clients* and a vulnerable server. All kinds of heap-allocated data could also be recovered: plaintext TLS communications (including authentication cookies and passwords), but also internal server data such as its private key.

As a side note, it it interesting to read the Heartbeat specification [STW12] along with the TLS specification [DR08], since this leads to a possible confusion concerning incomplete Heartbeat messages: the former claims such messages should be dropped whereas the general record framework defined by the latter allows records to be split. The Heartbeat specification could and should have been clearer on this.

7.1.5 CVE-2014-6321: remote code execution in SChannel

In November 2014, Microsoft published a security advisory, MS14-066, including several vulnerabilities. One of them was about SChannel, Microsoft's TLS stack, and could lead to remote code execution. The flaw was located in the `DecodeSigAndReverse` function, which parses ECDSA signatures in the context

[1] The first official OpenSSL version containing the Heartbeat extension was 1.0.1, published on March 14th 2012.

of client authentication using certificates. The vulnerability was dubbed Winshock; the name was chosen to look like Shellshock, a critical flaw against `bash` published several weeks earlier (CVE-2014-6271), even though these vulnerabilities have nothing in common.

When handling a client certificate using elliptic curves cryptography, the certificate, present in the `Certificate` message, is first parsed. This step allows the server to know the elliptic curve used for the signature. It is indeed described within the public key information, either using well known implicit identifiers (several curves can be identified by ASN.1 Object Identifiers, like `1.3.132.0.34` for `secp384r1`), or in an explicit manner (the description then contains the underlying finite field, and the curve equation). In both cases, the public key sets the coordinate size. In a second phase, the signature is read from the `CertificateVerify` message, which contains the coordinates of a point on the curve.

In SChannel, the vulnerable code allocated memory regions for the signature, based on the curve description from the certificate, then read the data from the signature, using this time the encoded ASN.1 length of the signature, *without checking the consistency between both lengths*. It was thus possible to trigger a buffer overflow using long coordinates within a crafted signature in the `CertificateVerify` message.

Smashing data in the heap using this vulnerability was proven exploitable in IIS servers: proofs of concept could even lead to remote code execution on vulnerable systems.

Theoretically, since the parsing issue was only present in the code handling client authentication using certificates, only servers proposing this form of authentication, by sending a `CertificateRequest` message, should have been affected. However, even *unsolicited* `Certificate` and `CertificateVerify` were parsed by SChannel. This is actually a second bug, that belongs to another class of flaws described in section 7.4.

7.1.6 Analysis and solutions

In the previous examples, we studied simple mistakes in critical functions, that were sometimes repeated in different independent projects.

Sometimes, it seems the used language or tools could and should have been more helpful, either by offering a *real* boolean type[2], or by warning the user of trivial errors (such as having obvious dead code in a function). In our study on programming languages, presented in appendix D, we found that languages and tools do not help the developer as much as they can, which means programmers should be careful with their code.

One way of hardening the code is to use more warnings in the build phase; for example, in C, one can enforce stricter checks using `-Wall -Wextra -Werror` and other `-Wxxx` options at all times. Another good practice is to always err on the safe side; this was done by GnuTLS developers to fix the presented bug, by only considering `1` as the valid case, instead of *everything that is not `0`*.

Some programming languages are better than others to avoid whole classes of bugs, but there is no silver bullets, and it is essential to always understand exactly what you are trading in exchange for what. As a first example, languages relying on a garbage collector to manage memory usually protect the developer from use-after-free and double-free errors; however, having no fine-grain control on memory management means that you cannot easily ensure that secrets (private keys, passwords) are erased as soon as possible from the memory. Actually, they could even be copied at multiple memory locations several times by the garbage collector during their lifetime!

Another example is the use of safe arrays, where each read or write access is checked at runtime to stay within the array boundaries. Even if this kind of mechanisms induces a small overhead, it is a good way to avoid classical errors such as buffer overflows. Yet, developers should be aware that several constructions evade these safety mechanisms, and stay away from such unsafe features. In the same vein, it is easy to miss such unsafe usage when they happen in a third-party library: it is thus necessary to understand the limits of the provided security mechanisms, by understanding how the runtime and the libraries work, especially when they interact with each other.

Finally, an interesting feature of many functional languages is the pattern matching construction, which can be seen as a sophisticated switch/case statement. If used correctly, it can signal when a case is forgotten, or when some code will never be executed. This particular feature proved useful in the `parsifal` development, as explained in section 6.6. It requires however that the developer activates the corresponding warnings in the compiler, and fix the resulting errors.

[2]The C language includes such a type since the C99 standard, but it seems almost no one makes use of it.

All in all, mastering programming languages and correctly using compilers and other tools can help improve code quality and avoid several classes of software bugs. Another best practice that can be improved in TLS stacks development is *negative* testing: when security is involved, it is not sufficient to check that what should work works, it is crucial to also check that what should not work does actually not. Checking that a Handshake fails as expected in case the `ServerKeyExchange` message is wrong would have prevented Apple's `goto fail` vulnerability. Since programmers seem to make the same mistakes time and again, it might be even more effective to cross-check *negative* and non-regression tests between implementations.

As a matter of fact, it would not be fair to conclude that all these bugs were only the result of poor programming practices; developers obviously bear their share of responsibility, but several errors were also the result of complex or ill-specified protocols (e.g. the Heartbeat specification) and formats (ASN.1 and DER[3] is well known to be hard to implement correctly). This will become even more true in the following sections.

7.2 Parsing bugs

In the previous section, we studied classical simple errors such as memory management issues. Such bugs can arise in the parser code, as can be seen by reading recent OpenSSL and GnuTLS security advisories: CVE-2015-1789 (OpenSSL, buffer overflow in `ASN1_TIME` manipulation), CVE-2015-0286 (OpenSSL, comparison functions choking on unexpected ASN.1 booleans), CVE-2015-3308 (GnuTLS, double free in CRL Distribution Point decoding), CVE-2015-6251 (GnuTLS, double free in Distinguished Name decoding), etc.

Beyond these somewhat common bugs, parsers trigger another class of vulnerabilities, when the parsed content does not correspond to its intended value. Such bugs can result from a confusion in the specification or a lack of precision in the parsing code.

7.2.1 CVE-2009-2408: Null characters in distinguished names

In 2009, Marlinspike presented several bugs in TLS stacks, leading to an authentication bypass [Mar09]. Amongst these vulnerabilities, he presented a difference in string interpretation between several X.509 implementations, regarding the presence of null characters.

The specification is clear on the subject: the length is explicitly set by a separate field and *not* by a terminating null character; moreover, most ASN.1 string types do not allow for null characters at all. However, several browsers, implemented using the C language, did not handle correctly null characters, which were interpreted as the end of the string, leading to different possible interpretations.

The presented scenario is the following. An attacker would request a certificate for the TLS server named `www.mybank.com\0.evil.com`, where `evil.com` is controlled by the attacker and `\0` is the null character. In some cases, the certification authority would simply look from the top-level domain (here `evil.com` and try to get confirmation from someone behind the `postmaster@evil.com` email address, which would here be the attacker. The certificate, which is legitimate, can then be used against vulnerable browsers to impersonate `www.mybank.com`.

Beyond the obvious misinterpretation from browsers, which should not rely on null characters to end ASN.1 DER strings, there is another bug: the CA should not accept ill-formed DER data. The only available string type are UTF-8, UTF-16, (essentially) UTF-32, `PrintableString` and `T61String`: the first three are Unicode representations, which should not be handled as mere octet strings, whereas `PrintableString` explicitly disallows null chars. `T61String` (or `TeletexString`), which allows null characters as *control characters*, is seldom used. In all the cases, it should be impossible to simply accept a null character as part of a fully-qualified domain name.

This example shows that, as soon as two implementations do not agree on the interpretation of a given element, there is a gap that an attacker can (and will) exploit. This is the reason why, contrary to what is usually taught, the Postel principle (*Be liberal in what you accept, conservative in what you send*) is dangerous and should be replaced by another, simpler, statement: be conservative, always (and report bugs in confusing specifications)[4].

[3] DER stands for Distinguished Encoding Rules. It is one possible concrete representation of the Abstract Syntax Notation ASN.1.

[4] Another way to put it is "to be conservative in what one does and paranoid in what one accepts from others — and

Layer	Field	Size	Value
Record	Type	1	`16` (Handshake)
	Version	2	`03 01` (TLS 1.0)
	Length	2	Record length
Handshake	Type	1	`01` (`ClientHello`)
	Length	3	Handshake message length
`ClientHello`	Version	2	`03 03` (TLS 1.2)
	...	...	...

Figure 7.1: Contents of the first bytes of a standard `ClientHello` message. It is the second version field, within the Handshake layer, that is used for the negotiation.

7.2.2 CVE-2014-3511: OpenSSL *downgrade attack*

In July 2014, Benjamin and Langley showed that OpenSSL exhibits a strange behaviour when it receives a `ClientHello` message split in very small records, even if the specification allows records from the same type to be split and merged in a very liberal way[5].

Yet, when parsing the first `ClientHello` fragment, an implementation needs at least 6 bytes in the record payload to identify the proposed protocol version (see figure 7.1). In the absence of this information, OpenSSL can not choose a proper version and systematically chooses TLS 1.0 instead of waiting for the rest of the `ClientHello` message.

Again, it is important to understand that a Handshake message can rightfully be split. Handshake messages can indeed be 16 MB long (their length is encoded using 3 bytes) whereas TLS records are limited to 16 KB.

To fix this bug, OpenSSL developers have chosen to simply reject such `ClientHello` messages, that did not propose a protocol version upfront; this is an incorrect behaviour with respect to the specification, but the alternative was deemed too complex to implement.

This attack shows that the complexity of TLS, combined with the need to support several protocol versions, can lead to subtle implementation difficulties.

7.2.3 CVE-2014-1568: NSS/CyaSSL/PolarSSL Signature Forgery

In September 2014, another vulnerability allowing to bypass server authentication on several TLS clients was published. This time, it affected NSS, the Firefox cryptographic library, as well as other TLS implementations such as CyaSSL and PolarSSL. The vulnerability takes its root in the code parsing DER-encoded RSA signature. DER is a concrete representation of ASN.1, which enforces a normal form: there should be one and only one correct representation for each abstract value. The vulnerability is a universal signature forgery relying on three elements to be exploitable:

- the attacker needs to find an RSA key with a public exponent equal to 3. This exponent can be anywhere in the certificate chain she is trying to spoof;
- the ASN.1 DER parser must be too liberal, i.e. accept non-canonically encoded values;
- DER length computation can silently overflow.

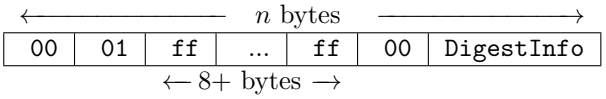
Within TLS, RSA signatures use the PKCS#1 v1.5 standard (it is also the standard commonly used in X.509 certificates, hence the ability to exploit the vulnerability anywhere in the chain). To sign a message m with a private key (N, d) using the hash function H, PKCS#1 v1.5 requires the following steps:

- Hash: compute $h = H(m)$;
- Format: prepare an ASN.1 DER block encoding a sequence containing the identifier of the used hash function and the hash value h. We denote by d this block, also called `DigestInfo`;

prone to freeze when surprised.", as Colin Percival states it on his blog [Per15].

[5] In practice, what is allowed and forbidden is not always clear in the specification, which led to the Alert attack [BFK+13]; it might also be seen as one of the causes of Heartbleed.

PKCS#1 v1.5 signature format:



`DigestInfo`:

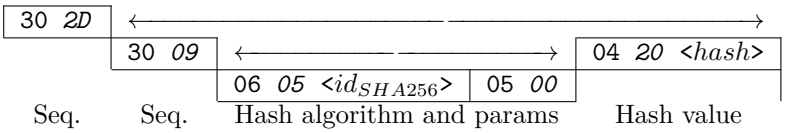


Figure 7.2: PKCS#1 v1.5 signature format. *Seq.* are ASN.1 sequences, length are represented in italic, and `<hash>` is the hash value of the message to sign (here using SHA-256).

- Pad: with n the length of the RSA modulus N, create an n-byte long octet string starting with `00 01 ff ... ff 00`, followed by d, where the number of `ff` bytes is adjusted accordingly. Let x be the integer represented by this value (see figure 7.2);
- Sign: the final result is $x^d[N]$.

In 2006, Bleichenbacher proposed an attack to exploit broken RSA implementations that did not check the absence of data beyond the `DigestInfo` block [Ble06]. Figure 7.3 presents an *incorrect* message that could be accepted by such implementations. In the case of a small public exponent (such as 3), it is easy to forge a signature for such a message. Assuming an attacker wishes to forge a signature for a given `DigestInfo` block, she can prepare a message m similar to the one presented in the figure, with a small value for p and the garbage part filled with zeroes[6]. Let s be the smallest integral value greater than the *real* cubic root of the big integer represented by m. As soon as n is big enough, s^3 will only differ from m in the garbage part of the message, leaving the `DigestInfo` part intact. With a fuzzy DER parser, it is thus possible to forge an arbitrary signature for an RSA key using 3 as its public exponent.

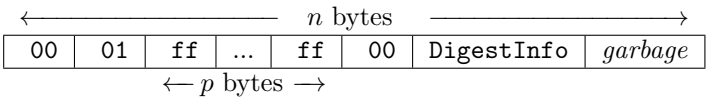


Figure 7.3: Example of a malformed PKCS#1 v1.5 message. With a liberal parser, messages of this form will be accepted, which leads to a Universal signature forgery attack.

The vulnerability exposed in September 2014 is subtler, since it relies on a non-canonical DER representation. For example, instead of representing the SHA-1 hash length by a simple byte (`0x14`), the attacker uses an alternative representation, which should be forbidden in DER, `XX 00 .. 00 14`, with `XX` equal to `0x80` + l, the number of bytes used to encode the length (here, the number of null bytes plus one).

Even if this lack of precision allows the attacker to mount an attack, it is still hard to exploit in practice. There was actually another flaw in the DER parsing code: when reading a very long length field (as the one described above), an integer overflow allowed the attacker to use arbitrary values for all but the four bytes, which would lead to a length field of the form `XX YY .. YY 00 00 00 14`, where `YY` is a sequence of $l - 4$ bytes controlled by the attacker.

Figure 7.4 shows an example of message targeted by this attack with a 1024-bit modulus (since the `XX` only allows for a 7 bit-long length field, the attack is more complex and requires splitting the target in more pieces with 2048-bit moduli). As in the original Bleichenbacher attack against PKCS#1 v1.5 signature forgery attack, the public exponent must be equal to 3 for the attack to succeed in practice.

[6] As a matter of fact, even if p is specified to be at least 8, some implementations allow the padding to be empty ($p = 0$), which fives the attacker more wiggle room.

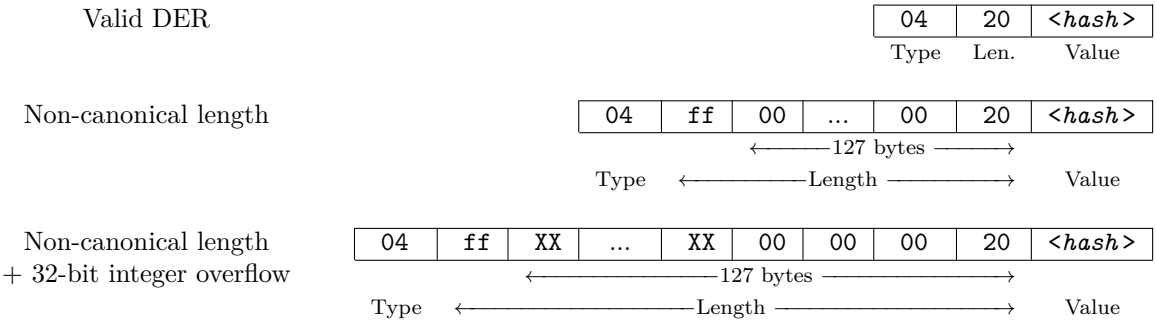


Figure 7.4: Example of a subtler malformed PKCS#1 v1.5 message, which was accepted by many implementations in 2014. In the ASN.1 DER part of the message describing the hash value, the length field can be mangled and still be accepted. In the third case, because of the integer overflow, `XX` can be chosen arbitrarily by the attacker. Of course, the rest of the `DigestInfo` must be built accordingly.

A common fix here is to reverse the comparison process while checking a signature: instead of computing $m = s^e$ then parse m and finally compare the encompassed hash value inside m, a robust implementation should produce the message $m^\star$ containing the expected `DigestInfo`, then compute $m = s^e$ and compare m to $m^\star$. By comparing concrete representations instead of abstract ones, we skip the parsing step and the only operations manipulating attacker-controlled data are the s^e computation and the trivial binary comparison. Moreover, since DER is a canonical representation of the abstract value, $m^\star$ is unambiguously defined[7].

As a side note, Parsifal PKCS#1 v1.5 rudimentary implementation was found to be subject to the original variant of Bleichenbacher attack, which led us to rewrite the comparison step in the verification function.

7.2.4 More on specification complexity

Some parsing bugs result from a confusion in the specification. In a sense, the Heartbleed buffer overflow could have been described in this section instead of the previous one. Let us look at two more examples where RFC were misinterpreted by developers.

In 2006, the IETF standardised the session tickets extension, allowing for session resumption without server-side state [SZET06]. However, no TLS stack implements the specification correctly: even if the specification described the *content* of the extension as a variable-length object (that is an opaque object prefixed by its length), every implementation ignores this second (useless) length field. Figure 7.5 shows the difference between the original RFC and the RFC published in 2008 to fix the gap between the specification and the implementations [SZET08].

```
The SessionTicket extension has been assigned
the number 35.  The format of the SessionTicket
extension is given at the end of this section.

  struct {
    opaque ticket<0..2^16-1>;
  } SessionTicket;

00 23          Ticket Extension type 35
01 02          Length of extension contents
01 00          Length of ticket
FF FF .. ..    Actual ticket
```

```
The SessionTicket extension has been assigned
the number 35.  The extension_data field of
SessionTicket extension contains the ticket.

00 23          Ticket Extension type 35
01 00          Length of extension contents (ticket)
FF FF .. ..    Actual ticket
```

Figure 7.5: Difference between RFC 4507 [SZET06] (on the left) and RFC 5077 [SZET08] (on the right) in the specification and in the encoding of a 256-byte long ticket.

[7] This is actually partially true, since some implementations still produce ill-formed `DigestInfo` where the algorithm parameters is omitted, instead of being a DER NULL element. To accommodate such pervasive deviations, a robust implementation should thus produce two versions of $m^\star$.

More recently, in September 2014, the Encrypt-then-MAC has been standardised for TLS [Gut14]. The idea is to reverse the order in which encryption and MAC are computed during the record preparation. During the specification, two different projects implemented the new extension to check for interoperability issues. However, one month after the publication, another set of implementers disagreed on the exact meaning of the record length field when using the Encrypt-then-MAC extension, leading to interoperability problems. An erratum was added to the RFC to clarify the language.

Both examples show that informal specifications can lead to interoperability issues, which could sometimes turn into security vulnerabilities. It is especially worrying in the second case, since the problem arose from the specification of a security extension.

7.2.5 Analysis and solutions

The bugs presented in this section show that parsing attacker-controlled data is an error-prone process that should never be overlooked. In the TLS and X.509 cases, this is amplified by many subtleties in the message encoding, namely ASN.1 DER representation and the ability to split messages in small records.

As soon as parsing is not straightforward and can lead to ambiguities, security vulnerabilities may arise, either because of different actors interpreting the same messages differently, or because it allows an attacker to tamper with the expected execution path. There again, the so-called robustness principle is a terribly wrong advice regarding writing software in general and parsers in particular.

Recipes to improve security would include writing strict parsers, avoid exposing them when possible (e.g. by comparing concrete representations instead of abstract, parsed ones), stress-test the parsers in corner cases. Yet, the real long-term advice is to simplify the specification and to express them using a more formal language, to reduce the possibilities of bugs and ambiguities in the resulting code.

Concerning the Universal Signature Forgery vulnerability, it is worth noting that TLS 1.3 will remove the old PKCS#1 v1.5 signature scheme from the protocol and replace it with PKCS#1 v2.1 signature scheme, PSS (Probabilistic Signature Scheme). It should however be noted that PSS will only be mandatory for in-protocol signatures, not for certificates: a flawed parser could thus still be subverted by the described vulnerability within TLS 1.3. The following section describes more cases where using obsolete cryptography is dangerous.

7.3 The real impact of obsolete cryptography on security

SSL/TLS is a rather old protocol, dating back 1995. The cryptography community has since learned a lot about algorithms, schemes and protocols. This knowledge has not always been taken into account in recent versions of the protocol, mostly for compatibility reasons: RSA certificates still use PKCS#1 v1.5 signatures, TLS 1.2 still (partly) relies on PKCS#1 v1.5 encryption, the CBC mode, and the MAC-then-Encrypt paradigm. In this section, we present the implications on implementations of using obsolete cryptography.

7.3.1 CVE-2013-0169 (Lucky13) and CVE-2014-3566 (POODLE): the dangers of MAC-then-Encrypt

As seen in chapter 2, the original cryptographic mechanisms used to protect the Record Protocol, namely MAC-then-RC4 and MAC-then-CBC, have been subject to many vulnerabilities. Some of them are the result of fundamental flaws in algorithms (e.g. RC4) or schemes (e.g. CBC with implicit IV), but it could be argued that Lucky13 is merely an implementation bug: fixing it is *only* a matter of having a constant-time procedure to decrypt and check the integrity of a record.

However, when one looks at the complex corresponding patch in OpenSSL [Lan13], one is forced to note that we have replaced a beautiful and modular, yet vulnerable, code with a vast amount of complex and intricated calls to hash compression functions and decryption primitives. We have traded a simple and intuitive decrypt/unpad/MAC-check sequence with low-level instructions.

This is the reason why researchers promote higher-level and secure by design constructions, such as AEAD ciphers, to obtain strong guarantees on both the confidentiality and the integrity of the protected

data. For example, Bernstein et al. propose new programming interfaces in their NaCl library [BLS12] to avoid putting an unbearable burden on each and every programmer using cryptography.

Another simpler path, presented in 2001 by Krawczyk [Kra01], reverts the order of encryption and authentication layers: instead of computing a MAC on the record and then encrypt the record and its MAC, he advocates for the Encrypt-then-MAC paradigm: first encrypt the data, and then compute a MAC on the ciphertext. We can then guarantee, as soon as the implementation first checks the authentication tag before doing anything else, that the attacker will need to break the MAC layer before the encryption key is exposed. Moreover, this composition is safe, even when used in a modular way.

So, the Record Protocol protection was known to be flawed in 2001 when TLS 1.0 was published, but was only partially fixed in 2008 with TLS 1.2 and the introduction of AEAD constructions[8]. Only TLS 1.3 will completely obsolete the flawed CBC mode and the biased RC4 algorithm, until you need backward compatibility using TLS 1.2 or earlier versions.

The impact on TLS stacks is a difficult choice between straightforward and modular, but flawed, code on the one side, and a complex, hard-to-follow and error-prone, but theoretically sound implementation on the other side. Actually, the portability of the OpenSSL fix is debatable, since Langley explains that he had to trick the compiler to avoid low-level optimisation related to modular reductions on small integers[9].

Considering the difficulty to fix this issue, it is worth looking at the story of `s2n`, a TLS implementation recently released by Amazon [Lab15a]. Despite including countermeasures against Lucky 13, Albrecht and Paterson presented evidence that the library was nevertheless vulnerable to a weaker, yet still exploitable, form of padding oracle [AP15]. To avoid writing too low-level code, `s2n` decryption code execution time was indeed not exactly constant. More recently, Somorovsky discovered new forms of CBC padding oracles in OpenSSL: the AES-NI[10] code path did not properly handle some exceptional cases, leading to different TLS alerts and to a timing attack [Som16] (CVE-2016-2107).

Letting the developer reuse and compose existing trusted primitives would also avoid silly bugs to appear. With CBC in SSLv3, the padding scheme is ad-hoc, and must not be checked entirely *by design*. This strange peculiarity led to TLS implementations to erroneously accept SSLv3 paddings as valid, even with more recent protocol versions: POODLE, which should be a SSLv3-specific issue, was found to also affect TLS 1.0 stacks [Pet15b]. Again, had Encrypt-then-MAC been used, all these attacks would have been easily prevented by a straightforward implementation.

As a side note, the CBC padding used in TLS is not the standard one: instead of signaling a n-byte padding with n bytes with the n value, each of the n bytes must contain the $n-1$ value. This led to an off-by-one error while checking the CBC padding in several TLS implementations: only $n-1$ bytes were checked, instead of n [Böc15].

However, even if we should not necessarily blame developers for flaws such as Lucky13 and POODLE (even the TLS 1.0 version), since the specification paves the way with numerous traps, they are sometimes at fault. For example, Pettersen describes an experiment called MACE (for MAC Error) [Pet15a], which shows that some implementations do not actually check *all* the bytes in a MAC computation! Such missing checks could allow for really unexpected attacks, since in all the model we usually consider, we rely on the fact that the MAC will eventually be checked.

7.3.2 CVE-2014-0411: PKCS#1 v1.5 and Bleichenbacher

Section 7.2.3 presented vulnerabilities related to the signature scheme from PKCS#1 v1.5. Let us now study the encryption scheme, which is also used in SSL/TLS, and which was also broken by Bleichenbacher.

A valid PKCS#1 v1.5 message is produced by formatting the plaintext and then encrypting it using the raw RSA operation. The expected format for an encrypted message is presented in figure 7.6: a

[8] One might also consider the recent Encrypt-then-MAC extension [Gut14] as an acceptable solution. It was however published too late, and should rather have been part of the TLS 1.1 standard.

[9] The details are presented at the end of the related blog post [Lan13]. In a nutshell, the DIV instruction takes a variable amount of time depending on its argument on Intel CPUs, which could be observable.

[10] AES-NI is a set of Intel CPU instructions allowing fast and secure AES operations: since the heart of the algorithm is executed in hardware, timing and cache attacks do not apply.

null character, followed by a block type byte (here, 2), then at least 8 padding random bytes, a null character and finally the message to encrypt.

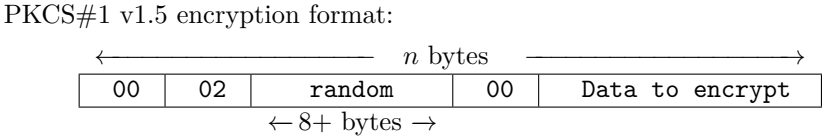


Figure 7.6: A valid PKCS#1 v1.5 encrypted message: the padding must contain at least 8 random non-null bytes.

It thus means that every correctly padded plaintext starts with `00 02`, which corresponds to a big integer between 2^{n-16} and 3^{n-16} (with an n-bit modulus). If an attacker wishes to recover the plaintext P associated to a given ciphertext C, she can multiply C by X^e and submit the new ciphertext to a decryption oracle: the padding will be correct as soon as $P \times X$ is between the expected bounds. By iterating such attempts, it is possible to aggregate information about the original plaintext P and recover it, as was shown by Bleichenbacher in 1998 [Ble98] in his so-called Million Message Attack. It was later improved to require less messages [KPR03, BFK+12]. The attack is applicable to RSA encryption key exchange in TLS.

As described in RFC 3218 [Res02], there are three classical countermeasures:

- group all possible errors so they lead to a unique signal, where the padding errors are indistinguishable from other errors;
- where possible, ignore all errors silently and replace the decrypted message by a random string (this is what is recommended for RSA encryption key exchange in TLS);
- use PKCS#1 v2.1 encryption (OAEP, Optimal Asymmetric Encryption Padding [JK03]) instead of the obsolete version.

Even if the Million Message Attack has been known since 1998, it is still a problem in recent TLS implementations. For example, in October 2012, Green tweeted the image in figure 7.7: instead of generating the random string *before* trying to decrypt the message, OpenSSL only generates the random string in case of an error, which could introduce a timing variation that could eventually be turned into a padding oracle.

The Bleichenbacher attack also resurfaces in the JSSE (Java Secure Socket Extension) SSL/TLS implementation [MSW+14]: by reusing standard cryptographic libraries, the JSSE implementation has to rely on them to handle padding errors, which could generate a specific error message or a timing difference due to the use of exception. These example show again a dilemma between code reuse and security: it is impossible to safely reuse standard libraries that throw exceptions. It is worth noting that the researchers also found new oracles in OpenSSL (the late random generation presented earlier) and Cavium hardware accelerators, with less efficient attacks.

Moreover, researchers have recently shown that PKCS#1 v1.5 padding oracles could even impact safe implementations, as soon as the same key was used by a vulnerable implementations. DROWN (CVE-2016-0800) shows that SSLv2 stacks including countermeasures against the Million Message Attack actually offer another form of padding oracle, by construction[11]; this oracle could then be used to decrypt a TLS `ClientKeyExchange` message, using the vulnerable SSLv2 stack as an oracle [ASS+16]. The same paper, as well as in an article presented in 2015 [JSS15], shows that a decryption padding oracle can also be used to forge a valid signature. This attack is not realistic in the TLS context, where the signature would have to be forged during the connection; however, QUIC [HISW16], an experimental transport protocol that was used as an input for TLS 1.3, is vulnerable.

It is thus clear that PKCS#1 v1.5 is inherently flawed, and, as with the MAC-then-CBC scheme described earlier, developers will get it wrong, time and again, until this obsolete algorithm is removed from the specification. In the mean time, it is crucial to avoid reusing the same RSA key in different contexts (decryption and signature, PKCS#1 v1.5 and PSS), since a vulnerability in one context may indirectly be used to attack the other.

[11] Moreover, the security issue was exacerbated by OpenSSL implementation bugs that could lead to efficient attacks.

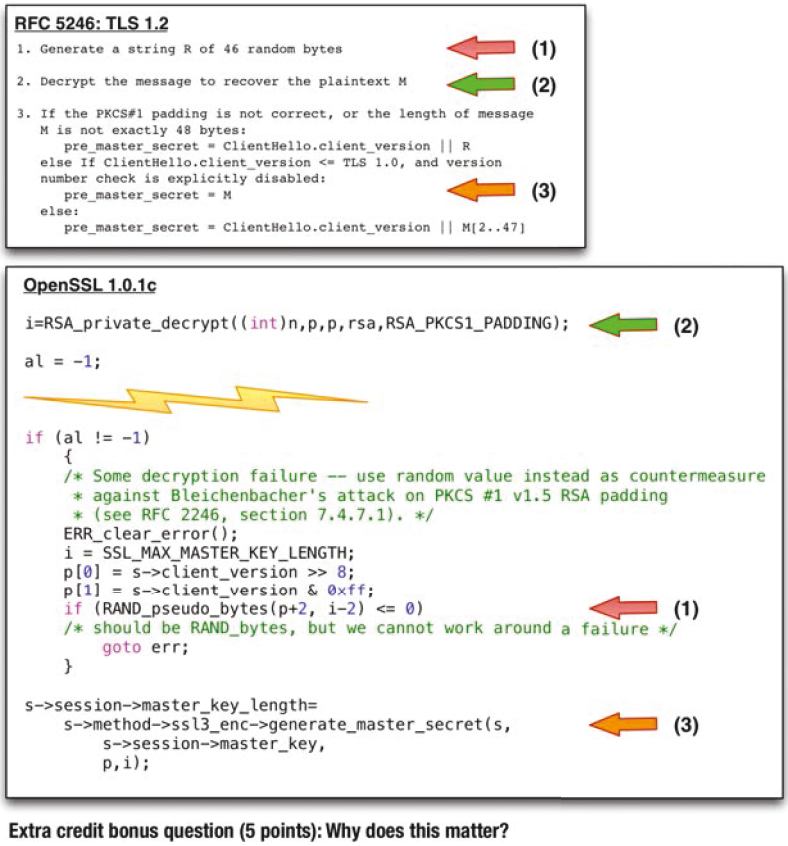


Figure 7.7: OpenSSL Fact tweet from October 2nd 2012 about OpenSSL 1.0.1c RSA decryption code.

7.3.3 Similar concerns in OpenPGP implementations

As a side note, we also studied another case where using obsolete cryptography led to security vulnerabilities that were very hard to fix: OpenPGP.

OpenPGP is a popular message format that specifies cryptographic transformations to encrypt and sign data [CDF+07]. Concretely, the encryption relies on blockcipher algorithms using the CFB mode[12], either in an encrypt-only mode, or in a hash-then-encrypt mode (the so-called *Encrypted Integrity-Protected* packets).

Our work [MRLG15] illustrates that OpenPGP CFB mode, both with or without the integrity protection, naturally leads to numerous format oracles. Implementations first need to decrypt OpenPGP packets, and then have to parse the resulting plaintext to check the plaintext structure: many constraints must indeed be checked, such as the presence of valid packet types (called tags) or the presence of hard-coded values signaling the hash value (which makes Integrity-Protected packets actually *more* vulnerable). As for TLS, a simple solution would have been to use the Encrypt-then-MAC paradigm, and to enforce that decryption only occurs *after* message integrity has been validated.

In practice, several OpenPGP libraries, including GnuPG, OpenPGP.js and End-to-End, are vulnerable to format oracles, because they output too much information in their error messages. An attacker can recover a plaintext message by submitting forged packets and analysing the resulting error messages. Even if the attacker model is debatable for some implementers, we argue that cryptographic library implementers should not presume of the usage that will be made of their work: as soon as an attacker can interact with an automated decryption process and observe error messages, this process is exploitable.

[12]The specification seems to impose a null IV, but at the same time specifies that the message be prepended by a random block, which will have the same effect as using and sending a random IV.

We thus learned two lessons: as before, the real solution is using up-to-date cryptography only (and handle the transition period properly); then, implementing cryptographic libraries is hard and should reveal as little information as possible (even innocuous-looking error messages can eventually become an exploitable side-channel).

7.3.4 Analysis and solutions

We can expect three properties from applications involving cryptographic mechanisms: security with regards to known attacks, compatibility with the existing ecosystem, and code modularity (i.e. the ability to reuse and combine existing high-level primitives). In practice, since TLS still partly relies on obsolete cryptography, it seems impossible to have the three properties at once. A developer must pick at most two out of three:

- modularity and compatibility, which corresponds to using standard primitives without specific countermeasures, which leads to attacks such as BEAST or Lucky 13;
- security and compatibility, which consists in rewriting large chunks of low-level cryptographic code to add complex countermeasures. The resulting code is complex and hard to maintain;
- security and modularity can be obtained by using only up-to-date robust cryptographic constructions (e.g. AEAD modes), at the expense of a compatibility loss.

As history showed with Bleichenbacher attacks and CBC padding oracles, attacks only get better over time: originally impractical attacks later become exploitable by more sophisticated attacks in more hostile environment. It is therefore necessary to choose the third way, and to only use sound algorithms and schemes to help developers do their job without having to jump through improbable hoops: a good cryptographic design should be easy to implement, in a modular and portable way.

Protocol specification committees should thus take cryptographer's advice into account, and as soon as possible ban flawed algorithms or constructions. The problem with most cryptographic flaws is not *whether* they are exploitable but *when* they will be. It took four years to turn the first MD5 practical collisions into a real-world certificate signature forgery, but almost 15 years to turn RC4 statistical biases to real-world attacks against TLS and WPA. Since 2015 saw the first practical collisions on the SHA-1 compression function, do we really want to take bets as to the date of a concrete threat against TLS or any other commonly used protocols? TLS 1.3 represents significant advances on the subject, since the new protocol versions should remove many cryptographic algorithms (such as RC4, MD5 and to a lesser extent SHA-1), modes (CBC for blockciphers, PKCS#1 v1.5) and parameters (arbitrary finite field group are replaced by properly sized named groups for the Diffie-Hellman key exchange).

7.4 The consequences of complex state machines

Since 2014, several attacks concerning flaws in TLS state machine implementations were published. Their impact can be catastrophic, either by skipping essential steps of the protocol or by exposing rarely used parts of code. Such attacks demonstrate how specification complexity can result in security issues.

7.4.1 CVE-2014-0224: *EarlyCCS*

In June 2014, Masashi Kikushi showed that OpenSSL state machine is vulnerable to a subtle attack: a man-in-the-middle between an OpenSSL client and an OpenSSL server, *both* vulnerable, could forge early `ChangeCipherSpec` messages and force the parties to use weak keys [Kik14]. The precise cinematics is described in figure 7.8.

The main idea behind this attack is to exploit OpenSSL state machine that accepts an early `ChangeCipherSpec` message (the real `ChangeCipherSpec`, which is still required, will be ignored in practice), both as a client and a server. At reception time, since no *master secret* is defined yet, session keys are derived from a null *pre-master secret* and public random values. Next, the attacker has to keep both connections in a consistent state, encrypting messages with the weak keys and keeping track of record numbers to compute correct MAC values.

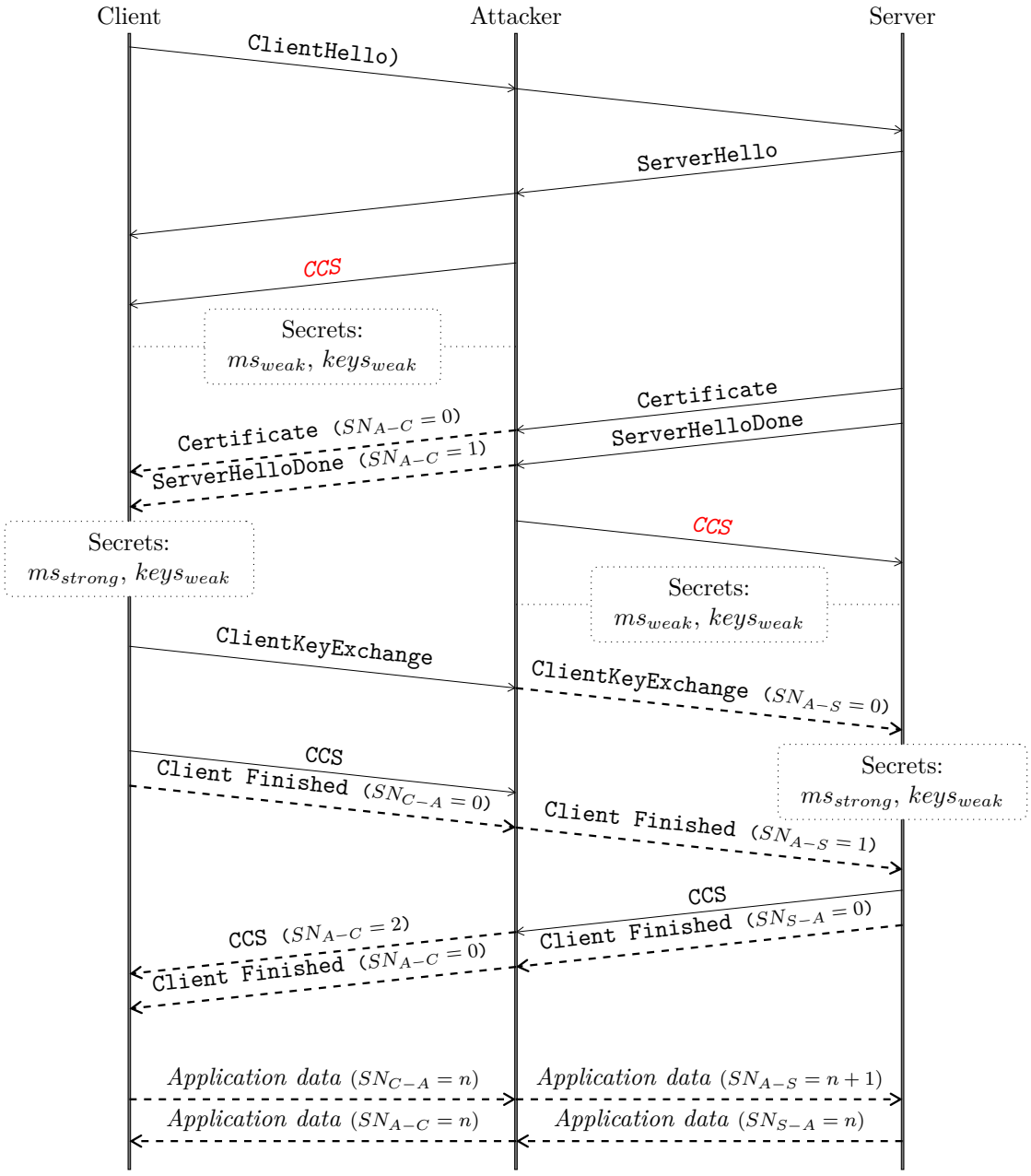


Figure 7.8: EarlyCCS attack cinematics. *ms* stands for *master secret* and SN_{X-Y} corresponds to the number of sent *record* between X and Y for the current epoch (it is reset with each `ChangeCipherSpec` message). The attacker needs in practice to keep track of four such numbers: between the client and the attacker and between the attacker and the server (with a counter for each direction).

In the end, for the handshake to terminate successfully, the attacker has to send correct `Finished` messages to the client and to the server. Since this message must contain a hash value covering, among other things, the *master secret* that was eventually agreed upon, the attacker needs both the client and the server to be vulnerable to complete the handshake.

Actually, as stated in the author's blog post, the corresponding code had already been fixed several times to handle wrongly-ordered `ChangeCipherSpec` messages: CVE-2004-0079 fixed a null-pointer assignment arising when the message was received before the ciphersuite was specified, CVE-2009-1386 fixed a similar problem in DTLS. Yet, only the direct consequence (a segmentation fault) was investigated in both cases, leaving aside the bigger picture.

As a side note, a reason why this attack is possible is because the `ChangeCipherSpec` is *not* a Handshake message, and as such it is *not* hashed in the transcript covered by the `Finished` message: thus, adding or removing a `ChangeCipherSpec` cannot be detected. TLS 1.3 solves this problem by removing `ChangeCipherSpec` altogether: the signaling sub-protocol is in fact useless. This was foreseen as early as in 1996 to be potentially problematic by Wagner et al. [WS96].

7.4.2 SMACK: State Machine AttaCKs

In January 2015, OpenSSL published a security advisory describing several security vulnerabilities: some of them were classical memory management errors, but there were also state machine-related flaws. These issues confirmed that the state automata implemented in OpenSSL was far from being robust.

More details were made available later on the SMACK website[13]: using FlexTLS, a flexible TLS stack developed by the INRIA Prosecco team, researchers tested the state machines of many different TLS stacks [BBD+15]. The results are especially worrying since they affect in practice all the known TLS stacks, to various degrees.

CVE-2014-6593: *Early Finished* (server impersonation)

In this simple, but devastating, attack, the attacker answers a vulnerable client with the following messages: `ServerHello`, `Certificate` (with the identity of the server to impersonate) and `Finished`, and *skips* the rest of the negotiation. Faced with such a shortened handshake, JSSE (Java) and CyaSSL TLS implementations consider the server authenticated and start sending cleartext `ApplicationData` messages!

CVE-2015-0205: *Skip Verify* (client impersonation)

In the case of a mutually authenticated connection, the server requests the client to present a certificate (using a `Certificate` message) and to sign the current Handshake transcript with his private key (`CertificateVerify`). Both these messages are required to properly authenticate the client. However, several implementations accept the `Certificate` message alone, where the client announces its identity, without the corresponding proof of identity: the Mono implementation indeed considers the second message as optional, but nevertheless authenticates the client; with CyaSSL, the attacker also needs to skip the client `ChangeCipherSpec` message; finally, with OpenSSL, the flaw is more subtle, since the attack only works when the client presents a certificate containing a static Diffie-Hellman public key.

Skip ServerKeyExchange (loss of the forward secrecy property)

Let us suppose a client and a server agree on an ECDSA/ECDHE ciphersuite: an ECDSA signature is used to authenticate the server and the key exchange relies on ECDHE. If an attacker in the middle removes the `ServerKeyExchange` message, some client implementations would reuse the ECDSA public key (the point contained in the certificate) instead of the expected ECDHE public parameter. This alone does not help the attacker, since removing a Handshake message creates a difference in the client and the server transcripts. It means that the connection will fail when `Finished` messages are exchanged, unless the attacker can break the Diffie-Hellman instance during the handshake.

However, when the False Start extension is used [LMM16], the client can send early application data to reduce latency, *before* checking the server `Finished`). By removing the `ServerKeyExchange`,

[13] `https://www.smacktls.com/`

the attacker thus obtains data encrypted using keys directly derived from static server-side ECDH parameters: the forward secrecy property is lost. The impacted implementations are OpenSSL and NSS.

CVE-2015-0204: FREAK (Factoring RSA Export Keys)

The last attack of the article is FREAK, which got some media coverage. As for the previous attacks, FREAK relies on an active network attacker able to modify the messages on the fly.

Section 1.1 describes a typical TLS connection using RSA encryption as its key exchange algorithm. Until the 2000's, strong encryption mechanisms were regulated by different countries. To comply with these legislations, TLS relies on an alternative key exchange algorithm: RSA-EXPORT: instead of simply sending its long-term RSA certificate (with a 1024- or 2048-bit modulus), the server also sends a second, short-term, RSA key, at most 512-bit long, signed by the long-term private key in a `ServerKeyExchange` message[14]. This way, the client encrypts its *pre-master secret* using a short RSA key, which allows for a strong server authentication (using the long-term RSA key) while respecting the rules limiting the size of encryption keys.

The FREAK vulnerability actually relies on three key elements:

- vulnerable TLS clients agreeing on a standard RSA encryption key exchange were willing to interpret a `ServerKeyExchange` containing a short-term weak RSA key, that should only appear when an RSA-EXPORT ciphersuite is selected by the server;
- (too) many TLS servers still accept to negotiate EXPORT ciphersuites; at the end of March 2015, they were around 26 % world-wide in March 2015, and 10 % when considering Top Alexa 1 Million domain names (Source: `https://freakattack.com/`);
- among these servers, many of them reuse the same short-term RSA-EXPORT key until the server reboots, which allows an attacker to factor the key while it is still used[15].

Combining all these factors, the FREAK scenario, as presented in figure 7.9, is the following:

- first the attacker A finds a server S accepting RSA-EXPORT ciphersuites, and using the same short-term RSA-512 key across sessions;
- A factors the RSA-512 key (this takes several hours with a reasonable budget);
- A leads a man-in-the-middle attack between a vulnerable client C and S, forces the client to negotiate an RSA key exchange while fooling the server into using an RSA-EXPORT ciphersuite;
- A then hands over the following messages until the `ServerHelloDone`;
- A receives the `ClientKeyExchange` she is able to decrypt with the weak private key;
- finally, A answers directly to C on behalf of S for the rest of the session.

Initially flagged as not critical for OpenSSL (which is rarely used as TLS client stack on desktop computers), FREAK affects in practice a lot of different TLS implementations: OpenSSL, BoringSSL, LibreSSL, Apple SecureTransport, Microsoft SChannel, the Mono TLS stack and Oracle JSSE (a Java cryptography provider). Beyond the original OpenSSL CVE (CVE-2015-0204), many security advisories were thus issued.

[14] As for the DHE/ECDHE `ServerKeyExchange` message, the signature also covered the client and random public values, to avoid the message to be replayed.

[15] Even if the short-term RSA should ideally be ephemeral, generating even a small RSA key on the fly is a costly operation. RSA-EXPORT implementations thus usually cache the short-term RSA key for a certain amount of time, typically between one hour and forever.

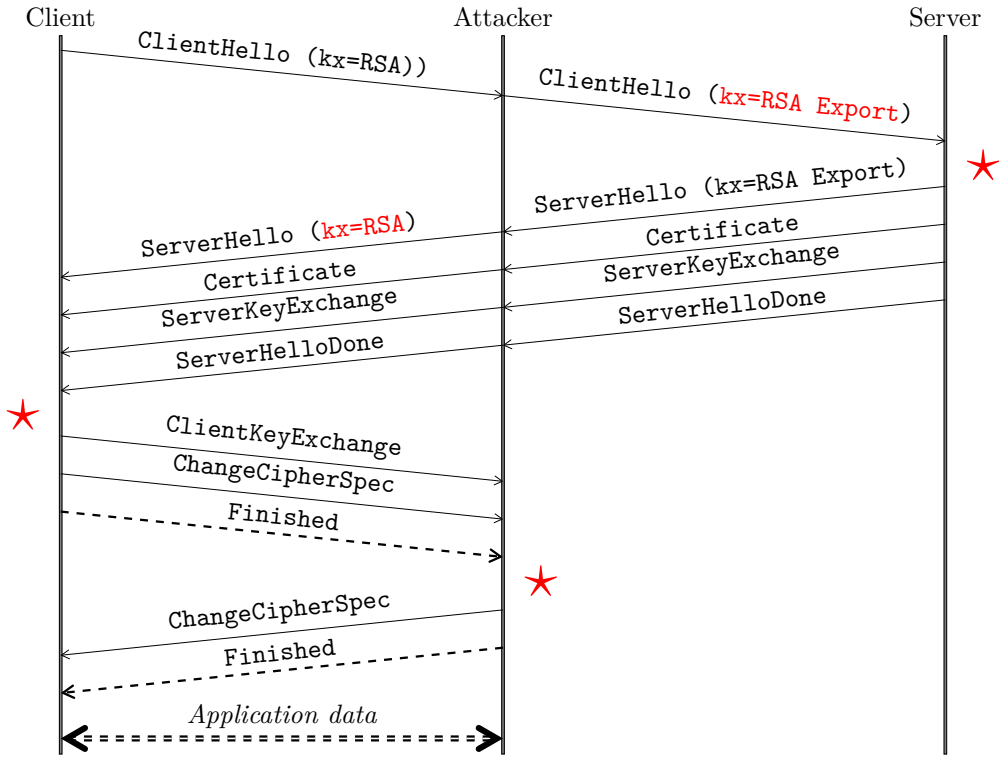


Figure 7.9: Description of the FREAK attack. Key steps of the attack are in red: Hello message mangling, server support for RSA-EXPORT ciphersuite, client tolerance to an unsolicited RSA-EXPORT `ServerKeyExchange` message, and the ability of the attacker to decrypt the `ClientKeyExchange`.

7.4.3 CVE-2014-6321: Winshock, from another perspective

Let us briefly reconsider Winshock, the remote code execution vulnerability in SChannel presented in section 7.1.5: the security bug relies on a simple buffer overflow in the code handling client authentication using ECDSA certificates. In a typical configuration, client authentication does not use TLS at all, but rather relies on application-level methods such as forms and passwords. So very few servers actually send a `CertificateRequest` message. However, an attacker could trigger the vulnerability in practice in any SChannel deployment, since even unsolicited Handshake messages are parsed and interpreted by SChannel, as in the FREAK or the EarlyCCS case. It can thus be seen as another form of state machine attacks.

7.4.4 Black-box fuzzing to evaluate TLS state machines

In 2015, Ruiter et al. described another approach to evaluate state machines in TLS stacks [dRP15]. They use state machine learning techniques to analyse different implementations as black boxes. To this aim, they choose an alphabet of abstract TLS messages (typical Handshake messages, application data and Heartbeat messages). Thanks to a software layer translating this abstract alphabet into concrete messages (the so-called test harness), they could build the observable state automata of different implementations.

The expected automata should be a straightforward “happy flow”, showing the different steps of a successful TLS session, which should typically consist in 5 states, and one more state to handle all the error cases. This is the observed behaviour for the RSA BSAFE Java library (see figure 7.10). The other studied libraries show more complex state machines (see for example the automata infered from GnuTLS in figure 7.11).

It is worth noting that, by studying the deviations of the implementations with regards to the expected simple automata, the researchers have been able to find two more vulnerabilities, including one

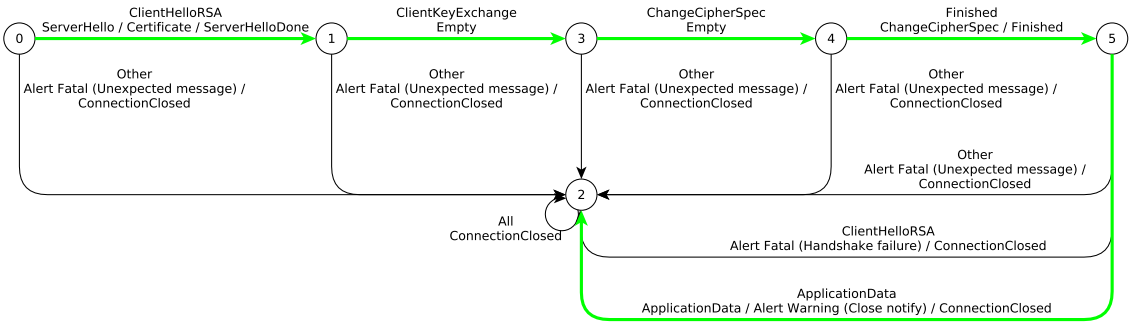


Figure 7.10: Observable state automata of the RSA BSAFE JAVA stack (version 6.1.1). 5 states clearly form the expected "happy flow", while the 2 state is the error state, where all invalid sessions eventually end. Source: [dRP15].

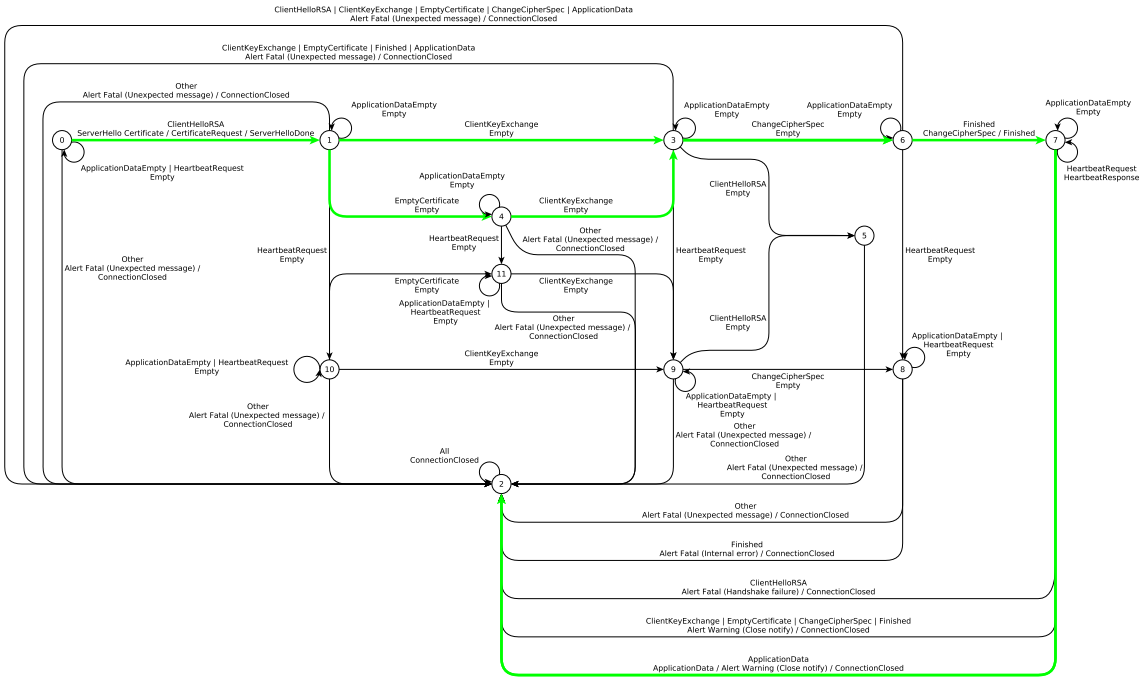


Figure 7.11: Observable state automata of GNU TLS 3.3.8. This time, the automata contains 12 states. In particular, states 8 to 10 form a shadow flow, where a Heartbeat message has led to a buffer reset. Source: [dRP15].

in GnuTLS 3.3.8, where sending a Heartbeat message would reset the buffer containing the handshake messages; this flaw could allow an attacker to mangle a handshake between a vulnerable client and a vulnerable server.

7.4.5 Analysis and solutions

The vulnerabilities presented in this section show that all major TLS implementations do not correctly keep track of the current *state* a session is in. Ideally, the TLS state machine should be driven by its current state only, not by the incoming messages: at each step, a client or a server should exactly know which messages are valid, and every other messages should trigger an `UnexpectedMessage` fatal alert.

It is worth noting that the EarlyCCS and FREAK attacks were found by research teams using formal method tools. As a matter of fact, since the TLS specification is rather complex, it seems that only a formal description of the automaton could lead to bug-free and compliant implementations. TLS 1.3 offers some progress here, by simplifying the protocol (no more `ChangeCipherSpec` messages, nor RSA key exchange) yet new features such as 0 RTT might introduce new difficulties. Moreover, RFC are usually written in natural (English) language, instead of using a formal language. Even if miTLS, the verified reference implementation of TLS developed in F# [BFK+13] can be seen as a formal description of the protocol, it is not the official one.

7.5 Concluding thoughts on implementation issues

In this chapter, we analysed different implementation bugs in TLS stacks that led to security vulnerabilities, from the most trivial ones to more complex flaws. We also offered ideas as to what could help the software community to produce more reliable tools.

There are huge classes of security flaws that rely on recurring trivial bugs such as memory management errors or integer overflows. To overcome them, there already exists type-safe programming languages that essentially avoid several kinds of bugs of being introduced in the first place. Even with unsafe programming languages, developers should take more advantage of the tools at their disposal, e.g. warnings emitted by the compiler or static analysers.

The recent attacks have also shown that the TLS ecosystem lacked an extensive, shared set of security tests, since multiple flaws were discovered, several years apart, in independent implementations of the protocol.

Finally, several vulnerabilities result from the complexity and the ambiguities of the TLS specifications. The situation is improving, in part because several versions or features have recently been obsoleted, but mostly because TLS 1.3 should be at the state of the art. Yet, the transition problem in a world where old implementations coexist with new ones will have to be dealt with for a long time.

Conclusion and perspectives

SSL/TLS has become an essential part of Internet security. The security of the protocol has been studied extensively by the research community, which resulted in the publication of many vulnerabilities in the last years. Another consequence of the attention drawn on SSL/TLS is the increasing activity to develop security extensions and to specify the next version of the protocol: TLS 1.3.

In this thesis, we studied the security of SSL/TLS from different points of view. We first analysed the protocol specifications. Beyond the original motivation of the protocol, namely allowing secure e-commerce transactions, it is now used for a vast number of applications. Since its inception, more than 20 years ago, SSL/TLS complexity has also dramatically increased: 5 protocol versions, more than 300 standardised ciphersuites, additional features such as renegotiation and session resumption, alternate authentication models beyond X.509 certificates, integration with multiple application protocols, etc. We also described known vulnerabilities, as well as the corresponding countermeasures, sorting them in different categories: handshake vulnerabilities, record protection flaws, certificate and trust issues and implementation bugs.

Among the security flaws, we focused in particular on those affecting the Record Protocol. The goal thereof is crucial, since it is responsible for the confidentiality and the integrity protection of application data. All the published attacks challenging these security properties target HTTPS, and more specifically the confidentiality of secret values repeated across different TLS sessions. This observation led us to propose defense-in-depth mechanisms to mask the repetition of such secrets. We also implemented these proposals to assess their cost and to validate their effectiveness against the described attacks. Even if our mechanisms were designed before the POODLE attack was published, they were nevertheless able to thwart the attack. This demonstrates that they are defense-in-depth countermeasures. However, we want to stress the importance of updating TLS stacks and configuring them to fix the known problems; defense-in-depth mechanisms must be used *in addition* to regular security updates, not *instead* of them.

We then studied the ecosystem from an experimental point of view. To this aim, we led several measurement campaigns in 2010, 2011 and 2014. There are several ways to interact with TLS servers in the field and we chose to scan the whole IPv4 address space in a documented and ethical manner. Since HTTPS is still the most common usage of SSL/TLS, and because HTTPS servers are usually publicly available services, our campaigns only targeted tcp/443 ports. Contrary to most of the available datasets, we chose to send multiple stimuli to each server we discovered. By comparing the answers received to different `ClientHello` messages, we were trying to understand the behaviour of TLS servers facing exotic parameters.

To analyse our datasets, as well as other public datasets, we wrote several tools, grouped into the `concerto` project. Since we had already published results on such datasets in 2012, we faced an interesting challenge: how to produce reliable and reproducible results on the same data, while taking into account the evolution of the state of the art. `concerto` was developed with the idea to combine efficiently simple programs to parse and analyse our datasets in a replayable fashion. This allowed us to compute results on various datasets, including external ones, in a similar manner. We could also compare our results in 2012 with our results four years later, when several parameters and algorithms had evolved.

The results found on TLS datasets from 2010 to 2015 show that the ecosystem is improving. For example, TLS 1.2 was supported by 30 % of the contacted hosts in 2014, and 47 % in 2015. However, this evolution is rather slow, and for the majority of the criteria we studied, we observe a small but

non-negligible proportion of insecure answers. In 2015, 3 % of the overall TLS servers still preferred SSLv3, 10 % of them chose weak ciphersuites (mostly RC4-based suites), and 30 % of them did not support the secure renegotiation standardised in 2010. All these servers are thus vulnerable to known attacks, even when fixes have been specified and implemented for a long time. These figures tend to be better for trusted hosts, but the situation is still far from perfect.

In the last part of this thesis, we focused on the implementation aspects of SSL/TLS. In particular, we looked at the parsing steps required to interpret TLS messages and X.509 certificates. To this aim, we developed several parsers, including `parsifal`, an open-source framework in OCaml designed to quickly write robust and efficient parsers. The main idea behind `parsifal` is to automatically derive most of the repetitive code from high-level descriptions of the messages to relieve developers from this task. They can then focus on the most interesting parts of the parsers. The `concerto` toolset described in the previous part relies on `parsifal`-powered dissectors to analyse the datasets. It is worth noting that beyond TLS and X.509, `parsifal` was also used to dissect and study other network protocols (e.g. BGP or NTP) and file formats (e.g. PNG images, PE binaries and OpenPGP packets) within ANSSI laboratories.

Since many implementation flaws were disclosed during the last year of this thesis, we studied them and proposed an in-depth analysis of their causes. We also discussed possible improvements. They were sorted by category:

- classic programming errors like memory management errors or trivial errors in the control flow;
- parsing bugs, resulting from complex or ambiguous specifications;
- implementation errors within the cryptographic code, induced by the use of obsolete constructions;
- faulty branches within state machines, resulting there again from the complexity of the standard.

TLS 1.3: a new hope?

The logical next step for our work on the specification aspects would be to study TLS 1.3. The current version of the draft has already benefitted from the growing knowledge of new vulnerabilities. Moreover, several changes were proposed in a proactive way, to finally get rid of obsolete primitives. This effort to clean up the standard is clearly a step forward for the protocol security. Indeed, the current version of the draft removes by construction most causes of known vulnerabilities:

- obsolete cryptographic algorithms and modes are removed (PKCS#1 v1.5, RC4, CBC);
- only ephemeral Diffie-Hellman key exchange using named groups is kept;
- the handshake flow is simplified and messages have been removed/renamed in a simpler fashion, allowing for a clean 1 RTT handshake;
- all the Handshake transcript is now included into the key derivation mechanisms;
- the `CertificateVerify` message is modified, so the signature now covers all the previous Handshake messages (and not only the random values);
- renegotiation is completely removed from the specification.

Several proposals improve the protocol "verifiability". It is important to note that these changes introduce more differences between TLS 1.2 and TLS 1.3, which might slow down the adoption of the new standard. Here are examples of these changes:

- the Pseudo-Random Function used to derive keys from the *master secret* is replaced by a more standard mechanism, HKDF, which simplifies the proofs of the protocol;
- client and server authentication are now very similar, with the presentation of the certificate, followed by a `CertificateVerify` message covering the transcript up to the certificate, and a `Finished` message to activate the ciphersuite in the considered direction;

- to prevent signatures to be reused in a different context, every signature used in TLS now covers a unique context string (e.g. “TLS 1.3, server CertificateVerify”);
- the `ChangeCipherSpec` message is removed, leading implementations to rely on safer, synchronous state machines (it also removes the last unauthenticated part of the handshake).

The TLS working group is now trying to propose an acceptable 0 RTT construction. It is a much awaited feature that could improve TLS 1.3 adoption rate. However, the mere existence of this mode is worrisome, since it cannot have the same security properties as the 1 RTT handshake (anti-replay, forward secrecy), but also because 0 RTT is re-introducing complex constructions that may be the cause of future implementation bugs.

All these changes will require an extensive security analysis, which could benefit from the use formal methods. It is worth noting that, in parallel to the specification efforts, the protocol has already been modeled and tested by several research teams using formal methods. For example, a workshop called TRON (TLS 1.3, Ready Or Not?) took place in February 2016 to present different analyses of the protocol. Pursuing this effort until the RFC is published is essential to guarantee that TLS 1.3 indeed possess the expected security properties.

Beyond the security analysis of TLS 1.3, such methods could even be extended to study other security protocols such as IKE/IPsec or SSH. Many network appliances indeed rely on the former, and the latter has become the universal administration methods for network equipments and servers.

Evolution of the SSLiverse in the field

Our experiments and other publications assessing the security of the TLS ecosystem have allowed to measure trends regarding the deployment of given features. Security researchers, protocol designers and software developers would benefit from recurring high quality TLS datasets and analyses. They would indeed allow to test in advance the intolerance of the deployed base to new features, before releasing them. This will especially be true for TLS 1.3, which may face the same kind of intolerance TLS 1.2 faced a few years back. Moreover, once a version, a feature or a cryptographic algorithm is considered obsolete, it is crucial to speed up the transition process towards their removal. Reliably measuring the adoption of new features could thus help developers enforce stronger defaults. Such changes are otherwise very hard to sell without without supporting data.

To this aim, collection and analysis methodologies must be described as rigorously as possible. This includes details concerning the selection of the contacted hosts and the nature of the `ClientHello` messages used to probe the servers. This is the reason why we released `concerto` as an open-source software. There are several ways in which our toolset could be improved. For example, revocation information and other relevant data concerning certificates should be included in the analysed data: CRLs, OCSP responses, Certificate Transparency logs.

Until recently, all the research efforts have been targeting the HTTPS servers. The situation has changed in 2015 since several research teams have started studying email protocols (SMTP, IMAP and POP). Such campaigns are required to take into account the distinctive aspects of every application protocol. Beyond HTTPS and email protocols, there might also be other TLS use cases worth investigating in the wild. When possible, these campaigns should use multiple stimuli, since they can improve the knowledge of the server behaviour.

Ideally, datasets should also be made public when possible, to allow independent researchers to reproduce the experiments and to run other algorithms on the data shared by the community.

Improving software implementations durably

The first lesson we should learn from recurring trivial bugs in critical software is that we need to use languages, tools and development methodologies allowing us not to reproduce the same bugs time and again. To this aim, we should improve our languages, compilers and static analysers, and use them more in existing projects. For example, a high impact vulnerability such as Apple’s `goto fail` could and should be detected using standard compilation options (`-Wunreachable-code` with `clang`).

Other areas in software engineering where TLS stacks seem to have been lacking behind is testing. Of course, testing is no silver bullet, especially in comparison to more formal development methodologies, but it is a good way to avoid making the same mistake twice. To this aim, we need more TLS test suites, including negative tests and non regression tests.

Negative testing should be the counterpart of functional (positive) testing: beyond checking that what should work really works, we should also check for security properties, and in particular check that what should not work actually fails: we should not ship TLS libraries accepting your favorite cat picture as a valid certificate (CVE-2008-5077 in OpenSSL *and* CVE-2014-0092 in GnuTLS) nor a library considering a `ServerKeyExchange` empty signature as valid (CVE-2014-1266 in Apple stack).

Non-regression testing are useful as safeguards, but since most of the commonly used TLS stacks are open-source, those tests should be shared when this makes sense: TLS and X.509 parsing errors can usually be tested in different implementations, and would have avoided the `BasicConstraint` vulnerability in Apple iOS, almost ten years after the same bug arose in SChannel; similarly, gathering attack scenarios such as those proposed by the SMACK study and check their outcome against every possible stack, and before every new release, would be a step forward.

Complexity is the enemy of security, be it in the specified state machine, in the cryptographic algorithms or modes, in the encoding specification or more generally in the manipulated structures. It seems TLS specifications could (and should) do a better job at defining the expected behavior of a TLS conforming implementation. By using the English language (instead of a formal description), important pieces of information are disseminated throughout the document and are hard to interpret in a unique way. For instance, it is not simple to know the exact state a message should be accepted in. Examples described in this thesis proved that it can lead to exploitable vulnerabilities.

Since many developers working on different TLS implementations regularly make the same mistakes (e.g. each of the SMACK attack affected several independent stacks), we should stop putting the blame only on the developers' shoulders and work on better, easy-to-get-right specifications, instead of the current confusing and error-prone informal description.

In TLS 1.3, the complexity introduced by several constructions such as 0 RTT, whose security implications are still heavily debated, will have significant repercussions on the corresponding code. Unless the expected behaviour of a compliant stack is made crystal clear (and hopefully formally specified) and unless relevant tests are published alongside the specification, we will certainly encounter new implementation flaws in the future.

The need for a smooth and safe transition

Finally, even if TLS 1.3 is perfect, we will still need to handle the transition period, with the co-existence of older versions. As a first step, it is important to guarantee that a recent client connecting to an up-to-date server ends up using the most recent version, using robust anti-downgrade mechanisms.

Then, we should try and update the SSL/TLS stacks wherever possible, which might require shutting down the access to old and unmaintained systems such as Windows XP (whose end-of-life was pronounced in April 2014) or Android 2 mobile phones.

Finally, when most of the clients and servers can safely negotiate recent versions of the protocol, it is possible to implement damage control by cryptographically separating old and new usages. For example, it was proposed on the TLS working group mailing list to use different certificates for pre-TLS 1.2 and TLS 1.3 sessions, which would avoid modern users to be affected by vulnerability from the past. In fact, the proposal can be extended to use a different certificate/private key for each distinct key exchange context (such a context should include the key exchange algorithm, the protocol version and any other relevant parameters). Such a measure would have limited the impact of vulnerabilities such as DROWN or FREAK.

All in all, this is certainly one of the most difficult challenge, since it requires taking into account real-life deployments, which can be very diverse, and where interoperability must be weighed against security considerations. This will require the people in charge of these obsolete and unmaintained deployments to identify the relevant systems, and isolate them properly using architectural decisions.

References

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

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *the ACM Conference on Computer and Communications Security, CCS'15, Denver, CO, USA*, pages 5–17, October 2015.

[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.

[Abr05] Jean-Raymond Abrial. *The B-Book - Assigning Programs to Meanings.* Cambridge University Press, 2005.

[ANS13] ANSSI. Internet Resilience in France – 2013 Report. Technical report, ANSSI, 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.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 16–26, May 2009.

[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.

[Atw15] Jeff Atwood. Coding Horror. `http://www.codinghorror.com/blog`, 2004-2015.

[Bar04] Gregory V. Bard. The Vulnerability of SSL to Chosen Plaintext Attack. *IACR Cryptology ePrint Archive*, 2004.

[Bar06] Gregory V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In *SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography, Setúbal, Portugal*, pages 99–109, August 2006.

[Bar11a] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[Bar11b] 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.

[BDK+15] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. Flextls: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies, WOOT'15, Washington, USA*, August 2015.

[BDLP+15] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627 (Proposed Standard), September 2015.

[Ber08] D. Bernstein. ChaCha, a variant of Salsa20. cr.yp.to/papers.html#chacha, 2008.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA*, pages 608–625, August 2012.

[BFK+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA*, pages 445–459, May 2013.

[BG05] Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases.* Addison-Wesley Professional, 2005.

[Bha16] Karthikeyan Bhargavan. Protecting Transport Layer Security from Legacy Vulnerabilities. In *Advances in Cryptology - EUROCRYPT 2016, 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria*, May 2016.

[BKB16] Karthikeyan Bhargavan, Nadim Kobeissi, and Bruno Blanchet. ProScript TLS: Building a TLS 1.3 Implementation with a Verifiable Protocol Model. *TRON Workshop - TLS 1.3, Ready Or Not*, February 2016.

[BKL11] L. Blunk, M. Karir, and C. Labovitz. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. RFC 6396 (Proposed Standard), October 2011.

[BKN04] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, 2004.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA*, February 2016.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In *18th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '98, Santa Barbara, CA, USA*, pages 1–12. Springer-Verlag, August 1998.

[Ble06] Daniel Bleichenbacher. Rump session at CRYPTO '06: Forging some RSA signatures with pencil and paper. Transposed by Hal Finney on the IETF Web mailing list: `https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html`, August 2006.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The Security Impact of a New Cryptographic Library. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile*, pages 159–176, October 2012.

[BMPvdM16] Remi Bricout, Sean Murphy, Kenneth G. Paterson, and Thyla van der Merwe. Analysing and Exploiting the Mantin Biases in RC4. Cryptology ePrint Archive, Report 20116/063, 2016.

[bms10] bushing, marcan, and sven. Console Hacking 2010. *27. Chaos Communication Congress*, December 2010.

[Böc15] Hanno Böck. A little POODLE left in GnuTLS (old versions). `https://blog.hboeck.de/archives/877-A-little-POODLE-left-in-GnuTLS-old-versions.html`, November 2015.

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

[Bou97] T. Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083 (Informational), March 1997.

[BPT15] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

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

[BTPL15] R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0. RFC 7568 (Proposed Standard), June 2015.

[BtSc16] Philippe Biondi and the Scapy community. Scapy. `http://www.secdev.org/projects/scapy/`, 2003-2016.

[BWBG+06] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFCs 5246, 7027, 7919.

[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.

[BZ14a] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Interface Generator for Data Formats. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA*, pages 158–166, May 2014.

[BZ14b] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA*, pages 615–628, October 2014.

[CC] ISO/IEC 15408: Common Criteria for Information Technology Security Evaluation. `http://www.commoncriteriaportal.org`.

[CDF+07] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. Updated by RFC 5581.

[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.

[Clu03] Jolyon Clulow. On the Security of PKCS#11. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany*, pages 411–425, September 2003.

[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.

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507, 7919.

[DAM+15] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J. Alex Halderman. Neither Snow Nor Rain Nor MITM...: An Empirical Analysis of Email Delivery Security. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan*, pages 27–39, October 2015.

[Dar09] Zaynah Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7, 2009.

[DCAT12] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Trans. Internet Techn.*, 12(1):1, 2012.

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

[Deu96] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[DH15] V. Dukhovni and W. Hardaker. SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS). RFC 7672 (Proposed Standard), October 2015.

[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.

[dRP15] Joeri de Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium, Washington, D.C., USA*, pages 193–206, August 2015.

[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.

[ELM16] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc: a pragmatic approach to PDF parsing and validation. In *37. IEEE Security and Privacy Workshops, SPW 2016, San Jose, CA, USA*, pages 126–139, May 2016.

[Ent11] Entrust. Entrust Bulletin on Certificates Issued with Weak 512-bit RSA Keys by Digicert Malaysia. `https://www.entrust.com/entrust-bulletin-on-certificates-issued-with-weak-512-bit-rsa-keys-by-digicert-malaysia/`, July 2011.

[EPS15] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), April 2015.

[ET05] P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), December 2005.

[FHBH+99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. Obsoleted by RFCs 7235, 7615, 7616, 7617.

[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.

[FM00] Scott R. Fluhrer and David A. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. In *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, pages 19–30, April 2000.

[FM11] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011. Updated by RFC 7936.

[For15] CA/Browser Forum. EV SSL Certificate Guidelines. `https://cabforum.org/extended-validation/`, 2007-2015.

[FS15] Max Fillinger and Marc Stevens. Reverse-Engineering of the Cryptanalytic Attack Used in the Flame Super-Malware. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand*, pages 586–611, December 2015.

[FSSF01] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The dos and don'ts of client authentication on the web. In *10th USENIX Security Symposium, Washington, D.C., USA*, August 2001.

[FT14] S. Farrell and H. Tschofenig. Pervasive Monitoring Is an Attack. RFC 7258 (Best Current Practice), May 2014.

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA*, pages 38–49, October 2012.

[Gil16] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919 (Proposed Standard), August 2016.

[Goo15] Mark Goodwin. Mozilla Security Blog: Revoking Intermediate Certificates: Introducing OneCRL. https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/, March 2015.

[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.

[HA04] Katia Hayati and Martín Abadi. Language-Based Enforcement of Privacy Policies. In *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Toronto, Canada*, pages 302–313, May 2004.

[HA13] Jacob Hoffman-Andrews. Forward Secrecy at Twitter. https://blog.twitter.com/2013/forward-secrecy-at-twitter-0, November 2013.

[HAM+15] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kâafar. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. *CoRR (Computing Research Repository)*, abs/1511.00341, 2015.

[HB15] P. Hallam-Baker. X.509v3 Transport Layer Security (TLS) Feature Extension. RFC 7633 (Proposed Standard), October 2015.

[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.

[HISW16] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02, January 2016.

[Hof02] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), February 2002. Updated by RFC 7817.

[HS12] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), August 2012. Updated by RFCs 7218, 7671.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium, San Jose, CA, USA*, pages 45–60, July 2008.

[IAN15] IANA. Transport Layer Security (TLS) Parameters. http://www.iana.org/assignments/tls-parameters/, 2005-2015.

[IEC] IEC 61508: Functional safety of electrical, electronic, programmable electronic safety-related systems. http://www.iec.ch/zone/fsafety/.

[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.

[ISO08] ISO. Document management — Portable Document Format — Part 1: PDF 1.7. Technical Report 32000–1:2008, International Organization for Standardization, Geneva, Switzerland, 2008.

[Jae10] Éric Jaeger. *Study of the Benefits of Using Deductive Formal Methods for Secure Developments*. PhD thesis, EDITE, 2010.

[JH08] Éric Jaeger and Thérèse Hardin. A few remarks about formal development of secure systems. In *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, Nanjing, China*, pages 165–174, December 2008.

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

[JL13] Éric Jaeger and Olivier Levillain. Langages et sécurité - généralités et cas des langages fonctionnels. In *24. Journées francophones des langages applicatifs, Aussois, France*, February 2013.

[JL14] Éric Jaeger and Olivier Levillain. Mind Your Language(s): A Discussion about Languages and Security. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA*, pages 140–151, May 2014.

[JLD+13] Éric Jaeger, Olivier Levillain, Damien Doligez, Christèle Faure, Thérèse Hardin, and Manuel Maarek. LaFoSec: Étude de la sécurité intrinsèque des langages fonctionnels. In *24. Journées francophones des langages applicatifs, Aussois, France*, February 2013.

[JLM+09] Éric Jaeger, Olivier Levillain, Benjamin Morin, Vincent Strubel, Silicom, Amossys, and INRIA. JavaSec: Étude de l'adéquation du langage java pour le développement d'applications de sécurité. Technical report, ANSSI, 2009.

[Jon12] Richard W. M. Jones. `ocaml-bitstring`. `https://github.com/mor1/ocaml-bitstring`, 2008-2012.

[JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA*, pages 1185–1196, October 2015.

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA*, pages 333–350, August 2010.

[Kar15] Hubert Kario. Security Pitfalls: Monthly Scan Results. `https://securitypitfalls.wordpress.com/`, 2014-2015.

[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.

[Kik14] Masashi Kikuchi. How I discovered CCS Injection Vulnerability (CVE-2014-0224). `http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html`, June 2014.

[KL00] R. Khare and S. Lawrence. Upgrading to TLS Within HTTP/1.1. RFC 2817 (Proposed Standard), May 2000. Updated by RFCs 7230, 7231.

[KMMMS15] David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation. In *24th USENIX Security Symposium, Washington, D.C., USA*, pages 223–238, August 2015.

[Koi11] Sami Koivu. (Slightly) Random Broken Thoughts. `http://slightlyrandombrokenthoughts.blogspot.fr`, 2007-2011.

[Kol02] Mitja Kolšek. Session Fixation Vulnerability in Web-based Applications. `http://www.acrossecurity.com/papers/session_fixation.pdf`, 2002.

[KPR03] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking rsa-based sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany*, pages 426–440, September 2003.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[Kra01] Hugo Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA*, pages 310–331, August 2001.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA*, pages 631–648, August 2010.

[Lab15a] Amazon Web Services Labs. `s2n`: an implementation of the TLS/SSL protocols. `https://github.com/awslabs/s2n`, 2015.

[Lab15b] SSL Labs. SSL Pulse: Survey of the SSL Implementation of the Most Popular Web Sites. `https://www.trustworthyinternet.org/ssl-pulse/`, 2012-2015.

[Lan11] Adam Langley. Unfortunate current practices for HTTP over TLS. `http://www.imperialviolet.org/2011/02/04/oppractices.html`, February 2011.

[Lan12] Adam Langley. Revocation checking and Chrome's CRL. `https://www.imperialviolet.org/2012/02/05/crlsets.html`, February 2012.

[Lan13] Adam Langley. Lucky Thirteen attack on TLS CBC. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`, February 2013.

[LC03] Kyung-suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, 2003.

[LCM+16] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905 (Proposed Standard), June 2016.

[LDM13] Olivier Levillain, Hervé Debar, and Benjamin Morin. Parsifal: Writing efficient and robust binary parsers, quickly. In *2013 International Conference on Risks and Security of Internet and Systems (CRiSIS), La Rochelle, France, October 23-25, 2013*, pages 1–6, 2013.

[LÉMD12] Olivier Levillain, Arnaud Ébalard, Benjamin Morin, and Hervé Debar. One Year of SSL Internet Measurement. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA*, pages 11–20, December 2012.

[Ler09] Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning (JAR)*, 43(4):363–446, 2009.

[Ler11] Xavier Leroy. Verified squared: does critical software deserve verified tools? In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA*, pages 1–2, January 2011.

[Lev12] Olivier Levillain. SSL/TLS : état des lieux et recommandations. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC, Rennes, France*, pages 3–42, June 2012.

[Lev13] Olivier Levillain. Parsifal : écriture rapide de *parsers* robustes et efficaces. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication SSTIC, Rennes, France*, June 2013.

[Lev14a] Olivier Levillain. Mind Your Language(s): A Discussion about Languages and Security. In *High Integrity Software, Bristol, UK*, October 2014.

[Lev14b] Olivier Levillain. Parsifal : une solution pour écrire rapidement des *parsers* binaires robustes et efficaces. In *25. Journées francophones des langages applicatifs, Fréjus, France*, pages 193–194, January 2014.

[Lev14c] Olivier Levillain. Parsifal: A Pragmatic Solution to the Binary Parsing Problems. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA*, pages 191–197, May 2014.

[Lev15] Olivier Levillain. SSL/TLS, 3 ans plus tard. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC, Rennes, France*, pages 227–273, June 2015.

[LGD15] Olivier Levillain, Baptiste Gourdin, and Hervé Debar. TLS Record Protocol: Security Analysis and Defense-in-depth Countermeasures for HTTPS. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore*, pages 225–236, April 2015.

[LKHG05] Alex X. Liu, Jason M. Kovacs, Chin-Tser Huang, and Mohamed G. Gouda. A secure cookie protocol. In *Proceedings of the 14th International Conference On Computer Communications and Networks, ICCCN 2005, San Diego, California, USA*, pages 333–338, October 2005.

[LLK13] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.

[LM15] Daan Leijen and Paolo Martini. Parsec, a fast combinator parser. `http://hackage.haskell.org/package/parsec`, 2001-2015.

[LMM16] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. RFC 7918 (Informational), August 2016.

[LWdW05] Arjen K. Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding X.509 Certificates. *IACR Cryptology ePrint Archive*, 2005.

[Man15] Itsik Mantin. Attacking SSL when using RC4: Breaking SSL with a 13-year-old RC4 Weakness. *Black Hat Asia*, March 2015.

[Mar02] Moxie Marlinspike. Internet Explorer SSL Vulnerability. `http://www.thoughtcrime.org/ie-ssl-chain.txt`, 2002.

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

[Mar11] Moxie Marlinspike. BasicConstraints Back Then. `http://www.thoughtcrime.org/blog/sslsniff-anniversary-edition/`, July 2011.

[Mav07] N. Mavrogiannopoulos. Using OpenPGP Keys for Transport Layer Security (TLS) Authentication. RFC 5081 (Experimental), November 2007. Obsoleted by RFC 6091.

[MB12] D. McGrew and D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655 (Proposed Standard), July 2012.

[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.

[MG11] N. Mavrogiannopoulos and D. Gillmor. Using OpenPGP Keys for Transport Layer Security (TLS) Authentication. RFC 6091 (Informational), February 2011.

[MH99] A. Medvinsky and M. Hur. Addition of Kerberos Cipher Suites to Transport Layer Security (TLS). RFC 2712 (Proposed Standard), October 1999.

[Mic01] Microsoft. MS01-017: Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard. `https://support.microsoft.com/en-us/kb/293818`, 2001.

[Mil84] Robin Milner. A Proposal for Standard ML. In *LISP and Functional Programming*, pages 184–197, 1984.

[ML15] B. Moeller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507 (Proposed Standard), April 2015.

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

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

[MRLG15] Florian Maury, Jean-René Reinhard, Olivier Levillain, and Henri Gilbert. Format Oracles on OpenPGP. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA*, pages 220–236, April 2015.

[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.

[MT91] Robin Milner and Mads Tofte. *Commentary on standard ML*. MIT Press, 1991.

[MVN06] K. Murchison, J. Vinocur, and C. Newman. Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP). RFC 4642 (Proposed Standard), October 2006.

[MVVP12] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA*, pages 62–72, October 2012.

[MZSH15] Wilfried Mayer, Aaron Zauner, Martin Schmiedecker, and Markus Huber. No Need for Black Chambers: Testing TLS in the E-mail Ecosystem at Large. *CoRR (Computing Research Repository)*, abs/1510.08646, 2015.

[New99] C. Newman. Using TLS with IMAP, POP3 and ACAP. RFC 2595 (Proposed Standard), June 1999. Updated by RFCs 4616, 7817.

[One96] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 49, 1996.

[orb13] orbitz. functional orbitz. `http://functional-orbitz.blogspot.fr`, 2005-2013.

[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.

[Per01] John Percival. Bugtraq mailing list. Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images). `http://seclists.org/bugtraq/2001/Jun/191`, June 2001.

[Per15] Colin Percival. The HTTP 500 Solution. `http://www.daemonology.net/blog/2015-11-27-the-HTTP-500-solution.html`, November 2015.

[Pet15a] Yngve Pettersen. The POODLE has friends. `https://vivaldi.net/en-US/userblogs/entry/the-poodle-has-friends`, July 2015.

[Pet15b] Yngve Pettersen. There are more POODLEs in the forests. `https://vivaldi.net/en-US/userblogs/entry/there-are-more-poodles-in-the-forest`, July 2015.

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

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

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

[Por15] Remy Porter. The Daily WTF. `http://thedailywtf.com`, 2004-2015.

[PPSP06] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry L. Peterson. `binpac`: A `yacc` for Writing Application Protocol Parsers. In *Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference, IMC 2006, Rio de Janeriro, Brazil*, pages 289–300, October 2006.

[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.

[PR15] R. Peon and H. Ruellan. HPACK: Header Compression for HTTP/2. RFC 7541 (Proposed Standard), May 2015.

[PRS11] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea*, pages 372–389, December 2011.

[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.

[Res13] Eric Rescorla. New Handshake Flows for TLS 1.3. `https://tools.ietf.org/html/draft-rescorla-tls13-new-flows-00`, November 2013.

[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.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFCs 6286, 6608, 6793, 7606, 7607, 7705.

[RM06] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006. Obsoleted by RFC 6347, updated by RFCs 5746, 7507.

[RM12] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. Updated by RFCs 7507, 7905.

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

[RRDO10] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[SAH11] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard), March 2011.

[SCM08] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), August 2008.

[SIR02] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the fluhrer, mantin, and shamir attack to break WEP. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2002, San Diego, California, USA*, February 2002.

[SKP15] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision on full SHA-1. *IACR Cryptology ePrint Archive*, 2015.

[SMA+13] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013.

[Som16] Juraj Somorovsky. Curious Padding oracle in OpenSSL (CVE-2016-2107). `http://web-in-security.blogspot.fr/2016/05/curious-padding-oracle-in-openssl-cve.html`, May 2016.

[SSA+09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA*, pages 55–69, August 2009.

[Sti15] Victor Stinner. Hachoir. `https://bitbucket.org/haypo/hachoir/wiki/Home`, 2007-2015.

[STW12] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520 (Proposed Standard), February 2012.

[SZET06] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507 (Proposed Standard), May 2006. Obsoleted by RFC 5077.

[SZET08] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), January 2008.

[tic09] ticalc.org. All ti signing keys factored. `http://www.ticalc.org/archives/news/articles/14/145/145273.html`, 2009.

[TP11] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011.

[Tru12] Trustwave. Clarifying The Trustwave CA Policy Update. `http://blog.spiderlabs.com/2012/02/clarifying-the-trustwave-ca-policy-update.html`, February 2012.

[TWMP07] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. RFC 5054 (Informational), November 2007.

[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.

[VP15] Mathy Vanhoef and Frank Piessens. All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS. In *24th USENIX Security Symposium, Washington, D.C., USA*, pages 97–112, August 2015.

[WG16] Mike West and Mark Goodwin. Same-site Cookies. `https://tools.ietf.org/html/draft-west-first-party-cookies-06`, January 2016.

[WL93] Pierre Weis and Xavier Leroy. *Le langage Caml*. InterEditions, 1993.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce, WOEC'96, Oakland, California, USA*. USENIX Association, 1996.

[WY05] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark*, pages 19–35, May 2005.

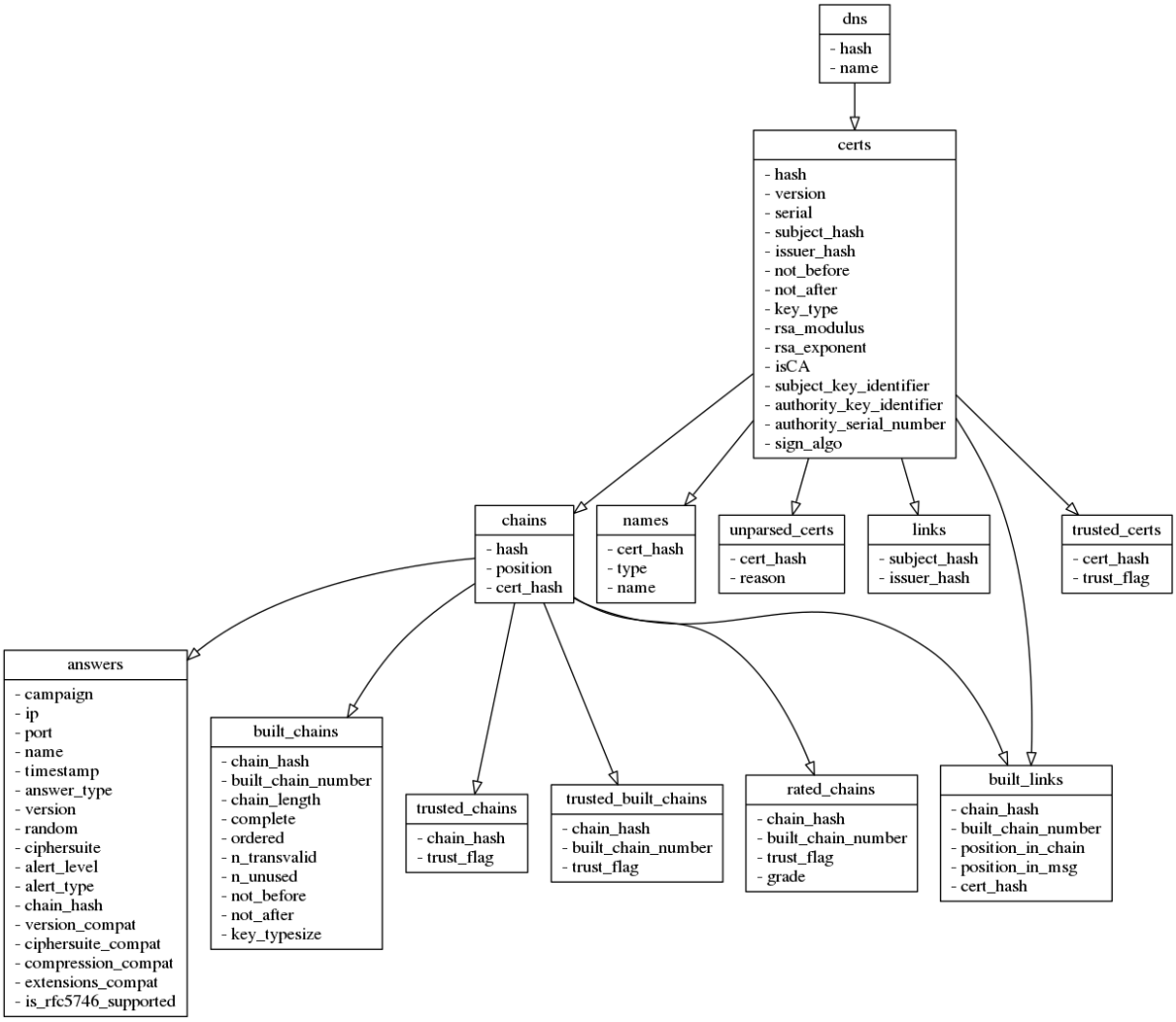
[WZKS13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA*, pages 260–275, November 2013.

[Zet12] Kim Zetter. How a Google Headhunter's E-Mail Unraveled a Massive Net Security Hole. `http://www.wired.com/2012/10/dkim-vulnerability-widespread/`, October 2012.

[ZK16] Dionysios Zindros and Dimitris Karakostas. Practical New Developments in the BREACH Attack. *Black Hat Asia*, April 2016.

Appendix A

`concerto` database schema



Appendix B

Detailed tables of the studied campaigns

B.1 Global statistics on the campaigns

In chapter 5, we presented the different campaigns. In this appendix, the same figures are presented, but with more details. First, table B.1 describes all the studied campaigns and the associated stimuli.

Campaign		Stimulus	Comments
Date	**Id**		
2010/07	000	TLS1.0	Small difference in the extensions sent
2010/08	001	EFF	*EFF campaign*
2010/12	002	EFF	*EFF campaign*
2011/07	003	TLS1.0-NoExt	
	004	DHE	
	005	TLS1.0	
	006	EC	
	007	SSL2	
	008	SSL2-TLS1.0	
	009	TLS1.2-modern	
2014/03	101	SSL2	
	102	SSL2-TLS1.0	
	103	TLS1.0-NoExt	
	104	TLS1.2-modern	
	105	TLS1.2	
	106	DHE	
	107	DHE	Stimulus sent 5 seconds after the previous one
	108	EC	
	109	EC-explicit	
	110	TLS1.0	`client_random = 00..00`
	111	TLS1.0	`client_random = FF..FF`
2015/08	200	TLS1.2	The stimulus was sent by ZGrab
	201	TLS1.2	The stimulus was sent by ZGrab (Top Alexa 1M)

Table B.1: Stimuli sent for each campaign.

Then, we sort the answers received for each campaign into several categories. Table B.2 shows global results for the ten campaigns. It partitions the answers obtained into the following classes.

Non-TLS answers are refined in *empty* and *non-empty* answers (HTTP headers, SSH banners or syntactically invalid TLS messages for example). In fact, it seems common to find non-HTTPS service listening on port 443.

SSL/TLS answers can be *Alerts* (the parameters proposed by the client did not suit the server), or *Handshake messages*.

Campaign Id	Campaign Stimulus	IPs with TCP/443	Non-TLS answers Empty	Non-TLS answers Other	SSL/TLS answers Alerts	SSL/TLS answers Handshake
000	TLS1.0	21,342,205	**40.62 %**	**13.21 %**	**0.80 %**	**45.37 %**
001	EFF	15,665,414	**0.00 %**	**26.74 %**	**0.10 %**	**73.08 %**
002	EFF	7,777,511	**0.00 %**	**0.92 %**	**0.01 %**	**99.07 %**
003	TLS1.0-NoExt	26,218,647	**47.29 %**	**9.35 %**	**0.20 %**	**43.15 %**
004	DHE	26,218,638	**62.80 %**	**3.20 %**	**16.72 %**	**17.29 %**
005	TLS1.0	26,218,630	**47.37 %**	**9.32 %**	**0.35 %**	**42.96 %**
006	EC	26,218,635	**54.27 %**	**9.38 %**	**33.16 %**	**3.19 %**
007	SSL2	26,218,653	**70.24 %**	**7.11 %**	**1.31 %**	**20.86 %**
008	SSL2-TLS1.0	26,218,653	**47.25 %**	**9.35 %**	**0.07 %**	**43.31 %**
009	TLS1.2-modern	26,218,638	**54.06 %**	**9.37 %**	**3.53 %**	**33.04 %**
101	SSL2	40,127,785	**62.13 %**	**4.89 %**	**3.07 %**	**29.43 %**
102	SSL2-TLS1.0	40,127,785	**17.23 %**	**10.43 %**	**0.19 %**	**72.14 %**
103	TLS1.0-NoExt	40,126,225	**17.11 %**	**10.43 %**	**0.28 %**	**72.18 %**
104	TLS1.2-modern	40,126,225	**20.99 %**	**10.42 %**	**5.61 %**	**62.98 %**
105	TLS1.2	40,126,225	**17.43 %**	**10.34 %**	**0.42 %**	**71.82 %**
106 & 107	DHE	40,126,225	**32.02 %**	**4.69 %**	**40.68 %**	**22.61 %**
108	EC	40,126,225	**21.20 %**	**10.39 %**	**59.92 %**	**8.49 %**
109	EC-explicit	40,126,225	**26.46 %**	**10.36 %**	**61.66 %**	**1.52 %**
110 & 111	TLS1.0	40,126,225	**17.33 %**	**10.31 %**	**0.35 %**	**72.01 %**
200	`scans.io`	46,231,300	**26.27 %**		**0.01 %**	**73.72 %**
201	`scans.io` (TA 1M)	684,365	**47.79 %**		**0.20 %**	**52.01 %**

Table B.2: Distribution of the answers collected for each campaign. The percentages are computed over the total number of IPs where we found a 443 open port (second column).

B.2 Analysis of TLS parameters

Tables B.3 and B.5 contain the detailed results concerning the versions and ciphersuites chosen by servers across the different campaigns, for each stimulus we sent. They respectively correspond to figures 5.2 and 5.3 in chapter 5.

The following table contains the list of TLS ciphersuites by category. This list does not include fixed or unauthenticated Diffie-Hellman, PSK, SRP, Kerberos and IDEA ciphersuites.

Ciphersuite Id	Ciphersuite Name	Compatible versions Min.	Compatible versions Max.	Export	Forward Secrecy
Strong					
`c02b`	`TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`	TLS 1.2	-	No	Yes
`c02c`	`TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`	TLS 1.2	-	No	Yes
`c02f`	`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`	TLS 1.2	-	No	Yes
`c030`	`TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`	TLS 1.2	-	No	Yes
`c05c`	`TLS_ECDHE_ECDSA_WITH_ARIA_128_GCM_SHA256`	TLS 1.2	-	No	Yes
`c05d`	`TLS_ECDHE_ECDSA_WITH_ARIA_256_GCM_SHA384`	TLS 1.2	-	No	Yes
`c060`	`TLS_ECDHE_RSA_WITH_ARIA_128_GCM_SHA256`	TLS 1.2	-	No	Yes
`c061`	`TLS_ECDHE_RSA_WITH_ARIA_256_GCM_SHA384`	TLS 1.2	-	No	Yes
`c086`	`TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256`	TLS 1.2	-	No	Yes
`c087`	`TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384`	TLS 1.2	-	No	Yes
`c08a`	`TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256`	TLS 1.2	-	No	Yes
`c08b`	`TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384`	TLS 1.2	-	No	Yes
`cc13`	`TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256`	TLS 1.2	-	No	Yes

Ciphersuite		Compatible versions		Export	Forward
Id	Name	Min.	Max.		Secrecy
cc14	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	TLS 1.2	-	No	Yes
cca8	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	TLS 1.2	-	No	Yes
cca9	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	TLS 1.2	-	No	Yes
OK / Acceptable					
000a	TLS_RSA_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	No
0016	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	Yes
002f	TLS_RSA_WITH_AES_128_CBC_SHA	SSLv3	-	No	No
0033	TLS_DHE_RSA_WITH_AES_128_CBC_SHA	SSLv3	-	No	Yes
0035	TLS_RSA_WITH_AES_256_CBC_SHA	SSLv3	-	No	No
0039	TLS_DHE_RSA_WITH_AES_256_CBC_SHA	SSLv3	-	No	Yes
003c	TLS_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	-	No	No
003d	TLS_RSA_WITH_AES_256_CBC_SHA256	TLS 1.2	-	No	No
0041	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	SSLv3	-	No	No
0045	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	SSLv3	-	No	Yes
0067	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	-	No	Yes
006b	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	TLS 1.2	-	No	Yes
0084	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	SSLv3	-	No	No
0088	TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	SSLv3	-	No	Yes
0096	TLS_RSA_WITH_SEED_CBC_SHA	SSLv3	-	No	No
009a	TLS_DHE_RSA_WITH_SEED_CBC_SHA	SSLv3	-	No	Yes
009c	TLS_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2	-	No	No
009d	TLS_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2	-	No	No
009e	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2	-	No	Yes
009f	TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2	-	No	Yes
00ba	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	TLS 1.2	-	No	No
00be	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
00c0	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	TLS 1.2	-	No	No
00c4	TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	TLS 1.2	-	No	Yes
c008	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	Yes
c009	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	SSLv3	-	No	Yes
c00a	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	SSLv3	-	No	Yes
c012	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	Yes
c013	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	SSLv3	-	No	Yes
c014	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	SSLv3	-	No	Yes
c023	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS 1.2	-	No	Yes
c024	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS 1.2	-	No	Yes
c027	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	-	No	Yes
c028	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS 1.2	-	No	Yes
c03c	TLS_RSA_WITH_ARIA_128_CBC_SHA256	TLS 1.2	-	No	No
c03d	TLS_RSA_WITH_ARIA_256_CBC_SHA384	TLS 1.2	-	No	No
c044	TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
c045	TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384	TLS 1.2	-	No	Yes
c048	TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
c049	TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384	TLS 1.2	-	No	Yes
c04c	TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
c04d	TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384	TLS 1.2	-	No	Yes
c050	TLS_RSA_WITH_ARIA_128_GCM_SHA256	TLS 1.2	-	No	No
c051	TLS_RSA_WITH_ARIA_256_GCM_SHA384	TLS 1.2	-	No	No
c052	TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256	TLS 1.2	-	No	Yes
c053	TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384	TLS 1.2	-	No	Yes
c072	TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
c073	TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384	TLS 1.2	-	No	Yes
c076	TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
c077	TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384	TLS 1.2	-	No	Yes
c07a	TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256	TLS 1.2	-	No	No
c07b	TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384	TLS 1.2	-	No	No
c07c	TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	TLS 1.2	-	No	Yes
c07d	TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	TLS 1.2	-	No	Yes
c09c	TLS_RSA_WITH_AES_128_CCM	TLS 1.2	-	No	No
c09d	TLS_RSA_WITH_AES_256_CCM	TLS 1.2	-	No	No
c09e	TLS_DHE_RSA_WITH_AES_128_CCM	TLS 1.2	-	No	Yes
c09f	TLS_DHE_RSA_WITH_AES_256_CCM	TLS 1.2	-	No	Yes
c0a0	TLS_RSA_WITH_AES_128_CCM_8	TLS 1.2	-	No	No
c0a1	TLS_RSA_WITH_AES_256_CCM_8	TLS 1.2	-	No	No
c0a2	TLS_DHE_RSA_WITH_AES_128_CCM_8	TLS 1.2	-	No	Yes
c0a3	TLS_DHE_RSA_WITH_AES_256_CCM_8	TLS 1.2	-	No	Yes
cc15	TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	TLS 1.2	-	No	Yes
ccaa	TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	TLS 1.2	-	No	Yes
feff	SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	No

Ciphersuite		Compatible versions		Export	Forward
Id	Name	Min.	Max.		Secrecy
ffe0	SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	No
Weak					
0004	TLS_RSA_WITH_RC4_128_MD5	SSLv3	-	No	No
0005	TLS_RSA_WITH_RC4_128_SHA	SSLv3	-	No	No
0013	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	SSLv3	-	No	Yes
0032	TLS_DHE_DSS_WITH_AES_128_CBC_SHA	SSLv3	-	No	Yes
0038	TLS_DHE_DSS_WITH_AES_256_CBC_SHA	SSLv3	-	No	Yes
0040	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	TLS 1.2	-	No	Yes
0044	TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	SSLv3	-	No	Yes
006a	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	TLS 1.2	-	No	Yes
0087	TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	SSLv3	-	No	Yes
0099	TLS_DHE_DSS_WITH_SEED_CBC_SHA	SSLv3	-	No	Yes
00a2	TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	TLS 1.2	-	No	Yes
00a3	TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	TLS 1.2	-	No	Yes
00bd	TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
00c3	TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256	TLS 1.2	-	No	Yes
c007	TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	SSLv3	-	No	Yes
c011	TLS_ECDHE_RSA_WITH_RC4_128_SHA	SSLv3	-	No	Yes
c042	TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256	TLS 1.2	-	No	Yes
c043	TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384	TLS 1.2	-	No	Yes
c056	TLS_DHE_DSS_WITH_ARIA_128_GCM_SHA256	TLS 1.2	-	No	Yes
c057	TLS_DHE_DSS_WITH_ARIA_256_GCM_SHA384	TLS 1.2	-	No	Yes
c080	TLS_DHE_DSS_WITH_CAMELLIA_128_GCM_SHA256	TLS 1.2	-	No	Yes
c081	TLS_DHE_DSS_WITH_CAMELLIA_256_GCM_SHA384	TLS 1.2	-	No	Yes
Broken					
0000	TLS_NULL_WITH_NULL_NULL	SSLv3	-	No	No
0001	TLS_RSA_WITH_NULL_MD5	SSLv3	-	No	No
0002	TLS_RSA_WITH_NULL_SHA	SSLv3	-	No	No
0003	TLS_RSA_EXPORT_WITH_RC4_40_MD5	SSLv3	TLS 1.0	Yes	No
0006	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	SSLv3	TLS 1.0	Yes	No
0008	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	SSLv3	TLS 1.0	Yes	No
0009	TLS_RSA_WITH_DES_CBC_SHA	SSLv3	-	No	No
0011	TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	SSLv3	TLS 1.0	Yes	Yes
0012	TLS_DHE_DSS_WITH_DES_CBC_SHA	SSLv3	-	No	Yes
0014	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	SSLv3	TLS 1.0	Yes	Yes
0015	TLS_DHE_RSA_WITH_DES_CBC_SHA	SSLv3	-	No	Yes
003b	TLS_RSA_WITH_NULL_SHA256	TLS 1.2	-	No	No
c006	TLS_ECDHE_ECDSA_WITH_NULL_SHA	SSLv3	-	No	Yes
c010	TLS_ECDHE_RSA_WITH_NULL_SHA	SSLv3	-	No	Yes
fefe	SSL_RSA_FIPS_WITH_DES_CBC_SHA	SSLv3	-	No	No
ffe1	SSL_RSA_FIPS_WITH_DES_CBC_SHA	SSLv3	-	No	No
010080	SSL_CK_RC4_128_WITH_MD5	SSLv2	SSLv2	No	No
020080	SSL_CK_RC4_128_EXPORT40_WITH_MD5	SSLv2	SSLv2	Yes	No
030080	SSL_CK_RC2_128_CBC_WITH_MD5	SSLv2	SSLv2	No	No
040080	SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	SSLv2	SSLv2	Yes	No
060040	SSL_CK_DES_64_CBC_WITH_MD5	SSLv2	SSLv2	No	No
0700c0	SSL_CK_DES_192_EDE3_CBC_WITH_MD5	SSLv2	SSLv2	No	No

		TLS		**Trusted**		**EV**	
000	TLS1.0	SSLv3	4.5 %	SSLv3	0.8 %	SSLv3	0.6 %
		TLS 1.0	95.5 %	TLS 1.0	99.2 %	TLS 1.0	99.4 %
001	EFF	SSLv2	0.1 %	SSLv3	0.9 %	SSLv3	0.6 %
		SSLv3	4.5 %	TLS 1.0	99.1 %	TLS 1.0	99.4 %
		TLS 1.0	95.4 %				
002	EFF	SSLv3	4.1 %	SSLv3	0.9 %	SSLv3	0.8 %
		TLS 1.0	95.9 %	TLS 1.0	99.1 %	TLS 1.0	99.2 %
003	TLS1.0-NoExt	SSLv3	3.6 %	SSLv3	0.7 %	SSLv3	0.6 %
		TLS 1.0	96.4 %	TLS 1.0	99.3 %	TLS 1.0	99.4 %
004	*DHE*	*SSLv3*	*2.6 %*	*SSLv3*	*0.2 %*	*SSLv3*	*0.2 %*
		TLS 1.0	*97.4 %*	*TLS 1.0*	*99.8 %*	*TLS 1.0*	*99.8 %*
005	TLS1.0	SSLv3	3.6 %	SSLv3	0.6 %	SSLv3	0.6 %
		TLS 1.0	96.4 %	TLS 1.0	99.4 %	TLS 1.0	99.4 %
006	*EC*	*SSLv3*	*15.8 %*	*SSLv3*	*0.3 %*	*TLS 1.0*	*100.0 %*
		TLS 1.0	*84.2 %*	*TLS 1.0*	*99.7 %*		
007	*SSL2*	*SSLv2*	*99.9 %*	*SSLv2*	*100.0 %*	*SSLv2*	*100.0 %*
008	SSL2-TLS1.0	SSLv3	3.9 %	SSLv3	0.8 %	SSLv3	0.8 %
		TLS 1.0	96.0 %	TLS 1.0	99.1 %	TLS 1.0	99.2 %
009	*TLS1.2-modern*	*SSLv3*	*1.4 %*	*SSLv3*	*0.3 %*	*SSLv3*	*0.2 %*
		TLS 1.0	*98.5 %*	*TLS 1.0*	*99.5 %*	*TLS 1.0*	*99.5 %*
		TLS 1.1	*0.1 %*	*TLS 1.1*	*0.2 %*	*TLS 1.1*	*0.2 %*
101	*SSL2*	*SSLv2*	*99.9 %*	*SSLv2*	*100.0 %*	*SSLv2*	*100.0 %*
102	SSL2-TLS1.0	SSLv3	3.2 %	SSLv3	0.4 %	SSLv3	0.6 %
		TLS 1.0	96.8 %	TLS 1.0	99.6 %	TLS 1.0	99.4 %
103	TLS1.0-NoExt	SSLv3	3.2 %	SSLv3	0.4 %	SSLv3	0.5 %
		TLS 1.0	96.8 %	TLS 1.0	99.6 %	TLS 1.0	99.5 %
104	TLS1.2-modern	SSLv3	0.9 %	SSLv3	0.2 %	SSLv3	0.3 %
		TLS 1.0	65.6 %	TLS 1.0	42.8 %	TLS 1.0	57.5 %
		TLS 1.1	0.5 %	TLS 1.1	0.5 %	TLS 1.1	0.8 %
		TLS 1.2	33.0 %	TLS 1.2	56.5 %	TLS 1.2	41.4 %
105	TLS1.2	SSLv3	3.2 %	SSLv3	0.3 %	SSLv3	0.5 %
		TLS 1.0	66.7 %	TLS 1.0	46.4 %	TLS 1.0	61.8 %
		TLS 1.1	0.5 %	TLS 1.1	0.5 %	TLS 1.1	0.7 %
		TLS 1.2	29.6 %	TLS 1.2	52.8 %	TLS 1.2	37.0 %
106	*DHE*	*SSLv3*	*2.3 %*	*SSLv3*	*0.3 %*	*SSLv3*	*0.3 %*
107		*TLS 1.0*	*97.7 %*	*TLS 1.0*	*99.7 %*	*TLS 1.0*	*99.7 %*
108	*EC*	*SSLv3*	*6.6 %*	*SSLv3*	*0.1 %*	*TLS 1.0*	*99.9 %*
		TLS 1.0	*93.4 %*	*TLS 1.0*	*99.9 %*		
109	*EC-explicit*	*SSLv3*	*36.9 %*	*SSLv3*	*0.8 %*	*SSLv3*	*0.2 %*
		TLS 1.0	*63.0 %*	*TLS 1.0*	*99.2 %*	*TLS 1.0*	*99.8 %*
110	TLS1.0	SSLv3	3.1 %	SSLv3	0.4 %	SSLv3	0.5 %
111		TLS 1.0	96.9 %	TLS 1.0	99.6 %	TLS 1.0	99.5 %
200	`scans.io`	SSLv3	2.3 %	TLS 1.0	27.1 %	TLS 1.0	30.9 %
		TLS 1.0	48.9 %	TLS 1.1	0.2 %	TLS 1.1	0.3 %
		TLS 1.1	1.8 %	TLS 1.2	72.7 %	TLS 1.2	68.7 %
		TLS 1.2	47.1 %				
201	`scans.io` (TA 1M)	TLS 1.0	24.0 %	TLS 1.0	22.3 %	TLS 1.0	19.7 %
		TLS 1.1	0.2 %	TLS 1.1	0.3 %	TLS 1.1	0.3 %
		TLS 1.2	75.8 %	TLS 1.2	77.4 %	TLS 1.2	80.0 %

Table B.3: Distribution of the TLS versions chosen by the servers for each campaign and each subset. Campaigns represented in italic correspond to stimuli that produced significantly less answers than standard stimuli. Percentages below 0.1 % are not represented

		TLS		**Trusted**		**EV**	
000	TLS1.0	Weak	34.1 %	Weak	40.6 %	Weak	43.7 %
		OK	65.9 %	OK	59.4 %	OK	56.3 %
001	EFF	Broken	0.2 %	Weak	34.3 %	Weak	36.4 %
		Weak	25.0 %	OK	65.7 %	OK	63.6 %
		OK	74.8 %				
002	EFF	Weak	25.9 %	Weak	34.6 %	Weak	39.5 %
		OK	74.0 %	OK	65.3 %	OK	60.5 %
003	TLS1.0-NoExt	Weak	26.4 %	Weak	31.5 %	Weak	33.4 %
		OK	73.6 %	OK	68.5 %	OK	66.6 %
004	*DHE*	*Not proposed*	*2.4 %*	*OK*	*99.9 %*	*OK*	*100.0 %*
		Weak	*0.2 %*				
		OK	*97.4 %*				
005	TLS1.0	Weak	26.5 %	Weak	31.6 %	Weak	33.4 %
		OK	73.5 %	OK	68.4 %	OK	66.6 %
006	*EC*	*Not proposed*	*17.0 %*	*Not proposed*	*1.2 %*	*Not proposed*	*0.7 %*
		Weak	*0.1 %*	*Weak*	*0.2 %*	*Weak*	*0.2 %*
		OK	*82.9 %*	*OK*	*98.6 %*	*OK*	*99.2 %*
007	*SSL2*	*Broken*	*99.9 %*	*Broken*	*100.0 %*	*Broken*	*100.0 %*
008	SSL2-TLS1.0	Broken	0.1 %	Weak	29.6 %	Weak	32.0 %
		Weak	20.5 %	OK	70.4 %	OK	68.0 %
		OK	79.3 %				
009	*TLS1.2-modern*	*Not proposed*	*1.3 %*	*OK*	*100.0 %*	*OK*	*100.0 %*
		OK	*98.7 %*				
101	*SSL2*	*Not proposed*	*0.1 %*	*Broken*	*100.0 %*	*Broken*	*100.0 %*
		Broken	*99.9 %*				
102	SSL2-TLS1.0	Weak	16.3 %	Weak	21.3 %	Weak	30.9 %
		OK	83.6 %	OK	78.6 %	OK	69.1 %
103	TLS1.0-NoExt	Weak	20.0 %	Weak	21.9 %	Weak	31.7 %
		OK	79.9 %	OK	78.1 %	OK	68.3 %
104	TLS1.2-modern	Not proposed	0.8 %	OK	98.2 %	OK	97.7 %
		OK	98.2 %	Strong	1.8 %	Strong	2.3 %
		Strong	1.0 %				
105	TLS1.2	Weak	19.7 %	Weak	21.1 %	Weak	31.2 %
		OK	80.3 %	OK	78.8 %	OK	68.8 %
106	*DHE*	*Not proposed*	*2.4 %*	*OK*	*99.9 %*	*OK*	*99.9 %*
107		*OK*	*97.4 %*				
108	*EC*	*Not proposed*	*8.4 %*	*Not proposed*	*0.2 %*	*Not proposed*	*0.1 %*
		Weak	*6.2 %*	*Weak*	*8.8 %*	*Weak*	*7.2 %*
		OK	*85.4 %*	*OK*	*91.0 %*	*OK*	*92.7 %*
109	*EC-explicit*	*Not proposed*	*47.1 %*	*Not proposed*	*2.2 %*	*Not proposed*	*1.7 %*
		Weak	*8.6 %*	*Weak*	*19.6 %*	*Weak*	*19.6 %*
		OK	*44.3 %*	*OK*	*78.1 %*	*OK*	*78.7 %*
110	TLS1.0	Weak	20.0 %	Weak	21.9 %	Weak	31.7 %
111		OK	80.0 %	OK	78.1 %	OK	68.3 %
200	`scans.io`	Weak	10.5 %	Weak	8.2 %	Weak	12.6 %
		OK	59.4 %	OK	43.2 %	OK	48.0 %
		Strong	30.1 %	Strong	48.6 %	Strong	39.4 %
201	`scans.io` (TA 1M)	Weak	6.2 %	Weak	6.8 %	Weak	7.3 %
		OK	45.7 %	OK	45.1 %	OK	45.5 %
		Strong	48.1 %	Strong	48.1 %	Strong	47.2 %

Table B.5: Distribution of the ciphersuites chosen by the servers for each campaign and each subset. Percentages below 0.1 % are not represented. The ciphersuites are grouped into categories: Strong, OK, Weak, Broken and the suites that were *not proposed* by the client.

Appendix C

`parsifal`: tutorial and references

Chapter 6 presented Parsifal, a generic framework to quickly write efficient and robust binary parsers. The present appendix presents additional resources about the project: tutorials presenting how to write TAR and PNG simple parsers, and the language grammar.

C.1 Step-by-step case study: the TAR archive format

Parsifal is publicly available under an open source license since June 2013 (`https://github.com/ANSSI-FR/parsifal`). The git repository contains examples of step-by-step code description concerning TAR archives, the DNS protocol, PNG images and PKCS#10 CSR (Certificate Signing Request).

To illustrate how to use the different constructions offered by Parsifal, this section describes how to handle a relatively simple format, TAR. It is a popular archive format in Unix systems. TAR allows to aggregate several files into one big file, which can then be compressed. A TAR archive is composed of several entries, each describing a file.

Enumerations

Each entry is defined by a file type. This type is encoded with one character; the possible values are given in the following table:

Char	Description
`<NUL>, '0'`	regular file
`'1'`	hard link
`'2'`	symbolic link
`'3'`	char device
`'4'`	block device
`'5'`	directory
`'6'`	FIFO
`'7'`	contiguous file

To describe such a value with Parsifal, we use the `enum` keyword, that creates types similar to C `enum`s, but the resulting values are statically typed. Thus, the following declaration:

```
enum file_type (8, UnknownVal UnknownFileType) =
  | 0 -> NormalFile
  | 0x30 -> NormalFile
  | 0x31 -> HardLink
  | 0x32 -> SymbolicLink
    ...
  | 0x37 -> ContiguousFile
```

will generate the following code:

```
type file_type =
  | NormalFile | HardLink | SymbolicLink | ...
  | UnknownFileType of int
```

It will also generate the `val parse_file_type : string_input -> file_type` function.

The arguments of the `enum` line respectively correspond to the size of the underlying integer (`8` in this example), and to the expected behaviour of the parser against unknown values (`UnknownVal` adds another constructor to handle the unexpected cases, but it is also possible to throw an exception by using the `Exception` parameter).

Structures

Each TAR entry starts with a header whose fields depend of the format version; the following table presents two such versions: the original specification (the TAR column) and a more recent version (the `ustar` column).

Offset (bytes)	Length (bytes)	Description	
		TAR	`ustar`
0	100	File name	
100	8	Permissions	
108	8	UID	
116	8	GID	
124	12	File size	
136	12	*Timestamp* of the last modification	
148	8	Header checksum	
156	1	Link indicator	File type
157	100	Name of the linked file	
257	6	-	`"ustar"` magic value
263	2	-	`ustar` version ("00")
265	32	-	Owner user name
297	32	-	Owner group name
329	8	-	Major device number
337	8	-	Minor device number
345	155	-	Prefix

It can be noted that the previously defined `file_type` enumeration is present in the header at offset 156. Moreover, the header is contained in a 512-byte "block". To define the TAR header in Parsifal, we simply use the `struct` keyword[1]:

```
struct tar_header = {
  file_name : string(100);
  ...
  file_type : file_type;
  linked_file : string(100);
  ustar_magic : magic("ustar\x0000");
  ...
  filename_prefix : string(155);
  hdr_padding : binstring(12);
}
```

[1] In addition to `file_type` and to basic $\mathcal{PT}$ypes such as char strings, the structure also uses the `magic` $\mathcal{PT}$ype (which is not related to OCaml's unsafe `Obj.magic` function) to define a magic number.

Unions

However, this structure requires the `ustar` fields to be defined, which will lead to parsing errors if the magic string is absent. To allow our parser to handle both versions of the format, we can split the structure in two chunks: the common part (everything before the `ustar_magic` field) and the `ustar`-specific part (which may only contain null bytes for the original TAR format).

To this aim, we must first parse the first fields, read the magic number, and handle the remaining bytes depending of the read value. The keyword allowing to parse different things in function of a discriminating value is `union`:

```
struct ustar_header = {
  owner_user : string(32);
  owner_group : string(32);
  ...
  filename_prefix : string(155);
  hdr_padding : binstring(12);
}

union additional_header (NoMoreHeader of binstring(247)) =
  | "ustar\x000" -> UStarHeader of ustar_header
  | ... (* other versions of TAR headers *)

struct tar_header = {
  file_name   : string(100);
  ...
  file_type   : file_type;
  linked_file : string(100);
  hdr_version : string(8);
  hdr_extra   : additional_header (hdr_version);
}
```

If `hdr_version` contains the `ustar` magic number, `hdr_extra` will be parsed as a `ustar_header` structure. If no magic number is recognised, `hdr_extra` will simply consists in a 247-byte long string, to pad the header up to 512 bytes.

A custom $\mathcal{PT}$ype

TAR stores integer values in a peculiar way: they are stored as a char string representing an octal value (the corresponding string must end with a null or a space character). In the above examples, we simply used classical strings to handle the TAR integer fields, which is not very practical, since the intended value of these fields is an integer.

To improve our parser, we can define a new OCaml type, as well as the required associated functions. We would thus get a custom $\mathcal{PT}$ype to replace the raw char strings. The following code presents how to write such a $\mathcal{PT}$ype called `tar_numstring` to handle these fields properly:

```
type tar_numstring = int

let parse_tar_numstring len input =
  let octal_value = parse_string (len - 1) input in
  drop_bytes 1 input;
  try int_of_string ("0o" ^ octal_value)
  with _ -> raise (ParsingException (CustomException "int_of_string",
    _h_of_si input))

let dump_tar_numstring len buf v =
  bprintf buf "%*.*o\x00" len len v

let value_of_tar_numstring v = VSimpleInt v
```

Next, it is possible to replace all the relevant `string(n)` fields with `tar_numtring(0)` ones.

Wrapping it up

Finally, it is simple to add a new structure to handle a whole entry. The following snippet allows to write a simple tool to list tar entries (where the file content must be padded to 512-byte blocks) from a `test.tar` file:

```
struct tar_entry =
{
  header : tar_header;
  file_content : binstring(header.file_size);
  file_padding : binstring(512 - (header.file_size mod 512))
}


let rec handle_entry input =
  let entry = parse_tar_entry input in
  print_endline (print_value (value_of_tar_header entry.header));
  handle_entry input

let _ =
  let input = string_input_of_filename "test.tar" in
  handle_entry input
```

C.2 Step-by-step case study: a PNG parser

This section briefly describes how to write a simple PNG parser. A PNG image is composed of a magic number (`"\x89PNG\r\n\x1a\n"`) followed by a list of chunks, which are described in the following table (see [Bou97] for the format specification).

Offset	Field	Size	Type
0	Chunk size `sz`	4	Big-endian integer
4	Chunk type	4	ASCII String
8	Chunk data	`sz`	Chunk-dependent
8 + `sz`	CRC	4	CRC 32

This first definition of the PNG format translates into the following code in `parsifal`:

```
struct png_chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  chunk_data : binstring(chunk_size);
  chunk_crc : uint32;
}

struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  chunks : list of png_chunk;
}
```

The first `struct` definition describes what a chunk is, and the `png_file` one what a PNG file is. Since `parsifal` automatically generates the associated `parse`, `dump` and `value_of` functions, a complete tool extracting PNG chunks can be written with the following lines:

```
let input = string_input_of_filename Sys.argv.(1) in
let png_file = parse_png_file input in
print_endline (print_value (value_of_png_file png_file))
```

Here is the result of the compiled programs on a PNG file:

```
value {
  png_magic: 89504e470d0a1a0a (8 bytes)
  chunks {
    chunks[0] {
      chunk_size: 13 (0x0000000d)
      chunk_type: "IHDR" (4 bytes)
      chunk_data: 000000140000001604030000000 (13 bytes)
      chunk_crc: 846176565 (0x326fa135)
    }
    chunks[1] {
      chunk_size: 15 (0x0000000f)
      chunk_type: "PLTE" (4 bytes)
      chunk_data: ccffffffcc99996633333333000000 (15 bytes)
      chunk_crc: 1128124197 (0x433dcf25)
    }
    chunks[2] {
      chunk_size: 1 (0x00000001)
      chunk_type: "bKGD" (4 bytes)
      chunk_data: 04 (1 bytes)
      chunk_crc: 2406013265 (0x8f68d951)
    }
    chunks[3] {
      chunk_size: 77 (0x0000004d)
      chunk_type: "IDAT" (4 bytes)
      chunk_data: 78da63602005b8000184c5220804... (77 bytes)
      chunk_crc: 466798482 (0x1bd2c792)
    }
    chunks[4] {
      chunk_size: 86 (0x00000056)
      chunk_type: "tEXt" (4 bytes)
      chunk_data: 436f6d6d656e7400546869732061... (86 bytes)
      chunk_crc: 1290335428 (0x4ce8f4c4)
    }
    chunks[5] {
      chunk_size: 0 (0x00000000)
      chunk_type: "IEND" (4 bytes)
      chunk_data: "" (0 byte)
      chunk_crc: 2923585666 (0xae426082)
    }
  }
}
```

Of course, this is only the beginning, and one would likely want to improve the description by enriching the chunk content. This is actually a strength of our methodology: it is generally easy to describe the big picture and then to refine the parser to take into account more details.

C.2.1 Union and progressive enrichment

To illustrate how to enrich the chunk data, let us start with the image header, corresponding to the `"IHDR"` type. It contains the following structure:

```
struct image_header = {
  width : uint32;
  height : uint32;
  bit_depth : uint8;
  color_type : uint8;
```

```
  compression_method : uint8;
  filter_method : uint8;
  interlace_method : uint8;
}
```

To use this new structure when dealing with a `"IHDR"` chunk, we have to create a union $\mathcal{PT}$ype, depending on the chunk type:

```
union chunk_content [enrich] (UnparsedChunkContent) =
| "IHDR" -> ImageHeader of image_header
```

Finally, we have to rewrite the `chunk_data` field in the `png_chunk` structure:

```
struct png_chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  chunk_data : container(chunk_size) of
                   chunk_content(chunk_type);
  chunk_crc : uint32;
}
```

Enriching other chunk types should be clear from now: write the $\mathcal{PT}$ype corresponding to the chunk content, and then add a line in the `chunk_content union`. When facing an unknown chunk type, `parse_chunk_content` will produce an `UnparsedChunkContent` value containing the unparsed string.

C.2.2 $\mathcal{PC}$ontainers: a useful concept

As we began using `parsifal` to describe various file formats and network protocols, it dawned on us that it might be useful to create another concept that could be reused: $\mathcal{PC}$ontainers. Initially, the only existing containers were lists, but the notion of containers can be broader: the abstraction of a container containing a $\mathcal{PT}$ype makes it possible to automate various kinds of transformations that must be applied at parsing and/or dumping time. Here are several examples:

- encoding: hexadecimal, base64;
- compression: DEFLATE, zLib or gzip containers;
- safe parsing: some containers provide a fallback strategy when the contained $\mathcal{PT}$ype can not be parsed;
- miscellaneous checks: CRC containers are useful to check a CRC at parsing time and to generate the CRC value at dumping time.

There again, the idea is to reuse code to reduce the time spent writing the same code time and again. Here is an example of an ancillary PNG chunk called `iCCP` (Embedded Profile), which contains a null-terminated string (implemented with the standard `cstring` $\mathcal{PT}$ype) that should not exceed 80 characters, a byte field and a compressed string. Using standard `parsifal` containers, the chunk can then be described as follows:

```
struct embedded_profile = {
  name : length_constrained_container(AtMost 80) of cstring;
  compression_method : uint8;
  compress : ZLib.zlib_container of string;
}
```

C.2.3 Custom $\mathcal{PT}$ypes

Finally, when it is not possible to describe a $\mathcal{PT}$ype using keywords like `struct` or `union`, it is always possible to write a $\mathcal{PT}$ype from scratch.

Assuming it does not exist already in the standard library, here is how you could describe the null-terminated string `cstring` as a custom $\mathcal{PT}$ype. The *intended* type is a simple string, but the corresponding `parse` and `dump` functions have to be written manually:

```
type cstring = string

let rec parse_cstring input =
  let next_char = parse_char input in
  if next_char <> '\x00'
  then (String.make 1 next_char) ^ (parse_cstring input)
  else ""

let dump_cstring buf s =
  POutput.add_string buf s;
  POutput.add_char buf '\x00'
```

Another example of custom $\mathcal{PT}$ype in the PNG description is the chunk itself. With the `struct` definition presented earlier, it is possible to create and dump inconsistent chunks where the length or the CRC do not correspond to the data contained. This might be useful for fuzzing purpose, but it makes it harder to produce valid values:

```
let data_chunk = {
  chunk_size = String.length png_data;
  chunk_type = "IDAT";
  chunk_data = UnparsedChunkContent png_data;
  chunk_crc = Crc.crc32 ("IDAT" ^ png_data);
}
```

An assumed design choice in `parsifal` is to simplify the manipulation (parsing and dumping) of valid files when possible, even if it makes other use cases (like fuzzing) less accessible. The following snippets present a custom $\mathcal{PT}$ype to replace `png_chunk` (only the parse function is detailed) and how the new, simpler, way to create a chunk:

```
type png_chunk = {
  chunk_type : string;
  chunk_data : chunk_content;
}

let parse_png_chunk input =
  let size = parse_uint32 input in
  let raw_data = peek_string (size + 4) input in
  let chunk_type = parse_string 4 input in
  let data = parse_container size
               (parse_chunk_content chunk_type) input in
  let crc = parse_string 4 input in
  let computed_crc = Crc.crc32 raw_data in
  if computed_crc <> crc then failwith "Invalid␣CRC";
  { chunk_type = chunk_type; chunk_data = data }
```

```
let data_chunk = {
  chunk_type = "IDAT";
  chunk_data = UnparsedChunkContent png_data;
}
```

The resulting variable can then be easily integrated in a chunk list to produce a PNG file.

As a side note, it would have been possible to obtain the same results without custom $\mathcal{PT}$ypes, had the PNG chunk structure length also covered the type and CRC fields:

```
struct raw_alt_chunk = {
  chunk_type : string(4);
  chunk_data : chunk_content(chunk_type);
}

alias alt_chunk = container[uint32] of crc_container of raw_png_chunk
```

We would indeed prefer a strict encapsulation of the different containers: here the length-constrained container (reading the length then the content) and the CRC container (reading the content and the CRC, then checking the consistency).

C.3 Reference: Parsifal grammar

The current appendix describes Parsifal grammar, that is the rules corresponding to the constructions added to the OCaml language by our `camlp4` preprocessor.

C.3.1 Common tokens

To describe Parsifal constructions, we will use the following standard tokens:

- ⟨*ident*⟩ for a variable identifier;
- ⟨*constr*⟩ for a type constructor;
- ⟨*exception*⟩ for an exception constructor;
- ⟨*expr*⟩ for an arbitrary expression;
- ⟨*int_expr*⟩ for an expression representing an integer value;
- ⟨*int*⟩ for an integer literal.

Moreover, all Parsifal constructions can take options, which are defined by the following rules, where *LittleEndian* only relates to enumerations, while *EnrichByDefault* and *ExhaustiveChoices* are specific to unions. It is also possible to specify parameters to pass to `parse` and `dump` functions:

⟨*option*⟩ ::= LittleEndian
| EnrichByDefault
| ExhaustiveChoices
| ParseParam ⟨*ident*⟩
| DumpParam ⟨*ident*⟩

Another pervasive token in this grammar is ⟨$\mathcal{PT}$ype⟩, which can either be an OCaml type, provided that the corresponding `parse` and `dump` functions exist, or a PContainer encapsulating a $\mathcal{PT}$ype:

⟨*PType*⟩ ::= ⟨*ident*⟩ [(⟨*params*⟩)]
| ⟨*ident*⟩ [(⟨*params*⟩)] of ⟨*PType*⟩

Finally, for the sake of simplicity, a token named ⟨*foos*⟩ should be interpreted as a list of ⟨*foo*⟩ tokens. The corresponding rules have been omitted.

C.3.2 Enumerations

An enumeration is characterised by the size in bits of the underlying integers, the way to behave when facing an unknown value (throw an exception or use a fallback constructor), and a list of values (the enumeration cases):

⟨*enum_def*⟩ ::= enum ⟨*ident*⟩ [⟨*options*⟩] (⟨*int*⟩, ⟨*e_undef*⟩) = ⟨*e_cases*⟩

⟨*e_case*⟩ ::= ⟨*int_expr*⟩ → ⟨*constr*⟩
| ⟨*int_expr*⟩ .. ⟨*int_expr*⟩ → ⟨*constr*⟩

⟨*enum_undef*⟩ ::= Exception ⟨*exception*⟩
| UnknownVal ⟨*constr*⟩

C.3.3 Structures

A structure is essentially a sequence of fields, some of which can be optional, temporary fields (*checkpoints*) or made-up fields (*parse_field*, which do not relate to something in the binary input):

⟨*struct_def*⟩ ::= struct ⟨*ident*⟩ [⟨*options*⟩] = ⟨*s_fields*⟩

⟨*s_field*⟩ ::= [⟨*decorator*⟩] ⟨*ident*⟩ : PType

⟨*decorator*⟩ ::= optional
| parse_checkpoint
| parse_field
| dump_checkpoint

C.3.4 Unions

A union is a list of union cases associating a discriminating value with a constructor, which usually contains a $\mathcal{PT}$ype (in the absence of a $\mathcal{PT}$ype, the constructed value is empty). In case the union can not be parsed, a default constructor is used. If no $\mathcal{PT}$ype is specified for the default constructor, `binstring` is used:

⟨*union_def*⟩ ::= union ⟨*ident*⟩ [⟨*options*⟩] (⟨*constr*⟩ [of PType]) = ⟨*u_cases*⟩

⟨*u_case*⟩ ::= ⟨*expr*⟩ → ⟨*constr*⟩ [of PType]

C.3.5 Aliases

Other constructions allow to describe $\mathcal{PT}$ypes: aliases to rename a $\mathcal{PT}$ype, ASN.1 structures and aliases are syntactic sugar to describe an ASN.1 DER type (header and content):

⟨*alias_def*⟩ ::= alias ⟨*ident*⟩ [⟨*options*⟩] = PType

⟨*asn1_alias_def*⟩ ::= asn1_alias ⟨*ident*⟩ [⟨*options*⟩]
| asn1_alias ⟨*ident*⟩ [⟨*options*⟩] = ⟨*asn1_header*⟩ ⟨*PType*⟩

⟨*asn1_struct_def*⟩ ::= asn1_struct ⟨*ident*⟩ [⟨*options*⟩] = ⟨*s_fields*⟩

C.3.6 Parsifal standard $\mathcal{PT}$ypes

Here is a list of standard $\mathcal{PT}$ypes provided by the standard library:

- `uintX`: unsigned integers on X bits;
- `string` / `binstring`: character strings, which can span across a given number of bytes or the remaining of the input. The distinction between `string` and `binstring` is the way to represent the value once parsed;
- `bitbool` and `bitint` to describe bit fields;
- `magic` to handle magic values (expected markers);
- `ipv4` / `ipv6`;
- `cstring` for null-terminated strings;
- `der_boolean`, `der_integer`, `der_bitstring`, `der_oid` and other basic ASN.1 types.

and here is a list of PContainers that are available in Parsifal:

- `list` / `array`: a list or an array of a given number of element. A list can also cover the remaining input;
- `base64_container` and `hex_container` to handle automatically these encoding;

- `zlib_container` to uncompress the input during parsing;
- `length_constraint`, for additional length checks;
- `safe_union`, to handle a failsafe mode in case a `parse` function fails with a union.

Appendix D

Mind Your Languages: a discussion about languages and security

This appendix presents studies led since 2007 about the impact of programming languages on software security. It is based on an article published in 2014 [JL14]. The subject has been presented several times in 2013 [JL13, JLD+13] and 2014 [Lev14a].

> An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.
>
> C.A.R Hoare

As part of its mission to raise the security level of IT infrastructures, ANSSI tries to identify, develop and promote methods or tools to improve correctness and robustness of software. In this area, several studies have been conducting since 2007 on the adequacy of languages (including formal methods) for the development of secure or security applications, namely JavaSec [JLM+09] and LaFoSec [JLD+13]. The objective of these studies was not to identify or specify the best possible language, but to understand the pros and cons of various paradigms, methods, constructions and mechanisms. During the work on these studies with language specialists and industrial users, one of the first lessons learned was that there was no common understanding about this notion of intrinsic security of languages, and furthermore that some of the concerns raised by the security experts were not understood at first by the other actors.

This chapter is intended to shed light on those concerns, as exhaustively as possible, through numerous illustrations. It attempts to summarise our journey among programming languages, identifying interesting features with potential impacts on the security of applications, ranging from language theory to syntactic sugar, low-level details at runtime, or development tools. Far from being conclusive, we expect no more than to promote, as far as security is a concern, alternative visions when dealing with programming languages.

The subject of this appendix is therefore quite broad, and to some extent difficult to structure in an appropriate manner. We have chosen to go from language theoretic aspects to low-level concrete details, then to discuss additional questions related e.g. to evaluation. After a short discussion about security in section D.1, section D.2 considers abstract features and paradigms of languages. We will then take a look under the hood, discussing syntax and more generally the concrete constructions provided by languages, that can have a noticeable impact on the quality of (or the confidence in) programs; this is the subject of section D.3. Next, section D.4 casts some light on those steps leading from source code to a running program that definitely deserve proper attention, such as the compilation phase and the execution environment. Once developed, a product has still to be tested and evaluated to derive an appropriate level of assurance, as it is discussed in section D.5. Finally, section D.6 ends this chapter with a few lessons learned and perspectives.

D.1 Language security

A *secure* software is expected to be able to cope with wilful aggression by intelligent agents. In this sense, security approaches can significantly differ from functional ones (e.g. when the objective is to ensure that for standard inputs the software returns the expected results) or safety and dependability ones. In any case, bugs have to be tracked and eradicated, but securing a software may require additional care, trying to prevent unacceptable behaviours whatever the circumstances (and regardless of their probability of occurrence).

Consider for example a program for compression and decompression. The *functional* specification of Compress and Uncompress is that for any file f we have:

$$\mathsf{Uncompress}(\mathsf{Compress}(f)) = f$$

However, it says nothing about the behaviour of Uncompress when applied to arbitrary inputs. Those can be the result of errors, but also maliciously crafted files, which are a common security concern (e.g. CVE-2010-0001 for `gzip`). In essence, we would also like to see the dual specification for resilience, that is for any file c we have:

$$\neg(\exists\, f,\, \mathsf{Compress}(f) = c) \Rightarrow \mathsf{Uncompress}(c) = \mathbf{Error}$$

We are also interested in *security* software, that is products offering security mechanisms. Such software has to be secure but has also to meet additional requirements to protect data or services. A good example discussed thereafter is a library providing cryptographic services, which is expected to protect the cryptographic keys it uses, preventing unauthorised modifications or information.

Finally, we have to take care of the question of the evaluation. Indeed, a secure software may not be of any use in practice if there is no way to know that it is secure. It is therefore important to have processes to obtain appropriate assurance levels (see for example the Common Criteria standard [CC]).

There exists numerous works concerning language safety, and to some extent one may consider that languages such as Erlang or Scade have been specifically designed to cope with resilience, safety or dependability. Many general purpose formal methods can also be seen as tools to adequately address functional or safety needs. However, to our knowledge, the question of the security of languages is not so popular, and the corresponding literature is still rather scarce. To be fair, there are books providing recommendations to develop secure applications in a given language, as well as proposals to natively include security features such as information flow control in languages [HA04]. But from our point of view, the security concerns are much broader than that, and should be addressed not only through coding recommendations or other software engineering methods, but also by due consideration during design phases of languages and associated tools.

This has led ANSSI to conduct several works dealing with security and development, looking at several programming languages or development methods with a critical (and admittedly unfair) eye. This includes a first study to better understand the contributions of the Java language [JLM+09], a second study on functional languages in general and OCaml in particular [JLD+13], as well as a review of the benefits and limits of formal methods [JH08, Jae10].

The sources of our concerns are numerous – and not always subtle: they range from syntax traps or false friends for inattentive developers to obfuscation mechanisms allowing a malicious developer for hiding a backdoor in a product while escaping detection by an evaluator, but also include theoretical properties not enforced in executable code or inappropriate specifications.

Code excerpts presented in the following sections use different programming languages. Our intent is not to criticise a specific language, but to illustrate concrete instances of our concerns (more often than not, the concepts are applicable to other languages). Furthermore, we may provide negative reviews for several mechanisms or concepts; the reader is expected to remember that we only deal here with *security* concerns for critical developments, and that our messages should not be generalised too quickly to other domains.

D.2 Abstract features and paradigms

> The tools we are trying to use and the language or notations we are using to express or record our thoughts, are the major factors determining what we can think or express at all!
>
> Edsger W. Dijkstra

D.2.1 Scoping and Encapsulation

We start our discussion with a family of features existing in nearly any mainstream languages, namely the scoping of identifiers and the encapsulation. They are rather simple mechanisms, and such information is helpful for the compiler to define an appropriate mapping or to apply possible optimizations. Furthermore, they are attractive for the developer dealing with security objectives, e.g. to protect confidentiality or integrity of data. We believe that developers rely on many assumptions about frontiers between various parts of software – for example, they expect proper separation between local variables of an application and a library it uses, and *vice versa*. Any mechanism blurring such frontiers is likely to translate into security concerns.

Variable scoping In general, languages come with variables of different sorts, in terms of scope (local *vs* global) and life cycle (constant, variable or volatile). This is closely related to various theoretical concepts such as bound variables and α-renaming, and is so common that it has become part of developers' intuition.

But for a few languages, the design choices appear rather unexpected, especially when syntax does not provide any clue. Consider the following snippet in PHP:

```
$var = "Hello␣World␣";
$tab = array("Foo␣", "Bar␣", "Blah␣");
{ foreach ($tab as $var) { printf($var); } }
printf($var);
```

The `foreach` loop enumerates the values of the array `$tab` and assigns them to the variable `$var`. This code prints successively `Foo`, `Bar`, `Blah`... and `Blah` at its last line: the variable `$var` in the `foreach` loop is not locally bound and its previous value `Hello World` is overwritten.

Python has a similar behaviour with the comprehension list construct. For example, in the expression `[s+1 for s in [1,2,3]]` (that yields list `[2,3,4]`) the variable `s` should be locally bound, but survives the definition. This is unexpected, and in fact inconsistent with other similar constructs in Python such as the `map` construct[1].

This is, in our view, sufficiently unexpected and intriguing to be dangerous. Developers rely on compositionality, and such poor managements of scopes mean that the semantics of closed pieces of code – using only locally defined variables – now depend upon their context.

Encapsulation The encapsulation, e.g. in object-oriented languages, is a form of extension of scoping, but with a different flavor. In Java for example, one can mark a field as `private` to prevent direct access except from other instances of the same class. Problems arise when a developer confuses this software engineering feature with a security mechanism.

Consider this example in Java, in which a class `Secret` has a private field `x`:

```
import java.lang.reflect.*;

class Secret { private int x = 42; }

public class Introspect {
  public static void main (String[] args) {
    try {
      Secret o = new Secret();
      Class c = o.getClass();
      Field f = c.getDeclaredField("x");
      f.setAccessible(true);
      System.out.println("x="+f.getInt(o));
    }
    catch (Exception e) { System.out.println(e); }
  }
}
```

[1]The behaviour of comprehension lists has been fixed in Python 3 but not in Python 2, for the sake of backward (bugward?) compatibility.

The printed result is `x=42`: introspection is used to dynamically modify the class definition and remove the `private` tag.

It is possible to forbid the use of introspection in Java with the so-called security monitor, yet this is a rather complex task – which is furthermore likely to have side effects on standard library services, as serialisation relies on introspection in Java for example.

OCaml provides encapsulation mechanisms through objects, but also a more robust version based on modules, whose public interface can be partial with respect to their actual implementation. For example, the following `C` module exports its `id` field but not `key`, as specified by its interface `Crypto`:

```
module type Crypto = sig val id:int end;;

module C : Crypto  =
struct
  let id=Random.self_init(); Random.int 8192
  let key=Random.self_init(); Random.int 8192
end;;
```

That is, `C.id` is a valid expression whereas `C.key` is not, being rejected at compilation time.

This is a pretty interesting feature to protect data, relying on well-studied type checking mechanisms: from the academic perspective, such a module is a closed box that cannot be opened. Yet there are in OCaml literature descriptions that do not fit together. In [WL93], for example, a polymorphic function is described as a function that does not analyse the whole structure of its parameters, whereas the reference manual indicates that functions such as `<` are polymorphic and compare the structure of their parameters. Enters version 3.12 of OCaml, introducing first-class modules (*i.e.* modules are values that are comparable).

This led us to write this little experiment:

```
let rec oracle o1 o2 =
  let o = (o1 + o2)/2 in
  let module O = struct let id=C.id let key=o end in

  if (module O:Crypto)>(module C:Crypto)
  then oracle o1 o
  else if (module O:Crypto)<(module C:Crypto)
  then oracle o o2
  else o;;

oracle 0 8192;;
```

The function `oracle` is parameterised by `o1` and `o2`, the lower and upper bounds. It creates a module `O` with `key` set to `o`, the mean value of the bounds, and compare it with `C`. If `O>C`, the function is invoked again replacing the upper bound by `o` (or the lower bound if `O<C`). In practice, the `oracle` function finds the hidden value of `C.key`[2]. The attack is efficient (logarithmic in time) but the main point is that we have been able to cross module frontiers using standard language constructs.

Adopting again the academic perspective, the box has not been opened, but its contents is revealed by using a weighing scale. Some consider that this argue against polymorphic comparison operators in OCaml; for us it shows that trusted theoretical notions are not automatically and easily translated into robust security properties. This particular example was discovered during the LaFoSec study, and led to a fix in OCaml 4.

Our comments regarding encapsulation mechanisms are not intended to pinpoint errors in the languages: such mechanisms are convenient design tools of software engineering. On the other hand, it should be clear for developers that they are not, in general, security mechanisms.

D.2.2 Side effects

A fundamental result of typed λ-calculus (a pure functional language) states that evaluation strategy has no influence on the result of computations. Conversely, in presence of side effects, the order of

[2]The complete explanation of why it works also relies on the existence of a not-so-appropriate compiler optimisation.

computations may become observable.

Provided this notion of evaluation strategy does not appear to be part of developer's common knowledge, we may expect some confusion when dealing with side effects. In [KR88] for example this is explicitly addressed by a clear and simple explanation of the difference of behaviour between the following macro and function when one computes `abs(x++)`:

```
#define abs(X) (X)>=0?(X):(-X)
```

```
int abs(int x) { return x>=0?x:-x; }
```

Yet we would like to discuss more intriguing situations:

```
{ int c=0; printf("%d␣%d\n",c++,c++); }

{ int c=0; printf("%d␣%d\n",++c,++c); }

{ int c=0; printf("%d␣%d\n",c=1,c=2); }
```

Well-informed C developers guess that the first line prints `1 0` as a consequence of right-to-left evaluation of parameters for the call-by-value strategy, but are generally surprised that the second line prints `2 2`.

At this stage, it would be cautious to admit that the pre- and post-increment operators can be too subtle to use. And, as in practice we will not miss them badly, why do they exist at all in the language?

We are not over yet with side-effects, as affectations are by definition the primitive form of side-effect, and as a bonus are also a value in many languages such as C – a cause of troubles for generations of developers that have accidentally inserted an assignment instead of a boolean test in their `if` statements. The third line of the example therebefore prints `1 1` (this is also the final value of the variable `c`), something not so easily explained except by discoursing on calling conventions of C. This type of code is confusing and explicitly discouraged by C standards: why then is it compiled without even a warning?

By extension, *anything* revealing the evaluation strategy can be considered as a side effect. This goes beyond assignments, and also includes for example errors or infinite loops.

Let us play again with macros and functions. What would be the difference between the two following versions of `first`?

```
#define fst(X,Y) X
```

```
int fst(int x,int y) { return x; }
```

As previously, they can be distinguished e.g. with expressions such as `first(x,y++)`. But there are other methods and one can reveal which one is used with `fst(0,1/0)`, as the call-by-value strategy for functions will throw an exception.

Pretty straightforward, so what do the following pieces of code (without even playing with macros)?

```
int zero(int x) { return 0; }
int main(void) { int x=0; x=zero(1/x); return 0; }
```

```
int zero(int x) { return 0; }
int main(void) { int x=0; return zero(1/x); }
```

There is unfortunately no simple answer to this simple question: using the `gcc` compiler, there is an exception for both with option `-O0`, with option `-O1` the first code returns `0` and with option `-O2` the second also returns `0`.

The fact that standard optimizations[3] can modify the observable behaviour of a program is worrying, and reveals in our view a lack of clear and non ambiguous semantics for the C language.

Let us go back to more basic observations on side effects, but with unexpected applications to OCaml. In this language any standard variable is in fact a constant, in the sense that it can only be declared, allocated and initialised at once, and that it cannot be assigned later. Yet all allocations are

[3]Admittedly, the `-O3` flag is known to have curious effects and its use is not recommended, but as far as we knew this was not the case with `-O2`.

managed by the garbage collector and variables live on the writable heap – this prevents the use of low-level security mechanisms such as storing constants in read-only pages for example.

OCaml also provides mutable strings (it is an impure functional language). Put together, this actually means that there is no way to have constant strings, which allows for dirty tricks such as this one:

```
let check c =
  if c then "OK" else "KO";;

let f=check false in
  f.[0]<-'O'; f.[1]<-'K';;

check true;;
check false;;
```

The first lines of this code declares a function `alert` with a boolean parameter and returns a string, either `"OK"` for `true` or `"KO"` for `false`. The second line computes `alert false` to get the reference to the `"KO"` string and overwrites both its characters. As a consequence, the last line returns the string `"OK"` instead of `"KO"` – as will any further invocation of `alert false`.

Again, this is logical, yet surprising: as far as the developer is concerned, the source code of the function `alert` appears to have been modified by a side effect.

This also applies to standard library functions of OCaml and one can modify for example `string_of_bool` – by the way also impacting the behaviour of `Printf.printf`. Similarly, it is a common practice in OCaml to parameter exceptions with strings, that can later be pattern-matched to make decisions; modifying such strings can then interfere with execution flow, as in the following snippet:

```
let alert test =
  if test then failwith "minor" else failwith "major";;

let reaction test =
  try ignore (alert test)
  with Failure "minor" -> ()
     | Failure "major" -> failwith "major";;

try alert false with Failure x -> (x.[1]<-'i'; x.[2]<-'n');;

reaction false;;
```

As a final example, let us mention that the strings returned by `Char.escaped`, which is a security mechanism, can also be meddled with. The following code is a program supposed to call `external_interpreter` with each of its arguments. Following best practices, strings are first escaped using `Char.escaped` on each character:

```
let escape_string req =
  let n = String.length req in
  let clean_req = Buffer.create (2 * n) in
  for i = 0 to n - 1 do
    Buffer.add_string clean_req (Char.escaped req.[i])
  done;
  Buffer.contents clean_req

let main () =
  for i = 1 to ((Array.length Sys.argv) - 1) do
    external_interpreter (escape_string Sys.argv.(i))
  done
```

However, as escaped strings returned by `Char.escaped` are mutable, executing `(Char.escaped '¨').[0]<-'.';` anywhere before the use of `escape_string` (for example in a module initialisation code) will change the observable behaviour of the program, which will incorrectly escape `"'"` as `"."`

instead of "
'", which may lead to security vulnerabilities. As with previous examples, it is disturbing to have a statement located anywhere in a program changing the behaviour of a standard library function.

These problems were also found during the LaFoSec study, and led to fixes in the standard library (where several *constant* strings are now copied before being returned), but the long term solution is to distinguish between constant strings (using the `string` built-in type) and mutable strings (using a new incompatible type called `bytes`), which has been added as an option in OCaml 4.02.

D.2.3 Types

> Don't you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thought-crime literally impossible, because there will be no words in which to express it.
>
> 1984, George Orwell

In mathematics, type theory was a proposal of *Bertrand Russel* to amend *Gottlob Frege*'s naive set theory in order to avoid *Russel*'s paradox. In computer science, type checking is a well-discussed subject, and provides a good level of assurance with respect to the absence of whole categories of bugs – especially when the associated theory is proven. It can statically reject syntactically valid but meaningless expressions, but can also enforce encapsulation, manage genericity or polymorphism, and so on.

In the family of ML languages [Mil84, MT91], such as OCaml, it also leads to type inference and static verification of the completeness and relevance of pattern matching constructs, actually providing great assistance to developers. Type-checking is therefore relatively powerful, efficient and, last but not least, it does not contradict developers' intuition (yet advanced type systems may induce subtle questions, e.g. when dealing with subtypes and higher-order constructs [Pie02]). Thus we consider that type-checking is a must for secure developments, preferably both static and strong to ensure early detection of ill-formed constructs.

Casts and overloading It is often considered that strict application of type-checking concepts leads to cumbersome coding standards, not-so-friendly to developers. OCaml for example distinguishes between integer addition `+` and floating-point addition `+.` and does not automatically coerce values. The expression `1. + 2` is therefore rejected, one correct version being `1. +. (float_of_int 2)`[4].

It is therefore standard for languages and compilers to *ease* developer's work by providing automated mechanisms allowing for overloading (using the same identifier for different operations) and automated casts or coercions. Yet we have various concerns about this approach. A first and trivial comment is that, whereas every student uses this type of trick, nearly none of them can explain what's going on. Casts and overloading are not a comfort anymore but a disguise.

> All animals are equal, but some animals are more equal than others.
>
> Animal Farm, George Orwell

But we are also worried by the consistency of design choices. Consider the example of Erlang, in which expressions such as `1+1`, `1.0+1.0` and `1+1.0` are valid, implicitly inviting developers not to care about the difference between integers and floats. Let us now consider the typical example of the factorial function, presented in all beginner lessons:

```
-module(factorial).
-compile(export_all).

fact(0) -> 1;
fact(N) -> N*fact(N-1).
```

[4] As a matter of fact, there is another possible interpretation, `(int_of_float 1.) + 2`, where the operator drives the cast and forces an integer context (resembling Perl way of typing). The mere fact that there can be multiple interpretation is actually the proof something is off and that this code should trigger a warning.

Without surprise, `factorial:fact(4)` returns `24`, but `factorial:fact(4.0)` causes a stack overflow.

The same type of remarks apply to JavaScript, in which the condition `0=='0'` (comparing an integer with a string) is true, but a `switch (0)` does not match with `case '0'`.

Beyond such inconsistencies, one can also wonder whether casts and overloads are worth missing fundamental and intuitive properties.

For example, equality is expected to be transitive, but this is not the case in JavaScript as `'0'==0` and `0=='0.0'` are true but `'0'=='0.0'` is false. Similarly `+` can represent integer addition, floating point addition or string concatenation. Consider now the following example:

```
a=1; b=2; c='Foo';
print(a+b+c); print(c+a+b); print(c+(a+b));
```

This code prints `3Foo`, `Foo12` and `Foo3`. Whereas both individual operations represented by `+` are associative, the property disappears for the composite (overloaded) operator.

Let us continue our discussion on casts and overloading with examples in the C language. Here again, It is notorious that unexperimented developers can be tricked, the canonical example being:

```
{ int x=3; int y=4; float z=x/y; }
```

Of course, it results in `z` being assigned float value `0.0`.

This one is maybe a little too easy to explain, so consider the following one, without plays on types:

```
#include <stdio.h>

int main(void) {
  unsigned char x = 128;
  unsigned char y = 2;
  unsigned char z = (x * y) / y;
  printf("(%d␣*␣%d)␣/␣%d␣=␣%d\n",x,y,y,z);
  return 0;
}
```

Some may be surprised to find out that `z` is assigned `128` and furthermore disappointed that compiler's optimisation has nothing to do with this result: even in this code where all values have the same type, there are implicit casts (the so-called promotion to a larger integer type).

The cast mechanism in itself can be quite subtle too, as the following code prints `1>=-1` and `1<-1`:

```
{ unsigned char a = 1; signed char b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }

{ unsigned int a = 1; signed int b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }
```

At this stage, we have to consider coercions and overloadings as false friends, on some occasions too devious to be managed properly. Are we too severe? Well, in practice, this leads to situations such as the following one, in which a `safewrite` function has been specifically designed to check that writes in an array are valid with respect to its bounds:

```
#include <stdio.h>

void write(int tab[],int size,signed char ind,int val) {
  if (ind<size) tab[ind]=val;
  else printf("Fail\n");
}

int main(void) {
  size_t size=120; int tab[size];
  write(tab,size,127,42);
```

```
  write(tab,size,128,42);
  return 0;
}
```

If `safewrite(tab,size,127,0)` indeed correctly produces the expected `Out of bounds` message, on the other hand `safewrite(tab,size,128,0)` succeeds – *why* and *where* the write occurs is left to reader's wisdom, as how the program would behave with `size=150` instead of `size=120`.

Of course several compiler options (such as `-Wconversion` for `gcc`) help to pinpoint such problems, but in practice there is a long history of bugs which are, in essence, similar to this one, leading to critical security vulnerabilities. This was the case for example with CVE-2010-0740, a really subtle bug in OpenSSL fixed by an even more subtle short patch:

```
-  /* Send back error using their
-   * version number :-) */
-  s->version=version;
+  if ((s->version & 0xFF00) == (version & 0xFF00))
+  /* Send back error using their minor version number :-) */
+  s->version = (unsigned short)version;
```

The Java language also allows developers for shooting themselves in the foot with overloadings as in this example, which prints `Foo`, `Bar` and `Foo`:

```
class Confuser {

  static void A(short i) { System.out.println("Foo"); }
  static void A(int i) { System.out.println("Bar"); }

  public static void main (String[] args) {
    short i=0;
    A(i);
    A(i+i);
    A(i+=i);
  }
}
```

PHP induces a whole new category of difficulties as far as types are concerned. One can play for example with string arithmetics, and increment with `++` a variable storing the string `"2d8"`. The variable value is successively the `"2d9"` string, the `"2e0"` string and finally the `3` float, as `"2e0"` is interpreted as a scientific notation for $2 \cdot 10^0$!

The confusion between strings and numerical values is consistently reflected by the comparison operator `==`, which can induce casts even when comparing values of the same type. For example, the `"0xf9"=="249e0"` condition yields true as it does not compare the two strings but convert them respectively into `int(249)` and `float(249)`.

Let us illustrate possible consequences of these mechanisms with the following piece of code:

```
$s1='QNKCDZO'; $h1=md5($s1);
$s2='240610708'; $h2=md5($s2);
$s3='A169818202'; $h3=md5($s3);
$s4='aaaaaaaaaaaumdozb'; $h4=md5($s4);
$s5='badthingsrealmlavznik'; $h5=sha1($s5);

if ($h1==$h2) print("Collision\n");
if ($h2==$h3) print("Collision\n");
if ($h3==$h4) print("Collision\n");
if ($h4==$h5) print("Collision\n");
```

When executed, it prints `Collision` four times. It is difficult to believe that we have indeed four distinct short strings with the same MD5 hash, and impossible to have a collision between an MD5 hash and a SHA-1 hash. The trick is that each of the computed hash is itself a string whose pattern

matches the $[0]^+e[0-9]^+$ regular expression. When compared with `==` they are therefore all converted into the same value, that is `float(0)`. This stunning behaviour was the cause of a vulnerability in Simple Machines Forum in 2013[5], and PHP was also impacted in 2011 by a similar security flaw.

To conclude this discussion, remember that we have presented type checking as a way to detect ill-formed expressions. To some extent, casts and overloadings weaken this detection, the compiler being authorised to manipulate the code until it has a meaning. But one should avoid a situation in which any syntactically valid construct would become acceptable!

JavaScript seems to be well advanced on this road, as all the lines in the following example are valid and have a distinct meaning, *including the first and the last ones*:

```
> {} + {};
NaN
> [] + {};
'[object Object]'
> {} + [];
0
> [] + [];
''
> ({} + {});
'[object Object][object Object]'
```

Type abstraction Types provide a level of abstraction of programs convenient for the developers, but can lead to over-simplistic analyses in some cases.

As a first example, consider the following SQL code:

```
SELECT CONCAT(IF(@X<=@Y,'X<=Y','X>Y'),
              ' and ',
              IF(@X>=@Y,'X>=Y','X<Y')) AS Test;
```

With `SET @X=1; SET @Y=2;` this code prints `X<=Y and X<Y`, as expected. But SQL also allows for representing the absence of value. With `SET @X=NULL;` we got `X>Y and X<Y` – a situation one may not expect when reading the code. The same type of comments can be made about the `NaN` value for floats in many languages.

Similarly, one can consider that a well-typed boolean expression will return either true or false. We would however argue that other behaviours are possible and should be considered as well, for example looping computations, crashes... or errors. Indeed, this short-sighted view can easily become a cause of concerns, as illustrated here:

```
#!/bin/bash
PIN=1234
echo -n "Please␣type␣your␣PIN␣code␣(4␣digits):␣"
read -s TYPED_PIN; echo

if [ "$PIN" -ne "$TYPED_PIN" ]; then
    echo "Invalid␣PIN␣code."; exit 1
else
    echo "Authentication␣OK"; exit 0
fi
```

This script checks whether a PIN code is valid or not for authentication. One can conclude that if the provided code is `1234`, authentication will succeed, whereas any other value is rejected. However, in practice the test relies on a shell command which is expecting numerical values. Should a non-numerical value such as `blah` be provided, this test would return an error code interpreted as being not true by the `if` statement, resulting in undue authentication.

In March 2014, an interesting vulnerability has been discovered in the GnuTLS library; let us just quote the analysis provided on Linux Weekly News[6]:

[5] http://raz0r.name/vulnerabilities/simple-machines-forum/
[6] http://lwn.net/Articles/589205/

> [This bug] has allowed crafted certificates to evade validation check for all versions of GnuTLS ever released since that project got started in late 2000.[...]
>
> The `check_if_ca` function is supposed to return true (any non-zero value in C) or false (zero) depending on whether the issuer of the certificate is a certificate authority (CA). A true return *should* mean that the certificate passed muster and can be used further, but the bug meant that error returns were misinterpreted as certificate validations.

As a matter of fact, a very similar issue, CVE-2008-5077, had been diagnosed in OpenSSL in 2008. The corresponding patch simply replaced a `if (!i)` line by `if (i <= 0)`.

Types at runtime As a final remark about types, one should note that in static type-checking systems, types are pure logical information that have no concrete existence in implementations. This topic and its practical consequences are discussed in section D.4.

D.2.4 Evaluators

Some languages offer mechanisms to dynamically modify or create code, e.g. relying on evaluators. This allows for meta-programming and other dynamic features.

This is the case of the `eval` command in PHP, transforming (possibly dynamically produced) strings into code which is executed. Other dynamic features exists in PHP. For example `$$x` refers to the variable whose name is stored in `$x` and `$x()` invokes the function whose name is stored in `$x`.

As a related subject, we will discuss in section D.4 the notion of attack by injection. For now, let us note that as far as security is concerned, the use of any form of evaluator makes a program impossible to analyse: in our view, language-embedded evaluators forbid security evaluation.

D.3 Syntax and syntactic sugar

The elements presented in the previous sections deal with relatively high-level concepts. However, on occasions, the syntax of the language can be confusing or even misleading either for developers or evaluators. We provide a short review of concrete details that can become important.

D.3.1 Syntax consistency and clarity

Identical keywords in different languages can have different semantics – something that we need to live with, but deserve adequate consideration e.g. for education of developers or evaluators.

But it may also be that the same keyword or concept is used with several semantics in a language, depending upon the context. One can consider for example in Java the various possible meanings of the `static` tag, or the use of the interface concept as a flag to enable some mechanisms provided by the standard library (e.g. `serializable`). We find such design choices confusing and therefore potentially dangerous.

As noted in section D.2, the `x=1` assignment is also an `int` value in C, whereas `x==1` is a boolean condition (that is an `int` value). Confusing the two constructs is a common mistake, due to syntax similarity and the absence of warning from the compiler using standard options.

This can also be a trick used by a malicious developer, as in this now classical example[7] of what appears to be an attempt to insert a Trojan horse in the Linux kernel:

```
+ if ((options==(__WCLONE|__WALL)) && (current->uid=0))
+ retval = -EINVAL;
```

This small insert in the code of a system call mimics an error check. Yet in practice, should the two options `__WCLONE` and `__WALL` be set together (an irrelevant combination), and only in this case due to the lazy evaluation of `&&`, then the process user id would become `0`, that is `root`. This trap is discrete both with respect to syntax and behaviour.

[7] `http://lwn.net/Articles/57135`.

D.3.2 No comments

> Don't get suckered in by comments, they can be terribly misleading.
>
> Dave Storer

Some other apparently irrelevant details of syntax may also be misleading to reviewers, e.g. the very basic concept of comments. Modern C compilers for example allows for different types of comments, `//` to comment the end of a line and (C++ style) `/* */` to comment a block of code. Problems arise when a program contains the sequence `//*`, as different tools can have different interpretation of this program.

To consider another language, in OCaml comments are surrounded by `(*` and `*)`, can be nested, but also have an intriguing feature. Consider the following piece of code:

```
(* blah blah " blah blah *)
let x=true;;

(* PREVIOUS VERSION ----------------------
(* blah blah " blah blah *)
let x=false;;
(* blah blah " blah blah *)
----------------------------------------*)

(* blah blah " blah blah *)
```

At first, it seems that the line `let x"=true` is executed, the rest of the code being commented out. But it is possible, in OCaml, to open a string in a comment – and this is exactly what we have done in our example. Therefore `let x"=true` is in fact commented out, whereas `let x"=false` is not. This is especially misleading when syntax colouring tools do not apply the appropriate rules (as this is indeed the case for several of them).

Similar tricks are possible in C, such as in this example:

```
// /!\ Do not uncomment /!\
/*********************************
const char status[]="Unsafe";
// /!\ Only for tests /!\
 *********************************/

// /!\ Important, do not remove /!\
const char status[]="Safe";
```

Of course, it assigns `Unsafe` to the string `status`[8]. This can be used by a malicious developer to trick an evaluator, but it may also mean that, either because of a genuine ambiguity or because of misunderstandings, different tools can have different interpretations for the same source file, a perspective we are worried about.

D.3.3 Encoding

Let us conclude this section with a silly discussion about encoding. Several compilers allow for the use of Unicode characters in source codes. We do not dispute the advantage of having many more symbols at hand, but we are definitely concerned by the numerous other possibilities offered by UTF-8-compliant tools.

Consider this valid Java code:

```
public class Preprocess {
  public static void ma\u0069n (String[] args) {
    if (false==true)
    { //\u000a\u007d\u007b
      System.out.println("Bad␣things␣happen!");
```

[8]Provided there are no blank spaces after the trailing backslashes, which in itself is an interesting observation.

```
      }
    }
}
```

Despite appearances, from the Java compiler's perspective the method name is `main`, the comment mark is immediately followed by a line feed, a closing bracket and an opening one. The `println` command, being out of the `if` block, is therefore executed. That is, using basic escape sequences, the structure of the code seems to be modified.

We have not yet investigated other interesting features such as right-to-left mark which allows for reverse printings (as well as overwrites), but we are definitively worried about them: there is a risk that different tools (such as the editor and the compiler in the previous example) provide different interpretations of the same source, leading to confusion or allowing for obfuscation.

D.4 From source code to execution

Even when the programming language used to develop critical parts of a system offers all the desirable properties, what really matters from the evaluation perspective is the behaviour of the program at runtime. Let us explore the long road from source code to execution, and some consequences for those desirable properties.

D.4.1 Which program?

Evaluators and interpreters For critical systems, we have mentioned in section D.2 our concerns regarding the presence of evaluators in a language, as they make program analyses impossible. But we also need to mention that, by definition, such constructs allow for code injection. This applies to languages such as PHP or Lisp that have built-in internal evaluators, but also to any other language which provides access to external evaluators.

Web applications are the canonical illustrations, transmitting strings interpreted as SQL queries by database engines. In PHP for example a common mistake is to build the query by concatenating constant strings with user-provided data, e.g. `$query="SELECT * FROM MyTable WHERE id ='".$login."'"`, without realising what would happen should the variable `$login` be assigned value `"' OR 1=1; DROP MyTable; - "`. But similar concerns exist e.g. when playing with `exec`-like features, using a shell as the interpreter, like `system` or `popen` in C.

There is no definitive conclusion about this type of vulnerabilities. Use of evaluators should be discouraged and submitted to thorough controls. It also justifies appropriate education for the developers, focusing not on recipes but on the principles, which are very generic: they apply not only to web applications in PHP but to any language with mechanisms blurring the frontier between data, metadata and code.

To illustrate the genericity of this notion, let us consider the shell: what would you expect as the result of the command `rm *` in a directory containing a file named `-fr`, for example?

Interpreted language Various forms of code injections or unexpected executions are discussed here. To palliate such vulnerabilities at system level, there are numerous known mechanisms and established practices. For example, one can control executable files, or rely on enforcing the $W \wedge X$ property to forbid a memory location to be both writable and executable.

But what become such mechanisms with an interpreted language? As the *code* is only read and not executed (from the processor perspective), they offer no protection; the use of interpreted languages therefore requires an update of common security practices and thumb rules. Worse, with Just-In-Time (JIT) compilers, modern interpreted languages need the memory to be writable and executable at runtime, preventing the use of $W \wedge X$ mechanisms, re-opening an avenue for attacks at the native level.

D.4.2 Undefined behaviour

Discussing side effects, we have illustrated unspecified or inadvisable constructs that can be compiled and executed without warnings. There are other meaningless but supported constructs in C, including very

intriguing (and worrying) ones, such as for example pointer arithmetic applied to function pointers[9]. We have also indicated that in the absence of strong typing, several languages try and define the meaning of many possible expressions.

A related problem in some languages is when the compiler leverages the presence of undefined behaviours to optimise the result. For example, in C, since dereferencing a null pointer makes no sense, the compiler can fairly assume that a dereferenced pointer is not null. Thus, in the following code, line 2 results in the assumption that `tun` is not null. The compiler can therefore remove the useless lines 3 and 4 for optimisation:

```
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
if (!tun)
     return POLLERR;
/* use *sk for write operations */
```

This particular code was part of the Linux kernel in 2009 and led to a vulnerability (CVE-2009-1897). A recent study [WZKS13] builds on similar examples to analyse the interaction between undefined behaviours and compiler optimizations. However, some of these dangerous optimizations do not show in the source code, but are created as a result of previous intermediate steps, which makes it very hard to fix the general issue without drastically impacting the performance of the generated code.

D.4.3 Encapsulation and compilation

Many languages offer a form of encapsulation, but in general this abstract characteristic does not survive compilation, producing a single, monolithic memory mapping. This is understandable, as the disappearance of encapsulation has no observable effect on the execution of the program, whereas performance are as good as possible. Yet this also explains why in dysfunctional situations (such as error injection following a buffer overflow) there is usually no protection nor any detection mechanism.

Java, being executed as a bytecode on a JVM, emulate such protections. Of course, as mentioned in section D.2, introspection can be a cause of concern, but let us put aside this question for a moment to consider a more intriguing fact.

Indeed, Java allows for defining inner classes. Instances of the inner class have direct access to private fields of the containing class, and *vice versa*, as illustrated by this code:

```
public class Innerclass {
  private static int a=42;

  static public class Innerinner {
    private static int b=54;
    public static void print() {
      System.out.println(Innerclass.a);
    }
  }

  public static void main (String[] args) {
    System.out.println(Innerinner.b);
    Innerinner.print();
  }
}
```

Interestingly, inner classes can be defined in Java but cannot be represented in bytecode – this is quite remarkable, as it means in theory that the Java language is more expressive than the Java bytecode and that the former cannot be compiled in the latter.

Yet the previous example can be compiled, so how is it done? The inner class is in fact extracted and compiled independently. In order to maintain accessibility of both the outer and the inner classes to the `a` and `b` fields, the private tags are removed. That is, those fields are now accessible from instances of other classes as well.

[9]It works... Well, it compiles, executes, and does *something*.

In our view, such silent modifications of the code by the compiler should be banned.

D.4.4 Memory protections in native binaries

Usually, when the compilation phase produces a native executable, all the compilation units used are grouped in the same memory space, where only coarse grain protection can be enforced: read-only regions and non-executable stacks or data.

As a matter of fact, the link between source code indications and binary memory mapping is sometimes not preserved, but can even be broken. Consider the following C code:

```
int main (void) {
    char* s = "Hello";
    printf ("%s\n",s);
    s[0] = 'h';
    printf ("%s\n",s);
    return 0;
}
```

When compiling it on recent systems, the string literal `"Hello"` is put in the read-only data section at compile-time, leading to a segmentation fault when executing line 4. The problem comes from the compiler that allows for a read-only string to be put into a read-write variable. This inconsistency vanishes when using `g++` instead of `gcc`, or by adding in the options the `-Wwrite-strings` flag, which is off by default due to compatibility issues.

We believe that in some cases, it would be interesting to go further than the current implementations of compilers. Critical parts of a system could benefit from finer-grain memory protections. For example, in object-oriented languages, different classes (or even different instances of the same class) could be compiled as different processes or at least as different threads, relying on operating system mechanisms to provide stronger isolation. This would mean that the compilation steps also consider non-functional properties (like encapsulation) as invariants to be preserved along the way, whereas today they only care to maintain observable behaviours in standard conditions.

D.4.5 About serialisation

In a static type-checking system, types are pure logical information which have no concrete existence in the actual implementation.

It is a pity because very often types, being so intuitive, are implicitly used in developers' reasonings – that is, some ill-typed situations may never be considered, whereas they can indeed occur. This is for example what make the wrap-and-decipher vulnerability of PKCS #11 [Clu03] so elusive: the attack relies on exporting a cryptographic key which is later re-imported as a message. Keys and messages can indeed be confused in the physical world at execution time, whereas theoretical models can implicitly forbid such a scenario (see [JH08, Jae10]). To some extent, this also explains why developers may omit to check the consistency of data in complex formats, e.g. trusting the provided uncompressed size field in the header of a compressed file, resulting in buffer overflow vulnerabilities when using C-like languages.

Many modern languages provide an easy way for the developer to store and load binary objects as strings or files, the serialisation. This mechanism allows for building complex structures (huge trees or hash tables) only once, saving them on disk and then relying on the deserialisation to unfold the object without having to code a parser.

In this context, our remarks about forgotten types apply. In practice, types are related to the interpretation method of such a string, and deserialising data is therefore often an act of faith, relying on the hope that the loaded string will be correctly interpreted as a value of the expected type. In general, modifying a serialised object on disk will lead to a memory error on loading, but the worst case scenario is to forge a serialised object directly pointing at memory cells it should not have access to.

As a matter of fact, the LaFoSec study [JLD+13] showed that the OCaml language was affected: during the serialisation, references between values are unfolded, yet no check insures the extracted references correctly point to deserialised values. It is therefore possible to forge a serialised blob containing pointers towards memory cells outside the blob – this is by the way the true meaning of the warning in the OCaml reference manual about `Marshal.from_channel`:

> Anything can happen at run-time if the object in the file does not belong to the given type.

The same is true with the Python `pickle` module, whose documentation[10] explicitly recognises the vulnerability:

> [The] pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

On occasions, type information would be interesting to preserve at runtime as metadata – provided their integrity is ensured. But even when types are enforced at runtime, as it is the case in Java bytecode, serialisation can still lead to serious security problems. Consider the following code:

```
import java.io.*;

class Friend { } // Unlikely to be dangerous!

class Deserial {
  public static void main (String[] args)
  throws FileNotFoundException, IOException, ClassNotFoundException {
    FileInputStream fis = new FileInputStream("friend");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Friend f=(Friend)ois.readObject();
    System.out.println("Hello␣world");
  }
}
```

This code is intended to serialise an instance of the class `Friend`, but this is not what happen in practice. Indeed, Java serialised files contains a reference to their class, and serialising such a file automatically loads the corresponding class and executes its initialisation code[11], before creating the instance in memory. It is later that this instance is casted – possibly causing an exception, should types not be compatible. But this is far too late if the initialisation code is malicious!

Thus Java serialised objects should not be seen as mere data. In 2008, a vulnerability was reported on the way the Calendar class deserialised foreign objects in a privileged context (CVE-2008-5353). In this particular example it was shown that deserialising an object could lead to load a new class and to execute initialisation code with the privileges of the standard library classes.

D.4.6 Low-level details matter

To understand how some vulnerabilities operate in order to prevent or patch them, it is often necessary to understand how memory management works in a computer.

Let us start with a format string attack in C[12]:

```
#include <stdio.h>

char *f="%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.\
%08x.%08x.%08x.%08x.%08x.%08x.%n";

void strfmtattack() { printf(f); }

int main(void) {
  int s=0x12345;
  int *p=&s;
  strfmtattack();
  if (s!=0x12345) printf("Bad␣things␣happen!␣s=%08x\n",s);
  return 0;
}
```

[10] `http://docs.python.org/3/library/pickle.html`

[11] That is code not included in any method and marked as `static`.

[12] Recent C compilers emit warnings when compiling this code because of the risk of format string attack.

At execution, this code prints various information, ending with `Bad things happen! s=126`. That is, the function `strfmtattack` uses a simple `printf` to read and overwrite variable `s` which is outside its scope and should therefore not be accessible.

Direct stack smashing is also possible in C and may have other interesting consequences, as in the following illustration:

```
#include <stdio.h>
#include <stdlib.h>

void set(int s,int v) { *(&s-s)=v; }

void bad() { printf("Bad␣things␣happen!\n"); exit(0); }

int main(void) {
  set(1,(int)bad); printf("Hello␣world\n"); return 0;
}
```

At execution, this program prints `Bad things happen!` and exits. That is, the `set` function gets a pointer on the stack to overwrite its own return address – modifying execution flow.

Once more, the frontier between code and data appears to be rather thin – here as a consequence of various historical choices such as *Von Neumann*'s architecture and the existence of a single stack mixing data and addresses. This is also possible because of the C language, which allows for low-level manipulations without associated controls. Describing how stack buffer overflows or format string attacks work [One96, LC03] requires knowledge on how variables, function arguments and return pointer are actually mapped in the memory. Without this culture, how to imagine that an out-of-bounds write in an array may result in code injection and execution?

We certainly appreciate high-level programming languages and the features they provide for developers, such as automated memory management by a garbage collector preventing many critical bugs. On the other hand, we are not very comfortable with the related idea of teaching only high-level programming languages. Our fear is that if developers no longer need to call `malloc` and `free` because the language they use has a garbage collector, there will be no point of teaching them what the stack and the heap are.

This is plain wrong, as high-level languages *in fine* rely on native binaries and libraries. In particular, it ends up with this type of code[13]:

```
public class Destruction {
    public static void delete (Object object) {
        object = null;
    }
}
```

By the way, using a garbage collector to handle memory allocations instead of letting the programmer manage memory manually may have security drawbacks.

Indeed, when a program manipulates secrets (passwords or cryptographic keys for instance), one has to minimise the duration of their presence in memory. It also has to ensure secure erasing by overwriting them with other values, so that they are no longer accessible via standard CPU instructions[14]. Those are common practices to tackle memory pages swapping, hibernation, crashes or other situations where the content of the memory is dumped and made accessible. For example, there exists network devices exporting their secrets in a core dump after an easy-to-cause crash. *Heartbleed* (CVE-2014-0160), the vulnerability affecting OpenSSL, is another recent example of possible sensitive data leakage: a buffer overflow allows an attacker for reading parts of server memory[15]. Even if zeroing data when memory is freed would not have blocked the attack entirely, it would have mitigated some of its consequences.

Yet it is very hard (or even impossible) to control lifetime of a secret data in presence of a garbage collector. For the sake of performances, freed areas are generally not cleared by the garbage collector,

[13]Source: `http://thedailywtf.com/Articles/Java-Destruction.aspx`; beyond the source code presented, many comments are worth reading to understand the level of culture regarding memory handling.

[14]We are not talking here about physical attacks on memory chips or mass storage systems, so-called *cold boot attacks*, as described in [HSH+08].

[15]To be precise, the data leaked is located in the process heap, and the flaw also affects OpenSSL clients.

for example. Some garbage collectors (e.g. the mark-and-copy flavor) might also spread the secrets several times across the memory.

Note that the same type of concerns may arise from other well-established mechanisms that have been developed to preserve observable behaviour of programs but not other types of properties related to security. Erasure by overwritings, for example, can be removed by a compiler during optimisation (as there are no later readings), neutralised by cache mechanisms or flash memory controllers moving around physical writings to avoid fatigue of memory cells, and so on.

D.5 About assurance

> Software and cathedrals are very much the same – first we build them, then we pray
>
> Sam Redwine

Developing correct and robust applications is difficult, and it has become even harder due to the increasing complexity of the manipulated concepts and systems. Furthermore, as mentioned in section D.1, it is in general useless in industrial environments to have a secure software without knowing for sure that it is indeed secure. This also applies to other domains such as transportation systems, and leads to the notion of certification process, such as [IEC]. In IT security, the equivalent standards are the Common Criteria [CC], which define different evaluation assurance levels (EALs).

Security certification relies on analyses of the program and of its development process by independent evaluators, a process known as security evaluation. Discussing the intrinsic security characteristics of languages therefore requires appropriate consideration of the helps and limits that applies to the evaluation process. This includes for example an appreciation of the genuine understanding of both developers and evaluators of the features and mechanisms used. This is why we do not consider most advanced language constructs fit for critical developments, simpler approaches generally allowing for higher level of assurance.

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
>
> C.A.R Hoare

D.5.1 About specifications

In our view, developing an adequate level of mastery of the constructs of a language requires both practice and theory. The latter has not to rely on academic publications but can be addressed by proper specifications for the language, e.g. with well-defined semantics for its main constructs, including the standard library.

But consider as a counterexample this extract of the Java specification for the `Object` class:

> The *general intent* is that, for any object `x`, the expression: `x.clone() != x` will be true, and that the expression: `x.clone().getClass() == x.getClass()` will be true, but these are *not* absolute requirements. While it is *typically* the case that: `x.clone().equals(x)` will be true, this is *not* an absolute requirement.

In essence, this specification provides no requirements, and it would be unreasonable to expect *anything* about `clone`.

This type of problems arises quite often, and various constructions in several languages appear to be only partially specified, voluntarily or not, explicitly or not. Let us consider for example two extracts of the C language unofficial specification [KR88]:

> The meaning of "adding 1 to a pointer" and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

As already mentioned, this first extract appears relatively clear, even if it is informal; yet it says nothing about the expected behaviour for function pointers.

> The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

This second extract is a good example of a non deterministic specification: at least, the reader is explicitly informed that there is no guarantee about the result e.g. of `-1/2`.

It is worth mentioning at this stage that non-deterministic specifications can be tricky to handle. How would you test a compiler to check that the result of `-1/2` complies with this specification? In general, the proposal is to check that it yields either `0` or `-1`, but we would argue that this is not sufficient. Indeed, an implementation returning one or the other according to other parameters (e.g. time of the day or a secret value) would not be rejected. Yet you can be sure that having `-1/2` not always equal to `-1/2` (that is losing reflexivity) would have consequences on the robustness of any software[16].

This is an instance of a problem known as the *refinement paradox* in formal methods such as B [Abr05]: given the non-deterministic specification $b \leftarrow \text{getb} \triangleq b := \top \,[]\, b := \bot$ (that is getb is an operation returning a value b which is either true or false), one often consider only the two constant solutions $b \leftarrow \text{getb} \triangleq b := \top$ and $b \leftarrow \text{getb} \triangleq b := \bot$. Yet there are other compliant implementations of getb in which the return value b is still in $\{\bot, \top\}$ but depends upon dynamic values. This notion of *refinement paradox*[17] applies to refinement-based methods such as B, but also to other formal methods such as Coq [Jae10] – and, as illustrated here, in many other, non formal situations.

As a summary, we prefer languages to be specified as completely, explicitly and formally as possible, avoiding non deterministic or partial properties. But we would also like to have tools such as compilers that provide errors and warnings when facing unspecified, discouraged or forbidden constructions.

D.5.2 About code signature

Code review by independent evaluators is a standard process in security evaluation.Yet one has to question its relevance e.g. when dealing with object-oriented languages.

Let us review the following Java code:

```
class StaticInit {
  public static void main(String[] args) {
    if (Mathf.pi-3.1415<0.0001)
      System.out.println("Hello␣world");
    else
      System.out.println("Hello␣strange␣universe");
  }
}
```

Of course, there is a trick – one that we have already mentioned when discussing deserialisation in section D.4. It has nothing to do with the value of the constant `Mathf.pi` but with the class `Mathf` itself:

```
class Mathf {
  static double pi=3.1415;
  static {
    // Do whatever you want here
    System.out.println("Bad␣things␣happen!");
    // Do not return to calling class
    System.exit(0);
  }
}
```

A Java program merely accessing `Mathf.pi` will dynamically load the `Mathf` class and execute its initialisation code. The scope of the evaluation is therefore much broader, as non-executable files representing Java classes on the execution platform at runtime (rather than on the development platform at compile-time) have to be considered as well.

[16] As a matter of fact, this particular non-determinism was solved with the C99 standard, which force the integral division to truncate towards zero in all the cases.

[17] The term paradox is an overstatement, see e.g. [MM05].

This is one of the mechanisms supporting *ad hoc* polymorphisms in object-oriented languages such as Java. By comparison, the Ada genericity (with code specialisation at compilation) or the OCaml polymorphism (a single code for values of different types) are much more easy to handle.

In some cases, to compensate the lack of guarantees on the program which is actually executed, it is possible to rely on signature – provided signature checks cannot be avoided, that the verifier cannot be tampered, that cryptographic keys are adequately protected, that cryptographic protocols ensure required properties, *etc.* It is usual for example in JavaCard to verify the bytecode and sign the applet off-line, to avoid embedding a bytecode verifier in the card.

Yet defensive checks of source code properties (or proof-carrying codes) and code signatures have very different goals: the former aims at ensuring that a program behaves correctly whereas the latter only deals with its origin. In particular, code signing says nothing about the quality of the code (in terms of correctness or robustness for example) or the competence of the signer.

All in all, code signing is useful for low-level libraries, when other defensive checks are not yet available. Here, the innocuousness of such code should be checked by source code audit and vulnerability analysis, signature only providing authentication on the execution platform (instead of any guarantee about the behaviour of the code). In other situation, signature is at best part of a defense-in-depth approach.

D.5.3 The critical eye of the evaluator

The different examples presented in the previous sections show that an evaluator should be aware of the numerous traps of programming languages.

When the evaluation process includes a source code audit, a good knowledge of the involved programming languages can help the evaluators, but we believe it is more important that they have a deep understanding of the *features* provided by the language and used by the developers. For example, object-oriented paradigm and serialisation mechanisms are pervasive. The potential security issues are the same across programming languages implementing them. We have also discussed the very generic concept of code injection.

As we saw earlier, as the developer's intuition might be shattered by some constructions, it is also important to question the properties advertised by the language and their practical robustness. For example, checking whether or not a private field is accessible may be rewarding, as in the Java inner class example discussed in section D.2.

Finally, as illustrated in section D.4, the evaluation of a product should depend on the intended final build process and the target runtime environment, since low-level details matter, and may vary from one platform to another. It is not rare to observe important differences between development and production builds, like the optimisation level; this may lead to different behaviours in practice, and even to real-world vulnerabilities as in the C undefined behaviour case.

D.5.4 About formal methods

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> Donald Knuth

At this stage, some may consider that the previous concerns raised would be solved by using deductive formal methods, allowing for proving program compliance with specifications, such as e.g. the B method [Abr05] or the Coq proof assistant [18].

According to our analyses, this would be overoptimistic, as some of the problems pointed out still apply, and formal methods come with their own traps for inattentive developers or evaluators. This is discussed at length in [JH08, Jae10], dealing with validity and completeness of the specification, the limits of expressivity of the formal languages, inconsistencies in the formal theory or bugs in the tools, trusted translations from formal to standard (executable or compilable) languages, and last but not least not satisfying explicit or implicit hypotheses when using proven software.

For example, in 2004 a variant of SSH was proven secure [BKN04], whereas in 2009 a plaintext-recovering attack against this very variant was discovered [APW09]. In the former paper, Bellare

[18] `http://coq.inria.fr`

et al. made an implicit assumption by working with binary streams already split into messages, whereas the attack presented indeed relied on this split operation.

D.6 Lessons learned and proposals

We have considered numerous aspects of programming languages, trying to identify concerns when security is at stake. Rather than to attempt to provide definitive conclusions about which language should be used, we just provide a list with a few lessons that we learned during our journey, as well as some proposal for the way ahead.

D.6.1 Languages and tools design

First of all, as mentioned earlier, we have been surprised to find that the security of languages, when considered at all, was often so with a very narrow scope or wearing blinkers[19]. It is already common knowledge that designing languages is *hard*, in all likelihood an order of magnitude harder than learning and mastering a language. However, in the light of our work, additional requirements should be taken into account.

If we certainly appreciate works related to more expressive type systems or the native integration of security mechanisms, we would like to avoid having feet of clay. This is why we advocate additional foundation works, defining which characteristics of languages are desirable to cope with security for critical developments.

This requires considering common vulnerabilities or limits, as well as studies of how properties can be ensured or preserved from the theory to the real execution environment, and careful consideration of the robustness of the proposed mechanisms. We do not expect (nor request) new languages to be defined, but at least that the evolution of existing ones are discussed with considerations about security in mind.

To add a few remarks about languages design when security is at stake:

- language principles, whatever their theoretical elegance, must never go against developers' (or evaluators') intuition;
- eliminate non-determinism in the specification of a language, and make explicit as far as possible undefined behaviours (if any);
- always balance the advantages of new constructions with the added complexity for developers to master their semantics.

Beyond the language design itself, attention should be paid to the associated tools, such as for example the compilers, debuggers, analysers or runtimes:

- avoid permissiveness, and track unspecified or undefined constructs to signal them through warnings or errors;
- ban silent manipulations;
- consider additional security options, e.g. to disable some optimisations or to enable non functional features (such as effective erasure);
- consider new compilation invariants or instrumentation, e.g. to preserve type information, encapsulation, execution flow (for example to cope with physical or logical fault injections);
- ease inspection of actual program executed at runtime and traceability with source code for evaluation;
- the tools should protect themselves against specifically crafted malicious inputs;

[19] Java 8, published in March 2014, advertises on *improved security*, that is new cryptographic algorithms, better random generators, support for TLS, or PKCS #11 to cite a few. Yet we worry about the actual gains security-wise as these evolutions, in practice, may translate into additional vulnerabilities in an ever-more complex stack of codes and protocols built on sand.

- long-term goals should include the development of trusted (or certified) tools, such as the CompCert initiative [Ler09, Dar09, Ler11].

The robustness principle (also known as *Postel*'s law) states that you should be conservative in what you do and liberal in what you accept. It aims at improving interoperability, but is not, in our view, appropriate when security is at stake.

D.6.2 Training developers and evaluators

Quite often IT security is expected to be managed only and independently by IT security experts through patches, intrusion detection systems, boundary protection devices, and so one. Yet we consider it is impossible to deal appropriately with security this way. Every developer, for example, should be security-literate enough in order not to introduce vulnerabilities to start with.

Therefore, beyond messages addressed to language and tool specialists, we would like to mention a few recommendations related to the developers' education (at least those dealing with critical systems and security concerns), that are also applicable for the evaluators' education.

Developers should also be ready, beyond functional approaches (checking that what should work works), to also adopt dual approaches (checking that what should not happen never happens). This includes for example worst-case reasonings related to unsatisfied pre-conditions, out-of-range values, ill-formed messages, *etc.* and can be supported by review of most common vulnerabilities and attacks.

Last but not least, as attackers are always looking for the weakest link, developers should be able to have a broad vision encompassing most of the avenues of attacks. To this aim, we have the feeling that one has to learn the basics in several domains such as language semantics, compilation theory, operating system principles, computer architecture. These are, basically, the subjects that have been discussed in this appendix when considering illustrations of our concerns.

Acknowledgement

Several illustrations have been derived from examples on different websites and blogs such as [Koi11, Atw15, orb13] and (last but not least) [Por15]. Another Java-oriented resource for the curious reader is Java Puzzlers [BG05].