Derivation and Analysis of Cryptographic Protocol Implementation

Aina Toky Rasoamanana

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAS005

Thèse de doctorat

Derivation and Analysis of Cryptographic Protocol Implementations

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 Ecole doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 08 juin 2023, par

AINA TOKY RASOAMANANA

Composition du Jury :

Marine MINIER Professeure, Université de Lorraine (LORIA)	Présidente
Karthikeyan BHARGAVAN Directeur de recherche, INRIA Paris (Equipe projet PROSECCO)	Rapporteur
Barbara FILA Maître de conférences, INSA Rennes, IRISA	Rapporteuse
Aurélien FRANCILLON Professeur, EURECOM (Software and System Security (S3) Group)	Examinateur
Arnaud FONTAINE Responsable d'un laboratoire, Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) (Laboratoire sécurité du logiciel)	Examinateur
Hervé DEBAR Professeur, Télécom SudParis (SAMOVAR)	Directeur de thèse
Olivier LEVILLAIN Maître de conférences, Télécom SudParis (SAMOVAR)	Co-encadrant de thèse

Derivation and Analysis of Cryptographic Protocol Implementations

Abstract

TLS and SSH are two well-known and thoroughly studied security protocols. In this thesis, we focus on a specific class of vulnerabilities affecting both protocols implementations, state machine errors. These vulnerabilities are caused by differences in interpreting the standard and correspond to deviations from the specifications, e.g., accepting invalid messages, or accepting valid messages out of sequence.

We develop a generalized and systematic methodology to infer the protocol state machines such as the major TLS and SSH stacks from stimuli and observations, and to study their evolution across revisions. We use the $L^\star$ algorithm to compute state machines corresponding to different execution scenarios.

We reproduce several known vulnerabilities (denial of service, authentication bypasses), and uncover new ones. We also show that state machine inference is efficient and practical enough in many cases for integration within a continuous integration pipeline, to help find new vulnerabilities or deviations introduced during development.

With our systematic black-box approach, we study over 600 different versions of server and client implementations in various scenarios (protocol versions, options). Using the resulting state machines, we propose a robust algorithm to fingerprint TLS and SSH stacks. To the best of our knowledge, this is the first application of this approach on such a broad perimeter, in terms of number of TLS and SSH stacks, revisions, or execution scenarios studied.

Remerciements

Tout d'abord, je remercie profondement mes encadrants, Olivier Levillain et Hervé Debar, pour leurs précieux conseils, leurs relectures et encadrement tout au long de ces années de thèse. Votre expertise et votre bienveillance ont été très précieuses pour m'aider à avancer dans mes recherches et à surmonter les difficultés rencontrées au fil des années.

Je souhaite également remercier Barbara Fila et Karthik Bhargavan qui m'ont fait l'honneur de rapporter cette thèse, ainsi que l'ensemble des membres du jury.

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 Angelot, Sarobidy, Grégory et Christophe.

J'aimerais exprimer ma gratitude envers mes collègues, à savoir Arthur, Adam et Nathanaël, pour toutes les discussions enrichissantes que nous avons eues, que ce soit en lien avec la thèse ou non. Je tiens également à vous exprimer mes meilleurs vœux pour la suite de votre thèse.

Je tiens également à remercier Sandra et Rania. Elles ont fait en sorte que chaque journée se déroule sans encombre.

Je souhaite remercier aussi tous mes collègues et amis. Parmis eux, je tiens particulièrement à remercier Nickson, Lamine, Amin, Lyes, Houda, Romain, Yannick, Chuan, Shurok et Penda. Je souhaite mes meilleurs vœux pour la suite de votre thèse.

Je n'oublie pas aussi de remercier particulièrement Dominique et Alice qui m'ont acceuilli à bras ouvert et m'ont beaucoup aidé sur tous les aspects.

Enfin, je remercie profondement mes parents, Faniry, Tafita, Israeline, Toavina et le petit Isaac pour tous vos soutiens et encouragements. Je vous aime d'un amour inconditionnel. Je peux enfin dire: vitaaaa !

Synthèse

TLS (Transport Layer Protocol) et SSH (Secure SHell) sont des protocoles cryptographiques utilisés pour assurer une communication sécurisée entre un client et un serveur. Ils ont des cas d'utilisation et des fonctionnalités différentes, mais ils partagent le même objectif : sécuriser la transmission de données entre deux dispositifs en réseau.

Au fil des années, TLS et SSH ont été confrontés à plusieurs types d'attaques, y compris les attaques liées à la machine à états des implémentations [SL+16,Lev20]. Comme la spécification ne spécifie pas un automate de référence, les developeurs doivent dériver par eux même leur machine à états à partir des descriptions et séquences informelles des messages du protocole. La tâche est si complexe que les erreurs sont courantes.

Les vulnérabilités peuvent être déclenchées par un attaquant envoyant des messages dans un ordre inapproprié (par exemple, EarlyCCS [Kik14]) ou en sautant des messages (par exemple, SkipVerify [BBD+15], qui contourne l'authentification du serveur en sautant les messages correspondants). Dans de rares cas, de telles vulnérabilités peuvent également être déclenchées par un attaquant envoyant des messages inattendus (par exemple, CVE-2018-10933 et CVE-2018-1000805), ce qui entraîne un accès non autorisé. Dans des cas plus complexes, interférer avec la machine à états permet de nouvelles attaques cryptographiques (FREAK [BBD+15], Factoring RSA Export Keys).

Toutes les piles majeures de TLS et SSH ont été vulnérables à au moins une faille de ce type au cours de la dernière décennie, ce qui prouve que ce sujet mérite d'être étudié plus avant. Nous proposons une méthodologie générique basé sur un algorithme appelé $L^\star$ à la fois pour analyser et identifier les implémentations de ces deux protocoles.

Dans cette synthèse, nous allons voir dans la Section 1 la description du protocole TLS et SSH. Dans la Section 2.2, nous décrivons rapidement un algorithm appelé $L^\star$, puis nous donnons plusieurs défis à relever pour son utilisation pour l'analyse des implémentations des protocoles. Dans la Section 3, nous proposons plusieurs méthodes pour améliorer la performance de l'inférence et dans la Section 4, nous décrivons les listes des vulnérabilités que nous avons détectées pendant

notre analyse des machines à états. Pour terminer, nous proposons une nouvelle méthode basée sur la machine à états pour identifier la pile/version d'une implémentation de TLS et SSH.

1 Description des protocoles TLS et SSH

1.1 Protocole TLS

Afin de permettre l'établissement d'un canal de communication chiffré et intègre entre un client et serveur TLS, les deux parties doivent s'entendre sur les algorithmes et les clés à utiliser. Dans cette étape de négociation, plusieurs messages sont échangés. Un exemple complet est donné à la figure 1 pour la version 1.3.

Avec les versions précédentes de TLS, le client initie la connexion avec le message `ClientHello` dans lequel il annonce les algorithmes et les fonctionnalités qu'il supporte. Le serveur choisit les paramètres qu'il retient pour la session, puis présente son certificat. Un second aller-retour sert ensuite à l'échange de clés. C'est seulement après que les données peuvent être échangées.

Avec TLS 1.3, l'échange de clés est réalisé avec les extensions `ClientKeyShare` et `ServerKeyShare`, des extentions incluses respectivement dans les messages `ClientHello` et `ServerHello`, ce qui permet de réaliser la négociation des paramètres, l'échange de clés et l'authentification du serveur avec un seul aller-retour.

Contrairement aux versions précédentes de TLS, tous les messages TLS 1.3 après le message `ServerHello` sont chiffrés. Pour que le client puisse authentifier le serveur explicitement, ce dernier envoie son certificat (dans le message `Certificate`) ainsi que la signature de tous les messages précédents (dans le message `CertificateVerify`). Le message `Finished` sert de confirmation que les deux parties utilisent la même clé.

La connexion peut échouer prématurément, soit lorsque le client et le serveur ne sont pas d'accord sur une suite cryptographique commune, soit lorsque le client ne fait pas confiance au certificat présenté par le serveur.

1.2 Protocole SSH

Le protocole Secure SHell (SSH) repose sur le protocole TCP et contient trois couches : couche Transport, couche Authentification et couche Connexion. La figure 2 décrit les messages échangés entre un client et un serveur SSH.

La couche transport sert à échanger la version SSH à utiliser, à faire des échanges de clés et aussi à authentifier le serveur. En outre, elle sert à établir un

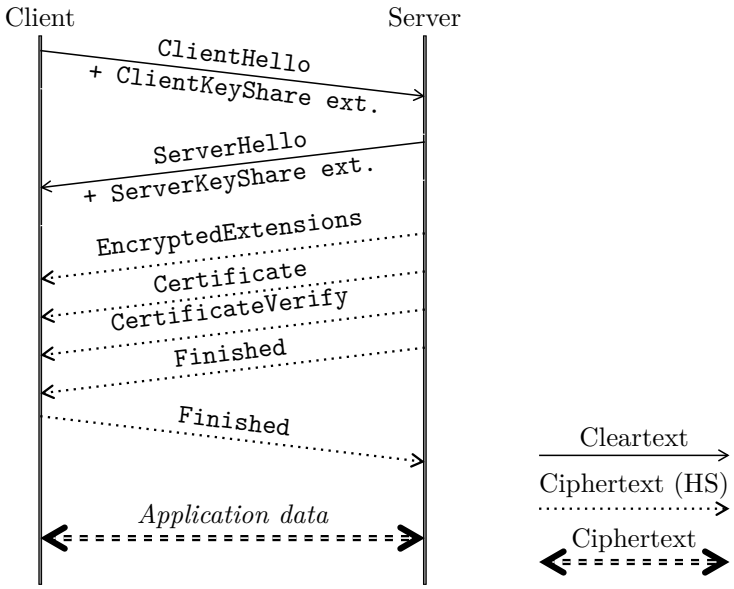


Figure 1: Exemple de négociation TLS 1.3.

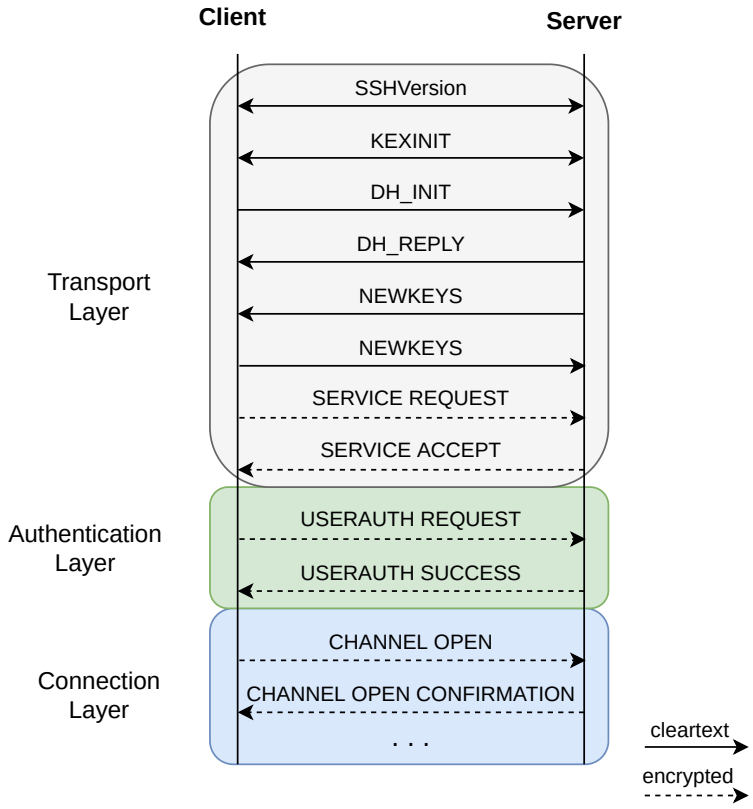


Figure 2: Exemple de négociation SSH.

canal sécurisé entre le client et le serveur, elle fournit la confidentialité, l'intégrité en utilisant le MAC et l'authentification du serveur.

Les clés sont dérivées en utilisant les messages `KEXINIT`, `DH_INIT` et `DH_REPLY`. Le serveur s'authentifie auprès du client en utilisant le message `DH_REPLY`. Tous les messages sont chiffrés après le message `NEWKEYS`.

La couche authentification sert à authentifier le client. Le client s'authentifie auprès du client en utilisant le message `USERAUTH_REQUEST`. Plusieurs options sont possible pour la methode d'authentification: par mot de passe ou clé publique par exemple. Le serveur utilise le message `USERAUTH_SUCCESS` en réponse à la requête d'authentification du client (s'il accepte la requête du client).

La couche connexion fournit des mécanismes pour gérer les canaux telles que l'exécution des commandes, transfert des fichiers, redirection de port, etc. Le client peut ouvrir autant de canaux qu'il veut, en théorie jusqu'à 2^{32}, en utilisant le message `CHANNEL_OPEN`.

2 L'algorithme L*

En 1987, Dana Angluin a proposé un algorithme [Ang87], appelé L*, qui infère un automate fini déterministe (DFA). L'algorithme nécessite la présence d'un Apprenti et d'un Sut (ou encore "System Under Test") tels que le Sut a une connaissance modélisée sous forme de DFA et l'Apprenti souhaite apprendre la connaissance du Sut en utilisant deux types de requêtes : requêtes d'appartenance et requêtes d'équivalence.

La requête d'appartenance permet à l'Apprenti de mettre à jour sa base de connaissance vis à vis du Sut et la requête d'équivalence lui permet de comparer ses connaissances par rapport aux connaissances réelles du Sut. Lors de la requête d'équivalence, l'Apprenti construit une hypothèse sous forme de DFA et fait une requête auprès du Sut si son hypothèse est équivalent à celui du Sut. Si elles sont équivalentes, le Sut retourne "vrai", sinon il retourne un contre-exemple et l'Apprenti remet à jour ses connaissances en faisant à nouveau des requêtes d'appartenance et ainsi de suite.

Dans son papier, Dana Angluin montrait que pour inférer un DFA avec n états et un contre-exemple de taille inférieure ou égale à m , alors $\mathcal{O}(mn^2)$ de requêtes d'appartenance et au plus $n-1$ requêtes d'équivalences sont nécessaires.

2.1 Utilisation de L* en pratique

En pratique, un composant supplémentaire, appelé Mapper, est nécessaire et qui sert principalement d'intermédiaire entre l'Apprenti et le Sut. La Figure 3 représente les composants nécessaire dans le cadre de l'apprentissage actif.

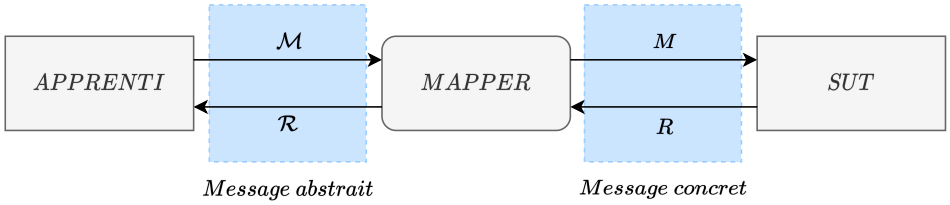


Figure 3: Configuration d'utilisation de L* en pratique.

L'Apprenti implémente l'algorithme L*. Le Mapper sert à concretiser les messages de l'Apprenti vers le Sut et aussi d'abstraire, en une chaîne de caractère, les réponses du Sut vers l'Apprenti.

En 2009, Shabaz et Goz ont adapté L* pour inférer une machine de Mealy (au lieu d'un DFA) [SG09]. Pour tout ce qui suit, nous choisissons d'utiliser cette variante de L*, qui est appropriée à l'étude de piles protocolaires.

2.2 Défis à relever pour l'utilisation de L*

Pour utiliser L*, nous devons relever plusieurs défis aussi bien théorique que pratique :

- Indépendance des connexions : pour éviter des inférences des connexions et pour obtenir des réponses déterministes.
- Méthode d'équivalence appropriée : en pratique, nous n'avons pas une requête d'équivalence parfaite telle que décrite ci-dessus. Nous utilisons des approximations basées sur la requête d'appartenance. Plusieurs méthodes existent déjà telles que: RandomWalk [Lá93], W(p)-method [Cho78,FvBK$^+$91], Distinguishing Bounds [RLM$^+$18].
- Efficacité et convergence : le nombre de requêtes dépend de la taille du vocabulaire d'entrée de L*. Plusieurs chercheurs ont proposées des optimisations que ce soit pour optimiser le nombre de requêtes ou pour accéler la vitesse de l'inférences [RS89, SG09, IHS14, VGRW22, HTJV15].
- Mapper : ce composant joue un rôle important pour l'inférence. Il doit être capable de concretiser n'importe quel message dans n'importe quel ordre. Il doit donc être flexible et robuste.
- Non déterminisme : comme nous souhaitons inférer une machine à états deterministe, tout comportement non deterministe n'est donc pas toléré. Cependant, nous avons constaté que plusieurs implémentations de SSH (par exemple `OpenSSH`, `AsyncSSH` et `paramiko`) se comportent de manière non-deterministe.

3 Amélioration de la performance

3.1 Optimisation du processus d'inférence

Notre premier résultat sur l'inférence de machine à états consiste en l'amélioration de la performance. Pour diminuer le nombre de requêtes d'appartenance, il est inutile de continuer d'envoyer des messages auprès de la cible une fois que la connexion est fermée. De plus, il est possible d'utiliser les informations que nous avons au fur et à mesure de l'inférence. Cela nous permet, non seulement, de détecter les timeouts ou encore de savoir à quoi s'attendre de la cible, mais aussi de détecter à quel moment nous devons arrêter d'envoyer des messages à la cible (quand nous savons, par exemple, que nous aurons des `EOF`); et donc nous pouvons améliorer considérablement la vitesse de l'inférence.

Nous avons évalué nos méthodes d'optimisation du processus d'inférence avec une implémentation d'un serveur TLS 1.2 d'OpenSSL 1.1.1k qui a six états en utilisant un vocabulaire d'entrée de taille douze et un timeout de 1 seconde.

Il apparaît que les deux optimisations améliorent les performances globales, avec une amélioration drastique (jusqu'à 20 fois plus rapide) par rapport à l'anticipation complète du délai d'attente. Nous avons réalisé des expériences similaires avec `statelearner` (même délai et même vocabulaire), sur le même matériel, et le temps nécessaire pour produire la machine à états était de $2,945$ secondes.

Pour une série de $1,400$ expériences concernant TLS (qui a duré environ 2 heures et demie au total, avec 30 inférences en parallèle), le temps moyen d'inférence était d'environ 3 minutes, la médiane était de 81 secondes et les percentiles 10 et 90 étaient respectivement de 27 secondes et d'environ 8 minutes.

3.2 Optimisation de la méthode d'equivalence Distinguishing Bounds

Nous proposons une amélioration d'une méthode d'équivalence appelé Distinguishing Bounds. Nous proposons au lecteur intéressé par cette amélioration de lire l'algorithme original décrit dans [RLM+18]. En quelque mots, cette méthode repose sur l'hypothèse d'une borne de distinction connue, alors que l'automate d'état résultant dépend de sa valeur. Si cette borne, qui caractérise la machine à états réelle, est estimée correctement, la méthode garantit l'obtention d'un résultat correct.

Au cours de la deuxième étape de l'algorithme Distinguishing Bounds, il vérifie si chaque paire d'états peut être distinguée à l'aide d'au moins une séquence d'une longueur inférieure ou égale à la borne de distinction B_{Dist}. Chaque état est vérifié de manière redondante à l'aide de séquences de longueur inférieure ou égale à $B_{Dist} - 1$.

Nous souhaitons avoir la même garantie que la méthode Distinguishing Bounds tout en proposons des améliorations de ce dernier que ce soit en nombre de requête ou temps nécessaire pour trouver un contre-exemple s'il existe.

Nous proposons trois méthodes pour améliorer la méthode Distinguishing Bounds:

1. Toujours utiliser des suffixes de longueur B_{Dist} pour tester chaque paire d'états.
2. Arrêter la vérification d'un état après la réception du message relatif à un fin de connexion.
3. Décomposer la vérification des boucles.

Nous avons mené plusieurs tests pour évaluer l'impact de ces trois méthodes en inférant plusieurs implémentations SSH et nous avons constaté que nous pouvous diminuer le nombre des requêtes d'appartenance lors de l'inférence par deux par rapport à la méthode originale dans les cas les plus favorables.

4 Analyses des machine à états TLS et SSH

4.1 Liste des scénarios

Pour analyser la machine à états des implémentations d'un protocole, nous considérons plusieurs scénarios. Un scénario est défini en fonction du protocole étudié.

Chacun de ses scénarios nous permet de détecter une ou plusieurs attaques liées à la machine à états comme le contournement de l'authentification, la présence de l'oracle de Bleichenbacher, etc.

Un autre domaine d'intérêt est la présence de boucles qui pourraient être utilisées par un attaquant pour bloquer un client, permettant des attaques cryptographiques complexes, telles que l'attaque LogJam [ABD+15]. Pour le côté serveur, la présence de boucles dans les machines d'état pourraient forcer le serveur à maintenir une connexion ouverte indéfiniment. Pour ces attaques par déni de service, nous nous concentrons uniquement sur les événements qui se produisent avant l'activation du chiffrement; de cette façon, l'attaquant n'a besoin que de très peu de ressources pour maintenir le canal ouvert.

a) Scénarios pour TLS

Pour TLS, nous définissons un scénario comme un tuple (rôle, version TLS, contexte) où le rôle peut être client ou serveur ; les versions du TLS que nous supportons sont les versions de TLS 1.0 à 1.3 ; et le contexte est representé par les vocabulaires d'entrées à utiliser pendant l'inférence.

b) Scénarios pour SSH

Pour SSH, nous définissons un scénario comme un tuple (rôle, pré-requis, vocabulaire d'entrée). Le rôle peut être client ou serveur. Le pré-requis represente l'ensemble des couches à éxecuter avant chaque requête d'appartenance. Le vocabulaire d'entrée est formé par la combinaison des messages de chaque couche.

En utilisant cette formaulation de scénario, nous pouvons donc inférer la machine à états des implémentations du client et serveur SSH couche par couche.

4.2 Méthodes pour détecter automatiquement des vulnérabilités

Une fois que nous avons inféré la machine à états d'une implémentations des protocoles, nous devons trouver une méthode pour les analyser. Nous classons en trois catégories les méthodes que nous pouvons utiliser pour les analyses des machines à états:

- Model checking et ses variantes : durant l'analyse de la machine à états, des propriétés requises par le protocole sont définies et vérifiées pour chaque machine à états examinée.
- *Happy path* et des heuristiques : cette méthode consiste à définir les séquences correctes voire attendues vis-à-vis du protocole étudié, que nous appelons *happy path* ; et chaque transition ou chemin qui n'est pas conforme au *happy path* est considéré comme un bug.
- En comparant entre elles plusieurs machines à états d'un même protocole : les différences entre deux ou plusieurs machines à états sont considérées comme des bugs.

Nous avons choisi d'utiliser la méthode basée sur le *happy path*. Cela nous permet de détecter et trier très facilement les vulnérabilités aux déviations à la spécification du protocle. Par contre, cette méthode n'est pas très pratique pour détecter automatiquement des vulnérabilités sur des grandes machines à états.

4.3 Reproduction et détéction des nouvelles vulnérabilités dans TLS et SSH

Nous avons analysé plus de 600 piles/versions de TLS et SSH (400 pour TLS et 200 pour SSH). En considérant les scénarios, nous avons obtenu plus de 6, 000 machine à états.

Nous choisissons d'utiliser la méthode basée sur le *happy path* décrite dans la Section 4.2. La figure 4 décrit la machine à état d'une implémentation TLS 1.3 d'un

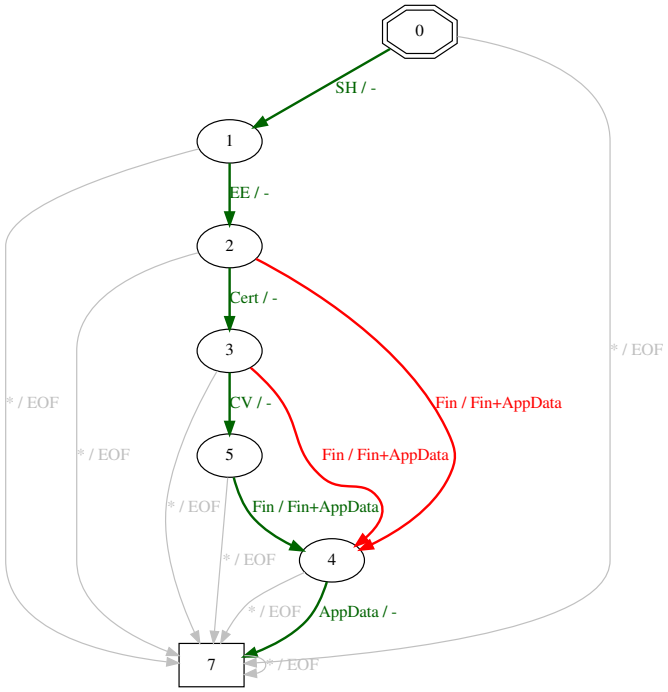


Figure 4: CVE-2020-24613, un contournement d'authentification d'un server TLS 1.3 de wolfSSL jusqu'à la version 4.4.

serveur wolfSSL v4.4.0. Le chemin vert correspond au *happy path* et les erreurs de la machine à états sont colorés en rouge. Notons qu'à partir du moment où nous recevons les messages `Finished` et `AppData` du client en passant par des chemins rouges, nous avons un bug. La figure 4 décrit le CVE-2020-24613. En effet, elle nous montre qu'en ignorant les messages `Certifcate` et `CertifcateVerify`, un attaquant peut contourner l'authentification du client TLS 1.3 de wolfSSL v4.4.0.

Nous avons reproduit et détecté automatiquement plusieurs vulnérabilités telles que (voir les tableaux récapitulatifs de chacune des vulnérabilité ci-dessous):

- le contournement d'authentification ;
- l'affaiblissement de l'authentification ;
- la désactivation du chiffrement (échange en clair) ;
- la fuite d'informations d'identification ;
- la présence des boucles ; et
- des oracles de Bleichenbacher.

En déhors de ces vulnérabilités, nous avons aussi détecté deux types de violation de la spécification intéressantes comme l'existence de la machine à états infinie et problème lié au rafraîchissement de clés. Une machine à états infinie n'est pas en soit une vulnérabilité, néanmoins nous pensons qu'une implémentation propre doit toujours avoir une machine à états finie. Lors de notre analyse, plusieurs implémentations SSH, comme OpenSSH, libssh, AsyncSSH et sshd-lite, ont des machines à états infinies.

a) Contournement d'authentification

CVE #	Piles	Versions	Commentaires
2014-0224	OpenSSL	$\leq$ 0.9.8za $\leq$ 1.0.0l $\leq$ 1.0.1h	EarlyCCS (CCS inattendu)
2015-0204	OpenSSL	$\leq$ 0.9.8zc $\leq$ 1.0.0o $\leq$ 1.0.1j	FREAK (EXPORT RSA downgrade côté client et serveur)
2020-24613	wolfSSL	$\leq$ 4.4.0	contournement d'auth. serveur TLS 1.3
2021-3336	wolfSSL	$\leq$ 4.6.0	contournement d'auth. serveur TLS 1.3
2022-25638	wolfSSL	$\leq$ 5.1.0	contournement d'auth. serveur TLS 1.3
2022-25640	wolfSSL	$\leq$ 5.1.0	contournement d'auth. client TLS 1.3
2018-10933	libssh	$\leq$ 0.8.3	contournement d'auth. serveur SSH
2018-7750	Paramiko	$\leq$ 2.4.0	contournement d'auth. serveur SSH
2018-1000805	Paramiko	$\leq$ 2.4.1	contournement d'auth. serveur SSH
2018-7749	AsyncSSH	$\leq$ 1.12.0	contournement d'auth. serveur SSH
en attente	wolfSSH	all	contournement d'auth. serveur SSH

b) Affaiblissement de l'authentification

CVE #	Piles	Versions	Commentaires
en attente	AsyncSSH	all	serveur SSH
en attente	wolfSSH	all	serveur SSH

c) Ignorance du Chiffrement

CVE #	Piles	Versions	Commentaires
en attente	AsyncSSH	all	serveur SSH
en attente	AsyncSSH	all	client SSH
en attente	wolfSSH	all	serveur SSH
en attente	wolfSSH	all	client SSH

d) Fuite d'informations d'identification

CVE #	Piles	Versions	Commentaires
en attente	wolfSSH	all	client SSH

e) À la recherche des boucles

CVE #	Piles	Versions	Description
2020-12457	wolfSSL	$\leq$ 4.4.0	DoS attaque contre le serveur TLS 1.2
-	erlang	24.0	La configuration par défaut autorise l'attaque DoS contre le serveur TLS
2022-25639	matrixSSL	4.0 - 4.3	DoS attaque contre le serveur TLS
-	fizz	2021 snapshots	Boucles inattendues côté client
en attente	NSS	3.15 - 3.78	DoS attaque contre le serveur TLS 1.0 à 1.2
en attente	wolfSSH	all	DoS attaque contre le serveur SSH

f) Oracle de Bleichenbacher

CVE #	Piles	Versions	Commentaires
2016-6883	matrixSSL	$\leq$ 3.8.2	
2017-13099	wolfSSL	$\leq$ 3.12.2	attaque ROBOT [BSY18]
2017-1000385	Erlang	20.0	attaque ROBOT [BSY18]

5 Identification des implémentations TLS et SSH

Nous constatons que les machines à états produites sont en réalité plus riches et spécifiques à chaque implémentations donnée.

Les différences viennent généralement dans les variations entre les implémentations sur la gestion des erreurs : différents messages d'alerte peuvent être émis. Il arrive parfois que plusieurs machines à états acceptent des messages inattendus et les ignorent silencieusement.

En utilisant une méthode décrite par Shu et Lee [SL11], nous pouvons calculer, pour un scénario donné, un ensemble de séquences de messages d'entrée qui permettent de distinguer les différentes piles que nous avons identifiées. Ensuite, nous pouvons calculer les empreintes de la pile en fonction de la réponse de chaque pile aux séquences distinctives.

Au-delà de révéler des différences intéressantes dans les mécanismes internes des piles, le fingerprinting des piles peut aider un attaquant à identifier, avec quelques séquences de messages, une version donnée (ou un ensemble de versions) d'une implémentation afin de sélectionner une vulnérabilité contre la pile identifiée.

Pour illustrer notre résultat sur le fingerprinting, nous présentons dans le tableau 1 les classes des piles/versions du serveur TLS 1.3 que nous arrivons à détecter en utilisant notre méthode.

Piles	**Versions**	*N*	**CVEs avec un score de sévérité elevé affetctant les serveurs**
erlang	24.0.3 - 24.2.1	9	*Aucun CVE de haute sévérité*
GnuTLS	3.6.16 - 3.7.2	4	*2021-20231 2021-20232*
matrixssl	4.0.0 - 4.1.0	4	*2019-10914* 2019-13470
	4.2.1 - 4.3.0	6	*Aucun CVE de haute sévérité*
NSS	3.39 - 3.40	4	2019-17006 2019-17007 2020-12403 2020-25648 2021-43527
	3.41 - 3.78	4	*2019-17006 2019-17007 2020-12403 2020-25648 2021-43527*
OpenSSL	1.1.1a - 1.1.1n	4	*2020-1967 2020-1971 2021-3449 2021-3711 2022-0778* 2022-1292
	3.0.0 - 3.0.2	4	*2022-0778 2022-1473* 2022-1292
wolfSSL	3.15.5 - 4.0.0	7	2019-11873 et tous ceux de la ligne suivante
	4.1.0 - 4.6.0	7	*2019-15651 2019-16748 2019-18840* 2021-38597 2022-25640
	4.7.0 - 4.8.1	7	*2021-38597* 2022-25640
	5.0.0 - 5.1.1	7	2022-23408 2022-25640
	5.2.0	6	*Aucun CVE de haute sévérité*

Table 1: Piles de serveurs TLS 1.3 regroupées par machines à états. N est le nombre d'états. Les CVEs en italique n'affectent que partiellement la classe d'équivalence.

Contents

List of Figures

List of Tables

Introduction

Transport Layer Security (TLS) and Secure SHell (SSH) are cryptographic protocols used to provide secure communication between a client and a server. Both protocols are maintained by IETF (Internet Engineering Task Force) since 2001 for TLS and 2006 for SSH. Both are widely used. They have different use cases and functionalities, but they share the same objective: securing data transmission between two networked devices.

TLS is a fundamental block of internet security. It encrypts and authenticates data transmitted between a web server and a client browser. The most recent version of the standard is TLS 1.3 [Res18]. It fixes many vulnerabilities uncovered in the last decade.

SSH is a protocol used to establish a secure remote channel between two networked devices. It is commonly used to provide secure access to remote servers and devices. SSH has two major versions, SSH v1 and SSH v2. Nowadays, SSH v1 has been abandoned in favor of SSH v2, which is the version standardized by IETF in 2006 [Ylo06b].

Over the years, TLS and SSH have faced several types of attacks [SL+16,Lev20]. Automata implementation errors represent one category of such implementation attacks. The RFC does not specify a reference automaton. Hence, implementers need to derive their state machine from the informal protocol message descriptions and sequences. The task is so complex that errors are easy.

Vulnerabilities can be triggered by an attacker sending messages in an inappropriate order (e.g., EarlyCCS [Kik14]) or skipping messages (e.g., SkipVerify [BBD+15], which bypasses server authentication by skipping the corresponding messages). In rare cases, such vulnerabilities can also be triggered by an attacker sending specific messages of the other-side-only (e.g., CVE-2018-10933 and CVE-2018-1000805), which leads to an unauthorized access. In more complex cases, interfering with the state machine enables new cryptographic attacks (e.g., FREAK [BBD+15], Factoring RSA Export Keys).

All major TLS and SSH stacks have been vulnerable to at least one such flaw in the last decade, proving that this subject deserves further study.

This thesis focuses on black-box testing using an active learning algorithm, $L^\star$,

initially described by Angluin [Ang87], and later adapted to Mealy machines [SG09], to infer the actual state machine through interactions with implementations.

In chapter 1, we study the state of the art related to TLS and SSH state machines. We also study in chapter 2 the background in using model learning for analyzing protocol implementations. Model learning has two categories, passive and active learning, which we discuss both categories and their applications.

This thesis proposes in chapter 3 a generalized methodology to analyze protocol implementation using active learning. Several research papers used active learning to analyze several protocol implementations. None of them proposed a systematic and automatic approach for security bug detection when analyzing protocol implementations. Actually using active learning to infer the state machines of the protocol implementations is not an easy task [HS18], but it allows to automatically extract state machines from protocol implementations.

In chapter 4, we propose an improvement of a component of the active learning called equivalence query. We also discuss the benchmarks of several equivalence query methods which allows us to select the most appropriate method for our use-case.

Chapter 5 allows to better understand how TLS and SSH implementations react to messages that diverge from an ideal message sequence. We extract state machine using $L^\star$ algorithm. We then compare these state machines with the expected behavior of an ideal stack. Despite the absence of a formal specification of such an ideal stack, a simple approximation of said ideal stack is to use so-called happy paths, which correspond to the expected message sequences for successful connections. A fully compliant stack should only contain happy paths and error transitions, leading to the end of the connection. Every other transition is deemed suspicious. We describe two use-cases of the methodology proposed in chapter 3 and we discuss two platforms, `TLS-inferer` and `SSH-inferer`, to systematically and automatically analyze TLS and SSH implementations.

Finally, beyond detecting bugs and vulnerabilities, we present in chapter 6 another application of our method, stack fingerprinting. We proved that it is possible to fingerprint stacks by using our method. Actually, this can be seen as a consequence of the absence of a reference automaton in the RFC.

Chapter 1

State of the Art on TLS and SSH State Machines

TLS and SSH protocols are both widely used. Both protocols provide confidentiality, integrity and authentication. These two protocols are known for having complex state machines. Moreover, several research papers have focused on the analysis of the TLS and SSH implementation to improve their security.

The organization of this chapter is the following: sec. 1.1 details how TLS and SSH work and describes the differences between TLS versions (from TLS 1.0 to TLS 1.3). Sec.1.2 summarizes and describes many known state machine attacks related to TLS and SSH. Finally, we describe the goal of this thesis followed by the related work on TLS and SSH state machine in sec.1.4.

1.1 Communication Protocols

In this section, we present in details two protocols: SSL/TLS and SSH. SSL/TLS is widely used to secure HTTP communications, whereas SSH allows secure remote access and server management.

1.1.1 Transport Layer Security Protocol

SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are security protocols whose main security goal is to provide confidentiality, integrity and authentication between a client and a server. 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.

The primary use of SSL/TLS is to secure web sites such as online-shopping and banking. Nowadays, it is used to secure widely a variety of internet protocols,

including HTTP, SMTP, IMAP, and FTP.

a) The SSL/TLS Handshake

A typical TLS 1.3 connection is shown in Figure 1.1: the client sends a `ClientHello` message to advertise the ciphersuites, i.e., a set of cryptographic algorithms, it supports and to propose a key share using one of the algorithms it supports. If the client and the server agree on capabilities, the server selects a suitable ciphersuite, and sends its own key share in a `ServerHello` message.

Once the client and server have agreed on algorithms and a common session key, messages are protected using authenticated encryption. The server carries on with several messages, including its certificate chain (`Certificate`) and a signature over the exchanged messages proving its identity (`CertificateVerify`). The `Finished` messages confirm keys in both directions. Then, session keys are updated, and application data can be exchanged.

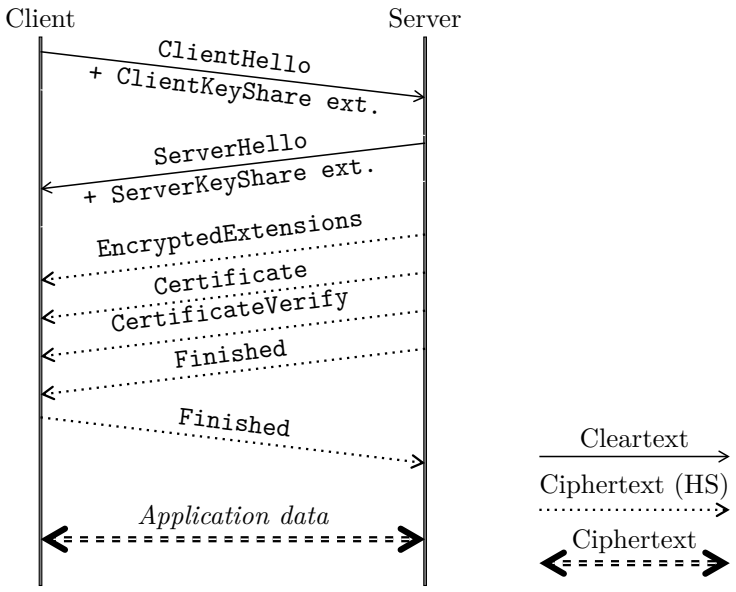


Figure 1.1: A typical TLS 1.3 connection.

Key Exchange TLS 1.3 uses a key exchange mechanism called DHE or ECDHE (Diffie-Hellman Ephemeral), which provides forward secrecy. The Client and the server exchange keys using the `KeyShare` extension. Both sides use their own private key and the other party's public key to compute a shared secret which is used to derive the *Pre Master Secret*.

Server Authentication Server authentication is a critical part of the handshake protocol. The server authentication is done using two messages `Certificate` and `CertificateVerify`. The server presents its certificate using the `Certificate` message and it adds a signature corresponding to the previous handshake messages to prove its identity via the `CertificateVerify` message.

Additionally, the client has to proceed in the same way in case of mutual-authentication i.e., it uses the `Certificate` and `CertificateVerify` messages to authenticate itself to the server.

Key Agreement The `Finished` message is part of the final stage of the handshake protocol. It is used by both sides to confirm keys. It contains the hash corresponding to the handshake protocol transcript.

This step is very important because it ensures that the handshake protocol has been completed without interception. This prevents man-in-the-middle (MiTM) attacks.

b) A History of Versions

The protocol SSL (Secure Sockets Layer) was created by Netscape in 1994 with the aim of securing HTTP connections using the new `https://` scheme. Initially, the first published version was SSLv2 [HE95]. It had significant conceptual issues that were fixed in the subsequent version SSLv3 [FKK11]. It is worth noting that SSLv2 and SSLv3 use distinct message formats, despite having a compatibility mode.

In 1999, the IETF took over the maintenance of the SSL protocol and renamed it to Transport Layer Protocol (TLS). Until today, the IETF has developed four version of TLS: TLS 1.0 [DA99], TLS 1.1 [DR06], TLS 1.2 [DR08] and TLS 1.3 [Res18].

In this thesis, we are not addressing SSLv2 and SSLv3.

TLS 1.0 The changes made to the SSL protocol when it was standardized as TLS 1.0 by the IETF in 1999 were minor ones.

The first modification is to change the pseudo random function to use an HMAC-based construction instead of an ad-hoc construction. The second modification is the use of a standard CBC padding to make TLS 1.0 less vulnerable to padding oracle.

One additional change is the addition of an implementation note about the Bleichenbacher attack on PKCS#1 v1.5 encryption scheme. Different new ciphersuites are also included in TLS 1.0 with respect to SSLv3.

TLS 1.1 The TLS 1.1 protocol is a minor update of TLS 1.0, it enhances the ability to counter known attacks on 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. An implementation note was included to minimize the potential of the padding oracle attacks.

TLS 1.2 This version brings significant changes. In addition to editorial changes and clarifications, such as merging TLS extensions and ciphersuite definitions, the first set of updates aimed to increase the customizability of parameters such as ciphersuites and extensions.

The first modification was the use of pseudo-random function (PRF) depending on the ciphersuite (by default, it uses SHA-256) instead of on a combination of MD5 and SHA-1.

Previously, TLS signatures in the `ServerKeyExchange` message depended on a combination of MD5 and SHA-1 to hash the data. However, with TLS 1.2, a single hash is negotiated with an extension to replace this approach.

Actually, the major improvement in TLS 1.2 is the addition of support for AEAD ciphersuites to protect the record payload, alongside the previously available CBC mode and stream ciphers.

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

TLS 1.3 This latest verion of TLS brings significant changes compared to its older versions. TLS 1.3 includes a new handshake protocol that reduces the number of roundtrips required to establish a connection. It is therefore fast and reduces latency.

TLS 1.3 removes support for all CBC mode and only supports AEAD ciphers, which are considered more secure. Moreover, TLS 1.3 also removes all insecure ciphersuites, including the RSA key exchange mechanism, to block attacks related to the Bleichenbacher attacks.

1.1.2 Secure SHell Protocol

The Secure SHell (SSH) Protocol is a protocol for secure remote login and other secure network services over an insecure network [Ylo06b]. It is mostly used to remotely access and manage servers, transfer files, and execute commands.

SSH has two major versions: SSH v1 and SSH v2. In 1995, Tatu Ylönen proposed the first version of SSH, SSH v1. The goal of SSH v1 was to replace the

rlogin, TELNET, FTP and rsh protocols, which did not provide strong authentication and nor confidentiality guarantee. SSH v1 was widely adopted in the late 1990s and early 2000s.

SSH v1 had several security flaws that were later discovered [SA98]. Thus, in response to these flaws, a new version, SSH v2, was developed and released in 2006 by the IETF [Ylo06b]. In this work, we only focus in SSH v2.

As described in Figure 1.2, the SSH protocol relies on the TCP protocol and is composed of three layers: the Transport layer, the Authentication layer, and the Connection layer.

Briefly, the Transport layer is used to exchange the SSH version and keys and to establish an encrypted and authenticated channel. It is also used to authenticate the server and to query a service such as *connection*, *authentication* (for the client-side).

The Authentication Layer provides mechanisms to authenticate the client to the server. The SSH protocol supports various authentication methods such as password, public key, host-based authentication, etc.

The Connection layer provides a mechanism for executing commands, transferring files, and forwarding TCP/IP connections between the client and server. It is used to manage channels between the client and the server which allow the client and server to communicate and forward data between them. Finally, the Connection layer supports various types of channels such as session, x11, direct-tcpip, etc.

Together, these three layers provide a comprehensive mechanism for remote access, file transfer, and communication over an insecure network. The overall SSH protocol is described in four specifications [Ylo06b, YL, Ylo06c, Ylo06a]. Figure 1.3 describes a typical exchange between a SSH client and server.

a) Transport Layer Protocol

The SSH Transport Layer is the first layer of the SSH protocol. It is described in RFC 4253 [Ylo06c]. The first stage of Figure 1.3 describes the Transport layer protocol.

After exchanging `SSHVersion` messages (which contain the SSH version supported and an identification string), both sides exchange supported algorithms using the `KEXINIT` message. Different types of algorithms are exchanged for key exchange, encryption, MAC and compression.

Once algorithms are exchanged, both sides select appropriate algorithms and perform key exchange using Diffie-Hellman on saved groups with `DH_INIT` and `DH_REPLY` messages. We notice that key exchange can also be performed using an arbitrary group in four steps using:

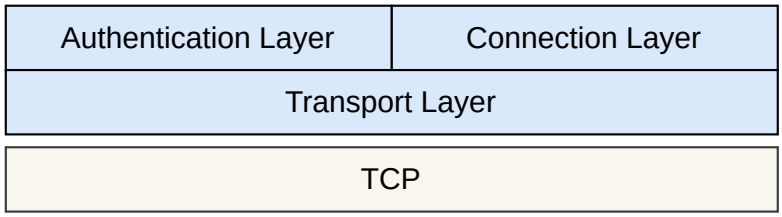


Figure 1.2: SSH protocol layers.

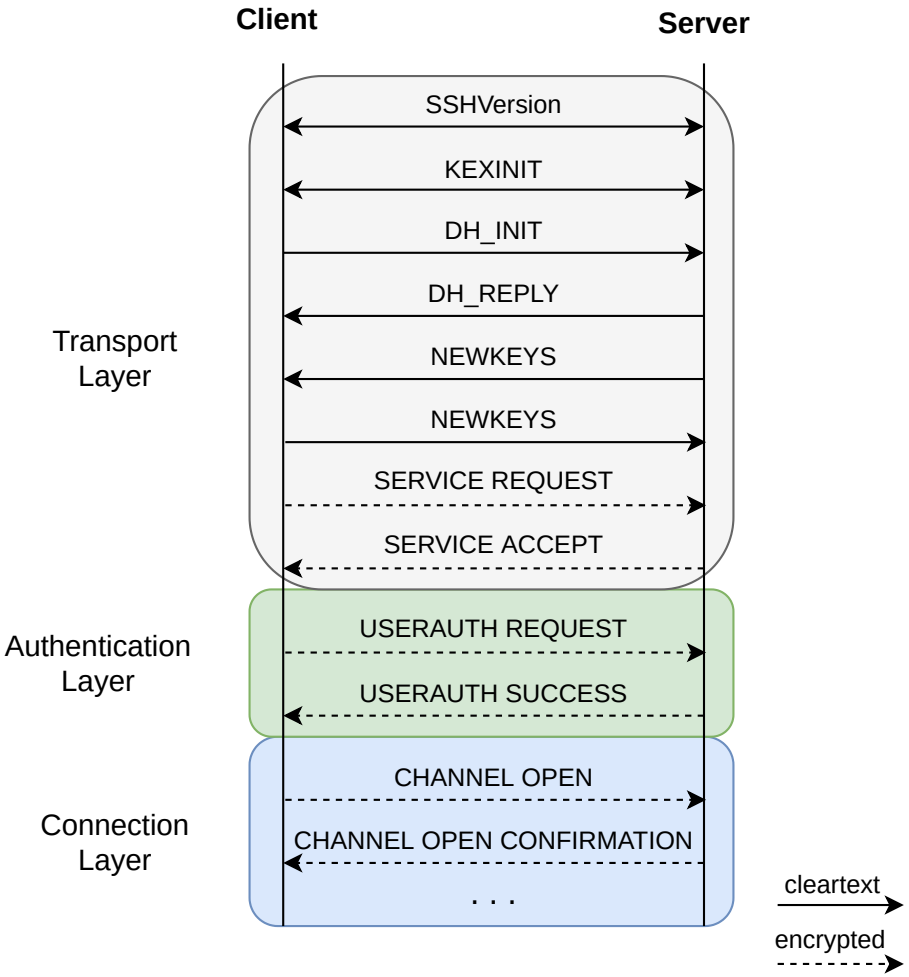


Figure 1.3: A typical SSH connection.

- `GEX_REQUEST`: sent by the client to the server to request the Diffie-Hellman parameters such as group and generator;
- `GEX_GROUP`: sent by the server to the client, contains Diffie-Hellman parameters;
- `GEX_INIT` which is the same as the `DH_INIT` message (sent by the client to the server); and
- `GEX_REPLY` which is the same as the `DH_REPLY` message (sent by the server to the client).

The `NEWKEYS` message marks the end of the key exchange and the begining of the encryption using the fresh keys derived from the key exchange.

Key Derivation Once the key exchange process is finished, then both sides derive the exchanged hash `H`, which is the hash of the parameters of the `KEXINIT`, `DH_INIT` and `DH_REPLY` messages.

The session identification, `session_id`, is the first value of the exchanged hash `H`. It is a unique identifier to distinguish a SSH session from another one. The value of the `session_id` remains static during the connection lifetime. In contrast to the `session_id`, the exchanged hash `H` changes when keys are re-exchanged.

The session keys are derived from the `session_id`, the exchanged hash `H` and the shared Diffie-Hellman secret obtained during the key exchange.

Server Authentication Server authentication is performed during the Transport layer. It is done using the message `DH_REPLY` or `GEX_REPLY`. These two messages contain both the server host keys and a signature over the exchange hash `H`.

Key Re-Exchange SSH offers the possibility to refresh keys. This process is done using the `KEINIT, DH_INIT, DH_REPLY` and `NEWKEYS` messages. It allows to refresh the session keys. The `session_id` remains unchanged even after a key refresh.

The Transport layer establishes a secure channel. It provides the following properties: confidentiality by using encryption, integrity by using MAC and server authentication (with the message `DH_REPLY` or `GEX_REPLY`). It also optionally provides compression of the data transmitted between the client and the server, reducing the amount of data transmitted and thus improving performance.

b) Authentication Layer Protocol

The SSH Authentication layer is the second layer of the SSH protocol and is described in the RFC 4252 [YL]. It is used to manage client authentication. The

messages exchanged between the client and the server are described in the second stage of Figure 1.3.

The client performs the authentication request using the `USERAUTH_REQUEST` message. The authentication method provides several options: password, public key , host based, etc. The server sends `USERAUTH_SUCCESS` if it accepts the client's authentication request message.

In case the public key method is used, the `USERAUTH_REQUEST` message contains the client's public key, the signature corresponding to the client's user name and the `session_id`.

If the client authentication is not required, the client can use the none method for the authentication request message.

c) Connection Layer Protocol

The SSH Connection layer is the third layer of the SSH protocol and is described in the RFC 4254 [Ylo06a]. It provides a mechanism for managing channels: commands executions, files transferring, TCP/IP forwarding, etc.

The client can open up to 2^{32} channels by using `CHANNEL_OPEN` message. The type of the channel can be a session, x11, forwarded-tcpip and direct-tcpip. The server sends the `OPEN_CONFIRMATION` message as a confirmation of the channel opening.

Once a channel is opened, the client can request a pseudo terminal (pty) or a shell and it also can execute commands, exchange data and close channels.

1.2 State Machine Attacks against TLS and SSH Implementations

In 2020, Levillain presented some lessons learned related to the implementation flaws in TLS stacks by describing several known attacks, including state machines attacks, related to TLS stacks [Lev20]. For SSH, we only identify few attacks related to the state machines attacks.

1.2.1 Padding Oracle Attack

A padding oracle attack is a type of cryptographic attack that exploits a vulnerability in the padding scheme. For this attack, an attacker can send crafted ciphertext to a server that decrypts the message and checks the validity of the padding corresponding to the decrypted message.

By analyzing the server's response, the attacker can determine if the padding is valid. It means that the existence of such an oracle can be discovered using state

machine representation. Basically, it is a cryptographic attack, but it can also be considered as a state machine problem because of the existence of the oracle.

Merget et al. summarized all TLS padding oracle attacks and they also scan and automatically classify them [MSA+19]. They identified five padding oracle attacks: Vaudenay's padding oracle [Vau02], BEAST attack [RD11], POODLE (more details about this attack are given in [MDK14]), Lucky 13 and its variants [AFP13, AP16, RPS18] and Bleichenbacher's attack and its variants [Ble98, BSY18, MSW+14].

Most of the padding oracle attacks apply to TLS. Only a few of them affect SSH. Moreover, we identify only one research paper which describes a padding oracle attack related to the CBC mode, affecting SSH stacks [APW09].

We give more details about how padding oracle attack works using the Bleichenbacher's attack example. We study this particular attack during a part of this thesis. We discuss our results in chapter 5.

Bleichenbacher's Attack

Let us first look at the way PKCS#1 v1.5 encryption scheme handles messages. 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 a ready-to-be-encrypted message is presented in Figure 1.4: a null character, followed by a block type byte (here, 2), then at least 8 padding random non-null bytes, a null character and finally the message to encrypt.

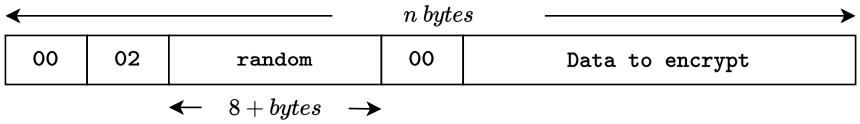


Figure 1.4: A valid PKCS#1 v1.5 encrypted message: the padding must contain at least 8 random non-null bytes.

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, Kel22]. The attack is in particular applicable to RSA encryption key exchange in TLS.

Even if the Million Message Attack has been known since 1998, it is still a problem that periodically reemerges in TLS implementations.

The Bleichenbacher attack made the news in 2014 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. This example shows a dilemma between code reuse and security: it is impossible to safely reuse standard RSA libraries that throw exceptions. It is worth noting that the researchers also found new oracles in OpenSSL and Cavium hardware accelerators, with less efficient attacks.

Moreover, researchers have 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 offers another form of padding oracle, by construction; this oracle could then be used to decrypt a TLS `ClientKeyExchange` message, using the vulnerable SSLv2 stack as an oracle [ASS+16], as soon as both sessions (SSLv2 and TLS) share the same RSA key.

The same paper, as well as an article presented in 2015 [JSS15], used this decryption padding oracle to forge a valid signature, as initially described by Manger et al. in 2001 [Man01]. This attack is not realistic in the standard TLS context, where the signature would have to be forged during the connection; however, some extensions discussed during TLS 1.3 standardization and early versions of Google-QUIC were vulnerable.

As a proof that this attack is still current today, we can cite two more publications targetting TLS: ROBOT (Return Of Bleichenbacher's Oracle Threats) [BSY18], relying on new signals from vulnerable state machines, and CAT (Cache-like ATtacks) [RGG+19].

It is thus clear that PKCS#1 v1.5 is inherently flawed, and, that developers will get it wrong, time and again, until this obsolete algorithm is removed from standards. 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 one.

1.2.2 CVE-2014-0224: `EarlyCCS`

In 2014, Masashi Kikuchi presented an attack affecting OpenSSL which enables a MiTM attack between an OpenSSL client and server. Figure 1.5 describes the cinematics of the attack [Kik14]. The goal of this attack is to force both sides to use weak keys by sending the `ChangeCipherSpec` earlier than expected.

The basic idea of this attack is to exploit the OpenSSL state machine error which accepts an early `ChangeCipherSpec` message, while it should discard it or end connection. If a TLS server or client accepts the `ChangeCipherSpec` message earlier than expected as described in Figure 1.5, then session keys are derived from a null secret and public random values because no shared secret is defined yet.

Once the weak keys is derived, the attacker follows the handshake by encrypting correctly the following messages. However, the attacker has to correctly build the `Finished` message (for the client and server) which contains the hash of the handshake transcript including the shared secret exchanged during the `ClientKeyExchange` and `ServerKeyExchange` between the client and server, to succeed the MiTM attack.

It is worth noting that the `ChangeCipherSpec` is not a part of the handshake message, thus adding or removing it cannot be detected by cryptographic means[1].

1.2.3 CVE-2014-6593: `EarlyFinished` (server impersonation)

When exchanging with the vulnerable client, the attacker only has to avoid sending all the handshake messages except `ServerHello`, `Certificate` and `Finished`. The `Certificate` contains the identity of the server to impersonate (which is public). Finally, after the `Finished` message, the `ApplicationData` is exchanged in cleartext. JSSE and CyaSSL were vulnerable to this attacks.

1.2.4 CVE-2015-0204: `FREAK` (Factoring RSA Export Keys)

Even after they were phased out, many servers kept accepting weak `RSA_EXPORT` ciphers for encryption and decryption process, which force the use of tiny RSA keys for "export" reasons. Using such weak ciphersuites increases the risk of an attack.

The basic idea of the `FREAK` attack is to force the client to use the `RSA_EXPORT` key exchange. As decribed in Figure 1.6, when a legimate client connects to a server accepting `RSA_EXPORT` ciphersuite, the attacker intercepts the client's `ClientHello` message, modifies the ciphersuite proposed by the client to add the `RSA_EXPORT` ciphersuite, and sends the crafted `ClientHello` message to the server.

Next, the server sends its `ServerKeyExchange` message using a shorter RSA key, at most 512 bits. It is worth noting that the `ServerKeyExchange` contains a signature using a strong RSA key, but the current key exchange is done using the weak RSA key. However the attacker has to factor the weak public RSA key used for the key exchange for the attack to succeed.

[1] Because it is ignored when building the `Finished` message.

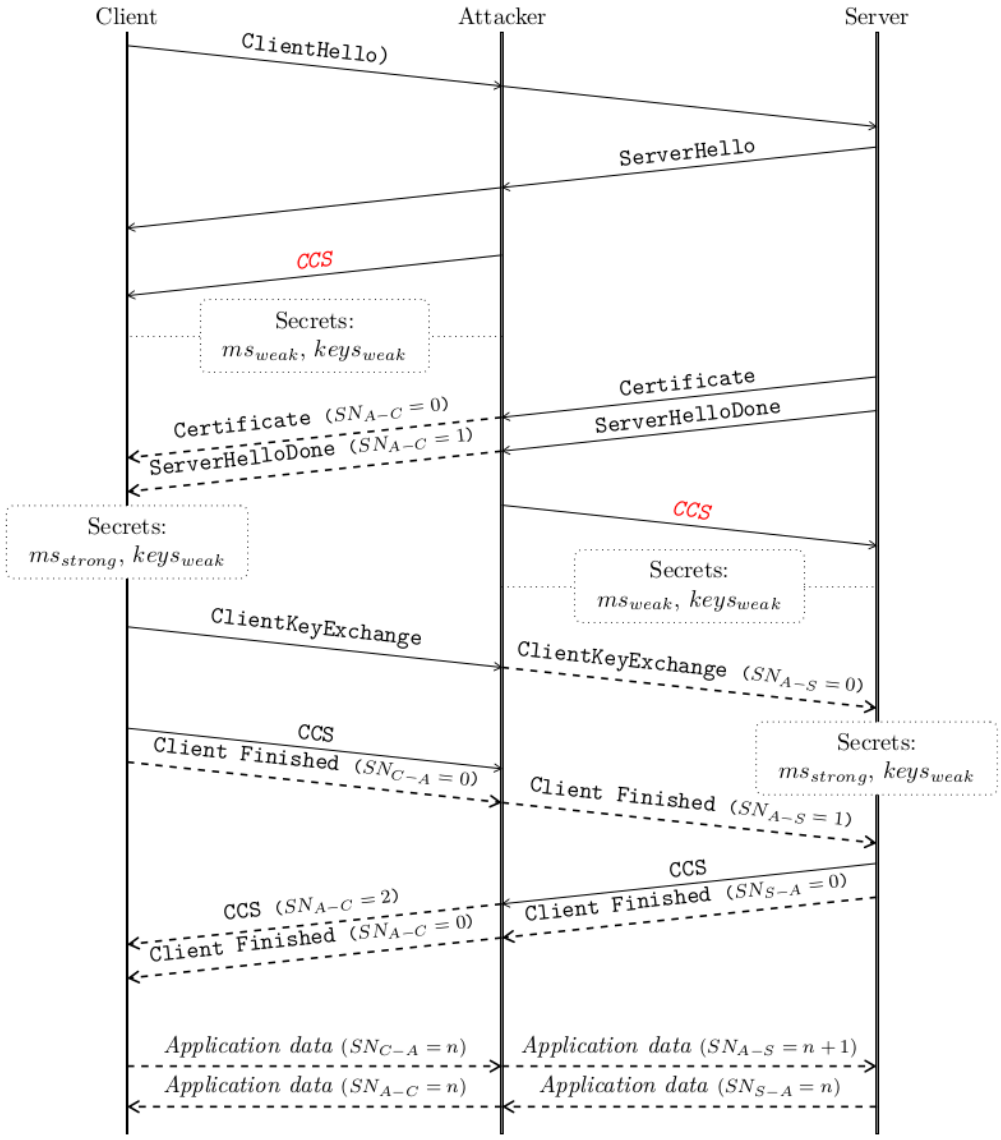


Figure 1.5: EarlyCCS attack cinematics. *ms* stands for *master secret* and SN_{X-Y} corresponds to the number of sent *record* between X and Y for the current epoch (it is reset with each `ChangeCipherSpec` message). The attacker needs in practice to keep track of four such numbers: between the client and the attacker and between the attacker and the server (with a counter for each direction) [Lev16].

Beyond the OpenSSL client, many TLS stacks were also vulnerable to the `FREAK` attack, such as BoringSSL, LibreSSL, Apple SecureTransport, Microsoft SChannel, the Mono TLS stack and Oracle JSSE.

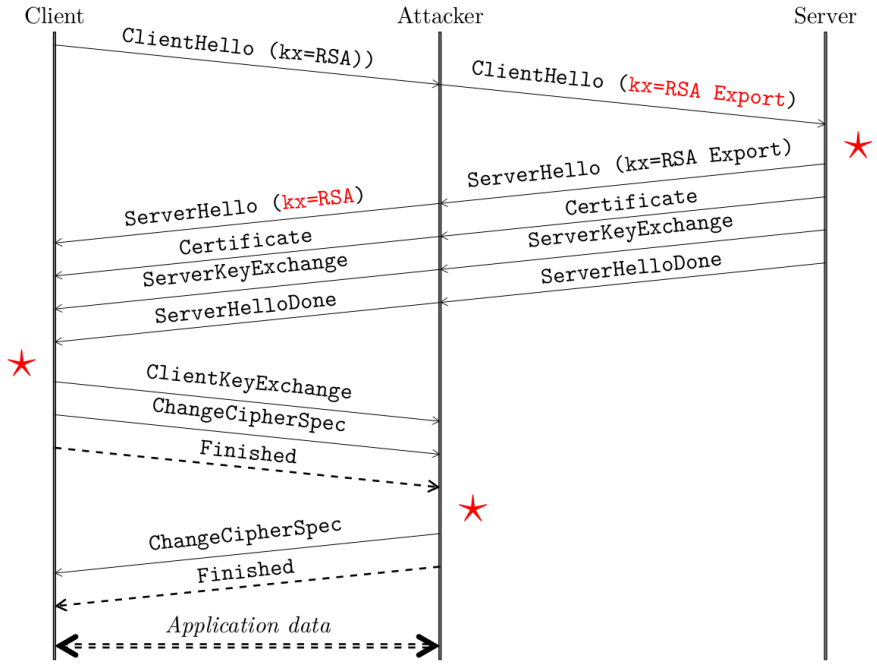


Figure 1.6: Description of the FREAK attack [Lev16].

1.2.5 `SkipVerify` (client impersonation)

When mutual authentication is required, the client must present a `Certificate` and its associated `CertificateVerify` messages which contains a signature over the entire handshake using the private key corresponding to the public key in the `Certificate` message.

The Mono server implementation considered the attacker authenticated without presenting its `CertificateVerify`. With CyaSSL, in addition to skipping the `CertificateVerify`, the attacker also has to skip the client `ChangeCipherSpec` message. This attack also works with OpenSSL, but only if the client's `Certificate` contains a Diffie-Hellman public key.

1.2.6 CVE-2014-6321: Winshock

This attack relies on a simple buffer overflow in the code handling client authentication using ECDSA certificates, which leads to a remote code execution.

When SChannel receives a ECDSA `Certificate` message, it sets the coordinate size based on the public key from the `Certificate`. Next, it parses the `CertificateVerify` message which contains the coordinates of a point on the curve, then reads the data from the signature without checking the consistency

between both lengths. Hence, it allows a buffer overflow using long coordinates within a crafted signature in the `CertificateVerify` message.

In the normal case, this attack is possible if the server request a client `Certificate`. However, this attack is possible even if the SChannel server does not request the client's `Certificate` because the SChannel server accepts and parses unsolicited `Certificate` and `CertificateVerify`.

1.2.7 CVE-2018-10933 and CVE-2018-1000805: Server Unauthorized Access

The attacker, as a client, sends the `USERAUTH_SUCCESS` message instead of the expected `USERAUTH_REQUEST` to have access to the SSH Connection layer and benefit from all its services. The `USERAUTH_SUCCESS` is normally a server-only message which is used to accept the client's authentication.

Unfortunately, this attack affected at least two SSH server implementations (libssh and Paramiko) which do not process separately some specific messages from client and server such as `USERAUTH_SUCCESS`.

1.3 Previous Known Methods to Analyze TLS and SSH Implementations

1.3.1 Related Work on TLS State Machine Analysis

Several methods have been used to analyze TLS implementations. In 2014, Kikuchi discovered the EarlyCCS vulnerability trying to prove state-machine-level properties using a proof assistant [Kik14]. This approach does not scale well, considering the huge work required to properly model the protocol.

Juraj Somorovsky presented TLS-Attacker [Som16], a framework for evaluating the security of TLS implementations. TLS-Attacker allows to forge customized TLS message sequence. It was successfully used to uncover several vulnerabilities in TLS libraries such as OpenSSL, Botan and matrixssl.

On its own, TLS-Attacker does not do state machine learning. It was nevertheless used in conjonction with `statelearner` by van Thoor et al. [vTdRP18] to infer TLS 1.3 state machines in 2018. Their study has several limitations: it only covers an internet draft of TLS 1.3, was only run on a few OpenSSL and wolfSSL servers, and included a reduced vocabulary[2].

Beurdouche et al. [BBD+15], developed a tool, FlexTLS, and proposed a method to test the behavior of mainstream TLS stacks against deviant traces consisting

[2]Vocabulary belongs to the input of the active learning algorithms.

in removing or adding messages from valid traces. They uncovered many bugs in different TLS stacks, including the EarlyCCS vulnerability discussed above and the infamous FREAK attack (Factoring `RSA_EXPORT` Keys). Tarun et al. [YS19] also used FlexTLS on Microsft SChannel, and they found bugs and vulnerabilities, including loops as those described in sec. 5.5.3. However, their approach is not generic and was not updated to be compatible with TLS 1.3.

De Ruiter and Poll used $L^\star$ in 2015 to infer TLS state machines for different TLS servers [dRP15]. They discovered various anomalies and security issues. Their study only covered server state machines and predates TLS 1.3.

Active learning methods have also been applied to other protocols and problems. In his thesis, Bossert developed `pylstar` and used it to reverse-engineer communication protocols between a malware and its server [Bos14]. He also studied the behavior of HTTP/2 clients to allow for robust fingerprinting [Bos16]. Fiterau-Brostean et al. applied model learning to SSH implementations [FLP+17] in 2017 and DTLS implementations [FJM+20] in 2020. In 2019, de Rasool et al. used `learnlib` (the library used by `statelearner`) to study Google's QUIC protocol [RAdR19].

1.3.2 Related Work on SSH State Machine Analysis

In 2007, Poll et al. proposed a method based on formal program verification to analyze SSH stacks, applied their method to MIDP-SSH (a Java implementation of SSH) and found a security flaw related to the Transport layer during their analysis [PS07]. They proposed another method in 2011, to manually review code of OpenSSH [PS11].

During his bachelor thesis, Erik Boss used model-based testing to analyze OpenSSH server [BP12]. He manually built SSH model from RFC 4253, and then generated test cases from it.

These three methods only studied the SSH Transport layer and not the other SSH protocol layers.

In 2017, Fiterau-Brostean et al. used active learning to analyze three SSH implementations and model checking to verify SSH properties described in the SSH specifications [FLP+17]. They did not found security flaws. However, they found that none of the three analyzed SSH stacks complied with the specifications. However, their work suffers from several limitations. They did not formalize the Mapper thus, model checking results cannot be fully transferred to the actual implementations. Moreover, it is not clear which messages were used to infer each SSH stacks, as they mentioned that for some SSH stacks, they only used *"restricted alphabet to reduce the learning time"*.

1.4 Looking for a Method to Analyze Protocol State Machines

Combining software development, network protocols and cryptography is a complex subject. TLS and SSH are examples of such combinaison. Both protocols have complex state machines, and the previous section shows that attacks related to the state machine are very recurrent and still relevant.

On the other hand, the IETF insists on letting the developers making their implementation choices, which leads them to make avoidable mistakes. Sometimes, specifications are not crystal clear, then the developers decide on their own on how to desambiguate the situation, which can lead to state machine attacks.

Several methods have been used to analyze communication protocol implementations. In this thesis we are looking for an efficient method which allows us to automatically and systematically analyze different protocol implementations. We believe that such a method is very useful for the community to improve protocol implementations.

From the related work discussed in sec. 1.3, active learning appears to be the most relevant technique. It seems relevant for analyzing the state machine of a protocol implementations. Most of the research papers using this method are not up to date with respect to studied protocol or they did not cover all features related the studied protocol.

Our research questions are therefore discussed below.

Generalized methodology based on active learning This first research subject allows us to:

- identify and classify bugs related to the state machine of a protocol implementations; and check if active learning is the relevant method to detect all the identified classes of possible state machine bugs;
- identify theoretical difficulties to overcome using the active learning method; and
- identify different method to automate bug detection from the state machine.

Limitations and optimization of the active learning method One legimate question is the performance of the active learning method. Fiterau-Brostean et al. discussed the performance of active learning when learning the SSH state machines of three SSH stacks. They mentioned that their inference took several days. Thus, this research question forces to propose different possible optimization for the learning process. It also allows to point to a new research direction on the possible improvement of the active learning method.

Application of the generic methodology to TLS and SSH Our research implements the theoretical results of the first research subject; and confirms the power of the methodology in reproducing known vulnerabilities and in detecting new ones.

It also allows us to evaluate the difficulties in using such a method in practice, especially with protocols with complex state machines as TLS and SSH.

Beyond detecting bugs and vulnerabilities automatically and systematically, this research subject also allows to uncover the problem in letting the developers making their implementation choices of the protocol. Even if a protocol implementation does not have a bug, we strongly believe that it remains possible to fingerprint protocol implementations by analyzing the error message related to an unexpected message.

Chapter 2

Model Learning – Theory and Application

As in this thesis, we propose a method based on $L^\star$ algorithm, a model learning algorithm. Thus, we give more details about model learning in this chapter.

We organize this chapter as follow: we present in sec. 2.1 two categories of model learning: passive and active learning. Then in sec. 2.2 we give more details about $L^\star$ the first algorithm of the active learning. Next, in sec. 2.3 we discuss how to use active learning in practice followed by an overview of other deterministic and non-deterministic active learning algorithm. Finally, in sec. 2.4, we summarize the application of this method in the field of network and software security.

2.1 Passive vs Active Learning

Model learning aims to build a software state diagram model from an implementation by observing the output corresponding to a given input. The state diagram model represents the observed behavior of the analyzed software against the given inputs.

This method is used to understand the behavior of a given component or a system without requiring access to the source code. It is also useful to generate test-suites for software testing [MS13]. Combined with fuzzing, it can be used to extract the state diagram model and then guide the protocol fuzzing process [PLJZ22]. It is also used for adaptive model checking [GPY02, PVY99]; the basic idea is to learn the state diagram model and then use model checking [Cla97] to verify if the learned model satisfies some properties.

To extract the state diagram model of a component or system, different approaches have been developed: analyzing code, mining system logs or performing tests. Methods for inference model may be classified as black-box or white-box. In

this thesis, we are only interested in the black-box methods. These approachs are relatively easy to use and can be used even if we do not have access to the source code.

Several possible representations can be used according to the use-case for the state model diagram, for example, Deterministic Finite Automaton (*DFA*), Nondeterministic Finite Automaton (*NFA*), Mealy/Moore Machine, Timed Automata, etc.

For the inference model approach, we present two techniques: passive (see sec. 2.1.1) and active learning (see sec. 2.1.2).

2.1.1 Passive Learning

Passive learning was first initiated by Gold in 1978 [Gol78]. Labeled data is provided to the LEARNER, who is supposed to find a model representing this data. The data are usually a set of logs from the *System Under Test* (SUT) and are labeled as accepted (positive samples S_+) or rejected (negative samples S_-) such that $S_+ \cap S_- = \emptyset$.

Thus, the model learned using passive learning is only accurate with respect to the logs provided. One of the drawbacks of using this method is the difficulty of finding negative samples in practice. These samples are often the result of fuzzing techniques applied on the implementation of the studied protocol [Bos14]. Hence, in 2019, Avellaneda and Petrenko proposed a method based only on positive samples (i.e., without negative samples) [AP19].

Most of the passive learning algorithm are based on the so-called Prefixe Tree Acceptor (PTA) [LMD05, HSL08, CWKK09, DlH10], which is a tree-like Deterministic Finite Automaton (DFA). For a given sample $S = \langle S_+, S_- \rangle$ such that $S_+ = \{aa, aba, bba\}$ and $S_- = \{ab, abab\}$, the corresponding PTA is given in Figure 2.1, where leaves are either an accepting or rejecting state. After the PTA is built, a DFA minimization is used to transform the PTA into an equivalent but smaller DFA which recognizes the same samples $S = \langle S_+, S_- \rangle$. This step is the key for most algorithms that deal with learning automata using samples [HSL08, Lan99, Xie03, AS94, LOW22] such as GOLD, RPNI, etc.

2.1.2 Active Learning

In *active learning*, querying actively the SUT, is required, the LEARNER is authorized to query the SUT whenever it is needed. The LEARNER derives the model from these observations.

Basically, the SUT should be a *Minimally Adequate Teacher* (MAT). It means that the SUT should be able to respond to two types of query: *membership queries*

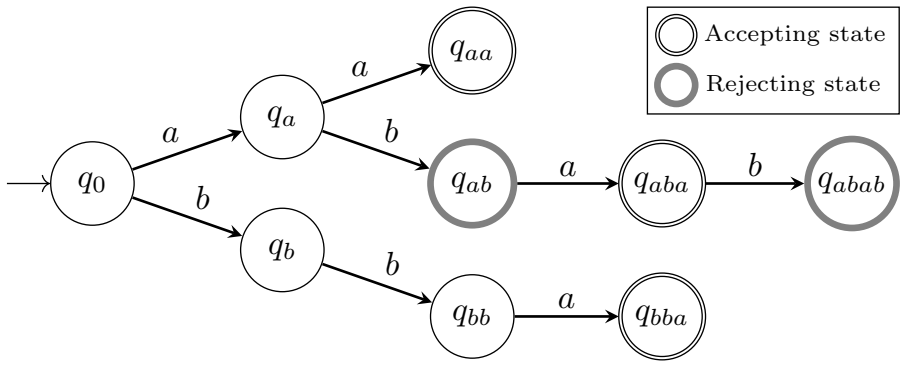


Figure 2.1: PTA($\{(aa, 1), (aba, 1), (bba, 1), (ab, 0), (abab, 0)\}$)

(MQ) and *equivalence queries* (EQ). In other words, a MAT contains two different oracles: a *membership oracle* and a *equivalence oracle*.

A *membership query* consists in querying if a word w is in MAT's language $\mathcal{L}$ (i.e., is $w \in \mathcal{L}$ *true*?). An *equivalence query* consists in quering if an hypothesis $H(\mathcal{L})$ is equivalent to $\mathcal{L}$ (i.e., is $H(\mathcal{L}) \equiv \mathcal{L}$ *true*?). If $H(\mathcal{L})$ is equivalent to $\mathcal{L}$, then the answer is *true*. In the opposite case, a *counter-example* is returned. MQ are used to update the LEARNER's knowledge of the SUT and EQ are used to check if the LEARNER's automaton is equivalent to the SUT's automaton.

In practice, the EQ does not exist, thus an approximation using MQ is required (see sec. 3.1.2 for more details). Hence, David Lee et al. [LY96] first proposed a method called *conformance testing* to tackle this problem by using a finite number of sequences of messages. Note that the role of the LEARNER is to build a hypothesis and the role of the conformance testing tool is to check the validity of the built hypothesis.

Some interesting questions have been addressed over the years: how to test efficiently and quickly the hypothesis using few sequences of messages? Several methods have been developped to solve the problem of the EQ method, e.g., RandomWord, RandomWalk, W(p)-method (see sec. 4.1 for more details).

Therefore, the learned model is only an approximation with respect to the input vocabulary. Using a different input vocabulary may lead to a different model. The quality of the learned model is also related to the EQ method used, which will be discussed in chapter 4.

Actually, in contrast to the active learning, all the passive learning approaches suffer from a scalability issue when applied on large automata due to the NP-completeness of such algorithms [Bos14]. Hence, we focused our work by using active learning for analyzing communications protocols implementations.

2.2 The L* Algorithm

In 1987, Angluin proposed L*, an algorithm that infers a deterministic finite automaton using membership and equivalence queries [Ang87]. This technique can be extended to extract the state machine of a protocol implementation using a Mealy machine representation [SG09], which can be seen as an automaton where transitions are labeled by both the messages sent and received.

L* is an automated black-box technique driven by a LEARNER. Figure 2.2 describes the experimental process. Lets assume the LEARNER wants to learn the SUT's language $\mathcal{L}$ over an input vocabulary Σ_I. To understand how the L* algorithm works, we define the *observation table* and two interesting properties about the *observation table*: closure and consistency.

Observation Table During the learning process, the LEARNER uses a table called the *observation table* to update its knowledge about the SUT. Table 2.1 describes an example of an *observation table*. Concretly, the *observation table* classifies sequences as members or non-members of the SUT's language $\mathcal{L}$.

It contains three elements (S, E, T) where $S \subset (\Sigma_I)^*$ is a non-empty finite prefix-closed[1] set of strings, $E \subset (\Sigma_I)^*$ is a non-empty finite suffix-closed[2] set of strings and T is a finite function: $((S \cup S{\cdot}\Sigma_I){\cdot}E) \longrightarrow \Sigma_O$, with Σ_O is the output vocabulary.

The *observation table* can be seen as a two-dimensional array with row labeled by the elements of $S \cup S{\cdot}\Sigma_I$ and columns labeled by the elements of E. We denote $row(s) = \prod_{e \in E} T(s.e)$ for all $s \in S \cup S{\cdot}\Sigma_I$.

Definition 2.2.1 (Closure)**.** An *observation table* is said to be closed if for all $t \in S{\cdot}\Sigma_I$ there exists $s \in S$ such that $row(t) = row(s)$.

Definition 2.2.2 (Consistency)**.** An *observation table* is said to be consistent if for all $(s_1, s_2) \in S \times S$ such that $row(s_1) = row(s_2)$ then for all $a \in \Sigma_I, row(s_1.a) = row(s_2.a)$.

Intuitively, rows represent states, closure means that all states are defined and consistency means that multiple representatives for a row lead to the same transitions.

2.2.1 How to Update the Observation Table?

During the learning process, the LEARNER checks if the *observation table* is closed and consistent. If one of these two properties is not met, then the *observation*

[1] A set is prefix-closed iff all the prefixes of every element of the set are also elements of the set.

[2] A set is suffix-closed iff all the suffixed of every element of the set are also elements of the set.

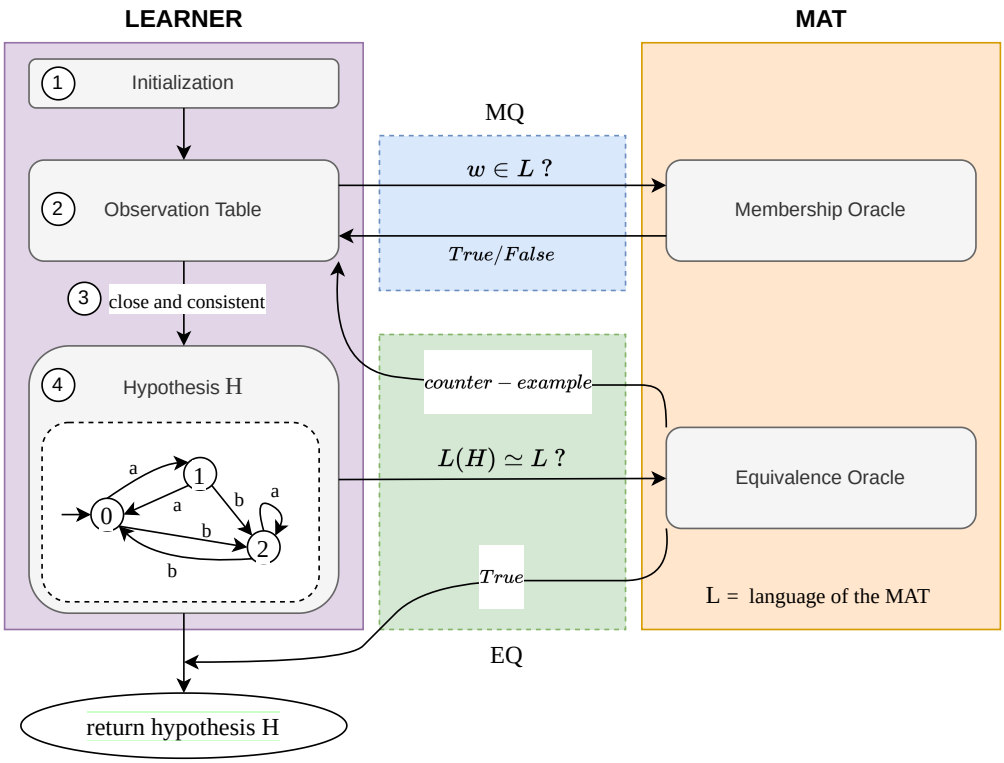


Figure 2.2: Principle of L* algorithm.

		E	
		a	b
S	ϵ	x	x
	a	y	x
$S.\Sigma_I$	b	x	x
	$a.a$	y	x
	$a.b$	x	x

Table 2.1: Example of a closed and consistent *observation table* (S, E, T)

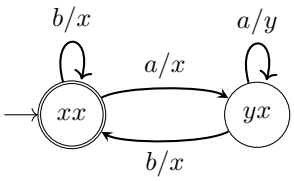


Figure 2.3: The Mealy machine derived from the *observation table* given in Table 2.1

table is updated:

- if the *observation table* is not closed, it means that there exists $t \in S \cdot \Sigma_I$ such that for all $s \in S$, $row(t) \neq row(s)$, then add t to S (i.e., $S \leftarrow S \cup \{t\}$) and fill in the table using MQ;
- if the *observation table* is not consistent, it means that there exists $(s_1, s_2) \in S \times S$ and $a \in \Sigma_I$ such that $row(s_1) = row(s_2)$ and $row(s_1.a) \neq row(s_2.a)$, then add a to E (i.e., $E \leftarrow E \cup \{a\}$) and fill in the table using MQ.

However, we also update the *observation table* if the EQ oracle returns a counter-example $\mathcal{C}$ (meaning that the hypothesis and the SUT disagree with the sequence $\mathcal{C}$). Thus we update the *observation table* by adding $\mathcal{C}$ and all its prefixes to S (i.e., $S \leftarrow S \cup prefixes(\mathcal{C})$) and then by filling the table using MQ.

2.2.2 The L* Learner

Figure 2.2 describes the principle of the L* algorithm. The first step of the algorithm is the initialization of the *observation table* (1), and the second step is filling the table by querying the SUT using MQ (2). When the *observation table* is closed and consistent (3), the LEARNER builds an hypothesis $\mathcal{H}$ (4) and queries the SUT to check the validity of the built hypothesis by using an EQ (5). If the SUT (the EQ oracle) returns a counter-example, then the LEARNER updates the *observation table* (2); otherwise, the LEARNER consider the hypothesis $\mathcal{H}$ as the SUT's language with respect to the input vocabulary.

2.2.3 Building an Automaton from the Observation Table

As discussed in sec. 2.1, there are several possible representations for the state model diagram according to the use-case. For network protocols, the Mealy machine is the most appropriate representation, selected by almost all researchers working in this field.

a) Mealy Machine

A Mealy machine $\mathcal{M}$ is a tuple $(Q, q_0, \Sigma_I, \Sigma_O, \delta, Out)$ where:

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ_I is a finite set of inputs,
- Σ_O is a finite set of outputs,

- δ is a transition function, $\delta : Q \times \Sigma_I \to Q$, and
- Out is an output function, $Out : Q \times \Sigma_I \to \Sigma_O$.

The transition function defines the modification of the current state $q \in Q$ given an input symbol $a \in \Sigma_I$. The output function can be extended to $Out : Q \times (\Sigma_I)^* \to (\Sigma_O)^*$. This extension allows to get the sequence of outputs corresponding to the sequence of inputs $\sigma \in (\Sigma_I)^*$ from a given state $q \in Q$. Formally, for all $q \in Q$, $i \in \Sigma_I$ and $\sigma \in (\Sigma_I)^*$:

- $Out(q, \epsilon) = \epsilon$
- $Out(q, i.\sigma) = Out(q, i).Out(\delta(q, i), \sigma)$

For all $(q, q') \in Q \times Q$, $i \in \Sigma_I$ and $o \in \Sigma_O$, we write $q \xrightarrow{i/o} q'$ to denote $\delta(q, i) = q'$ and $Out(q, i) = o$. A Mealy machine $\mathcal{M}$ is **complete** if for all $q \in Q$ and $i \in \Sigma_I$, there exists $q' \in Q$ and $o \in \Sigma_O$ such that $q \xrightarrow{i/o} q'$.

b) From an Observation Table to the Mealy Machine

It is possible to build an automaton from the *observation table* if and only if it is closed and consistent. Given a closed and consistent *observation table* (S, E, T), the construction of the Mealy machine $\mathcal{M} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, Out)$ from (S, E, T) is the following:

- $Q = \{row(s) : s \in S\}$
- $q_0 = row(\epsilon)$
- $\delta(row(s), i) = row(s.i), \forall (s, i) \in S \times \Sigma_I$
- $Out(row(s), i) = T(s, i), \forall (s, i) \in S \times \Sigma_I$

For all $(q, i) \in Q \times \Sigma_I$, $\delta(q, i) \in Q$ because the table is closed i.e., for all $s \in S \cup S \cdot \Sigma_I$, $row(s)$ has its representative in S. We notice that δ and T do not depend on the representative of $row(s)$ because of the consistency of the *observation table*.

Figure 2.3 illustrate the Mealy machine associated to the Observation Table in Table 2.1. We have: $Q = \{xx, yx\}$, $q_0 = xx$, $\Sigma_I = \{a, b\}$ and $\Sigma_O = \{x, y\}$.

2.3 Active Learning in Practice

In the previous section, we have seen a quick overview of active learning and the original algorithm proposed in this area. In this section, we will discuss the practical use of active learning followed by a list of some interesting active learning algorithms.

2.3.1 How to Use Active Learning?

In active learning, three components are required: Learner, Mapper and Sut (see Figure 2.4). The Learner implements the model learning algorithm, but most model learning algorithms take as input a list of abstract messages. It communicates with the Sut by selecting a sequence of messages built from the input vocabulary. The messages present in the selected sequence are sent one by one to the Sut.

However, since these messages are only an abstraction, a component is required to concretize these messages; we call this component Mapper. The Mapper is not only responsible for the concretization of abstract messages from the Learner to the Sut, but it is also responsible for the abstraction of concrete messages from the Sut to the Learner. It means that, interactions between the Learner and the Sut are mediated through the Mapper. Figure 2.4 resumes the role of the Mapper.

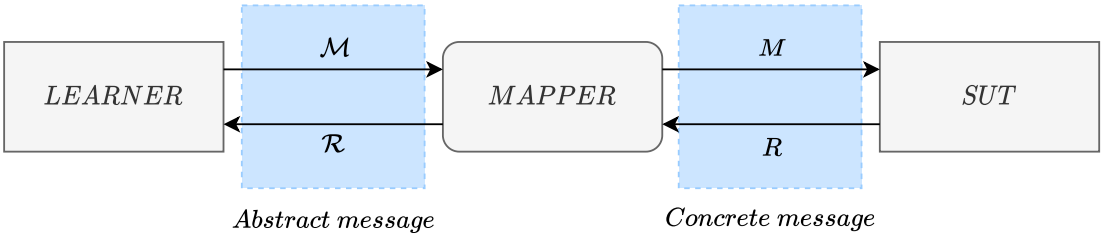


Figure 2.4: Principle of active learning.

There exist several algorithms of active learning; most of them were developped either as L* optimization or to learn non-deterministic automata. In the following section, we are going to see two categories of active learning: deterministic and non-deterministic active learning.

2.3.2 Overview of Active Learning Algorithms

Various active learning algorithms exist. Learning deterministic and non-deterministic automata is possible. The first and well-known active learning algorithm is L* (see sec. 2.2 for more details). From the original L* algorithm were derived different active learning algorithms, some of them offer performance improvement.

a) Deterministic Active Automata Learning

In addition to L*, there exists several deterministic active learning algorithms. A performance improvement was first proposed by Rivest and Schapire in 1989 [RS89].

When a counter-example $\mathcal{C}$ is returned, instead of adding all prefixes of $\mathcal{C}$ to the *observation table*, they only add a well-chosen suffix to the column E. L_M+ works in a similar way [SG09]. It divides $\mathcal{C}$ into its appropriate prefix and suffix such that the suffix contains the distinguishing sequence [3].

In 2014, Isberner Malte et al. proposed TTT, an algorithm which is an L*-like algorithm [IHS14]. In contrast to the traditional *observation table*, they used three tree-like data structures: a spanning tree, a discrimination tree and a discriminator tree. The spanning tree defines the access sequences of states, the discrimination tree is used to distinguish states and the discriminator tree is used to store the suffix-closed set of discriminators. In the worst case, TTT requires $\mathcal{O}(kn^2 + n\ log(m))$ membership queries where n is the size of the abstract model, k the size of the input vocabulary and m the longest counterexample returned by the Learner [Vaa17].

Recently, in 2022, two additional algorithms were proposed: L$^\#$ by Frits Vaandrager and Bharat Garhewal [VGRW22] and L^λ by Howar Falk and Steffen Bernhard [HS22].

b) Non-deterministic Active Automata Learning

Inferring non-deterministic automata is also possible using active automata learning. Researchers motivate the necessity of this kind of algorithm by two main problems of the deterministic active automata learning: (1) learning automata becomes infeasible for system with a large input and output alphabet [PA20], and (2) some systems behave non-deterministically (e.g., MQTT with the `PINGREQ` and `PINGRESP` messages [Sta14] and TFTP [KT14]).

The first algorithm to infer non-deterministic automata, called NL*, was proposed by Benedikt et al. in 2009 using membership and equivalence queries [BHKL09]. Instead of learning Deterministic Finite-state Automata (DFA) as the original L* algorithm, a Residual Finite-State Automata (RFSA) [DLT02] is learned.

In 2013, Pacharoen et al. proposed L$^\star_{NM}$ [PABS13] advancing NL* in performance. In 2014, Ali Khalili and Armando Tacchella proposed N*, an algorithm to infer non-deterministic automata in the form of Mealy Machines [KT14].

In 2020, Pferscher et al. proposed a new algorithm more complete than the past algorithm [PA20].

The basic idea of all non-deterministic active automata learning is to acquire all possible output of each input query. Multiple queries are required for a single sequence causing a decrease in performance in practice.

[3] A sequence allowing to distinguish at least two seemingly equivalent states of the conjecture and the real state machine.

2.3.3 Available Tools and Libraries

Many open-source tools implement the L* algorithm. The following is a brief overview of the most well-known tools:

- LearnLib[4]: a free and open-source Java library for active/passive learning [RSB05, IHS15]. It supports a large number of active learning algorithms such as L*, NL*, TTT, etc. and passive learning algorithms such as RPNI and OSTIA. It has been used for several model learning research publications [dRP15, FBJV16, FLP$^+$17, FJM$^+$20].
- pylstar[5]: a free and open source Python implementation of the L* algorithm. It focuses only on Mealy Machines and does not support other modeling formalisms such as non-deterministic automata. It has succesfully been used to infer automata for various protocols such as TLS [RLD22], Botnet protocols [Bos14] and Web Servers [Bos16].
- AALpy[6]: a lightweight active learning library written in Python [MAP$^+$22]. It implements several active and passive learning algorithms and modeling formalisms such as deterministic, non-deterministic and stochastic automata. This tool was successfully used to analyze several Bluethooth Low-Energy implementations [PA21, AKM$^+$22], and also to learn Input-Output behavior of Recurrent Neural Networks (RNN) [XWQH21].
- DroidStar[7]: a free active learning tool written in Java. It implements the L* algorithm only. It proposes and implements a new *EQ* method called Distinguishing Bounds [RLM$^+$18]. It is specially written for synthesizing behavioral specifications, in Android application programming, for event-driven framework classes that explain how and when their callbacks occur [RLM$^+$18].

2.4 Model Learning and Application

The applications of model learning have been multiple over the past year. Applications are found in black-box contexts as well as in grey-box and white-box scenarios (e.g., [MSTV$^+$22]). This field is becoming one of the well-established tools in the toolbox of the software engineer trained in security and formal methods. In this section, we describe the broad range of uses of model learning into different categories.

[4] `https://github.com/LearnLib/learnlib.git`
[5] `https://github.com/gbossert/pylstar.git`
[6] `https://github.com/DES-Lab/AALpy.git`
[7] `https://github.com/cuplv/droidstar.git`

2.4.1 Verification and Validation

a) Security

The state machine resulting from model learning can be used to identify vulnerabilities if they exist. Model learning is often used to automate state exploration of the SUT in a structured way. Several vulnerabilities were found in this manner over the past years.

In 2011, Chia Yuan Cho et al. proposed an approach based on active learning for concolic exploration of protocol behavior for discovering the so-called deep states in the SMB and RFB protocols behavior [CBP+11]. They applied their techniques to four different implementation of Remote FrameBuffer (RFB) [RL11] and Server Message Block (SMB) [Win] protocols. They found many vulnerabilities in different applications (Samba 3.3.4, Vino 2.26.1 and RealVNC 4.1.2).

In 2014, Georg Chalupar et al. proposed a method based on active learning and a LEGO robot to reverse engineer hand-held smartcard readers for Internet banking by interacting physically with the smart cards [CPPDR14]. A critical vulnerability, allowing a Man-in-the-Browser attack, was detected in a hand-held smartcard reader (e.edentifier2).

In 2015, Joeri de Ruiter and Erik Poll used $L^\star$ to infer TLS state machines for different TLS servers [dRP15]. They discovered various anomalies and security issues in several well-known TLS implementations.

In 2018, the same method was used, first, by McMahon Stone et al. to analyze the IEEE 802.11 4-Way Handshake protocol [MSCR18] and also by Wesley van der Lee et al. for detecting vulnerabilities in mobile applications [vdLV18]. Both papers detected many critical vulnerabilities.

In 2019, Jiaxing Guo et al. automatically detected vulnerabilities in the implementation of the IPsec protocol [GGCW19]. They combined model learning and model checking to completly automate the search of their vulnerabilities and RFCs violation properties.

In 2020, Fiterau-Brostean et al. proposed the first comprehensive analysis of DTLS implementations using active learning [FJM+20]. They uncovered several non-conformance bugs and security vulnerabilities in different DTLS implementations. Some of these vulnerabilities affect also the TLS part of these implementations.

b) Safety and Correctness

Analyzing complex state machines requires a rigorous method such as model checking. As described in Figure 2.5, the basic idea of combining model learning and model checking is to use model learning techniques to learn a system model and

then use model checking to verify if the learned model satisfies the properties described in the protocol specification.

If the properties do not hold, a counter-example trace is provided by the model checker. This counter-example trace is a sequence of messages violating one of the given properties. On the other hand, if the model checker terminates without returning any trace, then we conclude that all the given properties are satisfied by the SUT.

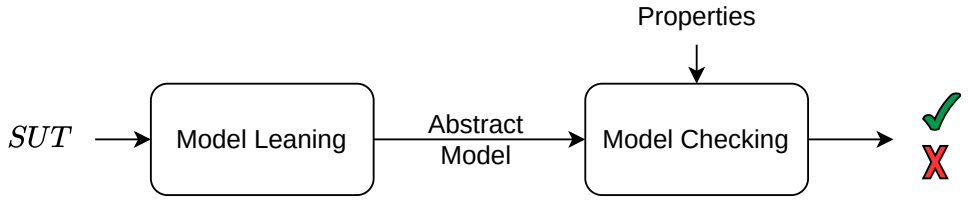


Figure 2.5: Principle of model learning combining with model checking.

This method was first applied to analyze different TCP implementations [FBJV16], then to SSH [FLP+17] and IPsec [GGCW19]. For the model checker tools, in the three studies, they used NuSMV[8] a reimplementation and extension of Symbolic Model Checking [CMCHG96]. Several RFC property violations were detected in an automatic way.

In 2021, Qinying Wang et al. [WJT+21] proposed a novel approach to systematically and automatically evaluate IoT protocols implementations using active learning and formal verification such as Tamarin Prover [MSCB13]. Their approach detected 252 property violations.

2.4.2 Learning-based Testing

Model-based testing is a structured testing technique in which models are used to guide the testing process [Mei18]. Test suites are generated using the model and conformance testing technique is used to test if the generated test suites are verified by the SUT.

Unlike model-based testing, learning-based testing allows to test systems without pre-existing models. It was applied for example to Web-based systems [RMSM09]. Models are extracted using model learning and then test suites are generated using the extracted model.

This line of application has been used for different types of systems such as Event-B [DIMS12, DIS12] and embedded components [SG14]. Hagerer et al. proposed the earlier application of model learning (active learning) for testing telecom-

[8] `https://nusmv.fbk.eu/`

munication systems [HMN+01, HHNS02]. Aichernig et al. discussed the relation between testing and learning [AMM+18].

Shahbaz and Groz used model learning for integration testing. They infer model and use them for test case generation [SG14]. Choi et al. use model learning for testing graphical user interfaces of smartphones and tablet application [CNS13]. Meinken et al. propose LBTest tool for test case generation, test execution and test verdict construction [MS13]. Their tool is based on model learning, model checking and random testing.

2.4.3 Learning-based Fuzzing

Recently, several researchers have proposed a novel network protocol fuzzing approach based on model learning. The main objective of black-box fuzzing is the assurance of sufficient test coverage. This novel approach is proved having a higher code coverage and vulnerability discovery than other well-known fuzzing tools such as AFLNWE, AFLNET and STATEFUZZ [PLJZ22]. The approach, as described in Figure 2.6, is based on two steps: (1) learning the abstract model and (2) fuzzing.

Model learning is used to infer the state machine of the SUT which is used to guide the fuzzing process. At the fuzzing stage, test cases are generated using the state machine, the MAPPER and the *Fuzzing Technique* components. The *Fuzzing Technique* component implements the mutation strategies like bitflip, shuffling, erasing, splicing, swapping and inserting. The results of each generated test case are checked by using the CHECKER component.

Aichernig et al. proposed a black-box learning-based fuzzing technique based on active learning and applied their method to test MQTT brokers implementation [AMP21]. They only learned one generic state machine for all MQTT brokers to fuzz every MQTT brokers implementations.

Unlike Aichernig et al., Andrea Pferscher et al. and Zhan Shu et al. proposed a fuzzing method based on individual learned state machines for every specific protocol studied. They successfully applied their method to different Bluetooth devices and Telnet implementations [PA22, SY22].

In contrast to the previous methods, Yan Pan et al. proposed a grey-box learning-based fuzzing method [PLJZ22]. The first step of their approach is done using black-box approach, while the second step uses grey-box approach. The grey-box is needed when checking the results of each generated test case. The CHECKER requires two additional components: *Address Sanitizer* for memory bugs and *Differention Checker* for semantic bugs.

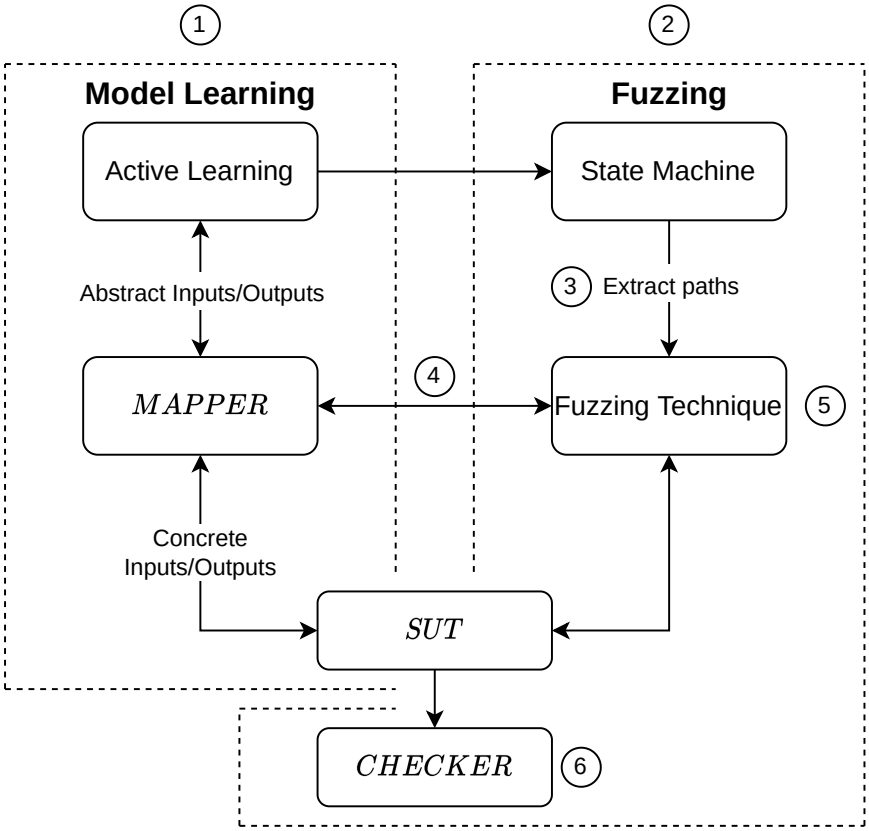


Figure 2.6: Principle of model learning combining with fuzzing.

2.4.4 Stack Fingerprinting

Another recent application of model learning is stack fingerprinting [DS22]. They used model learning to infer the SUT's state machine and then they applied different method such as the method proposed by Guoqiang Shu et al in 2011 which introduced a formal methodology for network protocol fingerprinting [SL11].

Guoqiang Shu et al. proposed different method to compute protocol network fingerprint based on Finite State Machine (FSM) or an extension of the FSM formalism, Parameterized Extended Finite State Machine (PEFSM), for modeling network protocol. From these models, they computed the so-called *distinguishing sequences* to distinguish one implementation from another.

Combining this method and model learning, during his master thesis, Jansen Erwin successfully compute the fingerprint associated to many TLS server implementations [JVdRP21]. This method requires computing the state machine of different TLS server implementations using model learning, and then applied the method proposed in [SL11] to compute the fingerprint of each studied implemen-

tation.

Andrea Pferscher et al. proposed a similar method to fingerprint Bluetooth Low Energy devices using active learning [PA21]. Unlike [JVdRP21], they only studied five BLE devices and manually computed the fingerprint associated to each BLE devices studied.

Chapter 3

Methodology – Automatic Validation and Black-box Analysis of Protocol Implementations

In this chapter, we propose a genereralized methodology based on active learning to analyze the state machine of a given protocol implementation. We begin this chapter with the practical challenges in using active learning, sec. 3.1; followed by the categories of state machine bugs that can affect every protocol implementations, sec 3.2. Then, in sec. 3.3, we discuss the minimum necessary knowledge of the studied protocol for the inference process and we also discuss three methods to detect bugs automatically. Finally, in sec. 3.4, we propose a generic method in writing the Mapper component.

3.1 Practical Challenges in Active Learning

We identified the challenges for the practical application of active learning. Active automata learning is characterized by its proactive way of querying the Sut. It requires some way to achieve this query-based interaction. Therefore, using model learning in practice is challenging.

3.1.1 Connection Independence

Active learning requires interactions with the Sut. The independence of the established connections means that the previous connection must not have influence (or an impact) on the current connection.

For this purpose, reseting the Sut after each membership query may be required to bring the Sut to its initial state. In other words, opening and closing

the SUT after each membership query would be a solution; it is naturally the case when learning the client-side state machine (see sec. 5.2.1).

For the server-side and for the learning performance, it is better to avoid opening and closing the SUT after each membership query because it slows down the learning process. Thus, the LEARNER often assumes that there is no dependence between connections in practice. However, reseting the SUT after each membership query becomes mandatory if the server crashes and/or is unable to ensure its availability.

3.1.2 Equivalence Query

The equivalence query plays an important role for the accuracy of the resulting state machine. Since the SUT's state machine is not known, equivalence queries do not really exist in practice, so we must approximate them by using membership queries. Conformance testing has been used to simulate equivalence queries [LY96].

A competition, called ZULU, was launched in 2010 attempting to intensify research on this field [CHJ09]. They justify such a competition by the need for an equivalence query method which garantees the correctness of the learned state machine. Sometimes state machines are huge, up to $20,000$ states the largest learned state machines to the best of our knowledge [RMSM09]. The winning solution for the ZULU competition is discussed in [HSM10]. Currently, we have not studied their winning solution because of the lack of details and accessible implementations.

If the state machines are not big, there exists several equivalence query methods that can deal with them, e.g., RandomWalk, W(p)-method and Distinguishing Bounds. Thus, we use these equivalence query methods because we do not have gig state machines (our biggest state machine is around 500 states and is related to `AsyncSSH` server).

3.1.3 Efficiency and Convergence

The number of queries sent during the learning process is correlated with the size of the input vocabulary. Several research papers describe methods to improve the number of queries and speed up the learning process.

In practice, each membership query of a given sequence $\mathcal{S}$ may require $t \times |\mathcal{S}|$ seconds if t is the timeout value in second. In such a case, minimizing the number of required membership queries and avoiding useless waiting (timeout) is the key to performance.

Several researchers proposed optimized algorithms to save membership queries (more details are discussed in sec. 2.1.2) [RS89, SG09, IHS14, VGRW22]. Practical tricks are discussed in [RLD22, HTJV15] to optimize the learning process.

The main goal of active learning algorithms is to infer a finite automaton. It is impossible to learn infinite automaton. It is worth noting that the LEARNER should make sure of the convergence of the learning process. For example, when learning SSH implementation state machines, we found state machines that could not be represented by a finite automaton (see sec 5.6.1).

3.1.4 Robust and Flexible MAPPER

The MAPPER plays an important role in the learning process: concretization and abstraction of messages between the LEARNER and the SUT. The MAPPER should not only be able to execute the protocol in the normal context, but also concretize messages in any order under any circumstances. Thus we seek a robust and flexible MAPPER. It means, from its knowledge at a given moment, it should be able to build a message complying with the protocol specification, in terms of message format and content, while using all the present knowledge.

The final results and its interpretation depends on how the MAPPER is written/built. Message concretization is important as its abstraction. When the MAPPER is not robust, non-deterministic behavior can occur not because the SUT does not behave deterministically but because of the MAPPER. For example, when working on TLS state machine inference, the TLS MAPPER can misinterpret an encrypted message. Thus it produces an unexpected response with a low probability. We found that when `scapy` could not properly decrypt a TLS packet, it tagged the packet either as `Unknown` or `ApplicationData` or `FatalAlert`, etc. depending on the random value of the packet type field it detected after the decryption.

3.1.5 Non-determinism

The most important requirement for deterministic active learning is that the SUT behavior must be *deterministic*, relative to the selected input vocabulary. For a given stack and a given set of parameters, a given input abstract message sequence should always produce the exact same abstract output sequence.

We identified three sources of non-determinism: non-deterministic SUT, timeout value and lack of robustness from the MAPPER.

The first case can be due to several root cases: interferences between connections, unavailability of the SUT, slowing down of the SUT to assure its task, or simply because the SUT is non-deterministic. We encountered such problems with `OpenSSH`, `AsyncSSH` and `paramiko`.

If the MAPPER is not robust enough, it can misinterpret the responses from the SUT which can lead to non-determinism behavior. An example of such problem has already been discussed above (see 3.1.4).

The most frequent cause of non-determinism comes from an insufficient timeout value in the socket while getting responses from the SUT. In this case, the LEARNER can miss additional responses from the SUT.

Thus, to analyze communication protocol implementations using active learning, we must overcome each of the five challenges listed above. These challenges concern only the LEARNER-side, but there exists also many challenges in state machines for the SUT's side. In the following section, we discuss three categories of the state machine attacks that can affect all protocol implementations.

3.2 Challenges in State Machines of Communication Protocols Implementations

In state machines, undesired transitions are indications of bugs which may lead to critical vulnerabilities in the system. These vulnerabilities can be further exploited by the attacker to degrade the security of the communication (e.g., availability, confidentiality, integrity, etc.) [YR15, BDD+19]. In the usual cases of communication protocols, the following types of bugs can exist in a state machines model:

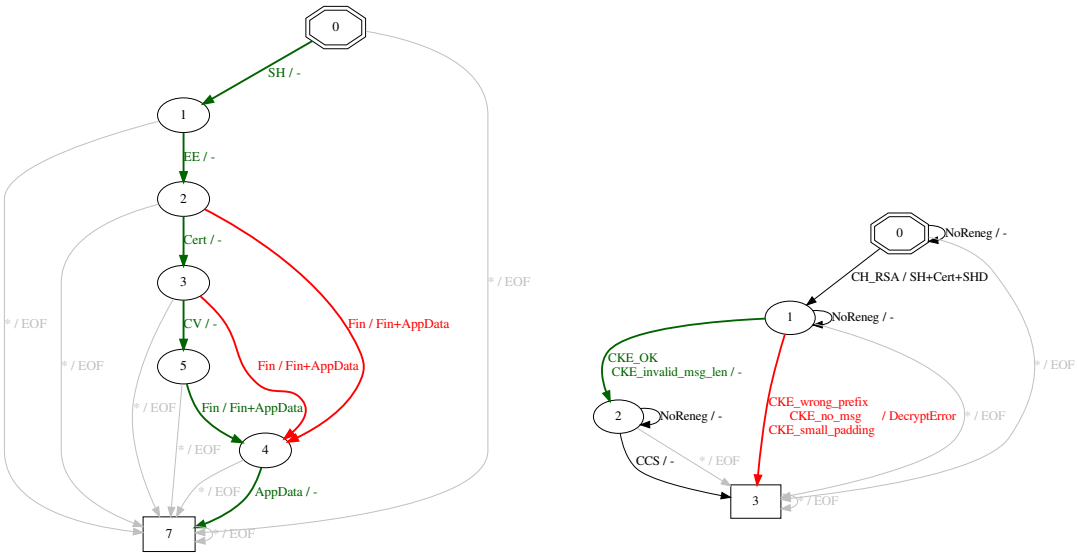
- undesired flows,
- undesired states and
- infinite executions in the state machines.

These kinds of bugs are an indication of a defective implementation where each state of the implementation is not checked against every possible input messages. Such an implementation first processes the messages and then checks if conditions are verified instead of processing the messages if and only if conditions are verified.

3.2.1 Undesired Flows

Undesired flows are transitions from a valid state to another valid state using invalid transitions (see Figure 3.1). Such path may lead to critical vulnerability when it allows to reach an important state by bypassing necessary transitions and thus violating protocol specifications e.g., CVE-2020-24613 described in Figure 3.1a.

Particularly, if the sink state is the end state of such flow, then undesired flows are synonyms of undesired responses to an invalid input (see Figure 3.1b). Such replies do not necessarily affect the security of the system, but they might introduce abnormal behavior violating protocol specifications. In some cases, such a bug may lead to critical vulnerability, when it allows an attacker to access an observable oracle e.g., CVE-2017-1000385 [Ble98, RD10, PY04, ASS+16, BSY18].



(a) CVE-2020-24613, a server authentication bypass in wolfSSL TLS 1.3 clients, up to version 4.4.

(b) CVE-2017-1000385, the erlang OTP-20.0 server exhibits a Bleichenbacher oracle

Figure 3.1: Examples of Undesired Flow.

3.2.2 Undesired States

States are called undesired if they are only reachable using unexpected sequence of messages from the initial state. It means that no expected sequence (described in the specification) allows to reach an undesired state. Hence, such a state is necessarily a vulnerability or a deviation from the specification.

For each state machine, we expect to have a unique sink state, which is related to the termination of connection. Hence, if there exists multiple sink states, only one of them is the desired sink state and all the others represent a specification violation.

For all the undesired sink states, the connection is always on and no message seems to be able to terminate the connection. If the server does not implement a protection for these unclosed connection (e.g., set up a timeout), it thus allows an attacker to carry out a denial of service attack.

Figures 3.2 and 3.3 illustrate two categories of undesired states. The state 4 in Figure 3.2 is an undesired state which allows an attacker to bypass server authentication. The state 3 of Figure 3.3 allows an attacker to lead a denial of service attack.

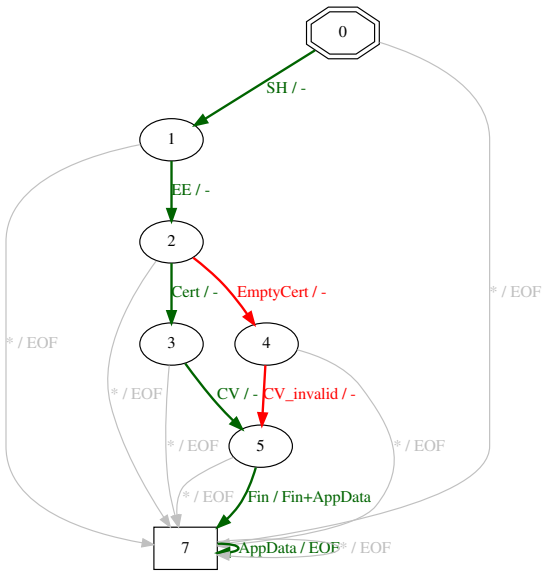


Figure 3.2: CVE-2022-25638, a server authentication bypass in wolfSSL TLS 1.3 clients, present up to version 5.2.0.

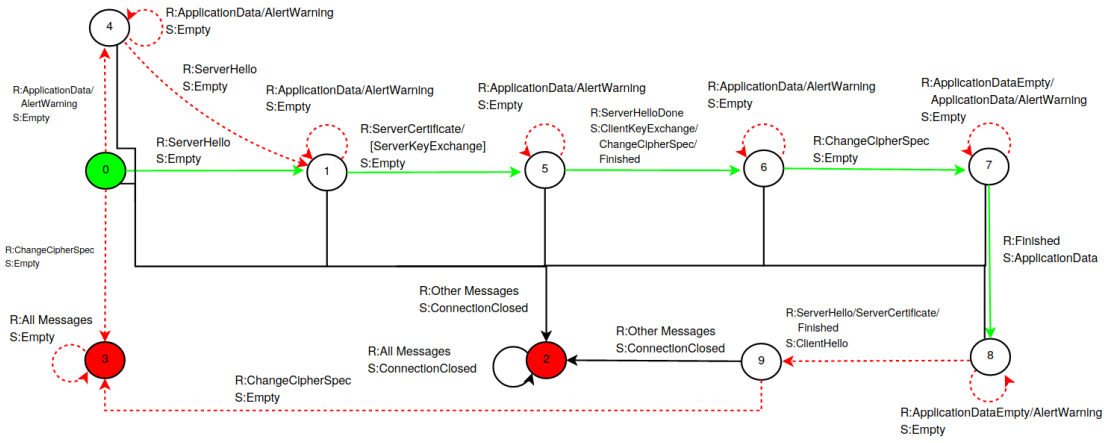


Figure 3.3: SChannel vulnerability present in Windows 8 and 10 (TLS 1.2) [YS19].

3.2.3 Infinite Executions in the State Machine

Infinity in the state machine are transitions or sequence of messages which can be reapeted indefinitely and never lead to connection termination.

In some cases, it may be considered as an expected behavior (behavior authorized by the specification) but if it is not authorized, it may lead to a denial of service attack e.g., CVE-2020-12457.

Infinity can also manifest itself in another way: the inference process may never converge for a given implementation, which means its state machine can not be represented using a finite state machine model. In our opinion, a clean specification (and the corresponding implementations) should always lead to a finite representation. It is the case for most of the studied implementations. We leave it as an open discussion for the community to enforce this constraint.

An implementation of a given protocol may have an infinite state machine for different reasons. One common reason we have encountered is described in situations such as the one depicted in the Figure 3.4. If the message $\mathcal{A}$ is sent n times followed by the message $\mathcal{B}$, then the SUT responds first by the message $\mathcal{X}$ followed by n times the message $\mathcal{Y}$.

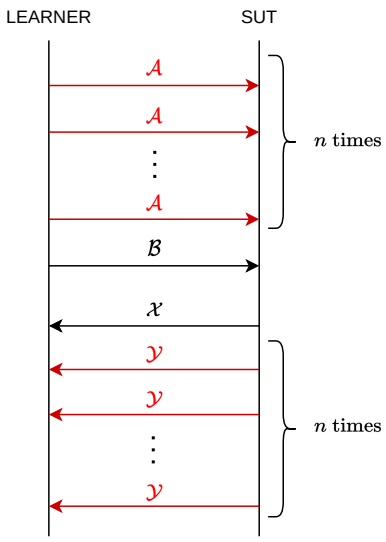


Figure 3.4: Example of an exchange causing infinite state machine.

We detected this kind of problem when learning the state machines of several SSH implementations such as `OpenSSH`, `AsyncSSH`, `wolfSSH` and `sshd-lite`. For example, when initiating a key refresh (re-key) in the connection layer with `OpenSSH` version 8.9.p1 server, we can highlight this problem with the following sequence input/output messages: `KEXINIT/KEXINIT`, `CH_OPEN/NoRSP` n times,

`DH_INIT/DH_REPLY+NEWKEYS+(OPEN_CONFIRMATION` n times)[1].

After describing all the practical challenges concerning the LEARNER and the SUT, we are going to discuss in the following section the knowledge required to analyze protocol implementation using active active learning.

3.3 Analysis of Communication Protocols Implementations in Black-box

Analyzing the implementation of a communication protocol in a black-box approach requires a lot of knowledge about the studied protocol depending on the chosen method. Using *Active learning* requires building a MAPPER. Building a MAPPER implies knowledge about the studied protocol. In this section, we describe such knowledge. We discuss also different possible methods to identify bugs in the resulting state machine.

3.3.1 Useful (and Minimum) Knowledge on the Studied Protocol

Active learning is a powerful method to learn state machines of a given protocol implementations (see sec. 2.1.2 for more details). A legitimate question about this method is: *What knowledge of the target protocol do we need to use this method?* In this section, we try to highlight the useful and minimun knowledge to learn the state machines of a given protocol implementations.

Since communications with the SUT are required, we thus need to be able to build valid messages in valid contexts. And, because abstractions of the SUT's response are also required, interpretations of the SUT's response are also mandatory (see sec. 2).

In the *active learning* context only the following knowledge is required:

i) the structure of all messages exchanged between client and server,

ii) the information needed to handle communication correctly (from message sent and received), and

iii) the application logic.

[1]We give more details about these SSH messages in sec. 5.2.3.

a) Message Structure

The LEARNER should know all possible messages that can be used during the learning process and be able to build valid messages for a given context of the communication. These are fundamental for this approach.

First, the LEARNER should select a list of messages of interest, then learn the packet format of each selected message.

Knowing the message structure is very important to build a valid message and ensure messages will be parsed by the SUT. The LEARNER should collect every useful parameter of a given message and fill them according to the specification.

For this purpose, parser generators would be a good solution that we investigated during a part of this thesis by building a platform to compare several existing parser generator tools [LNR21].

b) Find and Record Information

The first step is necessary but not sufficient. During the communication, past information may be needed to build the current packet. Thus, we need to find and record these over the execution lifetime for network protocols. Therefore, the LEARNER should be able to answer the following questions: *(a) Which information should be recorded for every packet sent and received to handle communication correctly? and (b) When and how to use them?*

Figure 3.5 illustrate well the importance of the knowledge described in this section. It represents the basic Diffie-Hellman protocol to exchange secret key and is the first key exchange published publicly in 1976 [Dif76].

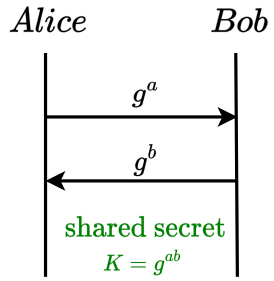


Figure 3.5: Diffie-Hellman key exchange scheme

To keep it simple, g is public; the protocol states that *Alice* generates an integer a, computes g^a and sends it to *Bob*. When *Bob* receives the message from *Alice*, he repeats similar actions as *Alice*, i.e., he generates an integer b, computes g^b and sends it to *Alice*. Then, both parties share a secret $K = g^{ab}$. The next messages rely on the computed shared secret K.

From *Alice*'s point of view, there are three important bits of information to record g, a and g^b (g, b and g^a for *Bob*) over the execution lifetime of the protocol. Two problems occur without considering these useful informations:

- whithout memorizing g, *Alice* and *Bob* cannot execute the protocol correctly because they are both unable to generate their message;
- whithout memorizing (a or g^b) for Alice's context, she can never compute the shared secret $K = (g^b)^a$, then she will never be able to build a valid message after exchanging the public Diffie-Hellman value.

We store these useful information in an *"internal state"*. So everytime the Learner has to build a message, it will find useful information in the *internal state* if present. If absent, the Learner uses default values.

c) Protocol Logic

Protocol logic operations are necessary for the transformation of the messages before or after receiving and/or sending a message. They might be also required for updating the *internal state*.

These operations must be performed to execute the communication correctly. For security protocols, key derivation, encryption, signature, MAC value, etc. are required. TCP is another good example to illustrate this where incrementation of the sequence number is required for each packet sent after the first `SYN` message [Joh81].

This knowledge is necessary for the Learner to know how and when to use information recorded in the *internal state*. These operations can take place when building a message (such as incrementation, encryption, signature, etc.), receiving messages from the Sut (e.g., decryption of the received packet) and updating the *internal state*.

d) Discussion

To summarize the useful knowledge when using *active learning*, Figure 3.6 sums up the information to consider and how to use it.

When building a message, the Learner should first learn the message structure, define all parameters corresponding to the current message and fill in them little by little starting from deriving the associated value from the *internal state* corresponding to a given parameter; then fill in the remaining parameters while remaining compliant with the specification. All these steps must be done while applying the protocol logic whenever applicable. Information that is useful for the execution of the communication should be stored in the *internal state*.

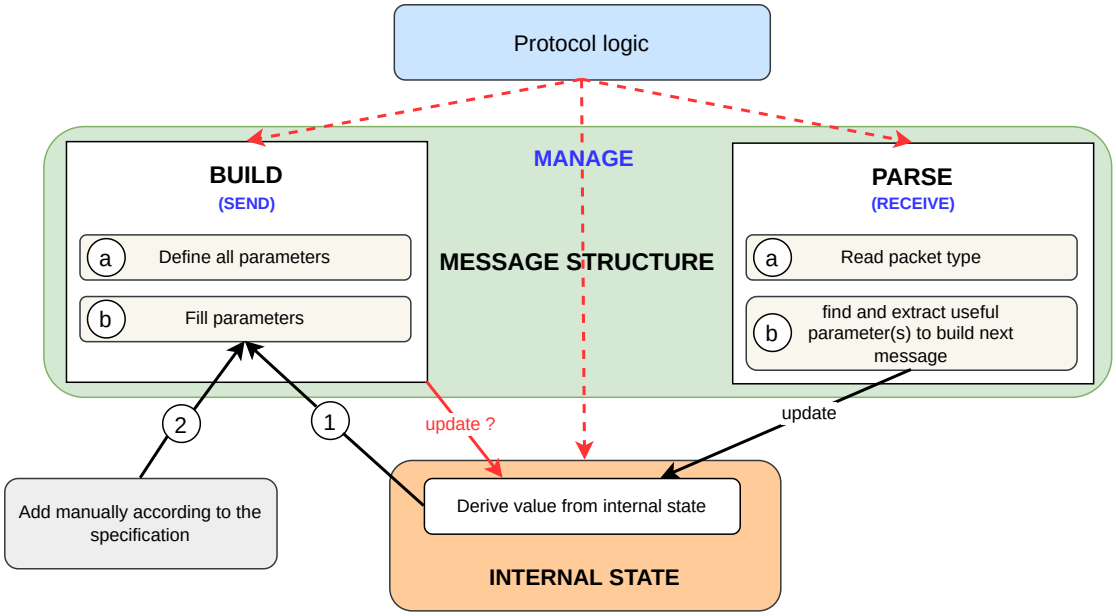


Figure 3.6: Useful knowledge on the studied protocol and description of how to use them for active learning state machine inference.

When receiving a message from the SUT, the LEARNER should first apply protocol logic if needed, then read packet type and, finally, find and store in the so called *internal state* all useful informations present in the current packet.

3.3.2 Finding Bugs from the Learned State Machines

Model learning is used to infer state machines for protocol implementations. However, finding bugs from the resulting state machines requires additional processing. Sometimes, the LEARNER has to deal with complex state machines, then robust and efficient method should be adopted, such as model checking which was used to analyze SSH state machines [FLP+17].

Analyzing the resulting state machines may require further knowledge of the studied protocol. It is also possible to analyze state machines without contextual information. In the following, we detail three strategies to analyze state machines successfully.

a) Model Checking

Model checking (see sec. 2.4.1) requires additional knowledge of the studied protocol because the user should define the properties that must be verified by the associated abstract model of the SUT (the resulting state machine).

These properties are manually derived from the specification of the protocol. An in-depth knowledge of the protocol is required to confirm bugs in the state machine, in contrast with other methods. The user must be aware of how the MAPPER was written when defining properties.

This method was successfully used to find specification violations in several protocols [FLP+17,FBJV16,GGCW19]. Because model checking always stop after finding the first counter-example, these applications can not achieve exhaustivity in finding bugs in the learned protocol implementations.

b) Happy Path

Most vulnerabilities/bugs research on state machines are based (directly or indirectly) on this method. This method is straight forward and only partial knowledge is required, compared to model checking. It only requires the knowledge of the correct or expected behavior (the legal state transitions) of the protocol for a given scenario (a.k.a. happy path).

All transitions/paths different from the happy path are considered vulnerabilities or specification violations.

The limitation of using this method is the lack of precision in the analysis of the error messages if required. Moreover, if the state machine is complex, it then becomes difficult to find vulnerabilities or bugs by just visualizing the state machine.

c) Without Considering Any Information

It is also possible to analyze state machines without considering any information about the studied protocol, but the results of the analysis can be less accurate. We distinguish two methods for this category:

(i) considering a reference implementation (if there exists one), and

(ii) comparing different state machine of the implementation of the same protocol.

In (i), the first step should be the choice of the reference implementation and the inference of its state machine. Then, all state machine are compared to this state machine and all discrepencies from the reference implementation state machine are considered as vulnerabilities or bugs.

In (ii), finding vulnerabilities and bugs is achieved by comparing several different state machines of the implementing the same protocol (of course, using the same input vocabulary).

By construction, except for model checking, these methods require manual inspection to characterize the nature of the detected deviations.

Before closing this chapter, we would like to discuss, in the following section, one of the most important component of the active learning approach, the MAPPER.

3.4 MAPPER

In this section we discuss our proposition to solve the challenge related to the MAPPER discussed in sec. 3.1.4. We identify two challenges when writting a MAPPER, the first challenge is how to build a robust and flexible MAPPER, and the second challenge is how and when to update its internal state.

3.4.1 Flexible MAPPER

Using *active learning* requires a robust and flexible MAPPER. The MAPPER implements the protocol while remaining modular and robust. Figure 3.7 gives more details on how a MAPPER should be written.

To find the accurate class/method for the abstract or concrete messages, the MAPPER refers to its first component. Then, in case of concretization, the MAPPER considers the useful knowledge related to the message and builds the concrete message by interacting with the *internal state*.

In case of the abstraction of a concrete message, the MAPPER should parse the packet first and then extract the useful knowledge if there exists and finally update the *internal state*. The update of the *internal state* is done according to sec. 3.4.2.

Each message is implemented in a class[2] wich implements at least `build`, `parse` and `update` functions. This structure allow us to (i) concretize the abstract message using the maximum amount of information from the *internal state*, and (ii) be flexible enough to send arbitrary messages at any state of execution of the protocol.

3.4.2 When and How to Update the Internal State?

The *internal state* has an important role when building a MAPPER. The choice of how we deal with it affects the final results of the learning process. If we do not manage it correctly, erronous results will be produced because of miscaculations or forgotten updates. If the LEARNER updates it incorrectly, then it will add some weird additional states to the final state machine, which may complicate the

[2]We refer to a class of a programming language.

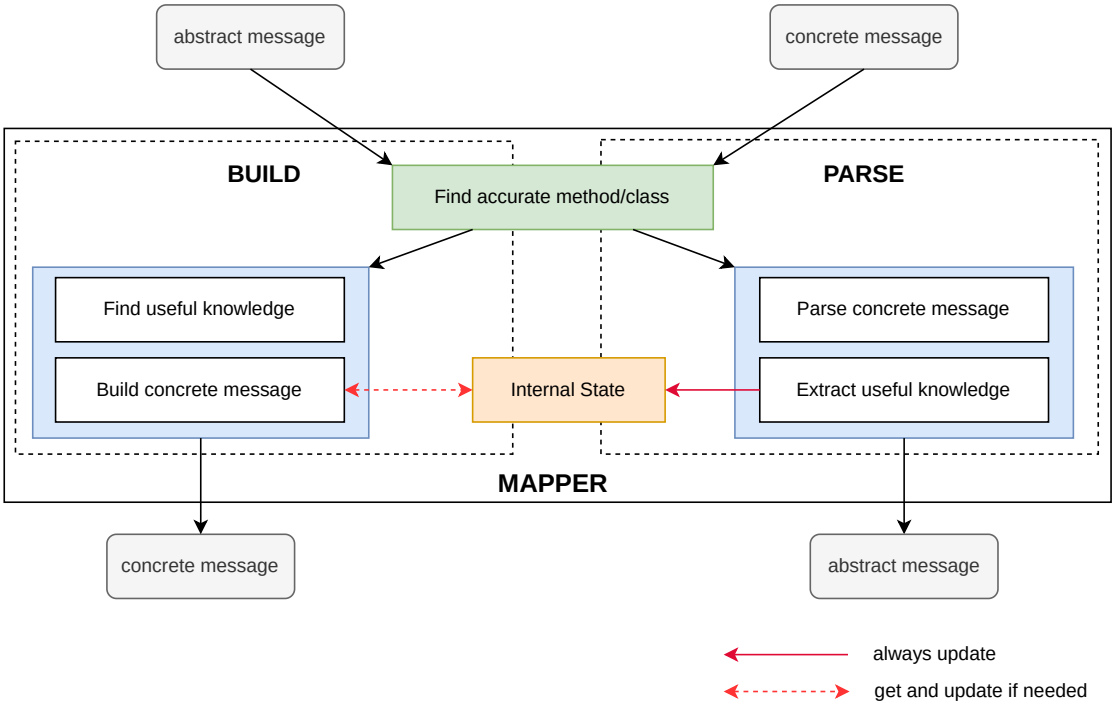


Figure 3.7: Abstract model of a MAPPER.

interpretation of the final result. The interpretation of the final state machine must depend on how the MAPPER is written.

Suppose we have to deal with the protocol described in Figure 3.8a. A legimate question is how and when we should update the *internal state* of the MAPPER? To tackle this problem, we consider three solutions:

Strategy 1: update the *internal state* every time we build and parse messages;

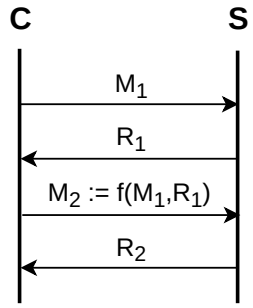
Strategy 2: add additional vocabulary for updating *internal state*; and

Strategy 3: consider additional knowledge on the studied protocol and add them to the MAPPER.

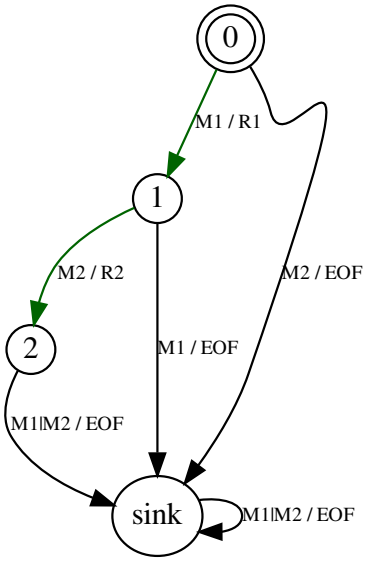
To illustrate each strategy identified above, we consider a buggy implementation of the protocol described in Figure 3.8a. We suppose that the implementation indefinitely accepts the message M_1.

a) Strategy 1: Always update

The *internal state* is always updated after each call to the `build` and `parse` functions. This first method is very simple to implement but it may cause additional



(a) Simple protocol to illustrate the problem of *internal state* update. It is a simple client-server protocol where $\mathcal{M}_1$ and $\mathcal{R}_1$ are non-static (they contain random value) and message $\mathcal{M}_2$ depends on $\mathcal{M}_1$ and $\mathcal{R}_1$.



(b) Clean and expected server-side state machines to the protocol described in Figure 3.8a.

Figure 3.8: Example protocol and its server expected state machine.

states to appear in the state machine such as the state 3 in the Figure 3.9a for example.

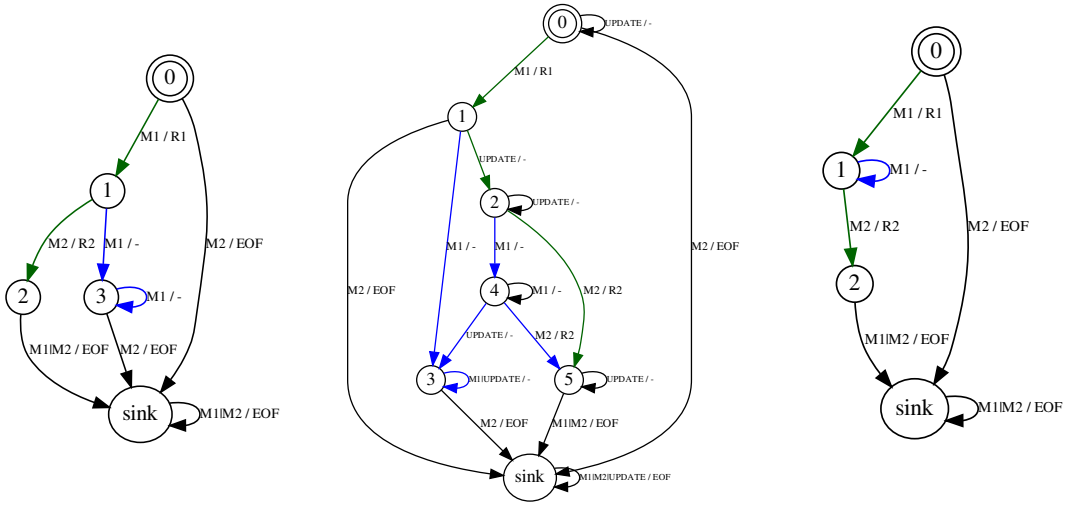
Analyzing the state machine is quite easy if the state machine is relatively simple and quite difficult otherwise because of the additional states due to the way the *internal state* is updated.

In Figure 3.9a, in state 3, the SUT indefinitely accepts the message $\mathcal{M}1$ and the message $\mathcal{M}2$ is rejected. It means that sending the message $\mathcal{M}1$ in state 1 has changed the state of the *internal state* of the MAPPER, hence $\mathcal{M}2$ is accepted in state 1 and rejected in state 3.

b) Strategy 2: Including additional and dummy messages

In contrast to the first proposed strategy, the update of the *internal state* is explicit and is done using additional and dummy message(s). In case of Figure 3.9b, the *internal state* of the MAPPER is done each time the dummy message `UPDATE` is sent.

Especially for this strategy, an intermediate variable which is used to store the last useful parameter of the message $\mathcal{M}1$, is required. And when the dummy mes-



(a) State machines using strategy 1.

(b) State machines using strategy 2.

(c) State machines using strategy 3.

Figure 3.9: State machines of the same implementation of the protocol described in 3.8a but with three different strategies of updating the *internal state*.

sage `UPDATE` is sent, the *internal state* is updated using the intermediate variable.

The resulting state machine is complex but the interpretation of the state machine is simple and explicit. In Figure 3.9b, in state 1, the message $\mathcal{M}2$ is rejected because the *internal state* is not yet updated after sending and receiving $\mathcal{M}1$ and $\mathcal{R}1$ respectively and the MAPPER is unable to correctly build the message $\mathcal{M}2$. However, it is accepted in state 2, because, in this state, the *internal state* was already updated, then the message $\mathcal{M}1$ is built correctly.

c) Strategy 3: Adding intelligence to the MAPPER

This approach require more knowledge of the studied protocol and is thus more difficult to implement. The *internal state* is updated if some conditions are verified. It means, that it requires the injection of some additional information to the MAPPER to handle perfectly the update of the *internal state*.

The resulting state machine is simpliflied with respect to the strategy 1 and 2 making easier the interpretation of the state machine compared to other methods.

Considering this strategy for our toy protocol with a flawed implementation (see Figure 3.8a), the *internal state* is updated in the client context if and only if after sending $\mathcal{M}1$, it receives $\mathcal{R}1$. It means that, sending message $\mathcal{M}1$ in state 1 does not trigger the update of the *internal state* (the *internal state* of the MAPPER remains unchanged).

Actually, almost all the research papers adopted the strategy 1 for the update of the *internal state*. For example, it is the case of `statelearner`[3] and `DTLS-Fuzzer`[4], tools developed for analyzing TLS and DTLS implementations [dRP15, FJM+20].

In the next chapter, before giving details on our use-case of the methodology described in this chapter, we will discuss the equivalence query method which plays an important role in the precision of the resulting state machines. This chapter allows us to choose an appropriate equivalence query method to use in our use-case.

[3] `https://github.com/jderuiter/statelearner.git`
[4] `https://github.com/assist-project/dtls-fuzzer.git`

Chapter 4

Comparative Study of Equivalence Query Methods

Equivalence query methods represent one of the biggest challenges of model learning. Since the resulting state machine is only an approximation, the equivalence query method is fundamental for the accuracy of the final results of the learning process (see sec. 3.1 for more details).

In this chapter, we discuss several known equivalence query methods and we also propose a new method. Sec. 4.1 describes the principle of an equivalence query method. In sec. 4.2, we discuss three equivalence query methods: RandomWalk, W(p)-method and Distinguishing Bounds. Then, in sec. 4.3, we propose a new method based on Distinguishing Bounds method. Finally, in sec. 4.4, we discuss our benchmark results while comparing three equivalence query methods.

4.1 Equivalence Query Methods

Equivalence queries do not really exist in practice, so we must approximate them. Several methods have been developed, such as the W-method [Cho78], the Wp-method [FvBK+91], Random Walk [Lá93] and Distinguishing Bounds [RLM+18].

The equivalence query method generates test cases, which are executed both on the SUT using membership queries and on the hypothesis. In case the executions produce different results, the corresponding message sequence is a counter-example invalidating the hypothesis.

To test the equivalence between the built hypothesis and the SUT's state machine, we need three components: the actual hypothesis (built by the LEARNER), a Test Generator and Analyzer, and the MAPPER. As described in Figure 4.1, the testing process follows the steps:

1. choose the equivalence method to use (with its requirements). Each equiv-

alence query method comes with its guarantees and takes an hypothesis to generate test cases (see sec. 4.2 for more details);

2. generate abstract tests from the Hypothesis according to the selected equivalence method;

3. concretize and execute the abstract messages generated; and then

4. analyze the SUT's responses; if it is the same as the expected response then regenerate a new test sequence and start again from step (2); otherwise, it returns the current test sequence as a counter-example.

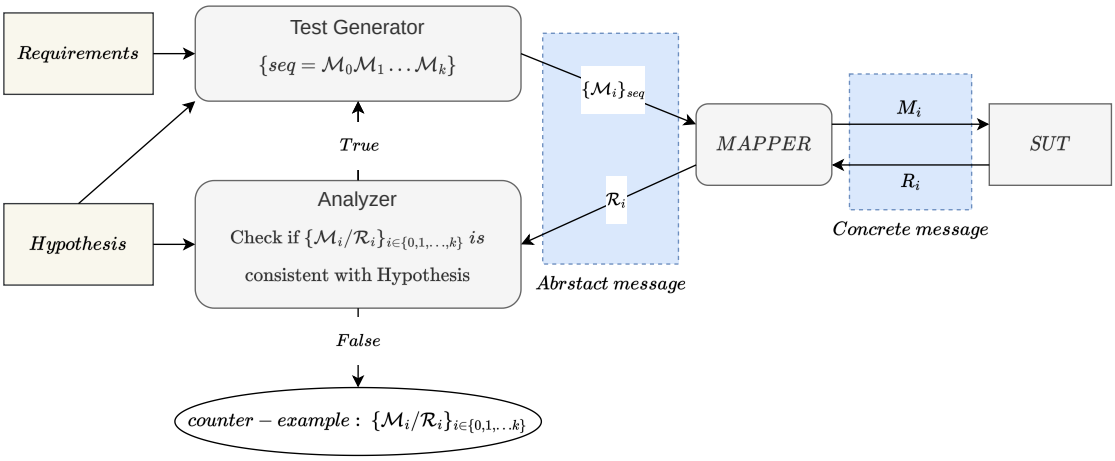


Figure 4.1: Equivalence Testing Process.

The first and the second step of the equivalence testing are detailed in the following sections. Each equivalence method comes with its requirements if they exist. The fourth step is simple, it consists of checking if the result corresponding to the actual test case is equal to the expected results from the hypothesis.

4.2 Avalaible Equivalence Query Algorithms

4.2.1 RandomWord and RandomWalk

The RandomWord method consists in generating random sequences of abstract messages from a given input vocabulary for a given length. Thus it requires the input vocabulary, the minimal and maximal length of sequences to generate and the number of test cases to generate in order to validate the hypothesis. It is the simplest method for equivalence testing.

RandomWalk [Lá93] is an equivalence method very similar to RandomWord. Instead of generating random test cases using input sequences only, it generates test cases by randomly walking to the hypothesis. A random walk starts with a fixed probability, called restart probability, which is the probability of restarting the random walk on the hypothesis. A random walk always starts from the initial state of the hypothesis.

Both methods provide no guarantee on the resulting state machines. Their complexity are related to the parameters defined by the user.

4.2.2 W(p)-method

The W-method [Cho78] and Wp-method [FvBK$^+$91] are based on the calculation of the so-called transition cover set P, state cover set Q and characterization set W (for definitions and a method of constructing such sets, see [FvBK$^+$91, Gil62]).

We recall the definition of these three sets as following:

i) The transition cover set P represents the set of sequences of input vocabulary which allow to reach a given state and test each possible transition from the actual state.

ii) The state cover set Q represents the set of input sequences which allow to visit each state in the automaton.

iii) The characterization set W represents the set of input sequences allowing to distinguish one state from another.

The characterization set is computed using the set of state identification W_i for each state s_i in the hypothesis (i.e., $W = \bigcup_{s_i} W_i$). As described in [FvBK$^+$91], a set of input sequences W_i is an identification set of state s_i if and only if for each state s_j in the hypothesis ($i \neq j$) there exists an input sequence p of W_i such that $Out(s_i, p) \neq Out(s_j, p)$ and no subset of W_i has this property.

The W-method and Wp-method assume a known upper-bound on the size of the exact SUT's state machine (which represents one of the drawbacks of using these methods).

Let us suppose that m is the state bound and n the exact size of the current hypothesis; and Σ_I the input vocabulary.

The W-method provides a set of test sequences formed by $P.Z$ (the concatenation of P and Z), where $Z = (\{\epsilon\} \cup \Sigma_I \cup \Sigma_I^2 \cup \cdots \cup \Sigma_I^{m-n}).W = \Sigma_I[m-n].W$.

The Wp-method improves upon W-method by requiring fewer test cases while providing the same guarantees. To this aim, Wp-method splits the counter-example search into two phases: the first step is used to ckeck that all states

in the hypothesis are identifiable in the SUT, and the second step is done uses test cases from a modified characterization set W.

Given an upper-bound m, the W-method and Wp-method prove the equivalence between hypothesis and a SUT of size up to m. However, the number of test cases generated by both methods is exponential in the bound m, thus it usually does not scale to large systems.

4.2.3 Distinguishing Bounds

The basic idea of Distinguishing Bounds is based on the so-called *distinguisher bound* value B_{Dist}. B_{Dist} is a property of the Mealy machine which guarantees that each pair (q, q') of states in the target Mealy machine can be distinguished by an input sequence $\sigma \in (\Sigma_I)^*$ of length $k \leq B_{Dist}$ i.e., $Out(q, \sigma) \neq Out(q', \sigma)$ [RLM+18].

It guarantees that the obtained state machine will be accurate as soon as two states in the real state machine can be distinguished in at most B_{Dist} steps. The Distinguishing Bounds method consists of two phases:

1. for each $q \in Q$, compute the representative $R(q)$ which is the shortest path to reach q from q_0; and

2. check the "fidelity" of the transition up to the distinguisher bound. It means for each $q \in Q$ and $i \in \Sigma_I$ such that $\delta(q, i) = q'$, check if $R(q).i$ and $R(q')$ can be distinguished using a suffix $\sigma \in (\Sigma_I)^*$ of length $k \leq B_{Dist}$.

This method assumes a known *distinguisher bound* while the resulting state machine depends on its value. If its value is smaller than the actual bound, the resulting state machine is certainly inaccurate. For a *distinguisher bound* B_{Dist} and a state machine of size n, this method requires $\mathcal{O}(n.|\Sigma_I|^{B_{Dist}})$ membership queries to find a counter-example.

4.3 Our New Method: DBBased method

During the second step of the Distinguishing Bounds algorithm, it checks if each pair of states are distinguishable using at least a sequence of length less than or equal to the *distinguisher bound* B_{Dist}. Each state is redundantly checked using sequences of length less than or equal to $B_{Dist} - 1$.

In this section we propose, DBBased method, an improvement of the Distinguishing Bounds method.

DBBased method Algorithm

Our proposed improvement is based on three principles:

Optimization (i) always use suffixes of length B_{Dist} to test each pair of states;

Optimization (ii) stop checking state after receiving *ConnectionClosed*; and

Optimization (iii) decompose loops during the verification of each state.

Optimization (i): Always Use a Suffix of Length B_{Dist}

The Distinguishing Bounds method consists in testing each state by using suffixes of increasing length, starting with 1 and continuing up to B_{Dist}. Instead, the DBBased method tests each pair of states with suffixes of length B_{Dist}.

As described in Table 4.1, with $B_{Dist} = 2$ and input vocabulary $\Sigma_I = \{\mathcal{A}, \mathcal{B}\}$, the Distinguishing Bounds method uses suffixes of length 1 and 2 to check each state and the DBBased method uses suffixes exactly of length 2 to check each state. Thus, the DBBased method only require 4 suffixes instead of 6 suffixes to test each state.

Distinguishing Bounds	$\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}.\mathcal{A}$, $\mathcal{A}.\mathcal{B}$, $\mathcal{B}.\mathcal{A}$, $\mathcal{B}.\mathcal{B}$
DBBased	$\mathcal{A}.\mathcal{A}$, $\mathcal{A}.\mathcal{B}$, $\mathcal{B}.\mathcal{A}$, $\mathcal{B}.\mathcal{B}$

Table 4.1: Suffixes required to test each state using Distinguishing Bounds vs DBBased with $B_{Dist} = 2$. $\Sigma_I = \{\mathcal{A}, \mathcal{B}\}$ is the input vocabulary.

This first modification allows us to considerably reduce the number of the required queries for searching for a counter-example, up to $n \times \sum_{k=1}^{B_{dist}-1} |\Sigma_I|^k$ if n is the size of the current hypothesis.

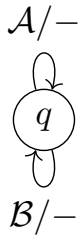
Optimization (ii): ConnectionClosed is Final

Finally, for each $q \in Q$ and $i \in \Sigma_I$ such that $\delta(q, i) = q'$, if $R(q).i$ and $R(q')$ lead to a *ConnectionClosed*, then we stop checking $R(q).i$ and $R(q')$. This allows us to reduce the number of queries up to $|\Sigma_I|^{B_{Dist}}$ for each state and each letter verifying such property.

Optimization (iii): Decompose Self Loops

We call self loop an edge in the Mealy machine that connects a state with itself. Our second modification is related to this type of loops.

If several self loops occur for the same state, we test each of them independently of the other self loops in the same state. In particular, we assume that each state is distinguishable if and only if for all $i \in \Sigma_I$, if i triggers a self loop for a state $q \in Q$, then for all $1 \leq k \leq B_{dist}$, $R(q).(i^k)$ and $R(q)$ can be distinguished using a suffix $\sigma \in (\Sigma_I)^*$ of length less than $B_{Dist} - k + 1$.



(a) Self loop example.

Distinguishing Bounds	DBBased
$\mathcal{A}.\mathcal{A}.\Sigma_I$	
$\mathcal{A}.\mathcal{B}.\Sigma_I$	$\mathcal{A}.\mathcal{A}.\Sigma_I$
$\mathcal{B}.\mathcal{A}.\Sigma_I$	$\mathcal{B}.\mathcal{B}.\Sigma_I$
$\mathcal{B}.\mathcal{B}.\Sigma_I$	

(b) Test cases using Distinguishing Bounds vs DBBased with $B_{Dist} = 3$. Σ_I is the input vocabulary.

Figure 4.2: Comparison of the test cases, of length equal to B_{Dist}, associated to the figure on the left and generated using both methods.

To illustrate this, let us consider the self loops depicted in Figure 4.2a. With $B_{Dist} = 3$, Table 4.2b describes all test cases required for both method, Distinguishing Bounds and DBBased, to check the state depicted in Figure 4.2a. We notice that we only consider suffixes of length B_{Dist} in the table given in Figure 4.2b.

If $B_{dist} > 2$, this change allows us to reduce the number of queries for finding a counter-example by up to $2|\Sigma_I|((n_{sloop})^{B_{dist}-1} - n_{sloop})$ for each state containing n_{sloop} self loops. It means that if each state of the hypothesis contains less than one self loop, this change does not bring any performance improvement. However, the gain is up to $\mathcal{O}(|\Sigma_I|(n_{sloop})^{B_{dist}-1})$ for each state containing n_{sloop} self loops.

Algorithm

Algorithm 1 is an equivalence oracle for Mealy machines using the membership oracle for a given distinguisher bound B_{Dist}. This algorithm implements the three optimizations described above.

In Algorithm 1, the `ANALYZE_SUFFIX` function is used to sort the suffixes interesting counter-example search. Actually, DBBased method does not provide the same guarantee as Distinguishing Bounds method, but it stores the missing sequences in the `validation_suffix` variable of the Algorithm 1.

Thus, in case the user wants to have the same guarantee as the Distinguishing Bounds method, it should first proceed the counter-example searching using the

Algorithm 1 DBBased method Algorithm

Require: Mealy machine $\mathcal{M} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, Out)$, Distinguisher bound B_{Dist} and Membership oracle MOracle

Ensure: True if $\mathcal{M} \simeq \mathcal{M}_{\text{SUT}}$ and counter-example $cex \in (\Sigma_I)^*$ otherwise

1: **for** $q \in Q$ **do**
2: $R[q] \leftarrow w_q$ where $\delta(q_0, w_q) = q$ s.t. $|w_q|$ is minimal
3: validation_suffix[q] $\leftarrow \{\}$
4: **for** $q \in Q$ **do**
5: **for** $i \in \Sigma_I$ **do**
6: $w_i \leftarrow R[q].i$
7: **if** $Out(q, i) \neq$ last symbol of $Out(MOracle(w_i))$ **then**
8: **return** $R[q].i$
9: $q' \leftarrow \delta(q, i)$; $w'_i \leftarrow R[q']$
10: **if** $Out(q, i) =$ ConnectionClosed **then**
11: **if** last symbol of $Out(MOracle(w'_i)) \neq$ ConnectionClosed **then**
12: **return** $R[q']$
13: **go to** 5
14: **for** suffix $\in (\Sigma_I)^{B_{dist}-1}$ and $l \in \Sigma_I$ **do**
15: analyze_later = ANALYZE_SUFFIX(suffix, q)
16: **if** analyze_later = *true* **then**
17: validation_suffix[q] $\leftarrow$ validation_suffix[q] $\bigcup$ {suffix}
18: **go to** 14
19: $out_0 \leftarrow Out(MOracle(w_i.\text{suffix}.l))$
20: $out_1 \leftarrow Out(MOracle(w'_i.\text{suffix}.l))$
21: **if** $out_0 \neq out_1$ **then**
22: **if** $out_0 \neq Out(q', w_i.\text{suffix}.l)$ **then**
23: **return** $w_i.\text{suffix}.l$
24: **else**
25: **return** $w'_i.\text{suffix}.l$
26: **return** True
27: **procedure** ANALYZE_SUFFIX(suffix, q)
28: $start\_state \leftarrow q$
29: $start\_letter \leftarrow$ suffix[0]
30: $i \leftarrow 1$
31: **while** $i <$ |suffix| **do**
32: $end\_state \leftarrow \delta(start\_state, start\_letter)$
33: **if** $end\_state = start\_state$ **then**
34: $tmp\_end\_state \leftarrow \delta(end\_state,$ suffix[i])
35: **if** $tmp\_end\_state = end\_state$ and $start\_letter \neq$ suffix[i] **then**
36: **return** *true*
37: $start\_state \leftarrow end\_state$
38: $start\_letter \leftarrow$ suffix[i]
39: $i \leftarrow i + 1$
40: **return** $false$

DBBased method and if no counter-example is found, then checks the remaining tests sequences present in the `validation_suffix`.

In the worst case, Distinguishing Bounds, DBBased and DBBased with validation require $\mathcal{O}(n.|\Sigma_I|^{B_{Dist}})$ membership queries to find a counter-example. Actually, this is only a theoretical complexity, because in practice, DBBased and DBBased with validation require less membership queries than Distinguishing Bounds to find a counter-example.

In the following section, we evaluate the efficiency of finding counter-example and the performance of our new method against the Distinguishing Bounds method.

4.4 Benchmark and Discussion

Aichernig et al. proposed a comparaison of four active learning algorithms against different equivalence query methods [ATW20] (excluding the Distinguishing Bounds method). Their goal was to identify which equivalence query method works well with which active learning algorithm. They demonstrated that pure random testing, such as RandomWord and RandomWalk, does not scale, and Wp-method performed well with all types of their compared active learning algorithms.

In this section, in contrast to their works, we focused on comparing three equivalence query methods using L* algorithm. We also discuss an interesting open problem about the performance of the learning process.

4.4.1 Experimental Setup

We evaluate three different methods for the equivalence query: Distinguishing Bounds (DB), DBBased method and DBBased method with validation (see Table 4.2). We evaluate these three methods by inferring three different SSH stacks (`Net::SSH`, `wolfSSH` and `ssh2`) using different scenarios (cf sec. 5.2.3).

We selected these three stacks according to the following conditions: they all required multiple hypotheses in the learning process; but they contain various amount of self loops (few, moderate, many as described in Table 4.3). We only apply our benchmark to SSH stacks because most of TLS stacks only have 1 or 2 hypotheses and contain very few self loops.

We have not included RandomWalk and W(p)-method for the benchmarking because they do not share the same parameters as Distinguishing Bounds and DBBased methods. However, we believe that it is an interesting direction for a future work.

Appendix A summarizes the results of the inferences using DB (for Distinguishing Bounds), DBBased with validation (for DBBased method with validation) and

DBBased (for DBBased method) methods. For each result, we have used a B_{dist} equal to 3.

We denote counter-example by "cex". We split the results of searching for a counter example into two: "Searching cex" (when a counter-example is found) and "Hypothesis Validation" (when no counter-example is found). Hence, the exact results of searching for a counter example are earned by aggregating these two results (Searching cex and Hypothesis Validation).

Stacks	**SUT**	**Size of the vocabulary**	**Size of the Automaton**	**Nb loops by state**	**Nb hypothesis**
`Net::SSH` v7.1.0	client	8	48	1.17	7
`wolfSSH` v1.4.12	server	5	18	1.76	4
		10	74	3.45	5
`ssh2` v1.11.0	client	8	26	0.92	7

Table 4.2: SSH stacks and scenarios to evaluate equivalence query methods.

	Size of the Automaton	**Nb states with loops = 1**	**state with loops ≥ 2**	
			Nb of states	Nb loops by state
exp-1	48	4	13 (27.08%)	3.92
exp-2	18	4	12 (66.67%)	2.16
exp-3	74	0	72 (97.30%)	3.50
exp-4	26	1	9 (34.62%)	2.44

Table 4.3: Overview of the self loops by experiments. The number in parenthesis represents the percentage of states containing self loops ≥ 2.

To simplify, we denote:

- *exp-1*: the experiment related to `Net::SSH` v7.1.0 client using the transport and authentication vocabulary;
- *exp-2*: the experiment related to `wolfSSH` v1.4.12 server using the transport vocabulary;
- *exp-3*: the experiment related to `wolfSSH` v1.4.12 server using the transport and authentication vocabulary; and
- *exp-4*: the experiment related to `ssh2` v1.11.0 client using the transport and authentication vocabulary.

4.4.2 Experimental Results

Before starting our analysis, we confirm that the three methods have found the same state machines for the four inferred SSH stacks after the same number of hypotheses.

When analyzing our results, we focus on the following three aspects:

- the number of membership queries required for each equivalence query method;
- the time required to find counter-example if there exists one; and
- the distribution of the time and queries during the learning process.

a) Number of Membership Query

The results of our benchmarking confirm that DBBased method and DBBased method with validation require a smaller number of membership queries than the Distinguishing Bounds method to validate an hypothesis.

Table 4.4 shows that around 50% of the queries of the Distinguishing Bounds method are useless to find a counter-example compared to the DBBased method with validation; whereas these two methods provide the same guarantee.

Actually, the length of the queries are almost the same for the three methods. However, as expected, DBBased method has slightly longer queries than the Distinguishing Bounds method.

b) Finding a Counter-Example

Table 4.5 shows the ability of DBBased method (with and without validation) to quickly find a counter-example compared to the Distinguishing Bounds method if one exists. We merge the results of DBBased method and DBBased method with validation in the Table 4.5, they are the same for all of our four experiments.

Only very few queries are necessary to find a counter-example, up to 89% less than Distinguishing Bounds method (for *exp-4*), using DBBased method. On average, both DBBased methods use only 34% queries of Distinguishing Bounds method to find a counter-example. Both DBBased method use quite long letters per query to find counter-example.

Finally, considering the 4 experiments, DBBased method find counter-example up to 2 times faster than the Distinguishing Bounds method. Table 4.5 presents the detailed results.

		DB	**DBBased** with validation	**DBBased**
exp-1 ($M_{sloop} = 1.17$)	Nb query	394 000 (100%)	180 769 (45.88%)	152 033 (38.59%)
	Mean query length (in messages)	7.63	8.75	8.92
exp-2 ($M_{sloop} = 1.76$)	Nb query	22 720 (100%)	13 609 (59.90%)	12 469 (55.88%)
	Mean query length (in messages)	5.57	6.11	6.06
exp-3 ($M_{sloop} = 3.45$)	Nb query	972 000 (100%)	725 388 (74.63%)	616 380 (63.41%)
	Mean query length (in messages)	6.89	7.27	7.21
exp-4 ($M_{sloop} = 0.92$)	Nb query	213 952 (100%)	42 334 (19.78%)	40 478 (18.92%)
	Mean query length (in messages)	6.97	8.97	8.98
Average percentage of the Nb query		100%	50.05%	44.20 %

Table 4.4: Evaluation of the queries required for validating an hypothesis. The number in parenthesis represents the percentage of required queries compared to the Distinguishing Bounds. M_{sloop} is the mean self loops by state.

c) Repartition of the Time and Queries

Tables 4.7 and 4.6 summarize the distribution of the time spent while searching for a counter-example, validating and building hypothesis.

The DBBased method with validation loses a lot of time to validate an hypothesis (up to 60% of the time with *exp-1* and 83% with *exp-3*) compared to the two other methods (38% of the time with *exp-1* for Distinguishing Bounds method and 36% with *exp-1* for DBBased method). In addition to the queries specific to DBBased method, DBBased method with validation also checks all the combinations of self loops that were skipped in the last occurence of DBBased method.

During the learning process, the three methods spent almost 90% of the time on searching for counter-eaxmple and validating hypothesis. Only 10% of the time is used to build hypothesis. The larger the state machine, the longer the equivalnece query methods take for searching for a counter-example and validating hypothesis.

However, for our four experiments, we notice that DBBased method with and

		DB	**DBBased** with and without validation
exp-1	Duration (in secondes)	31 274 (100%)	17 290 (55.28%)
	Nb query	609 440 (100%)	207 521 (34.05%)
	Mean query length (in messages)	5.89	6.72
exp-2	Duration (in secondes)	3 124 (100%)	2 681 (85.24%)
	Nb query	24 754 (100%)	13 056 (52.74%)
	Mean query length (in messages)	5.23	5.64
exp-3	Duration (in secondes)	67 715 (100%)	36 067 (53.27%)
	Nb query	361 106 (100%)	139 634 (38.67%)
	Mean query length (in messages)	6.94	7.23
exp-4	Duration (in secondes)	2 704 (100%)	2 321 (85.84%)
	Nb query	263 657 (100%)	27 419 (10.40%)
	Mean query length (in messages)	4.89	7.19

Table 4.5: Evaluation of the ability of the three methods in finding counter-examples.

without validation brings a small improvement compared to the Distinguishing Bounds method either on the proportion of the time or the queries while searching for a counter-example and validating hypothesis (see Tables 4.6 and 4.7). But, more benchmarks should be done to confirm this affirmation.

		DB	**DBBased** with validation	**DBBased**
exp-1	Searching cex	60.16%	52.15%	56.21%
	Hypothesis Validation	38.89%	45.43%	41.18%
	Building Hypothesis	0.95%	2.42%	2.61%
exp-2	Searching cex	50.96%	46.86%	48.86%
	Hypothesis Validation	46.77%	48.85%	46.67%
	Building Hypothesis	2.27%	4.29%	4.47%
exp-3	Searching cex	26.87%	22.41%	25.32%
	Hypothesis Validation	72.32%	76.40%	73.34%
	Building Hypothesis	0.81%	1.19%	1.34%
exp-4	Searching cex	54.60%	36.47%	37.40%
	Hypothesis Validation	44.31%	56.31%	55.21%
	Building Hypothesis	1.09%	7.22%	7.39%

Table 4.6: Repartition of the queries while searching counter-example, validating and building hypothesis.

4.4.3 Discussion

The DBBased method is at least two times faster than the Distinguishing Bounds method for discovering a counter-example. It only requires a very few queries, up to 20% less queries than the Distinguishing Bounds method with *exp-1*. It also improves the overall learning process, up to two times faster than the Distinguishing Bounds method.

The DBBased method is the most efficient in terms of speed in finding a counter-example but it does not provide the same guarantee as the DBBased method with validation and Distinguishing Bounds method. It offers the same

		DB	**DBBased** with validation	**DBBased**
exp-1	Searching cex	54.73%	33.14%	52.76%
	Hypothesis Validation	38.62%	60.16%	36.13%
	Building Hypothesis	6.65%	6.70%	11.11%
exp-2	Searching cex	57.56%	49.08%	54.83%
	Hypothesis Validation	33.60%	41.12%	34.27%
	Building Hypothesis	8.84%	9.80%	10.90%
exp-3	Searching cex	19.02%	14.15%	17.54%
	Hypothesis Validation	78.84%	83.59%	79.67%
	Building Hypothesis	2.14%	2.26%	2.79%
exp-4	Searching cex	25.57%	22.09%	25.25%
	Hypothesis Validation	59.99%	62.11%	56.65%
	Building Hypothesis	14.44%	15.80%	18.10%

Table 4.7: Repartition of the time while searching counter-example, validating and building hypothesis.

quarantee as the two other methods, under the assumption that two states do not need complex paths with different self loops to be distinguished.

Based on our four experiments, the DBBased method works in practice. It finds the same state machines as the two other methods because the validation step does not provide a new counter-example in our experiments.

The DBBased method with and without validation share the same complexity when finding a counter-example, but in contrast to DBBased method, DBBased method with validation, loses a lot of time to validate an hypothesis. However, the DBBased method with validation remains more efficient than the Distinguishing Bounds method either in finding counter-example or in the overall learning process;

whereas both methods provide the same guarantee.

Actually, our results about the distribution of the time spent and the number of queries while searching for a counter-example, validating and building hypothesis show the necessity of having an efficient equivalence query method which allows to decrease the number of queries.

Chapter 5

Black-box Analysis of TLS and SSH Implementations

This chapter details our results on TLS and SSH state machine inference using the generalized methodology proposed in chapter 3.

We use the Mealy machine representation for the state machines of TLS and SSH. A detailed example is given in Figure 5.1.

Figure 5.1, describes a typical TLS 1.3 connection and the corresponding expected client state machine. On the right, transitions are labeled with the messages *sent to* / *received from* the client. The path in green is the expected flow, also called *happy path*, described on the left, ending with a request (the `AppData` received from the client between states 4 and 5) and the answer (the `AppData` sent between states 5 and 6). A transition with `*` aggregates the behaviors for the remaining input messages.

The *happy path* in Figure 5.1 does not take session resumption or the so-called 0-RTT mode into account. It also ignores common error cases, such as the impossibility for the client and the server to agree on a common ciphersuite.

From this description, we represent the expected behavior of a TLS 1.3 client with the state machine on the right side of Figure 5.1. The happy path, in green, starts with the client outputting a `ClientHello`, and leads to the `Finished` messages and the exchange of Application data. Outside of this happy path, all other messages (denoted by `*`) lead to the sink state with a fatal alert. This figure is identical to the state machine inferred on OpenSSL 3.0.1 using our methodology.

This chapter is organized as follows: sec. 5.1 gives more details about how we built our Mapper. Then, in sec. 5.2, we describe our TLS and SSH platform followed by several TLS and SSH scenarios considered for the inference and the list of our studied stacks. sec.5.3 presents how we detect bugs from TLS and SSH state machines and how we confirm them. In sec. 5.4, we discuss and evaluate our optimization proposal for the learning process. In sec. 5.5, we provide more details

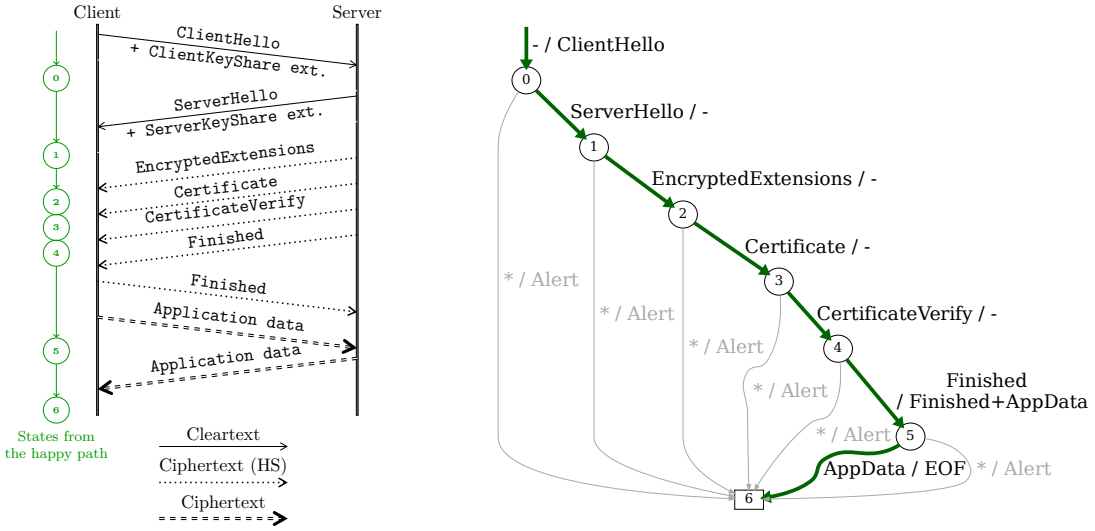


Figure 5.1: A typical TLS 1.3 connection and the corresponding expected client state machine.

about reproduced and new vulnerabilities we have detected. Finally, in sec. 5.6, we summarize the list of reproduced and new vulnerabilities we have detected; and we also discuss about the limitation of our method.

5.1 Mapper Implementation

As discussed in chapter 3, one major challenge with the L* approach is that the Mapper used to concretize the abstract messages has to be flexible enough to send arbitrary messages at any state of execution of the protocol (even ones that would clearly be invalid). In this section, we describe our implementation of the Mapper for TLS and SSH.

5.1.1 TLS Mapper

For the TLS platform, we use `scapy`, a Python-based network tool, to forge and decode packets [Bio05]. `scapy` implements all SSL/TLS versions (from SSLv2 to TLS v1.3) and allows us to easily build customized packets (e.g., a `Certificate-Verify` with a wrong signature).

Apart from small patches[1], `scapy` proved to be up to the task. We added two patches to `scapy`: variable initialization and fix bug related to Diffie-Hellman.

[1]Our patches can be seen at `https://github.com/pictyeye/scapy/tree/pylstar`

Variable initialization Initializing variables is required to build TLS messages in an arbitrary order. During the concretization, if a variable is required but its value is none, the default value (initial value) is used when building the message.

Fix bug: Diffie-Hellman When using `scapy` as a TLS Mapper, we detected a bug related to the *Pre Master Secret* (PMS) derivation for Diffie-Hellman key exchange in TLS (only the Finite Field DHE, not the ECDHE variant). Indeed, the RFC states that the leading zero bytes must be stripped when storing the PMS. This is bad practice, since it leads to exploitable timing attacks such as the Raccoon Attack[2], which is CVE-2020-1968 for OpenSSL, but required to be compatible with other stacks.

In contrast to TLS, we built from zero our SSH Mapper which we describe in the next section.

5.1.2 SSH Mapper

In this section, we describe the design of our SSH Mapper following the solution introduced in chapter 3.

a) Message Structure

Because the Mapper concretizes and abstracts messages, we need to know the structure of the SSH messages. We identified two different SSH messages structures: without and with encryption [Ylo06c].

Without encryption, each SSH message has the structure described in Figure 5.3, where the `packet length` is the length of the packet in bytes (not including the `packet length` field itself); the `padding length` is the length the `padding` (random bytes) and the `SSH payload` covers the SSH message type and the `payload` (the `payload` depends on the SSH message type).

The SSH message structure changes when encryption is enabled (see Figure 5.4). First, a counter is incremented for each message sent and received. Second, SSH messages are built using the message structure without encryption. Then, the output of the second operation is encrypted and the MAC value corresponding to the counter concatenated with the output of the second operation is computed. Finaly, the encrypted SSH packet is the concatenation of the encrypted packet and the MAC value. This corresponds to the encrypt and MAC paradigm, used by default in SSH (but it can change with some algorithms).

We implemented in `ssh_generic_packet` and `parse_generic_ssh_msg`, two functions that handle the SSH message structure either with or without encryption.

[2] `https://blog.min.io/raccoonattack/`.

As described in Figure 5.2, we also implemented in each SSH message class two functions, `build` and `parse`, for the SSH payload structure which is specific to each SSH message type.

b) *Internal state*

We recall that the SSH protocol has three layers: Transport layer, Authentication layer and Connection layer. To handle the *Internal state*, we create `SSHConnState`, a class which contains three "sub-internal state": `TransportState`, `CryptoState` and `ConnectionLayerState`. The authentication layer does not change the state of the *Internal state* of the MAPPER.

The `TransportState` is used to store all parameters related to the `KEXINIT` (from both side) and the `DH_INIT` and `DH_REPLY` messages. These parameters are useful for the derivation of the session identification (`session_id`) and the session keys.

The `CryptoState` is used to store all parameters related to the cryptography such as the `session_id` (which is static during the SSH connection), the symmetric cryptographic material (`local_sym_crypto` and `remote_sym_crypto`) which is used to store the selected encryption algorithm, the session keys from client to server and server to client, the material required for the computation of the MAC value (`macstate`) such as the current sequence number (i.e., the counter described in Figure 5.4) and the selected MAC algorithm (from the LEARNER to the SUT).

Finally, the `ConnectionLayerState` is used to correctly handle the channels in the connection layer. We use `open_channels`, a dictionary which allows us to handle the association of each recipient and sender channel. Especially when analyzing the SSH server, we offer the possibility to limit the number of channels to open. It allows us to easily handle the channel association.

For the update of the *Internal state*, we choose the strategy 3 described in sec. 3.4.2. Since SSH state machines are big, this strategy allows us to get a simplified state machines (in terms of number of states).

c) Protocol Logic

To correctly implement the SSH MAPPER, we need the following operations: (i) derivation of the exchanged hash `H` and the `session_id`, (ii) derivation of the session keys, (iii) encryption, (iv) decryption, (v) signature and (vi) computation of the MAC value. These six operations are implemented in different functions of the `CryptoState`.

The exchanged hash `H` is the hash of $V(C)||V(S)||I(C)||I(S)||K_S||e||f||K$ where C is the client, S is the server, $V(x)$ is the identification string of x (which is present in the `SSHVersion` message), $I(x)$ is the payload of the `KEXINIT` message of x, K_S

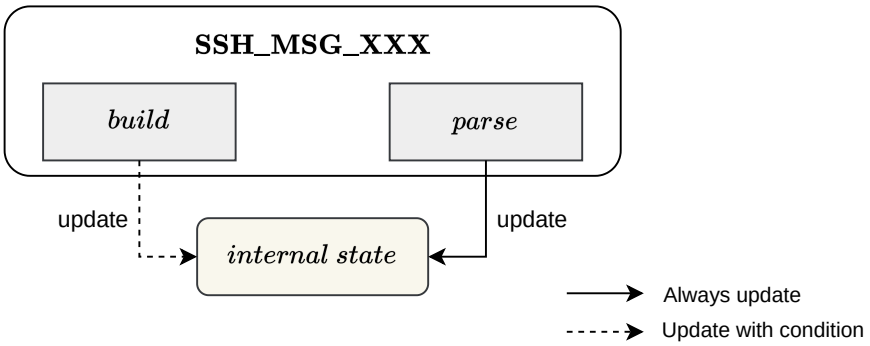


Figure 5.2: Design of the SSH Mapper.

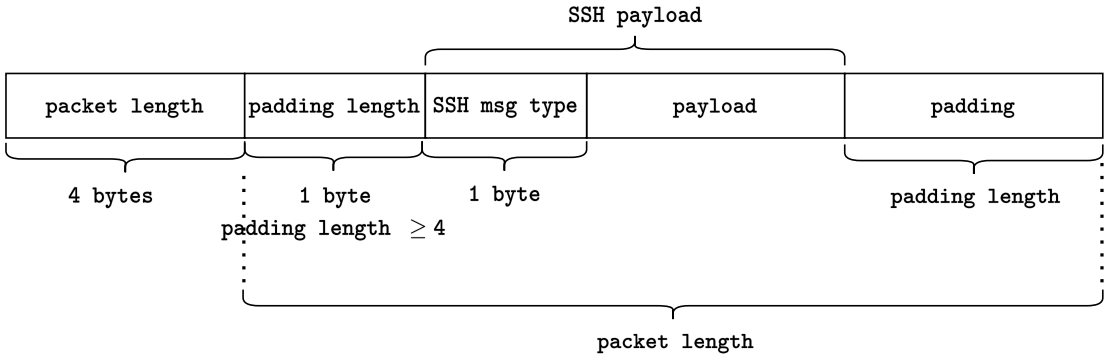


Figure 5.3: SSH message format before encryption is activated.

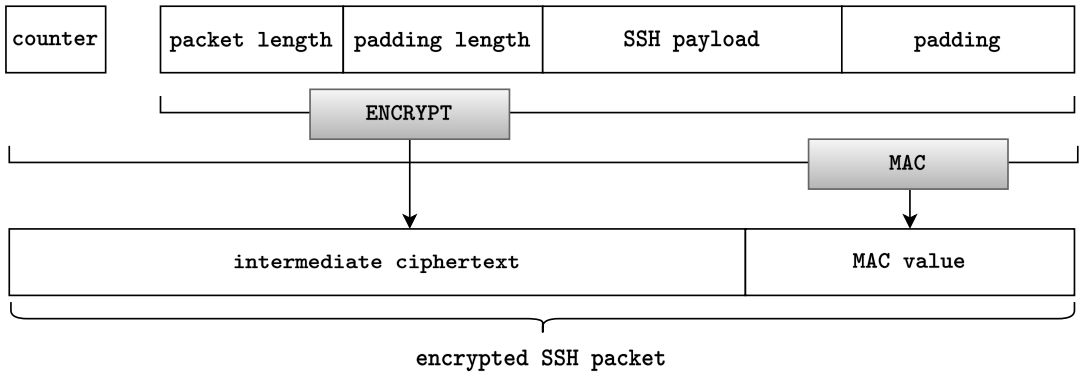


Figure 5.4: SSH message format with encryption is activated.

is the public host key of the server, e (resp. f) is the public Diffie-Hellman value of the client (resp. the server) and K is the shared secret (between the client and server) from the execution of the Diffie-Hellman protocol. The value of `H` is updated during each key re-exchange.

The `session_id` (which is a unique identifier for this connection) is the first value of H from the first key exchange. Session keys are then derived from the `session_id` and the exchanged hash `H`.

To complete the implementation of the MAPPER, we implement different functions for the encryption, the decryption, the computation of the signature (for the server and client authentication) and the MAC value (see Figure 5.4).

In the following section, we discuss our TLS and SSH platform; we also discuss about different scenarios used to systematically analyze TLS and SSH stacks.

5.2 Experimental Setup and Experiments

The learning setup comprises the LEARNER, the MAPPER and the SUT. The SUT and the LEARNER are configured following the scenario detailled in sec. 5.2.3. In this section, we discuss three things: architecture of our platform, the implementations tested and the learning alphabet.

5.2.1 Architecture of our Platform

Our two platforms: `TLS-inferer` and `SSH-inferer`, are based on `pylstar` as an implementation of the L*.

a) TLS Platform

In our platform, a TLS stack is defined as a container running at least one of the following scripts: `run_server`, which launches a TLS server, ready to be sollicited; `run_client`, which starts a so-called trigger server, a service listening to signals from the inference tool, so a TLS client can be spawned each time we want to test a message sequence. Table 5.1 lists the TLS stacks currently included.

Figure 5.5 describes a typical workflow of our platform to infer a client state machine.[3] First, we start a client container running the trigger script (step 1). Then, we start our inference tool containing the L* engine (the LEARNER) and the TLS MAPPER (step 2).

[3]Inferring a server works in a similar, but simpler, way. Indeed, we can simply start the server and have the inference tool open a connection for each sequence to test.

Each time the algorithm needs to learn information from the SUT (the TLS client) using a sequence of messages, it first resets the client (step 3), which spawns a fresh TLS client in the client container (step 4). This client establishes a TCP connection to the TLS server within the harness (step 5) and sends its `ClientHello`. From now on, the $L^\star$ engine drives the MAPPER by transmitting abstract messages to send to the client (step 6). The harness concretizes those messages and sends them to the client (step 7). In return, the concrete answer from the client (step 8) are abstracted by the harness (step 9).

Steps 3 to 9 are repeated until the $L^\star$ engine is able to produce a valid hypothesis regarding the client state machine, that is to generate an automaton accurately describing the client behavior (step 10).

b) SSH Platform

We built a very similar platform as TLS for the analysis of the SSH server. Client inference works exactly as TLS client inference.

We observed that several SSH servers crashed after a few membership queries and when inferring the Connection layer, a few SSH stacks do not properly ensure its functionality (they behave non-deterministically) after several membership queries.

For these reasons, we added a *proxy* to the SSH server container which is used to open and close the server. This allows the LEARNER to restart the server if necessary.

Figure 5.6 describes a typical run of our platform to infer a SSH server state machine. First, we start a server container running the trigger script and the *proxy* (step 1). Then, we start our inference tool containing the $L^\star$ engine (the LEARNER) and the SSH MAPPER (step 2).

At the begining, the LEARNER establishes a TCP connection to the *proxy* and sends the message `OPEN` to the *proxy* (step 3). Then, the *proxy* resets the server (step 4) which spawns a fresh SSH server in the server container (step 5). After the SSH server is opened, the *proxy* sends a signal `OK` to the LEARNER (step 6), which notifies the LEARNER that the server is ready. Hence, the MAPPER establishes a TCP connection to the SSH server and sends its `SSHVersion` and gets the SUT's `SSHVersion` (step 7). From now on, we proceed exactly as with the TLS platform.

However, after a membership query, the LEARNER has the possibility to close and re-open the server by sending the message `CLOSE` and `OPEN` to the *proxy*. When the *proxy* receives the message `CLOSE`, it closes the server and sends the signal `OK` to the LEARNER. And again, if the LEARNER wants to re-open the server, then it sends `OPEN` to the *proxy* and so on.

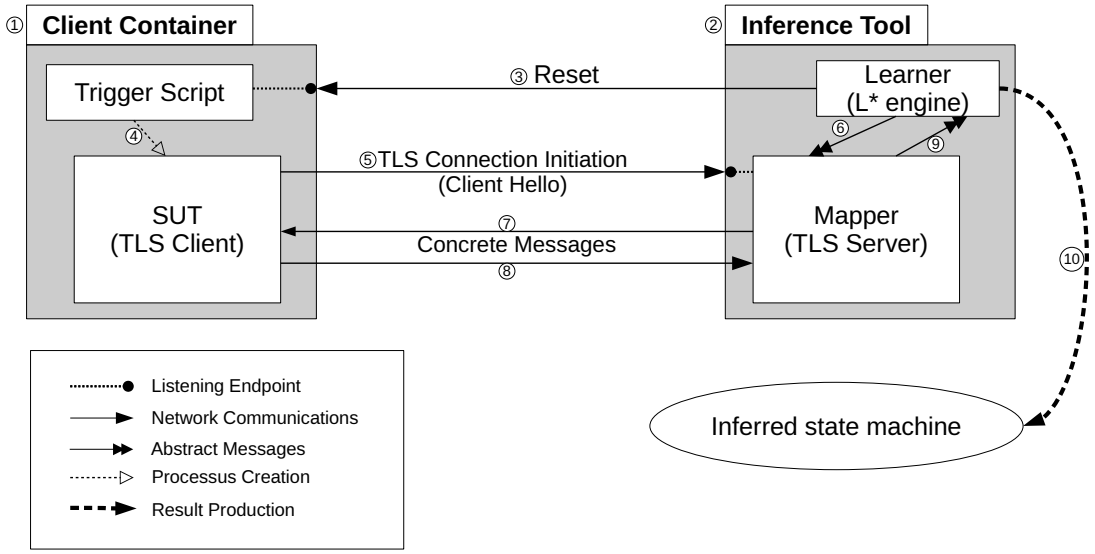


Figure 5.5: TLS client inference workflow.

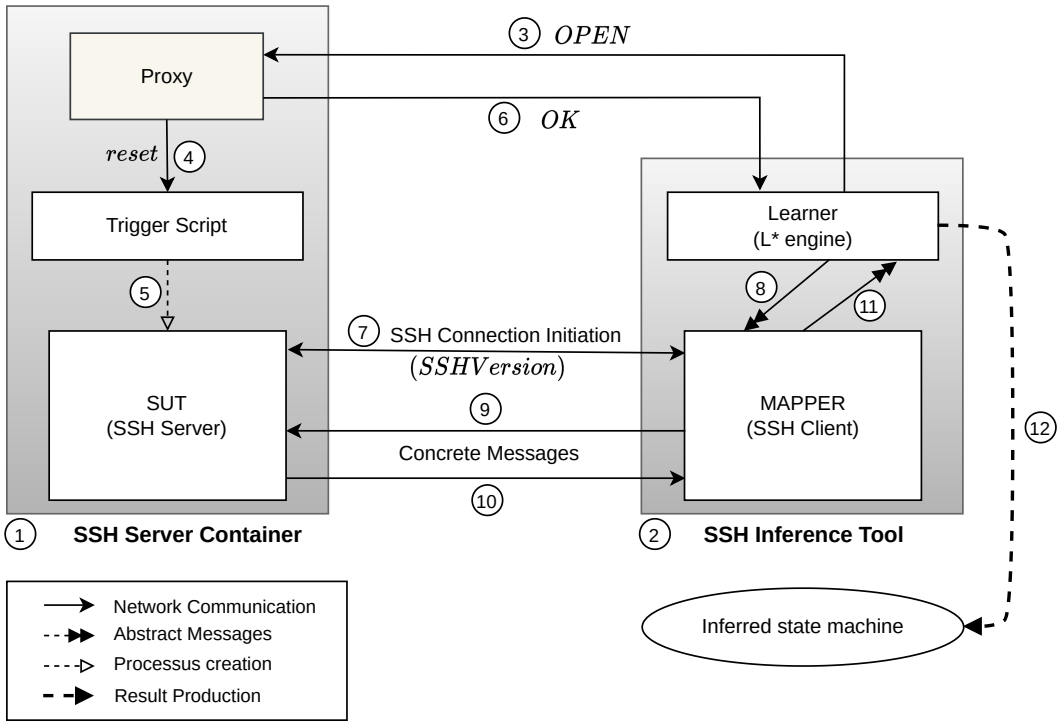


Figure 5.6: SSH server inference workflow.

5.2.2 Implementations Tested and Analyzed

We create containers for more than 600 stacks (400 TLS stacks and 200 SSH stacks). Table 5.1 details the TLS stacks currently included in our TLS platform and Table 5.2 for SSH platform.

For each stack, we reuse the tools or the example code available within the project to build and run a TLS or SSH client and/or a TLS or SSH server[4]. Such pieces of code are representative of the way the libraries are used in practice.

Each TLS container is customized to select the protocol version and the ciphersuites, and to include the required cryptographic material (certificate, keys, trusted certification authority), which allows us to study different scenarios.

In addition to the "example" client, we create for each TLS stack a second container, using `curl` dynamically linked with each TLS stack. These curl-based images provide a unified interface across stacks, removing small differences in the example provided by the projects, e.g., missing certificate checks. All server-side examples include a functional and sufficiently customizable example for our needs.

In contrast to the TLS container, each SSH container is customized to select the cryptographic algorithm (from client to server and vice versa), include the required cryptographic material (keys and authorized client keys) and the SSH configuration file. We notice that we have only analyzed SSH v2.0.

5.2.3 Learning Alphabet

Our goal is to systematically analyze TLS and SSH stacks. To this aim, we try to consider different scenarios allowing to further analyze stacks in terms of state machine by trying to cover the maximum security perimeter as possible.

a) TLS Learning Alphabet

To analyze TLS implementations, we define different scenarios. A scenario is defined by the following three parameters:

(i) the role (client/server) and the configuration (protocol version, ciphersuites, etc.) of the SUT;

(ii) the input vocabulary (the list of abstract messages) used during the inference; and

(iii) a set of security properties to test on the resulting graph.

[4]e.g., `openssl s_server` for the OPenSSL project.

Stack Name	Versions	Client	Server	Comments
OpenSSL	0.9.8m - 1.0.0t (41)	✓	✓	Only TLS 1.0
	1.0.1a - 1.1.0l (53)	✓	✓	Only TLS 1.0 and 1.2
	1.1.1a - 1.1.1n (14)	✓	✓	
	3.0.0 - 3.0.2 (3)	✓	✓	
curl+OpenSSL	1.0.0a - 1.0.0t (20)	✓		Only TLS 1.0
	1.0.1a - 1.1.0l (53)	✓		Only TLS 1.0 and 1.2
	1.1.1a - 1.1.1n (14)	✓		
	3.0.0 - 3.0.2 (3)	✓		
GnuTLS	3.6.16 - 3.7.2 (4)	✓	✓	
curl+GnuTLS	3.6.16 - 3.7.2 (4)	✓		
mbedtls	1.3.10 - 1.4 (17)	✓	✓	Only TLS 1.0
	2.0.0 - 3.0.0p1 (96)	✓	✓	Only TLS 1.0 and 1.2
wolfssl	3.12.0 - 3.14.4 (10)	✓	✓	Only TLS 1.0 and 1.2
	3.15.5 - 5.2.0 (20)	✓	✓	
curl+wolfssl	3.12.0 - 3.14.4 (10)	✓		Only TLS 1.0 and 1.2
	3.15.5 - 5.1.1 (20)	✓		
matrixssl	3.7.2 (1)		✓	Only TLS 1.0
	4.0.0 - 4.3.0 (7)		✓	
NSS	3.15 - 3.38	✓	✓	Only TLS 1.0 and 1.2
	3.39 - 3.78	✓	✓	
erlang	20.0 (1)		✓	Only TLS 1.0
	24.0.3 - 24.2.1 (2)		✓	
fizz	2021.02 - 2021.06	✓		Only TLS 1.3 Weekly snapshots

Table 5.1: List of TLS Stacks included in our TLS Platform (the number in parentheses is the number of stacks).

Stack Name	Versions	Client	Server
`OpenSSH`	6.5.P1 - 9.2.P1 (30)	✓	✓
`dropbear`	2014.64 - 2022.83 (20)	✓	✓
`libssh`	0.7.6 - 0.10.4 (23)	✓	✓
`paramiko`	2.4.0 - 3.1.0 (30)	✓	✓
`AsyncSSH`	1.12.0 - 2.13.1 (38)	✓	✓
`wolfSSH`	0.1 - 1.4.12 (21)	✓	✓
`SSH2`	1.0.0 - 1.11.0 (12)	✓	✓
`sshd-lite`	1.3.1 - 1.2.0 (6)	✗	✓
`Net::SSH`	4.0.0 - 7.1.0 (34)	✓	✗

Table 5.2: List of SSH Stacks included in our SSH Platform (the number in parentheses is the number of stacks).

Client Scenarios In these scenarios, the SUT is a client, running a given version of TLS. The client is configured with a trusted certification authority and is expected to check the certificate presented by the server. The inference tool acts as a server, with the following input vocabulary: `ServerHello`, `Certificate` messages (valid, empty, invalid — trusted but for the wrong domain —, and untrusted), other server-side Handshake messages, `ApplicationData` and `CloseNotify`.

In these scenarios, we ensure that the client only sends application data to a correctly authenticated server. We look for paths leading to `ApplicationData` messages and check for proper authentication.

Another area of interest is the presence of loops that could be used by an attacker to stall a client, enabling complex cryptographic attacks, such as the LogJam attack [ABD+15]. Since the goal of such attacks is to delay the completion of the TLS Handshake, we only focus on loops happening early in the connection.

Server Scenarios In these scenarios, the SUT is a server, running a given version of TLS. The server can be configured to require mutual authentication (with respect to a given certification authority). The inference tool acts as a client, and uses the following vocabulary: different `ClientHello`s, various `Certificate` messages (empty, trusted, untrusted), other client-side Handshake messages, `ApplicationData` and `CloseNotify`. We also include alerts and unexpected messages such as server-side messages.

When client authentication is required, we want to ensure that the server properly authenticates the client. Only paths with a valid certificate and the corresponding signature should be accepted.

We are also interested in the presence of loops in server state machines, which could force the server to maintain a connection open indefinitely. For such denial of service attacks, we only focus on occurrences happening before encryption is activated; this way, the attacker only needs to spend very few resources to keep the channel open.

Moreover, keeping the server in an early stage of the connection reduces the chances of something being logged. Note that these loops are different from the ones created through TCP segmentation or TLS `ClientHello` fragmentation, which would be limited by the length of the data to send.

b) SSH Learning Alphabet

To analyze SSH implementations, we consider seven scenarios. A scenario is defined by the following four parameters:

(i) the role (client/server) and the configuration (list of algorithms, etc.) of the SUT;

(ii) the input vocabulary (the list of abstract messages) used during the inference;

(iii) the layer supposed to be completed (so it is executed before the inference); and

(iv) a set of security properties to test on the resulting graph.

It means that these scenarios offer us the possibility of inferring the SSH stacks state machine layer by layer. Table 5.3 gives an overview of the input vocabulary we considered during the SSH inference.

`Transport` **Scenario** In this scenario, the input vocabulary is the vocabulary concerning only the transport layer and the Learner is only interested in the transport layer.

By considering this scenario, we quickly ensure that the server or client properly implements the key exchange, the server authentication if the Sut is the client and if it accepts skipping the `NEWKEYS` which is used to enable encryption. If this first layer is not properly implemented, it opens a potential MiTM attacks.

`Authentication` **Scenario** In this scenario, we suppose that the transport layer is completed. We only use authentication layer vocabulary (including invalid authentication request messages when the Sut is the server) as an input vocabulary of the $L^\star$ algorithm.

This scenario allows us to quickly check whether the authentication layer is well-written and whether it also allows to check if the authentication message is correctly checked when it comes in an appropriate time, in case the Sut is the server. If the server does not correctly check the client's authentication, then it allows a MiTM attack.

However, this scenario can not help a lot when learning the client's state machine apart from checking if the client implements the protocol as expected.

`Connection` **Scenario** In this scenario, the first SSH two layers are supposed to be completed and the input vocabulary is the connection layer vocabulary. Actually, especially in case the Sut is the server, the input vocabulary is parameterized by the number of channel the Learner want to test.

In case the Sut is the server, this scenario allows us to check if the server correctly handles the channel as described in the specification which is a complex task. This scenario allows for both sides (client and server) to check if a DoS attack is possible in the connection layer.

`Transport and Authentication` **Scenario** In this scenario, the first two layers of SSH are simultaneously infered using the transport layer and the authentication layer vocabulary.

In addition to the `Transport` scenario, this scenario allows us to automatically detect if replay attack is possible (see sec. 5.5.2) in case the SUT is the server; and it allows to check if the client does not leak credentials to a non-authenticated server in case the SUT is the client.

`Authentication and Connection` **Scenario** In this scenario, the transport layer is supposed to be completed (i.e., out of interest) and the input vocabulary is the union of the authentication layer vocabulary and the connection layer vocabulary.

In addition to the `Connection` scenario, this scenario allows us to ensure that only an authenticated server has access to the connection layer and benefits from its services. Only paths with valid authentication messages should be accepted.

`Transport and Connection` **Scenario** In this scenario, the transport layer and the authentication layer are supposed to be completed and the input vocabulary is the union of the transport layer vocabulary and the connection layer vocabulary.

It allows us to check if the key re-exchange is implemented correctly in the connection layer. By considering this scenario, we are also interested in the presence of paths described in sec. 3.2.3 which might allow an honest but curious client or server to lead a DoS attack.

`Transport, Authentication and Connection` **Scenario** We combine all the SSH message given in table 5.3 to analyze the SUT. This scenario allows us to check simultaneously all the security properties described above. However, compared to the scenario introduced above, this scenario is very costly in terms of the performance of the learning process, which is expected because of the size of the input vocabulary.

5.2.4 Configuration and Adaptation of our Platform

Before discussing our method to automatically detect bugs in state machines, we present different solutions that we have implemented in our platform to avoid non-determinism and infinite state machines.

a) Non-determinism

Since we use $L^\star$ algorithm, we have to avoid all non-deterministic behavior of the SUT (see sec. 3.1.5). TLS and SSH stacks behave deterministically, with a few exceptions.

	Server Inference	Client Inference	Comment
Transport	KEXINIT_DH DH_INIT NEWKEYS SRV_REQ DISCONNECT	KEXINIT DH_REPLY NEWKEYS SRV_ACCPT DISCONNECT	SRV_REQ and SRV_ACCPT are parameterized by {auth, connection, none}.
Authentication	AUTH_none AUTH_PW AUTH_PW_NOK AUTH_RSA AUTH_RSA_NOK	AUTH_SUCCESS AUTH_FAILURE AUTH_PK_OK	AUTH_PW_NOK and AUTH_RSA_NOK contain a valid *username* and an arbitrary password and signature.
Connection	CH_OPEN CH_DATA CH_EOF CH_CLOSE CH_SHELL CH_pty CH_EXEC	OPEN_CONFIRM OPEN_FAILURE CH_SUCCESS CH_FAILURE CH_DATA CH_EOF CH_CLOSE	CH_EXEC contains the command line for creating a file test.txt using linux command line touch.

Table 5.3: SSH input vocabulary by layer. Messages in the server inference are client-side messages and vice versa.

When a SUT takes too long to answer a stimulus, we can misinterpret its silence as the absence of messages, whereas output messages were actually expected. This requires to get the timeout parameter right. During each step of the membership query, we must ensure that we have received all the messages the SUT has sent. The usual solution is to wait for a long period of time before inferring "no response", which makes the inference slow. To improve the performance of our tools, we introduce heuristics to reduce the timeouts when possible.

In rare cases, an encrypted message can be misinterpreted and produces an unexpected response. To avoid this, we always tag reception of encrypted packets that cannot be properly decrypted with a dedicated letter UnknownPacket.

For SSH, we also detected two specific additional sources of non-determinism. For the same query, we sometimes get the SUT's responses in a different order. For example as a response to the message $\mathcal{M}$, sometimes we get $\mathcal{A} + \mathcal{B}$ and sometimes we get $\mathcal{B} + \mathcal{A}$ [5]. From the L* point of view, these two sequences of messages are

[5]When the LEARNER sends a CH_EXEC message to an OpenSSH server after a CH_OPEN, we observed the following responses WINDOW_ADJUST+CH_SUCCESS+CH_DATA+CH_EOF+CH_EXIT +CH_CLOSE and WINDOW_ADJUST+CH_SUCCESS+CH_DATA+CH_EXIT+CH_EOF+CH_CLOSE.

different.

To avoid this problem, for each step of the *membership query*, we store the outputs from the SUT in `setOfOutput`[6] variable. And if the set of the SUT's responses is already present in `setOfOutput`, we consider this element as the corresponding response.

A few SSH stacks behave non-deterministically when processing the message `CH_DATA`. In rare cases, data are encapsulated in one `CH_DATA` message and sometimes the SUT splits the same data into multiple `CH_DATA` message. Hence we always consider only one `CH_DATA` message even if we receive multiple consecutive `CH_DATA` to avoid this non-determinism problem.

However, this modification slows down the performance of the learning process because we have to disable our optimization (discussed in sec. 5.4) when a `CH_DATA` message is present in the SUT's response. If a `CH_DATA` message is present in the SUT's response, we have to ensure that we have received all `CH_DATA` the SUT has sent.

b) Infinite State Machines

As discussed in sec. 3.1.3, $L^\star$ cannot infer infinite state machines. Several SSH stacks do not have a finite state machine (e.g., OpenSSH, AsyncSSH and sshd-lite). They all suffer from the same problem discussed in sec. 3.2.3.

To quickly recall the common reasons related to infinite state machines, the latter can occur when the SUT accepts a message $\mathcal{M}_0$ indefinitely and it does not directly respond to the LEARNER until it receives message $\mathcal{M}_1$. Then the LEARNER receives multiple times the response corresponding to $\mathcal{M}_0$ in addition to the response corresponding to $\mathcal{M}_1$ itself.

To avoid this problem, which amounts to counting parenthesis, when we receive a message $\mathcal{M}$ more than twice in the SUT's responses, we aggregate the messages to "$\mathcal{M} + \mathcal{M}.\mathcal{M}^*$"[7]. Thus, during the analysis of the state machine, we must be aware of the meaning of "$\mathcal{M} + \mathcal{M}.\mathcal{M}^*$" which means that the LEARNER has received the message $\mathcal{M}$ at least twice.

Hence, by using this compression, we can infer the state machine of an implementation which would otherwise be infinite. However, we loose the exact information about the SUT because no distinction is made between a message received twice and three or more times.

[6] The variable `setOfOutput` contains a set of string.

[7] This does not apply to the `CH_DATA` message.

c) Discussion

As described above, for some cases, we modify the SUT's response to avoid either non-determinism or infinite state machines. These modifications are not neutral for the performance of the learning process, as they prevent us from taking advantage of the optimization discussed in sec. 5.4.

Consequently, to be able to take advantage of our optimization, we create a variable `real_knowledge_tree`[8] to store the exact knowledge of the LEARNER from the SUT even if we modify the exact responses from the SUT as discussed below. It is worth noting that `pylstar` use the `knowledge_tree` variable to store its knowledge of the LEARNER.

To illustrate this, if for example we get $\mathcal{M} + \mathcal{M} + \mathcal{M} + \mathcal{M}$ as response to $\mathcal{A}$ from the SUT, then we store this sequence, $\mathcal{A}/\mathcal{M} + \mathcal{M} + \mathcal{M} + \mathcal{M}$, to the `real_knowledge_tree`. And following the modification to avoid infinite state machines, we modify $\mathcal{M} + \mathcal{M} + \mathcal{M} + \mathcal{M}$ into $\mathcal{M} + \mathcal{M}.\mathcal{M}^*$ and store it to the default variable `knowledge_tree` of `pylstar`.

5.3 Vulnerability Detection and Confirmation

The learned state machine is not easy to analyze, as it requires a rigorous method to automatically find bugs. In this section, we discuss our method to analyze state machines by considering the different methods introduced in chapter 3.

5.3.1 Vulnerability Detection

To identify bugs using the learned TLS model, we first identify RFC violations and then we analyze whether these violations represent bugs with the following steps:

(i) color in green the happy paths representing successful connections;

(ii) color in gray error transitions leading to the sink state, which are expected;

(iii) color all remaining transitions in red since they are RFC violations, and may correspond to vulnerabilities.

For SSH, instead of coloring all the remaing transitions in red (after step ii), we color them in three different colors according to the input vocabulary and the corresponding layer. Thus we chose three different colors for the three SSH layers. This allows us to check easily if there exists a shortcut between two layers.

[8] `real_knowledge_tree` is a tree-like representation of the LEARNER's knowledge.

When the state machine is big (about 100 states), it becomes difficult to exhaustively find a bug from the state machine using this technique. It does remain efficient to find a shorcut between two layers.

Hence, we created a tool based on `NuSMV` which is a reimplementation and extension of Symbolic Model Checking [CMCHG96]. In contrast to the classical model checking method, as described in Figure 5.7, we extract paths from the state machine and check if the path verifies the property. This simple modification allows us to exhaustively and automatically find a bug from the learned state machine.

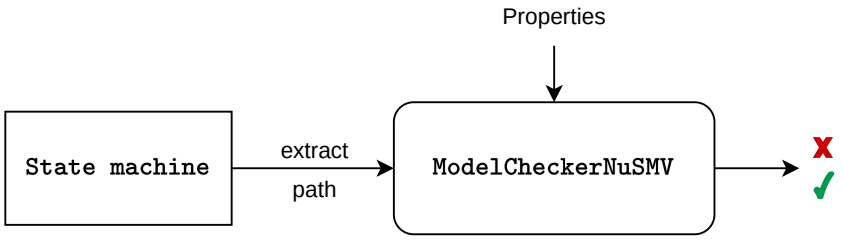


Figure 5.7: `ModelCheckerNuSMV`, a `NuSMV`-based tool for exhaustive bug finding using state machine.

However, we do not yet defined all the required properties for the SSH Connection layer, which we leave out for a future work.

5.3.2 Vulnerability Confirmation

These scenarios identify potential implementation issues which need to be independently confirmed as security flaws. $L^\star$ is an algorithm that produces a state machine which represents the behavior of the SUT. However, the produced state machine is only an approximation due to the (limited) set of abstract messages selected in the scenario and the equivalence query method used. We thus use tools to independently check whether a potential security issue, discovered by the inference, actually translates into a real security flaw.

For authentication bypass issues, we extract the potentially dangerous paths and replay them to the SUT, in a context where we do not have access to the authentication secret. If we can trigger the tested stack to emit for example an `ApplicationData` in case of TLS, the flaw is confirmed.

For SSH, we extract the potentially dangerous paths followed by `CH_OPEN` (according to the vulnerability) and replay them to the SUT, if we trigger the tested stack to emit an `OPEN_CONFIRM`, we confirm the vulnerability.

For loops, we send precomputed packets to the SUT at a given pace (typically one message per minute), and for a given duration (e.g., several hours). If we can maintain the connection open, we have a proof that the loop can be weaponized.

Before dicussing our results on the TLS and SSH state machine inference, we present in the following section different methods to optimize the learning process.

5.4 Optimization of the Learning Process

Before discussing our optimizations in `pylstar`, we can already improve the performance by running in parallel several inferences. Since we use containers, running multiple instances of SUTs and inference tools is essentially free, so we can benefit from a multi-core architecture.

5.4.1 `EOF` is Final

When we receive a network error, which indicates that the SUT has shut down the communication channel, we can conclude that all subsequent messages will trigger the same signal (`EOF`), so it is not necessary to build and emit the corresponding messages.

In [dRP15], de Ruiter and Poll actually proposed a similar improvement in the equivalence method implemented in `statelearner`, which resulted in measurable performance gains. By also applying the idea to the first phase of the algorithm (the membership queries), we further improve the performance.

5.4.2 Exploiting the Determinism

As discussed earlier, $L^\star$ relies on the fact that the SUT is deterministic. So we propose another optimization, which is a direct consequence of this assumption. During its execution, $L^\star$ often sends sequences that are extensions of already sent sequences.

Let us assume that we have already observed that sending `A` to the SUT triggers two messages, `x` and `y`. When evaluating the input sequence `A B`, we can send `A`, read `x` and `y` *without waiting after the reception of* `y`, then send `B` and observe the answer using the timeout.

A restricted version of this optimization consists in skipping the timeout only when we know sending a message will not trigger any message back.

5.4.3 Optimizations' Evaluation on TLS stacks

Table 5.4 describes the time required for a typical inference with different optimizations. We infer the TLS 1.2 server state machine for OpenSSL 1.1.1k (which contains 6 states) with 12 input messages and a 1-second timeout. The machine

	No EOF optim.	**EOF optimization**
No anticipation	1,885 s (100%)	1,598 s (85%)
Skip timeouts on empty responses	1,081 s (57%)	862 s (46%)
Skip timeouts on all known responses	128 s (7%)	77 s (4%)

Table 5.4: Average time required to infer TLS 1.2 server state machine for OpenSSL 1.1.1k. Percentages are the fraction of the unoptimized time.

hosting the experiment is an 16-core AMD EPYC 7302P at 3GHz, with 128 GB of RAM and all the storage on SSDs.

It appears that both optimizations improve the overall performance, with a drastic improvement from the fully-fledged timeout anticipation. We ran similar experiments with `statelearner` (same timeout and vocabulary), on the same hardware, and the time required to produce the state machine was 2,945 seconds.

Obviously, the time required for our inferences can vary, depending on the complexity of the SUT state machine (which can count as much as 30 states in some cases), the size of the input vocabulary (the scenario), and the speed of the SUT. The default timeout used is 1 second, but to get a stable inference, we must raise this value to 3 seconds for several stacks.

For a 1400-experiment run (which took around 2 and a half hours overall, with 30 inferences in parallel), the average inference time was around 3 minutes, the median was 81 seconds, and the 10th and 90th percentiles were respectively 27 seconds and around 8 minutes.

5.4.4 Discussion

We did not evaluate the impact of our optimization for SSH. Using our optimization, several SSH inferences take days to obtain the final state machine. It is therefore out of scope to make benchmarks for lack of time and for priority reasons.

However, we disable the **skip timeout on empty responses** optimization for few SSH stacks (especially when learning the Connection layer) because we detected a non-deterministic behavior of some SSH stacks (including `OpenSSH`) due to this specific optimization. We still apply all the other optimization tricks to speed up the learning process.

As described in Figure 5.8, OpenSSH server has two types of behavior depending on the **skip timeout on empty responses** optimization (enable and disable). It responds by `CH_EXIT` when the message `CH_CLOSE` is sent 1 second after the `CH_EXEC`, whereas it responds by `CH_EXIT_SIGNAL` if `CH_CLOSE` is directly

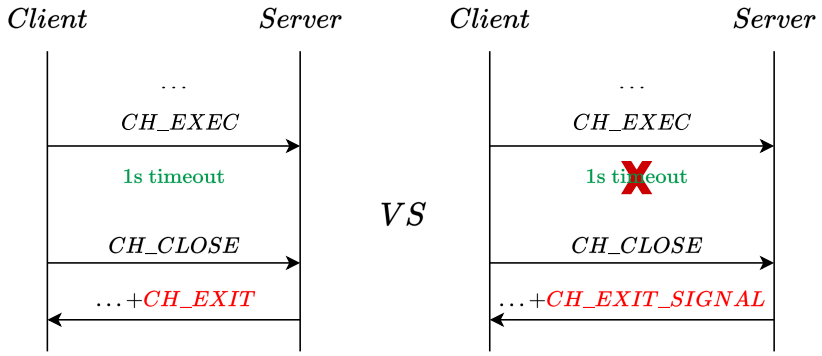


Figure 5.8: OpenSSH server non-deterministic problem detected when enabling the **skip timeout on empty responses** optimization.

sent after `CH_EXEC` without waiting 1 second.

We have discussed all the components required for the learning process, Learner, Mapper and the Sut. We also discussed how we proceed to detect bugs automatically (and how we confirm them) from the learned state machine and just above, we have indroduced our method to speed up the learning process. Thus, we are now up to analyze the resulting state machine following all the processes discussed previously.

5.5 Analysis of the Resulting State Machines

Our inference tool use the Distinguishing Bounds equivalence method (for TLS) and DBBased method (for SSH) to find counterexamples. As explained in chapter 4, these two methods offer the same guarantee and for the performance reason, since the SSH state machine is big, we use DBBased method for the SSH inference. To get relevant results, we must thus assume that the chosen B_{dist} value is sufficient.

We analyze over 600 different versions of different TLS and SSH stacks using different client and server scenarios and get over 3,000 automata. Sec. 5.6.3 summarizes the vulnerabilities reproduced and discovered during our study.

5.5.1 Authentication Bypasses

a) Server authentication bypasses in wolfSSL

Around 2015, authentication bypasses in state machines seemed to be pervasive in TLS stacks [BBD+15, dRP15]. In 2020, CVE-2020-24613, an authentication

bypass affecting wolfSSL TLS 1.3 client, caught our eye, and we decided to try and reproduce it using L*.

To this aim, we infer the state machines for wolfSSL TLS 1.3 clients, for different versions, with standard Handshake messages. Figure 5.9a represents the state machine corresponding to wolfSSL 4.4, which is vulnerable to CVE-2020-24613. By skipping the `CertificateVerify` message, an attacker can bypass server authentication, and thus impersonate any server to a vulnerable client. The vulnerability was fixed in version 4.5, as can be seen on Figure 5.9b, which corresponds to the inferred state machine for the patched version, using the same vocabulary.

However, in other scenarios, we also use a broader input vocabulary including an empty `Certificate` message, that should never be sent by the server. We could thus discover another vulnerability in wolfSSL, present in all versions at the time. As shown in Figure 5.9c, instead of skipping the `CertificateVerify` message, the attacker can send an empty `Certificate` message, followed by a `CertificateVerify` message signed by an arbitrary RSA key.[9] This new bug was confirmed, reported as CVE-2021-3336, and fixed.

By adding new messages to the input vocabulary, we also discover another alternate path to reintroduce the initial bug. Figure 5.9d shows that an attacker can send an empty `Certificate` message, followed by an invalid `CertificateVerify` message, containing an unknown signature algorithm and an arbitrary payload to bypass server authentication. This bug, identified as CVE-2022-25638, has been fixed in version 5.2.0.

All these attacks were reproduced by sending the identified transcript to the vulnerable SUTs. The program replaying the attack was not given access to the server private key, and we checked both wolfSSL and curl+wolfSSL stacks to make sure the authentication bypasses were real.

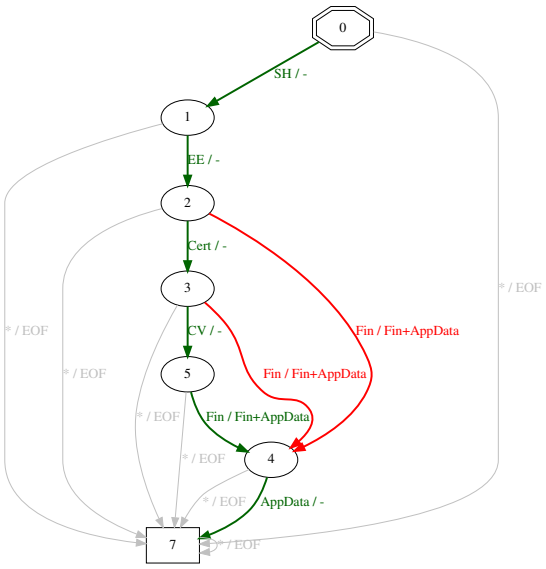
b) Other bypasses in TLS

In OpenSSL, several paths are incorrectly identified as invalid bypasses: the client seems to be accepting any certificate from the server. However, when we analyze a real TLS client using OpenSSL (the curl+OpenSSL stack), these dangerous paths disappear. Indeed, in our OpenSSL containers, TLS clients use the `s_client` application, which does not enforce any checks regarding the certificate.[10]

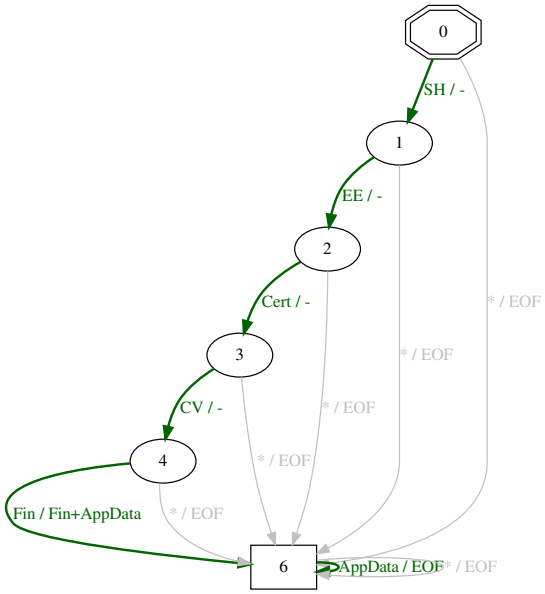
We use the same approach to assess the quality of TLS servers authenticating clients. We get an issue in wolfSSL TLS 1.3 servers, as shown in Figure 5.10,

[9] In our inference tool, sending a `Certificate` message selects the corresponding RSA key to be used in the subsequent `CertificateVerify`. For `EmptyCertificate`, the selected RSA key is a fresh key generated for the experiment.

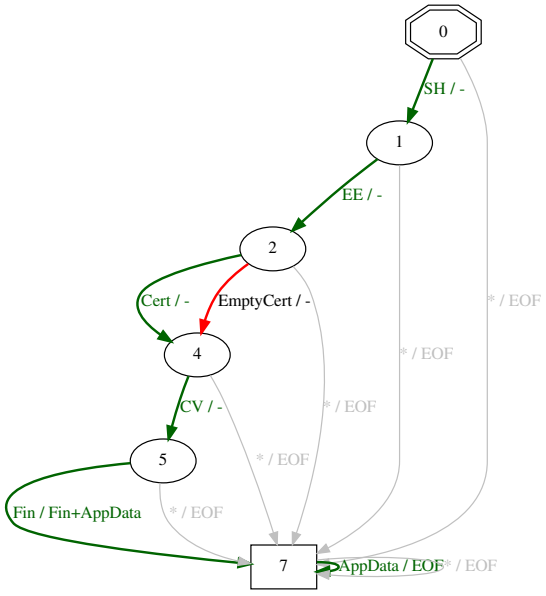
[10] It is possible to add options such as `-verifyCAfile` to the command line, but they do not end an unauthenticated handshake and merely produce a warning message.



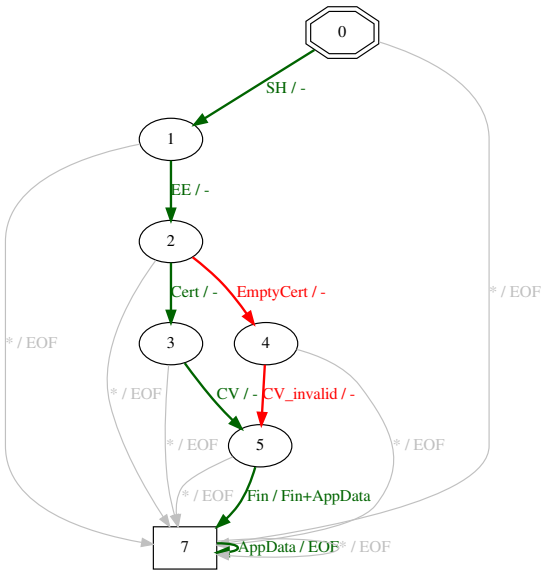
(a) CVE-2020-24613, a server authentication bypass in wolfSSL TLS 1.3 clients, up to version 4.4. An attacker can impersonate any server to a vulnerable client by skipping the `CertificateVerify` message.



(b) CVE-2020-24613 fixed in version 4.5. With the same vocabulary used in Figure 5.9a, the dangerous transitions have indeed disappeared.



(c) CVE-2021-3336. By sending an empty `Certificate` message, followed by an arbitrary `CertificateVerify`, server impersonation is also possible.



(d) CVE-2022-25638. Adding a completely invalid `CertificateVerify` message reintroduces a dangerous transition.

`SH`	:	`ServerHello`	`EE`	:	`Encrypted Extensions`
`Cert`	:	`Certificate`	`CV`	:	`CertificateVerify`
`Fin`	:	`Finished`	`AppData`	:	`ApplicationData`
`EOF`	:	End of the connection			

Figure 5.9: Attacks against wolfSSL TLS 1.3 clients.

which is the transposition of CVE-2020-24613 to the server. By skipping the `CertificateVerify` message (and optionally the `Certificate` message), an attacker can bypass the authentication and impersonate any legitimate client. It is worth noting that the server correctly rejects untrusted certificates and empty `Certificate` messages (since client authentication is required in this scenario). This bug, CVE-2022-25640, has been fixed in version 5.2.0.

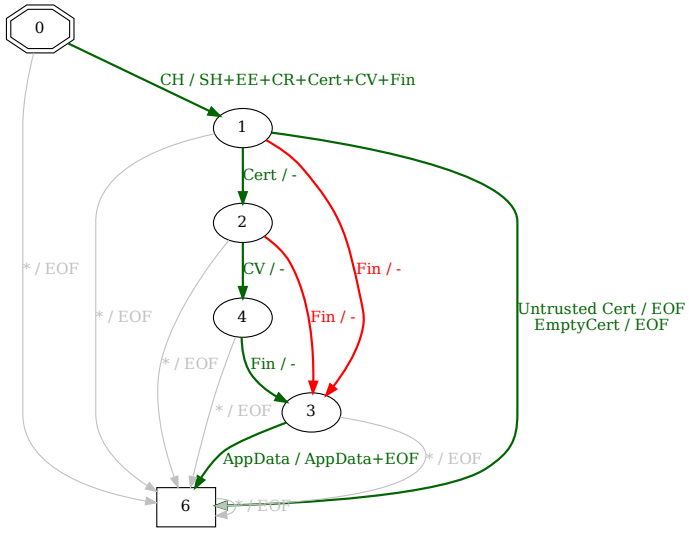


Figure 5.10: CVE-2022-25640. In versions, up to 5.1.0, client authentication can be bypassed in wolfSSL TLS 1.3 servers, using the same idea as in CVE-2020-24613.

c) Server authentication bypasses in SSH stacks

In 2018, at least 3 different SSH stacks, `libssh`, `AsyncSSH` and `Paramiko`, suffered from authentication bypasses in state machines. Using L*, we are able to reproduce these vulnerabilities (CVE-2018-10933, CVE-2018-7749, CVE-2018-7750 and CVE-2018-1000805). To this aim, we infer the state Machines for `libssh`, `AsyncSSH` and `Paramiko` server using only the authentication and connection layer messages.

We reproduce CVE-2018-7750 and CVE-2018-1000805 against the `Paramiko` server in Figure 5.11. As described in Figure 5.11a, CVE-2018-7750 consists in skipping the authentication request messages (`AUTH_Passwd` or `AUTH_RSA`), thus an attacker can bypass the `Paramiko` server authentication and directly performs the connection layer operations. This vulnerability affects all `Paramiko` servers versions up to 2.4.1.

In the same year, another vulnerability against `Paramiko` server was discovered, CVE-2018-1000805. The server authentication bypass attack consists in sending

the AUTH_SUCCESS[11], which is normally only sent by the server, instead of the authentication request message. The attacker can then benefit from all services offered by the Connection layer. This vulnerability affects all Paramiko server version up to 2.4.2.

CVE-2018-10933 is similar to CVE-2018-1000805, but unlike CVE-2018-1000805, the attacker can only open channels and send data using CH_DATA message. We provide the state machine related to this attack in Appendix B.

CVE-2018-7749 is similar to CVE-2018-7750. As in CVE-2018-7750, the AsyncSSH server performs operations related to the Connection layer before authentication has completed, but an attacker can not have access to the corresponding responses until the authentication is completed. It is worth noting that we are not able to completly reproduce the bug because of the black-box approach.

However, we also analyzed wolfSSH using authentication_connection scenario and thus we discovered a new vulnerability affecting all wolfSSH versions at the time. As shown in Figure 5.12, an attacker can successfully open a channel without authenticating to the server. However, to benefit from the connection layer services, the attacker should present, before or after the CH_OPEN_SESS message, an arbitrary authentication request message (AUTH_PW_NOK and AUTH_RSA_NOK) containing a valid *username*.

When the attacker presents an AUTH_PW_NOK or AUTH_RSA_NOK, the server rejects its authentication request message by sending AUTH_FAILURE meaning that the server seems to correctly reject an invalid authentication request message. However, if the attacker ignores the error message, and sends an CH_OPEN, then the server accepts to open a session channel and considers that the client is authenticated.

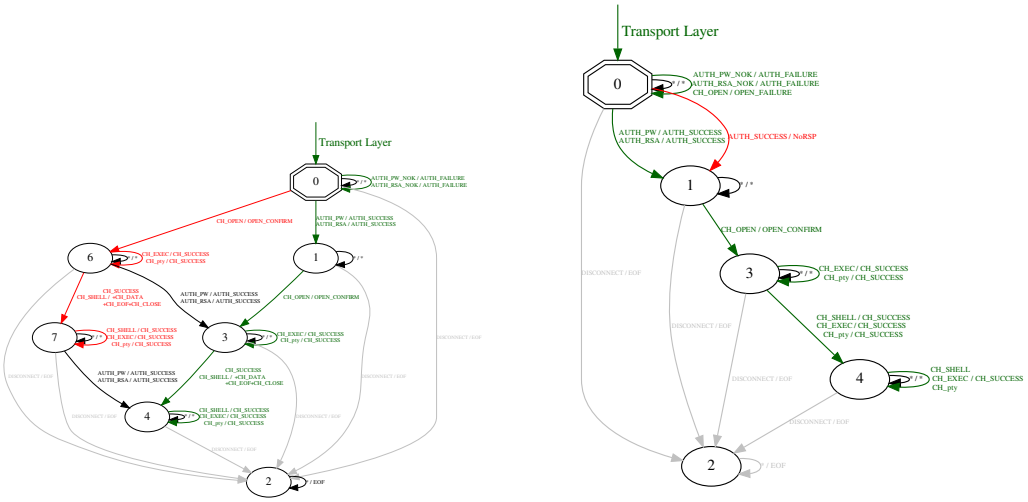
5.5.2 Weakened Authentication in SSH

As discussed in sec. 1.1, for the SSH protocol, authentication must take place after the execution of the transport layer, otherwise, attacks are possible. If a server accepts and processes a *pubkey* authentication message before the cryptographic material is defined, this makes replay attacks possible.

As shown in Figure 5.13, if the *pubkey* authentication message is sent before sending DH_INIT message, the session_id is not yet defined. It follows that:

- the signature presents in the authentication message is static (i.e., it remains unchanged during the key lifetime); and
- the *pubkey* authentication message is sent in cleartext.

[11] We added the AUTH_SUCCESS message to the Learner's input vocabulary.



(a) CVE-2018-7750. An attacker can impersonate any `Paramiko` server version up to 2.4.1 by skipping the authentication request message.

(b) CVE-2018-1000805. An attacker can impersonate any `Paramiko` server version up to 2.4.2 by sending `AUTH_SUCCESS` message instead of the authentication request message.

Figure 5.11: Attack against `Paramiko` server.

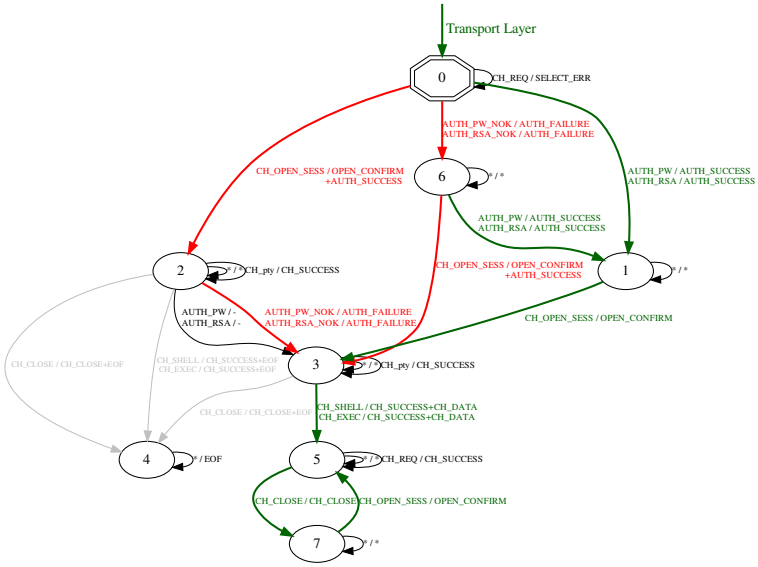


Figure 5.12: A server authentication bypass in `wolfSSH` in all versions at the time.

Finding a valid signature for such an attack is not easy but still possible. An attacker having such capabilities once can then bypass the server authentication any time later.

This attack has a similarity to `EarlyCCS` [Kik14] in the sense that the attacker tries to convince the other side to use a weak key (*pre master secret* for TLS and `session_id` for SSH). In the TLS context, both sides have to be vulnerable to such an attack for the MiTM attack to succeed.

When analyzing `AsyncSSH` and `wolfSSH`, we discover that they are also vulnerable to this kind of replay attack.

We also suspect `libssh` server version up to 0.8.4 to be vulnerable to such an attack. The `libssh` example server does not implement the *pubkey* authentication, it only implements the *password* authentication. The `libssh` server, `ssh_server_fork`, accepts the `AUTH_PASSWORD` message before the first key exchange process, that's why we suspect that it is also the case for the *pubkey* authentication message. It is however only a suspicion because we have not built a `libssh` server accepting the *pubkey* authentication.

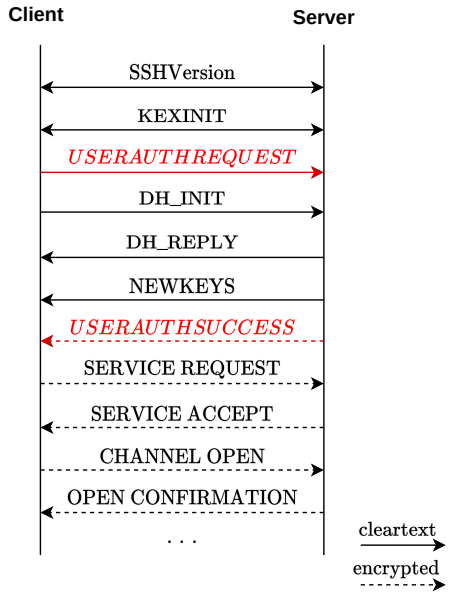


Figure 5.13: Weakened authentication in `AsyncSSH` server in all versions at the time.

5.5.3 Loops in the Automata

As discussed in Sec. 5.2.3, exploiting loops in TLS and SSH state machines can be used to mount sophisticated cryptographic attacks [ABD+15]. Loops have also

been considered as a potential vector for denial of service attacks (e.g., CVE-2020-12457). We thus identify such loops in our state machines, and we focus on those occuring before messages are protected.

Analysis of a False Positive

The inferred state machine for wolfSSL TLS 1.2 server (all versions) seems to exhibit a loop on the initial state, tagged with the `NoRenegotiation` warning. However, when we repeatedly send such warnings to the SUT, the server actually closes the connection after 4 warnings. This situation exhibits the fact that $L^\star$ is only an approximation, which can not always capture behavior happening very deep in the state machine. This justifies our approach, to always confirm potential vulnerabilities identified on the generated state machine.

For SSH, four stacks contain loops before messages are protected: OpenSSH, dropbear, AsyncSSH and wolfSSH. However, three of them have timeout protection, OpenSSH, dropbear and AsyncSSH. When timeout is expired, they terminate the connection. OpenSSL and AsyncSSH terminate connections after 120 secondes, and dropbear after 300 secondes.

Real bugs

After careful verification, we confirm the presence of several loops in different TLS and SSH stacks, which are summarized in Table 5.5.

Stack	**Scenario**	**Messages**	**Max. Time Between Msgs**
erlang 24	1.0/1.2 Server	`NoRenegotiation` Alert or `ApplicationData`	$>$ 1 hour*
fizz 22.01.24	1.3 Client	`ChangeCipherSpec`	$>$ 1 hour
matrixssl 4.0 - 4.3	1.0/1.2 Server	`NoRenegotiation` Alert	$\approx$ 40 seconds
NSS 3.15 - 3.78	1.0/1.2 Server	`NoRenegotiation` Alert	$>$ 1 hour
OpenSSL $<$ 1.1.0	1.0/1.2 Server	Empty `ApplicationData`	$>$ 1 hour
wolfSSH $\leq$ 1.4.12	server	`SERVICE_REQUEST` `DISCONNECT`	$>$ 1 hour

* Erlang has a `Timeout` parameter that can thwart the attack. It was added to the official tutorial.

Table 5.5: Description of confirmed loops in TLS stacks.

For servers, loops can lead to Denial of Service attacks against TLS and SSH services, while requiring very few resources from the attacker. Indeed, the attacker can easily establish TCP connections and regularly send the right payload.

Beyond the payload and the SUT identification (IP address and port), the attacker needs to store, for each connection, the source port and the associated sequence numbers only. With stacks keeping a connection alive for several minutes

between packets (most probably because they do not enforce any kind of timeout), this represents a tiny amount of CPU, memory and network resources for the attacker. Moreover, distributing this attack is trivial. Finally, with vulnerable stacks, the attack can be run indefinitely and does not usually generate logs.

Beyond adding reasonable timeouts (both per-message and per-handshake) within the affected stacks, firewalls or other network devices should be used to detect and block such extreme behavior. For the affected stacks, the issues have been reported and fixed when deemed relevant.

5.5.4 Unsolicited Client Authentication

TLS client authentication is an optional feature. The client can only present its certificate after the server sends a `CertificateRequest`. Servers may however be accommodating, and accept `Certificate` and `CertificateVerify` messages from the client, even when they were not solicited.

Such behavior may expose parts of the code that are not normally used. In 2014, a critical security flaw was found in Microsoft SChannel: a buffer overflow in the ECDSA signature check, triggered by client authentication, led to remote code execution. Accepting unsolicited client authentication messages made this obscure bug actually reachable in most deployments.

In our corpus, several versions of wolfSSL exhibit a similar behavior. Even if these paths do not necessarily lead to security issues, they should be removed, and considered bad practice, as they are a deviation from the specification.

5.5.5 Skip Encryption in SSH

Protocols such as TLS and SSH are designed to ensure confidentiality, integrity and authentication. By skipping encryption, it is thus possible to break the integrity guaranteed by these protocols.

We discover two sources of vulnerabilities in SSH stacks allowing to skip encryption either on the client or server side. Skipping the SSH `NEWKEYS` message allows server/client impersonation attack possible. We noticed that skipping encryption allows server/client impersonation attacks if the attacker is able to perform an additional step that we discuss below.

a) Skip Server Encryption

As described in Figure 5.14, skipping the server `NEWKEYS` message allows server blind impersonation attack because server authentication is already done in the `DH_REPLY` message. In such an attack, the client's messages are encrypted. It means that, for the server blind impersonation attack to succeeed, the attacker

has to guess two things: the client's *username* and the requested *recipient channel* (which is the channel number given in the `CHANNEL OPEN` message).

The messages coming after the first `CHANNEL OPEN` message are hard to guess, which limits the attacker's capabilities. However, if an attacker is able to guess the commands executed by the client, then it can mislead the client by sending fake responses. In practice, this assumption may make sense for a client executing a script known to the attacker when connecting to an SSH server.

b) Skip Client Encryption

Performing a similar attack against an SSH server is much more difficult than against an SSH client because it requires either building a valid signature (which is cryptographycally hard) or injecting faults to force the client stack to skip the `NEWKEYS` message.

For the second option, if an attacker is able to force a legitimate client to skip the `NEWKEYS` message, it can benefit from all priviledges of the user. Although the attacker cannot decrypt the server's responses, it can execute commands or push data to the SSH server.

During our analysis of the state machine, we discovered that `AsyncSSH` and `wolfSSH` client- and server-sides (all versions at the time) are vulnerable to both attacks (skip server/client encryption).

5.5.6 Credential Leakage in SSH

We identified a new vulnerability affecting all wolfSSH version at the time which allows an attacker to obtain the client's credential. Hence, the attacker can use the leaked credential to impersonate the client to the real server.

Using the `Transport and Authentication` scenario to the wolfSSH client, we found that the client leaks the client's password before the server's authentication (i.e., before the `DH_REPLY` message). As described in Figure 5.16, the attacker only have to send the `SSHVersion` message followed by an `AUTH_FAILURE` message to obtain the client's credential[12]. Concretly, the wolfSSH client always responds with `AUTH_PW` to an `AUTH_FAILURE` message at any state of execution of the protocol.

5.5.7 Detection of Bleichenbacher Oracles in TLS

In this use case, our goal was to automatically identify Bleichenbacher oracles in TLS stacks, by finding states in the state machine where we could distinguish

[12]We noticed that we failed to configure the wolfSSH client to accept the `DH_REPLY` message. Thus, the state machine given in Figure 5.16 is "incomplete".

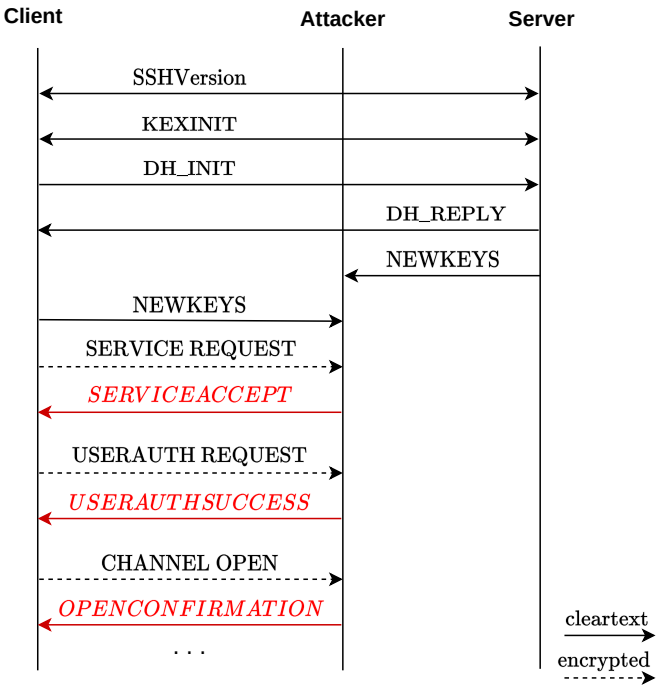


Figure 5.14: Skip the `NEWKEYS` message of the server.

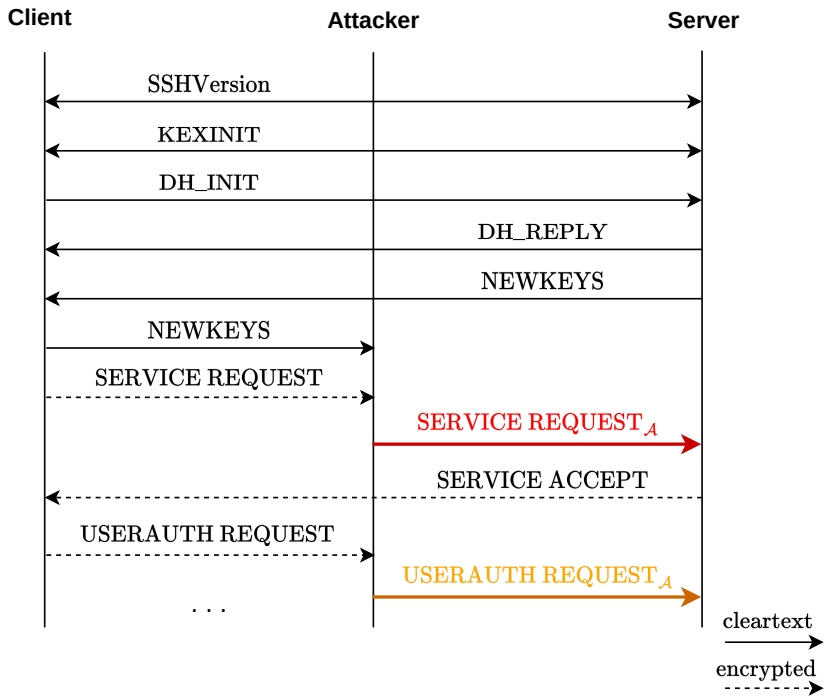


Figure 5.15: Skip the `NEWKEYS` message of the client. We denote $\mathtt{M}_{\mathcal{A}}$ the message built by the attacker. Since the `SERVICE_REQUEST` and `USERAUTH_REQUEST` are encrypted, the attacker can not decrypt them, then to succeed, the attacker has to build himself these two message in cleartext.

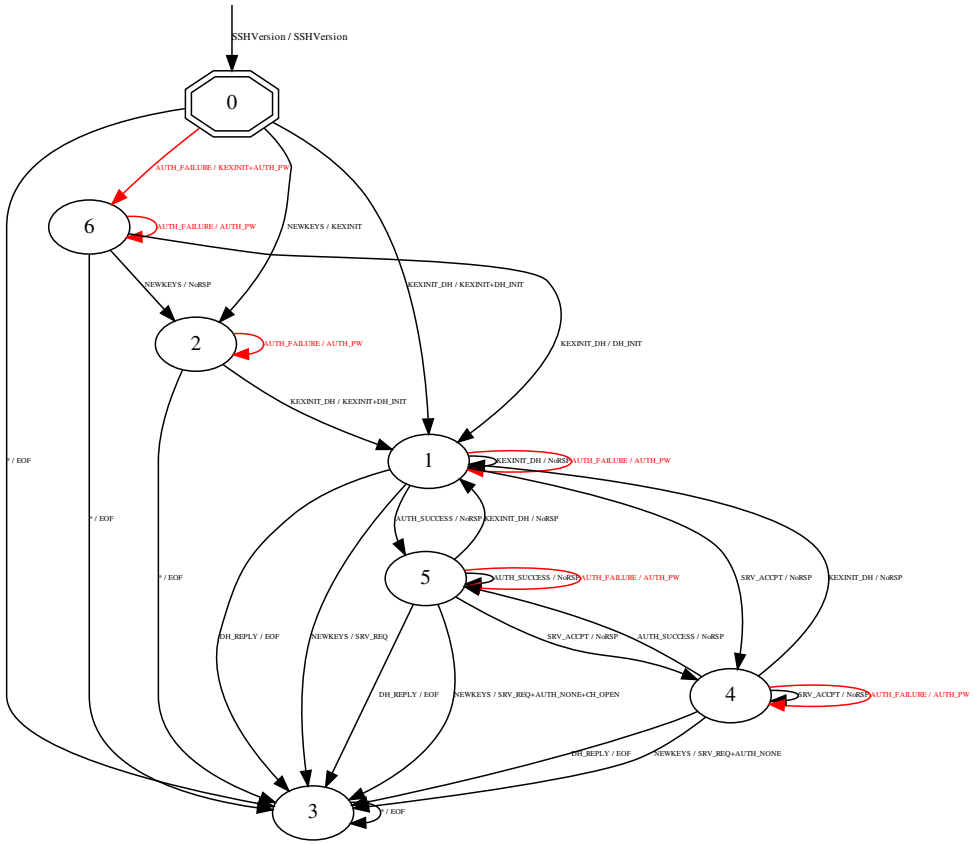


Figure 5.16: wolfSSH client leaks creadential by responding `AUTH_PW` to `AUTH_FAILURE` before the server authentication.

well-padded from wrongly-padded messages.

Our approach was to develop a scenario including forged `ClientKeyExchange` messages with various properties, and then to use our platform to run this scenario against TLS stacks, including ones known to be vulnerable to the Bleichenbacher attack. We were confident we would reproduce existing flaws (i.e., identify known oracles), but we hoped to also find new oracles, that may not be visible at first sight, but that might be found in strange corners of the state machines (e.g., when a specific message was sent after the `ClientKeyExchange`).

We indeed found oracles that were already documented: erlang OTP-20.0, matrixssl 3.7.2 and wolfSSL v3.12.*. Figure 5.17 shows the inferred state machine

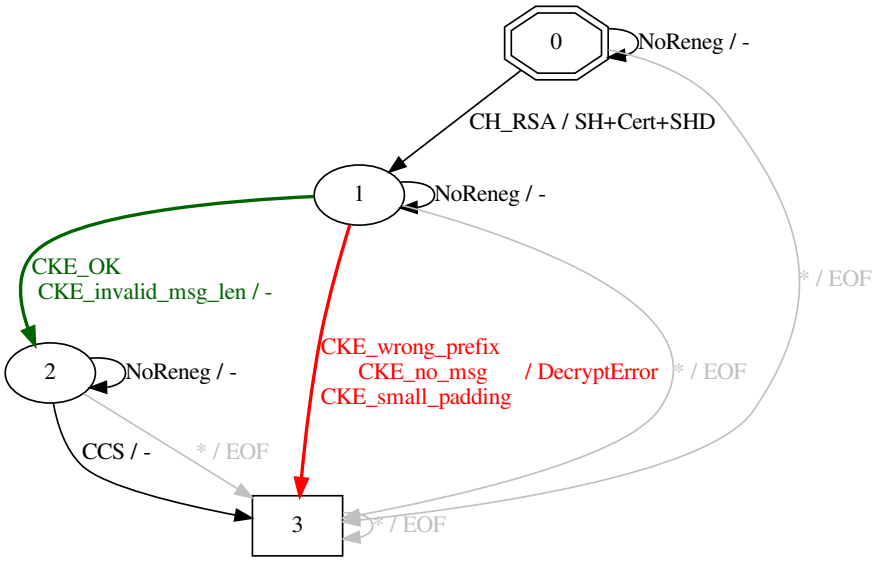


Figure 5.17: The erlang OTP-20.0 server exhibits a Bleichenbacher oracle: CVE-2017-1000385.

for the erlang stack, where it is possible to distinguish between correctly and incorrectly padded ClientKeyExchange messages. In this case, the distinction corresponds to an explicit alert instead of the expected silence from the server.

Depending on the kind of messages that are distinguishable, Bardou et al. defined a classification [BFK+12]. Thus, the obtained state machines allow us to evaluate the quality of the identified oracle, that is to approximate the number of messages required to recover a plaintext message.

Our efforts did not raise any new oracle, neither in terms of affected stacks nor in terms of message sequences to observe, but we confirm that all existing oracles were detected by our approach.

5.6 Discussion

There exists other bugs which are not vulnerabilities but correspond to unexpected behavior. Maehren et al. had investigated a lot for detecting bugs and unexpected behavior in TLS stacks [MNH+22]. They detected several specification violations and bugs. In contrast to them, we did not focus on the specification violations but on detecting vulnerabilities. However, we detect additional bugs which are

not necessarly vulnerabilities and that are worth mentionning such as the infinite state machine and the missing key refreshement.

In this section, we discuss these two bugs identified during our analysis of the resulting state machine. We also discuss the limitations of our approach.

5.6.1 Infinite State Machine in SSH

During our analysis, we discover that several SSH stacks had an infinite state machine. We summarize our findings in Table 5.6.

Stack	Versions	Comments
OpenSSH	$< 8.8.p1$	SSH server
OpenSSH	all	SSH client
libssh	$< 0.8.4$	SSH server
AsyncSSH	all	SSH server
AsyncSSH	all	SSH client
sshd-lite	all	SSH server

Table 5.6: List of SSH stack having an infinite state machine.

For example, after opening a session channel with a `sshd-lite` server, if the Learner sends multiple times the message `CH_DATA`, the server accepts the message and it does not answer the Learner until it receives a `CH_SHELL` message. Thus the Learner receives `CH_SUCCESS` (meaning that the `sshd-lite` server accept to open a shell) followed by `WINDOW_ADJUST` (which allows to extend the size of data the Learner can send) and multiple `CH_DATA` depending on the amount of data sent by the Learner earlier.

In the Connection layer, if the server starts a key refreshement by sending `KEINIT` to an `OpenSSH` client, it gets `KEINIT+DH_INIT`. However, if the server sends multiple `CH_OPEN` after the `KEINIT` instead of a single `DH_REPLY` message, the `OpenSSH` client accepts them but it does not answer until the server sends `DH_REPLY` which answer the server by sending `NEWKEYS` followed by multiple times the message `OPEN_FAILURE`.

5.6.2 Missing Key Refreshment in SSH

Especially for SSH, during the key exchange, the message `NEWKEYS` is used to enable the sender's encryption; and during key re-exchange, it is used to confirm the key refreshment (intuitively, a re-key is meant to renew keys in both directions).

Actually, missing `NEWKEYS` during key re-exchange is not a violation of the RFC because the RFC does not explicitly disallow this behavior [Ylo06c]. However, it defeats the purpose of the rekey procedure.

Using the `Transport and Connection` scenario, we reproduce the missing `NEWKEYS` bug affecting OpenSSH server discussed in [FJST23]. We also discover new ones affecting `AsyncSSH` and `OpenSSH` client.

5.6.3 New and Reproduced Vulnerabilities

In this section, we summarize the list of new and reproduced vulnerabilities we detected during our analyzis of TLS and SSH implementations. New vulnerabilities uncovered during our study are tagged "New". Previously known vulnerabilities are tagged with one of the following status. "Not Reproduced" means we could not reproduce the issue, either because we did not include the vulnerable stack or because of a limitation in our approach (e.g., the absence of a given abstract mesage); "Detected" means the infered state machine shows an unexpected transition related to the vulnerability; "Reproduced" means that the infered state machines provides evidence that the vulnerability is present and can be exploited, should the state machine be accurate.

For TLS, we only focus on TLS 1.0 to 1.3 versions, we do not investigate several vulnerabilities such as DROWN [ASS+16], a cryptographic attack using flaws (including state machine bugs) in SSLv2 servers to recover TLS-encrypted plaintext.

a) Unexpected Loops

CVE #	Stack	Versions	Description	Status
2020-12457	wolfSSL	$\leq$ 4.4.0	Reproduced	TLS 1.2 server DoS
-	erlang	24.0	New	Default configuration allow for TLS server DoS
2022-25639	matrixSSL	4.0 - 4.3	New	TLS server DoS
-	fizz	2021 snapshots	New	Unexpected client loops
pending	NSS	3.15 - 3.78	New	TLS 1.0 to 1.2 server DoS
pending	wolfSSH	all	New	SSH server DoS

b) Credential Leakage

CVE #	Stack	Versions	Status	Comments
pending	wolfSSH	all	New	SSH client

c) Weakened Authentication

CVE #	Stack	Versions	Status	Comments
pending	AsyncSSH	all	New	SSH server
pending	wolfSSH	all	New	SSH server

d) Authentication Bypasses

CVE #	Stack	Versions	Status	Comments
2014-0224	OpenSSL	$\leq$ 0.9.8za $\leq$ 1.0.0l $\leq$ 1.0.1h	Detected	EarlyCCS (unexpected CCS transitions)
2015-0204	OpenSSL	$\leq$ 0.9.8zc $\leq$ 1.0.0o $\leq$ 1.0.1j	Detected	FREAK (client- and server-side EXPORT RSA downgrade)
2015-0205	OpenSSL	$\leq$ 1.0.0p $\leq$ 1.0.1j	Not Reproduced	Client auth. bypass. Requires DH certificate support
2020-24613	wolfSSL	$\leq$ 4.4.0	Reproduced	TLS 1.3 server auth. bypass
2021-3336	wolfSSL	$\leq$ 4.6.0	New	TLS 1.3 server auth. bypass
2022-25638	wolfSSL	$\leq$ 5.1.0	New	TLS 1.3 server auth. bypass
2022-25640	wolfSSL	$\leq$ 5.1.0	New	TLS 1.3 client auth. bypass
2018-10933	libssh	$\leq$ 0.8.3	Reproduced	SSH server auth. bypass
2018-7750	Paramiko	$\leq$ 2.4.0	Reproduced	SSH server auth. bypass
2018-1000805	Paramiko	$\leq$ 2.4.1	Reproduced	SSH server auth. bypass
2018-7749	AsyncSSH	$\leq$ 1.12.0	Detected	SSH server auth. bypass
pending	wolfSSH	all	New	SSH server auth. bypass

e) Skip Encryption

CVE #	Stack	Versions	Status	Comments
pending	AsyncSSH	all	New	SSH server
pending	AsyncSSH	all	New	SSH client
pending	wolfSSH	all	New	SSH server
pending	wolfSSH	all	New	SSH client

f) Bleichenbacher Padding Oracles

The vulnerabilities described here affect TLS servers offering RSA key exchange (which was removed in TLS 1.3).

CVE #	Stack	Versions	Status	Comments
2016-0800	OpenSSL	$\leq$ 1.0.1t $\leq$ 1.0.2f	Not Reproduced	Requires SSLv2 messages
2016-6883	matrixSSL	$\leq$ 3.8.2	Reproduced	
2017-13099	wolfSSL	$\leq$ 3.12.2	Reproduced	ROBOT attack [BSY18]
2017-1000385	Erlang	20.0	Reproduced	ROBOT attack [BSY18]

5.6.4 Limitations of Our Approach

Our systematic black box approach is very efficient to find bugs and vulnerabilities related to the state machines. We are able not only to reproduce known vulnerabilities but also to generalize the search for a class of bugs and to find new ones.

However, we are not able to totaly reproduce some known vulnerabilities such as `EarlyCCS`, `FREAK`, `DROWN`, CVE-2015-0205 related to OpenSSL and CVE-2018-7749 related to `AsyncSSH`, for different reasons.

For most of them, the limitation is related to the way the Mapper is built (it is the case for `EarlyCCS`, `FREAK`, CVE-2016-0800 and CVE-2018-7749). For example, we can not reproduce CVE-2015-0205, because it requires a DH certificate which is not supported by our Mapper.

Moreover, to fully reproduce `EarlyCCS` and `FREAK`, we should re-write and adapt our Mapper for these specific use-cases. There already exists a better, dedicated, tool suitable for detecting these bugs.

Finally, for any black box approach, it is impossible to fully reproduce the CVE-2018-7749 related to `AsyncSSH`. Checking if a command line is silently executed in the server-side is required to reproduce this vulnerability which goes beyond the black blox context.

Chapter 6

Stack Fingerprinting with Application to TLS and SSH

As described by Smart et al., fingerprinting is similar to identifying an unknown person by taking his or her unique fingerprints and finding a match in a database of known fingerprints [SMJ00].

In this chapter, we discuss our results on stack fingerprinting. The applications of stack fingerprinting are multiple. For example, this is the first step an attacker might use before running an attack. This step quickly allows an attacker to identify the target. It is also used to identify the sources of an attack. This enables administrators to determine the legitimate actors to connect to administrated systems/networks.

Several tools have been developed for stack fingerprinting such as `Nmap`[1]. `Nmap` is a powerful network exploration and security auditing tool. In 1998, Fyodor proposed a method to fingerprint Operating Systems (OS) and implemented his method in `Nmap` [Lyo98]. Fyodor proved that it is possible to identify OS through TCP/IP stack fingerprinting. The idea is to analyze the target's behavior against malformed or exotic TCP/IP packets. Several researchers followed this line of research to fingerprint TLS [JVdRP21], SSH [GGD19], Bluetooth Low Energy (BLE) [PA21], etc.

There exists several methods to fingerprint protocol implementations. We identify and discuss three classes of these methods:

- message-based fingerprinting: the stack fingerprint is computed by analyzing protocol messages;
- feature-based fingerprinting: the stack fingerprint is computed by analyzing several features related to the implementations of the studied protocols; and

[1] `https://nmap.org/`

- state-machine-based fingerprinting: the stack fingerprint is computed by analyzing state machines of the studied protocol.

The organization of this chapter is as follows: sec. 6.1 presents the method based on the message structure and its content and on the features of the implementations of protocols for fingerprinting stacks. Then, we present in sec. 6.2 our results related to the state-machine-based fingerprinting. Finally, we present the limitations of our approach in sec.6.3.

6.1 Message- and Feature-Based Fingerprinting

For these methods, fingerprints are mostly computed from the messages structure and content. This is completely unrelated to the results of our work on TLS and SSH state machines.

6.1.1 TLS Fingerprinting

To identify a TLS client, Husák et al. [HCJC16] use the list of ciphersuites proposed by the client to fingerprint TLS stacks. The idea has been generalized by Kotzias et al. and by Frolov and Wustrow [KRA+18, FW19] to include other fields of the `ClientHello` to fingerprint the client. The method was applied successfully to detect malware, censorship circumvention tools and web browsers. Salesforce proposed two formats to capture the idea: JA3 for passive fingerprinting and JARM for active fingerprinting.[2]

Durumeric et al. [DMS+17] presented the impact of HTTPS interception on security. They identified the nature of the client by identifying a mismatch between the HTTPS User-Agent header and TLS client behavior (supported ciphersuites, declared extensions).

In 2022, Sosnowski et al. proposed an optimization of JARM and a methodology for acquiring and leveraging TLS metadata with the help of large scale active measurement [SZS+22]. In 2023, they proposed DissecTLS, an active TLS scanner to reconstruct the configuration and capabilities of the TLS stack [SZSC23]. They compared their method to four active TLS scanners and fingerprinting tools such as JARM, ATSF, SSLyze[3] and testssl.sh[4].

[2] `https://github.com/salesforce/ja3` and `https://github.com/salesforce/jarm`.

[3] SSLyze is tool written in Python for SSL/TLS scanning (`https://github.com/nabla-c0d3/sslyze.git`)

[4] testssl.sh is a tool for testing SSL/TLS encryption (`https://testssl.sh`).

6.1.2 SSH Fingerprinting

Ghiëtte et al. proposed a method to identify SSH client stack by dissecting cipher suites and SSH version strings to generate unique fingerprints for bruteforcing tools used by an attacker [GGD19].

Two research papers claimed identifying SSH stack by using only the SSH version string [CWB+19,KW22]. However, we notice that the identification of the SSH stack is not their main contribution, but these two papers focused their discussion and conclusion arround that. Relying on the banner sent by an implementation usually lacks robustness, since it is often easily customizable.

When working on SSH state machines, we sent multiple invalid messages and we discovered that we were able to separate almost all SSH server stacks (`dropbear` and `sshd-lite` are not separable using these messages) by using invalid messages such as `NEWKEYS_NOK`, `Empty_msg`, `SRV_REQ_NOK` and `DH_INIT` messages containing a public Diffie-Hellman equal to 1 and g (which is the group generator). Table 6.1 gives more details about our results.

In Table 6.1, `NEWKEYS_NOK` is a `NEWKEYS` message containing additional unexpected data before the SSH padding; and `SRV_REQ_NOK` is a service request message where the *service name* is left empty.

Server	$dh = 1$	$dh = g$	`NEWKEYS_NOK`	`Empty_msg`	`SRV_REQ_NOK`
OpenSSH	✗	✗	✗	✗	✗
dropbear	✗	✓	✓	✗	✗
libssh	✗	✓	✓	✓	✓
Paramiko	✓	✓	✓	✗	✗
AsyncSSH	✓	✓	✗	✗	✗
wolfSSH	✗	✓	✓	✗	✓
SSH2	✗	✓	✓	✓	✗
sshd-lite	✗	✓	✓	✗	✗

Table 6.1: Server fingerprint corresponding to each SSH stack present in our platform.

Beyond the fingerprinting results discussed previously, we also detected different aspects which might help for the identication of SSH stacks.

a) Deprecated SHA-1

Most SSH implementations still use the `SHA-1` hash algorithm, although [LP20] and [BL16] strongly recommend to remove `SHA-1` support in any security protocol including SSH.

Following these recommandations, `OpenSSH` removed algorithms related to `SHA-1` from version 8 and upwards. Such a modification is a source of fingerprint because by looking at the algorithms proposed by the target, it is possible to identify the class of version of `OpenSSH` used by the target [GGD19].

b) Diffie-Hellman Group Exchange

Several SSH stacks, such as `OpenSSH`, `libssh`, `AsyncSSH` and `wolfSSH` still support the `diffie-hellman-group-exchange` key exchange method.

`AsyncSSH` does not implement `diffie-hellman-group1`[5] but still accept Diffie-Hellman based on a group of size 1024 bits when negotiating group using the `diffie-hellman-group-exchange` key exchange algorithm. Accepting a group of size 1024 bits is problematic for security because of the LOGJAM attack [ABD+15] and also the specification is explicitly against it [VB17].

`wolfSSH` always uses a group exchange of size 2048 bits, even if the client negotiates a size greater than 2048 bits.

Finally, `AsyncSSH` and `wolfSSH` do not add an additional check on the parameters (the generator g and the group P) received from the server. `Paramiko` does not check the validity of the parameter g, thus a confinement attack is possible [VAS+17].

c) Generation and Verification of Diffie-Hellman Value

Except for `OpenSSH`, all the SSH stacks installed in our platform do not check the public Diffie-Hellman value (they thus accept a weak Diffie-Hellman value). They blindly trust the other side (client or server) that they have correctly generated their secret Diffie-Hellman value. `OpenSSH` seems to be the only one SSH stack implementing an additional verification on the public Diffie-Hellman value of the other side.

The security of Diffie-Hellman key exchange depends not only on the public parameters such as the group P and the generator g, but also on the choice of the secret Diffie-Hellman value [FPS06, VOW96]. For example, `AsyncSSH` does not generate the secret Diffie-Hellman value correctly[6].

As stated in Section 6.2 of [FPS06] on the generation of the private Diffie-Hellman exponents:

> "*To increase the speed of the key exchange, both client and server may reduce the size of their private exponents. It should be at least twice as long as the key material that is generated from the shared secret.*"

[5] `diffie-hellman-group1`, which allows to negotiate Diffie-Hellman based on a group of size 1024 bits, is deprecated, thus it is removed from almost all SSH stacks

[6] `AsyncSSH` generates its secret Diffie-Hellman value using `range(1, q)`.

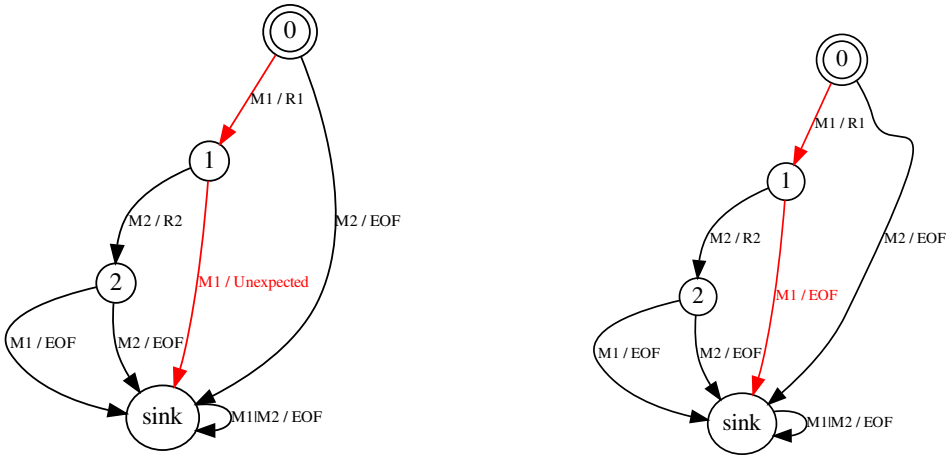
6.2 State-Machine-Based Fingerprinting

Before discussing our results related to this stack fingerprinting method, we first introduce an algorithm which is useful for the calculation of the fingerprint.

6.2.1 Distinguishing Sequences

In 2011, Shu and Lee [SL11] proposed a method to actively fingerprint network protocols using finite state machines. The goal of their algorithm is to compute the *distinguishing sequences* which is a set of sequences allowing to distinguish all candidate state machines for the fingerprinting.

Formally, given a set of Mealy machines $\mathcal{C} = \{M_1, M_2, \ldots, M_n\}$, which each Mealy machine M_i having the same input vocabulary Σ_I, a sequence $seq \in (\Sigma_I)^*$ allows to separate M_i and M_j $(i \neq j)$ if and only if $Out_{M_i}(q_0, seq) \neq Out_{M_j}(q_0, seq)$ where Out_M is the output function of the Mealy machine M introduced in sec.2.2.



(a) Implementation example 1 (b) Implementation example 2

Figure 6.1: Two different implementations of the protocol described in Figure 3.8a.

Figure 6.1 gives an example of such a sequence allowing to distinguish the two state machines described in Figures 6.1a and 6.1b. In this case, the sequence $seq =$ `M1, M1` allows to distinguish the two implementations because we get `R1, Unexpected` from implementation 1 and `R1, EOF` from implementation 2.

The basic idea of their algorithm is to find a sequence seq such that $Out_{M_i}(q_0, seq) \neq Out_{M_j}(q_0, seq)$ and then put in the same set all candidates having

the same output (i.e., $Out_{M_{k_i}}(q_0, seq) = Out_{M_{k_j}}(q_0, seq)$). It means that they create a partition where each element (a set of state machines) have the same output to the sequence *seq*. They repeat this process for each element of the *partition* until the size of the partition is equal to the number of candidates in the group.

In the worst case, for n candidate state machines, this method requires $\mathcal{O}(n^2)$ *distinguishing sequences*.

6.2.2 TLS and SSH Stack Fingerprinting

The differences usually lie in variations among implementations about error handling: different alert messages can be emitted. Several state machines accept unexpected messages and silently ignore them.

Using the method described in sec. 6.2.1, we can compute, for a given scenario, a set of input message sequences separating the different stacks we inferred. Then, we can compute the stack *fingerprints* as the answer on each stack to the *distinguishing sequences*.

In addition to the algorithm discussed in sec. 6.2.1, when searching for a sequence separating different state machines, we use a breadth-first search algorithm to browse the state machines.

If possible, we try to find the minimal number of *distinguishing sequences* by removing sequence that are prefixes of another sequence in the *distinguishing sequences*. If `A,B` and `A,B,C` are both in the *distinguishing sequences*, then we only consider `A,B,C` (i.e., we remove `A,B` from the *distinguishing sequences*).

a) TLS Stack Fingerprinting

We expect the state machines to be rather simple, as shown in Figure 1.1, with less than 10 states, a restricted number of happy paths and the rest of the transitions representing fatal errors pointing towards the sink state. Yet, as surprising as it may seem, we observe that the produced state machines are actually richer, with up to 31 states, and that each of them is specific to a given TLS stack[7].

Beyond revealing interesting differences in TLS stack internals, fingerprinting TLS stacks can help an attacker pinpoint, with a few message sequences, a set of versions of a TLS implementation to select an effective exploit against this particular target. This may also help identify the underlying TLS stack in network appliances.

Fingerprinting also allows to detect the presence of interception middleboxes that can be used for censorship. Indeed, such middleboxes may produce unique

[7]Of course, within a given project, successive versions may share the same automaton.

fingerprints, either at the message-level or at the state machine-level. It is also possible to look for discrepancies between the TLS stack and the application-layer stack to detect middleboxes, as described by Durumeric et al. [DMS+17].

Application to TLS 1.3 Servers

To illustrate our state-machine-based fingerprinting, Table 6.2 presents the classes we identify for the simple TLS 1.3 scenario with no client authentication.

Stack	Versions	N	High-severity CVEs affecting the servers
erlang	24.0.3 - 24.2.1	9	*No high-severity CVE referenced*
GnuTLS	3.6.16 - 3.7.2	4	*2021-20231 2021-20232*
matrixssl	4.0.0 - 4.1.0	4	*2019-10914* 2019-13470
	4.2.1 - 4.3.0	6	*No high-severity CVE referenced*
NSS	3.39 - 3.40	4	2019-17006 2019-17007 2020-12403 2020-25648 2021-43527
	3.41 - 3.78	4	*2019-17006 2019-17007 2020-12403 2020-25648 2021-43527*
OpenSSL	1.1.1a - 1.1.1n	4	*2020-1967 2020-1971 2021-3449 2021-3711 2022-0778* 2022-1292
	3.0.0 - 3.0.2	4	*2022-0778 2022-1473* 2022-1292
wolfSSL	3.15.5 - 4.0.0	7	2019-11873 and all the ones in the next row
	4.1.0 - 4.6.0	7	*2019-15651 2019-16748 2019-18840* 2021-38597 2022-25640
	4.7.0 - 4.8.1	7	*2021-38597* 2022-25640
	5.0.0 - 5.1.1	7	2022-23408 2022-25640
	5.2.0	6	*No high-severity CVE referenced*

Table 6.2: TLS 1.3 server stacks grouped by state machines. N is the number of states. CVEs in italic only affect part of the equivalence class.

Separating these 13 classes requires sending the following 7 distinguishing sequences only:

```
CloseNotify
ClientHello, Certificate
ClientHello, ClientHello
ClientHello, CloseNotify
ClientHello, Finished, CloseNotify
ClientHello, EmptyCertificate, CertificateVerify
ClientHello, EmptyCertificate, InvalidCertificateVerify
```

Janssen et al. [JVdRP21] proposed an approach similar to ours to fingerprint TLS servers, with a tool called `tlsprint`,[8] based on state machines inferred with `statelearner`. However, the studied stacks are limited to `OpenSSL` and `mbedTLS`

[8] `https://github.com/tlsprint/tlsprint`

servers, without TLS 1.3 support. Furthermore, we observed that `tlsprint` has a non-deterministic behavior against several OpensSSL stacks from our testbed.

b) SSH Stack Fingerprinting

To fingerprint an SSH server stack, we apply the method described in sec. 6.2.1 to the SSH Transport layer state machines only, thus an unauthenticated attacker can identify an SSH server stack. To the best of our knowledge, this is the first method using such approach to fingerprint SSH stacks.

To illustrate our state-machine-based fingerprinting, Table 6.3 presents the classes we identify for the SSH server using the `Transport` scenario. In the Transport layer, client authentication is not yet required.

Stack	**Versions**	*N*	**High-severity CVEs affecting the servers**
sshd-lite	1.2.0 - 1.3.1	8	*No high-severity CVE referenced*
SSH2	1.0.0 - 1.11.0	8	*2020-26301*
wolfSSH	0.1 - 1.4.12	18	*2022-32073*
Paramiko	2.4.0 - 3.1.0	5	*2018-7750 2018-1000805*
dropbear	2014.64 - 2022.83	6	*2021-36369 2020-36254 2017-9078 2017-2659 2016-7408 2016-7407 2016-7406*
OpenSSH	6.5.p1 - 6.7.p1	4	*2021-41617* 2016-6515 2016-10708 2016-10012 2016-10010 2016-10009 2016-0778 2015-8325
	6.8.p1 - 7.1.p1	9	2021-41617 2016-6515 2016-10708 2016-10012 2016-10010 2016-10009 2016-0778 2015-8325
	7.2.p1 - 7.3.p1	10	2021-41617 *2016-6515* 2016-10708 2016-10012 2016-10010 2016-10009 *2015-8325*
	7.4.p1	6	2021-41617
	7.5.p1 - 9.2.p1	6	*2021-41617 2019-16905*
libssh	0.7.6 - 0.7.7	6	2019-14889
	0.8.0 - 0.8.3	6	2019-14889 2018-10933
	0.8.4 - 0.8.9	6	*2019-14889*
	0.9.0 - 0.10.4	6	*2019-14889*
AsyncSSH	1.12.0 - 1.16.1	15	*2018-7749*
	1.17.0 - 1.17.1	15	*No high-severity CVE referenced*
	1.18.0 - 2.13.1	13	*No high-severity CVE referenced*

Table 6.3: SSH server stacks grouped by state machine. N is the number of states. CVEs in italic only affect part of the equivalence class.

Separating these 17 classes requires sending the following 7 distinguishing sequences only:

```
DH_INIT
NEWKEYS
SERVICE_REQUEST
KEXINIT_DH, KEXINIT_DH
KEXINIT_DH, DH_INIT, KEXINIT_DH
KEXINIT_DH, DH_INIT, NEWKEYS, DH_INIT
KEXINIT_DH, DH_INIT, SERVICE_REQUEST, SERVICE_REQUEST
```

6.3 Advantages and Limitations of the Approach

We believe that feature-based fingerprinting methods are more robust than the method discussed in sec. 6.1 because they are not customizable and are specific for the implementation; but they require further study. Our current results on SSH have been found manually.

We also believe that state machine-based fingerprinting methods are rather robust, since they rely on the way TLS and SSH stacks handle messages at their core, and not on easily customizable parameters such as the list of supported ciphersuites.

However, for TLS, there exist configuration parameters that can impact the structure of the state machine. We already handle several of them, such as server-requested client certificate authentication or TLS 1.3 middlebox compatibility (which consists of sending useless `ChangeCipherSpec` messages). Other features might affect the accuracy of our tool, such as the renegotiation mechanisms, which we leave out for future work.

Conclusion and Perspectives

In this thesis, we propose a methodology to systematically and automatically analyze protocol state machines. We implemented and successfully applied our method to analyze several TLS and SSH stacks. We analyzed over 600 stacks (400 TLS stacks and 200 SSH stacks), representing several versions of open source projects.

Using our platform and our methodology, we reproduced known bugs on TLS and SSH stacks, uncovered new implementation errors, including security vulnerabilities such as authentication bypasses, possible denial-of-service vectors, credential leakage, weakened authentication and encryption disabling.

We also found the existence of infinite state machines in several SSH stacks such as OpenSSH, AsyncSSH and sshd-lite.This has not been detected before. In our opinion, a clean implementation should always lead to a finite representation. However, we leave it as an open discussion for the community to enforce this property.

Moreover, since the state machine we infer are sufficiently precise to spot differences between implementation families, this supports the concept of state-machine-based fingerprinting, an alternative to the more classical approach based on ciphersuite-based fingerprinting, and offers a more robust characterization.

To the best of our knowledge, our work is the most extensive and systematic application of model learning to an important corpus of TLS and SSH implementations.

Overall, we believe that these deviations from the specification, even when they do not lead to exploitable security vulnerabilities, are detrimental to the overall quality of the implementation. They represent an unnecessary complexity that has been known to facilitate the introduction of security issues in the future when features are added. To reduce these deviations (and to limit fingerprinting opportunities), standards should produce more formal definitions of the expected state machines in future specifications.

Since our tools have been published as open-source software, we hope our work can help build a common test-bed for the community where we can compare and improve different approaches and tools.

Finally, we also studied and proposed an equivalence query method of the

active learning. Our benchmark results showed a significant improvement of an interesting equivalence query method. We are aware of the need of an efficient equivalent query method because it takes almost 90% of the number of queries and execution time of the overall learning process.

Application of our Methodology to Other Protocols

The natural next step for our work is the extension of the use-case to other protocol to benefit from our methodology. In particular, lowering the time required to infer a state machine allow us to explore more complex protocols with a rich input vocabulary, such as the recently standardized QUIC protocol [IT21] or the widely used IoT protocols, such as MQTT [Sta19] and CoAP [BLT+18], or industrial protocols: OPC-UA [EM11].

Rigorous Analysis of Large State Machines

Several SSH state machines are big, containing tens or hundreds of states, that is the case when simultaneously inferring the three layers, thus we can not use the same method as TLS to analyze the resulting state machine.

For this reason we have started developping `ModelCheckerNuSMV`, a tool described in sec. 5.3.1, but we have to define all properties related to the Connection layer. Fiterau-Brostean et al. defined many properties of the Connection layer in [FLP+17], but their model is incomplete and they defined properties are restricted to only one channel. Actually, defining SSH properties when several channels are opened is a difficult task, but we strongly believe that it is very useful to check if there are no interference between channels.

In 2023, Fiterau-Brostean et al. [FJST23] proposed an interesting method to automatically detect bugs in state machines. To this aim, they defined two things: the list of bugs they are interested in and the list of expected behavior with respect to the sudied protocol, then they transform these two things to a DFA. Bug detection is performed by computing the intersection of the DFA to the Mealy machine[9]. Using this approach, they are able to automatically detect vulnerabilities and specification violations. However, among the specification violations, they are not able to specify which property has not been verified.

Generic Mapper

Since we propose a generalized methodology for analyzing a protocol implementation, it is also interesting to propose a generic Mapper. Combining both ap-

[9]Of course, they transform the Mealy machine into a DFA before computing the intersection.

proaches are very interesting because it would pave the way for a fully automated methodology for new protocols. This work propose the specification of how such a MAPPER should be written, which is already an interesting start to write a generic MAPPER.

Building a generic MAPPER is similar to building a parser generator, which is a topic that was discussed during this thesis [LNR21]. To this aim, we should be aware of how to build a generic *internal state* and a generic *protocol logic*. This is actually difficult but remains possible.

Beyond the Client-Server Context

Most of the research papers use model learning to analyze a protocol with client-server context. But one legimate question is:

"*Is it possible to use model learning to analyze protocol beyond the client-server context?*"

We are thinking in particular of the Security Assertion Markup Language (SAML) protocol [CMPM05]. For the SAML protocol, three parties participate in the communication: a user, an application or service and an identity provider. Briefly, when a client wants to connect to an application or a service, it initiates a connection to the application/service by querying a service and then gets token from the identity provider which is required to succeed the authentication to the application. Later, the application verifies itself the validity of the user's message based on the token (provided by the identity provider) by communicating to the identity provider.

It seems to be difficult for an active learning approach to analyze such a protocol. However, to succeed in inferring such protocol, we think about two solutions:

- infer one by one the state machines of each parties then analyze each state machine and try to find attacks by combining each bug from each state machine; or
- infer two parties simultaneously, but it seems to be difficult because we have first to find the most appropriate representation of the state machine of both parties, and we also have to find a better solution of how the inference should be done.

Thus, extending active model learning to multi-party protocols represents an interesting challenge, and would probably require new models and tools.

List of Publications

Conference and Workshop

- Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks. In *European Symposium on Research in Computer Security*, pages 637–657. Springer, 2022
- Olivier Levillain, Sébastien Naud, and Aina Toky Rasoamanana. Work-in-Progress: Towards a Platform to Compare Binary Parser Generators. In *35. IEEE Security and Privacy Workshops, SPW (LangSec) 2021, San Jose, CA, USA*, May 2021
- Olivier Levillain and Aina Toky Rasoamanana. Wombat : one more Bleichenbacher Toolkit. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication, SSTIC 2020, Rennes, France*, June 2020
- Angelot Behajaina, Roghayeh Maleki, Aina Toky Rasoamanana, and Andriaherimanana Sarobidy Razafimahatratra. 3-setwise Intersecting Families of the Symmetric Group. *Discret. Math.*, 344(8):112467, 2021

Software

- `Wombat` (`https://gitlab.com/pictyeye/wombat`)
- `langsec-pf` (`https://gitlab.com/pictyeye/langsec-pf`)
- `TLS-inferer` (`https://gitlab.com/gaspians/pylstar-tls`)

Bibliography

[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 et Paul Zimmermann : Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. *In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.

[AFP13] Nadhem J Al Fardan et Kenneth G Paterson : Lucky thirteen: Breaking the TLS and DTLS record protocols. *In 2013 IEEE symposium on security and privacy*, pages 526–540. IEEE, 2013.

[AKM+22] Bernhard K Aichernig, Sandra König, Cristinel Mateis, Andrea Pferscher, Dominik Schmidt et Martin Tappler : Constrained Training of Recurrent Neural Networks for Automata Learning. *In International Conference on Software Engineering and Formal Methods*, pages 155–172. Springer, 2022.

[AMM+18] Bernhard K Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler et Masoumeh Taromirad : Model learning and model-based testing. *In Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 74–100. Springer, 2018.

[AMP21] Bernhard K Aichernig, Edi Muškardin et Andrea Pferscher : Learning-based Fuzzing of IoT Message Brokers. *In 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58. IEEE, 2021.

[Ang87] Dana Angluin : Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[AP16] Martin R Albrecht et Kenneth G Paterson : Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. *In Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*, pages 622–643. Springer, 2016.

[AP19] Florent Avellaneda et Alexandre Petrenko : Inferring DFA without Negative Examples. *In International Conference on Grammatical Inference*, pages 17–29. PMLR, 2019.

[APW09] Martin R Albrecht, Kenneth G Paterson et Gaven J Watson : Plaintext Recovery Attacks Against SSH. *In 2009 30th IEEE Symposium on Security and Privacy*, pages 16–26. IEEE, 2009.

[AS94] R Alquezar et A Sanfeliu : Incremental Grammatical Inference from Positive and Negative Data Using Unbiased Finite State Automata. *In In Proceedings of the ACL'02 Workshop on Unsupervised Lexical Acquisition*. Citeseer, 1994.

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni *et al.* : DROWN: Breaking TLS Using SSLv2. *In 25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, 2016.

[ATW20] Bernhard K Aichernig, Martin Tappler et Felix Wallner : Benchmarking Combinations of Learning and Testing Algorithms for Active Automata Learning. *In Tests and Proofs: 14th International Conference, TAP 2020, Held as Part of STAF 2020, Bergen, Norway, June 22–23, 2020, Proceedings 14*, pages 3–22. Springer, 2020.

[BBD+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub et Jean Karim Zinzindohoue : A Messy State of the Union: Taming the Composite State Machines of TLS. *In IEEE Symposium on Security and Privacy, SP*, pages 535–552, 2015.

[BDD+19] Pooneh Nikkhah Bahrami, Ali Dehghantanha, Tooska Dargahi, Reza M Parizi, Kim-Kwang Raymond Choo et Hamid HS Javadi : Cyber kill chain-based taxonomy of advanced persistent

threat actors: Analogy of tactics, techniques, and procedures. *Journal of information processing systems*, 15(4):865–889, 2019.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel et Joe-Kai Tsay : Efficient Padding Oracle Attacks on Cryptographic Hardware. *In Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 608–625. Springer, 2012.

[BHKL09] Benedikt Bollig, Peter Habermehl, Carsten Kern et Martin Leucker : Angluin-Style Learning of NFA. *In IJCAI*, volume 9, pages 1004–1009, 2009.

[Bio05] Philippe Biondi : Packet generation and network based attacks with Scapy. *In CanSecWest Applied Security Conference*, 2005.

[BL16] Karthikeyan Bhargavan et Gaëtan Leurent : Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH. *In 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[Ble98] Daniel Bleichenbacher : Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. *In Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[BLT+18] Carsten Bormann, Simon Lemay, Hannes Tschofenig, Klaus Hartke, Bilhanan Silverajan et Brian Raymor : CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. *RFC*, 8323:1–54, 2018.

[Bos14] Georges Bossert : *Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols*. Thèse de doctorat, MATISSE, 2014.

[Bos16] Georges Bossert : Comparison and attacks against HTTP2. *In Symposium sur la Sécurité des Technologies de l'Information et de la Communication*, 2016.

[BP12] Erik Boss et Erik Poll : Evaluating implementations of SSH by means of model-based testing. *Bachelor's esis. Radboud University*, 2012.

[BSY18] Hanno Böck, Juraj Somorovsky et Craig Young : Return Of Bleichenbacher's Oracle Threat ROBOT. *In 27th USENIX Security Symposium (USENIX Security 18)*, pages 817–849, 2018.

[CBP+11] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu et Dawn Song : MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. *In 20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[CHJ09] David Combe, Colin de la Higuera et Jean-Christophe Janodet : Zulu: An interactive learning competition. *In International Workshop on Finite-State Methods and Natural Language Processing*, pages 139–146. Springer, 2009.

[Cho78] Tsun S. Chow : Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[Cla97] Edmund M Clarke : Model checking. *In International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[CMCHG96] Edmund Clarke, K McMillan, Sérgio Campos et Vasiliki Hartonas-Garmhausen : Symbolic model checking. *In International conference on computer aided verification*, pages 419–422. Springer, 1996.

[CMPM05] Scott Cantor, Jahan Moreh, Rob Philpott et Eve Maler : Metadata for the OASIS security assertion markup language (SAML) V2. 0, 2005.

[CNS13] Wontae Choi, George Necula et Koushik Sen : Guided gui testing of android apps with minimal restart and approximate learning. *ACM SIGPLAN Notices*, 48(10):623–640, 2013.

[CPPDR14] Georg Chalupar, Stefan Peherstorfer, Erik Poll et Joeri De Ruiter : Automated Reverse Engineering using Lego®. *In 8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[CWB+19] Phuong Cao, Yuming Wu, Subho S. Banerjee, Justin Azoff, Alexander Withers, Zbigniew T. Kalbarczyk et Ravishankar K. Iyer : CAUDIT: Continuous Auditing of SSH Servers To Mitigate Brute-Force Attacks. *In* Jay R. Lorch et Minlan Yu, éditeurs :

16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019, pages 667–682. USENIX Association, 2019.

[CWKK09] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel et Engin Kirda : Prospex: Protocol Specification Extraction. *In 2009 30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

[DA99] Tim Dierks et Christopher Allen : The TLS protocol version 1.0 (RFC 2246). *IETF Request For Comments*, 1999.

[Dif76] Whitfield Diffie : New direction in cryptography. *IEEE Trans. Inform. Theory*, 22:472–492, 1976.

[DIMS12] Ionut Dinca, Florentin Ipate, Laurentiu Mierla et Alin Stefanescu : Learn and test for Event-B–a Rodin plugin. *In International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 361–364. Springer, 2012.

[DIS12] Ionut Dinca, Florentin Ipate et Alin Stefanescu : Model learning and test generation for Event-B decomposition. *In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 539–553. Springer, 2012.

[DlH10] Colin De la Higuera : *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[DLT02] François Denis, Aurélien Lemay et Alain Terlutte : Residual finite state automata. *Fundamenta Informaticae*, 51(4):339–368, 2002.

[DMS+17] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman et Vern Paxson : The Security Impact of HTTPS Interception. *In 24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[DR06] T Dierks et E Rescorla : RFC 4346: The Transport Layer Security (TLS) protocol, version 1.1 (2006), 2006.

[DR08] Tim Dierks et Eric Rescorla : The transport layer security (TLS) protocol version 1.2. Rapport technique, 2008.

[dRP15] Joeri de Ruiter et Erik Poll : Protocol State Fuzzing of TLS Implementations. *In 24th USENIX Security Symposium*, pages 193–206, 2015.

[DS22] Carlos Diego N Damasceno et Daniel Strüber : Family-Based Fingerprint Analysis: A Position Paper. *In A Journey from Process Algebra via Timed Automata to Model Learning*, pages 137–150. Springer, 2022.

[EM11] Udo Enste et Wolfgang Mahnke : OPC Unified Architecture. *Autom.*, 59(7):397–405, 2011.

[FBJV16] Paul Fiterău-Broştean, Ramon Janssen et Frits Vaandrager : Combining model learning and model checking to analyze TCP implementations. *In International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[FJM+20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas et Juraj Somorovsky : Analysis of DTLS Implementations Using Protocol State Fuzzing. *In 29th USENIX Security Symposium*, pages 2523–2540, 2020.

[FJST23] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas et Fredrik Tåquist : Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. *In 30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[FKK11] Alan Freier, Philip Karlton et Paul Kocher : RFC 6101: The secure sockets layer (SSL) protocol version 3.0, 2011.

[FLP+17] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager et Patrick Verleg : Model learning and model checking of SSH implementations. *In Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 142–151, 2017.

[FPS06] Markus Friedl, Niels Provos et William Allen Simpson : Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol. *RFC*, 4419:1–10, 2006.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou et Abderrazak Ghedamsi : Test Selection Based

on Finite State Models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.

[FW19] Sergey Frolov et Eric Wustrow : The use of TLS in Censorship Circumvention. *In 26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[GGCW19] Jiaxing Guo, Chunxiang Gu, Xi Chen et Fushan Wei : Model learning and model checking of IPSec implementations for internet of things. *IEEE Access*, 7:171322–171332, 2019.

[GGD19] Vincent Ghiëtte, Harm Griffioen et Christian Doerr : Fingerprinting Tooling used for SSH Compromisation Attempts. *In 22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 61–71. USENIX Association, 2019.

[Gil62] Arthur Gill : Introduction to the theory of finite-state machines. 1962.

[Gol78] E Mark Gold : Complexity of Automaton Identification from Given Data. *Information and control*, 37(3):302–320, 1978.

[GPY02] Alex Groce, Doron Peled et Mihalis Yannakakis : Adaptive model checking. *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer, 2002.

[HCJC16] Martin Husák, Milan Cermák, Tomás Jirsík et Pavel Celeda : HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. *EURASIP J. Inf. Secur.*, 2016:6, 2016.

[HE95] Kipp Hickman et Taher Elgamal : The SSL Protocol. 1995.

[HHNS02] Andreas Hagerer, Hardi Hungar, Oliver Niese et Bernhard Steffen : Model generation by moderated regular extrapolation. *In International Conference on Fundamental Approaches to Software Engineering*, pages 80–95. Springer, 2002.

[HMN+01] Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune et Hans-Dieter Ide : Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication*, 55:1033–1040, 2001.

[HS18] Falk Howar et Bernhard Steffen : Active automata learning in practice: an annotated bibliography of the years 2011 to 2016. *In Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, pages 123–148. Springer, 2018.

[HS22] Falk Howar et Bernhard Steffen : Active Automata Learning as Black-Box Search and Lazy Partition Refinement. *In A Journey from Process Algebra via Timed Automata to Model Learning*, pages 321–338. Springer, 2022.

[HSL08] Yating Hsu, Guoqiang Shu et David Lee : A Model-Based Approach to Security Flaw Detection of Network Protocol Implementations. *In 2008 IEEE International Conference on Network Protocols*, pages 114–123. IEEE, 2008.

[HSM10] Falk Howar, Bernhard Steffen et Maik Merten : From ZULU to RERS. *In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 687–704. Springer, 2010.

[HTJV15] Marco Henrix, J Tretmans, D Jansen et F Vaandrager : *Performance improvement in automata learning*. Thèse de doctorat, Master thesis, Radboud University, Nijmegen, 2015.

[IHS14] Malte Isberner, Falk Howar et Bernhard Steffen : The TTT algorithm: a redundancy-free approach to active automata learning. *In International Conference on Runtime Verification*, pages 307–322. Springer, 2014.

[IHS15] Malte Isberner, Falk Howar et Bernhard Steffen : The open-source LearnLib. *In International Conference on Computer Aided Verification*, pages 487–495. Springer, 2015.

[IT21] Jana Iyengar et Martin Thomson : QUIC: A UDP-Based Multiplexed and Secure Transport. *RFC*, 9000:1–151, 2021.

[Joh81] Postel John : Transmission Control Protocol. *RFC 793*, 1981.

[JSS15] Tibor Jager, Jörg Schwenk et 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*, pages 1185–1196, 2015.

[JVdRP21] Erwin Janssen, Frits Vaandrager, Joeri de Ruiter et Erik Poll : Fingerprinting TLS Implementations Using Model Learning. 2021.

[Kel22] Evgnosia-Alexandra Kelesidis : An Optimization of Bleichenbacher's Oracle Padding Attack. *In Innovative Security Solutions for Information Technology and Communications: 14th International Conference, SecITC 2021, Virtual Event, November 25–26, 2021, Revised Selected Papers*, pages 145–155. Springer, 2022.

[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`, 2014.

[KPR03] Vlastimil Klíma, Ondrej Pokornỳ et Tomáš Rosa : Attacking RSA-based sessions in SSL/TLS. *In Cryptographic Hardware and Embedded Systems-CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings 5*, pages 426–440. Springer, 2003.

[KRA+18] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez et Juan Caballero : Coming of Age: A Longitudinal Study of TLS Deployment. *In Proceedings of the Internet Measurement Conference, IMC*, pages 415–428, 2018.

[KT14] Ali Khalili et Armando Tacchella : Learning nondeterministic mealy machines. *In International Conference on Grammatical Inference*, pages 109–123. PMLR, 2014.

[KW22] Mandy Knöchel et Sandro Wefel : Analysing Attackers and Intrusions on a High-Interaction Honeypot System. *In 2022 27th Asia Pacific Conference on Communications (APCC)*, pages 433–438. IEEE, 2022.

[Lan99] Kevin J Lang : Faster Algorithms for Finding Minimal Consistent DFAs. *NEC Research Institute, Tech. Rep*, 1999.

[Lev16] Olivier Levillain : *A study of the TLS ecosystem.* Thèse de doctorat, École doctorale informatique, télécommunications et électronique (EDITE) de Paris, septembre 2016.

[Lev20] Olivier Levillain : Implementation Flaws in TLS Stacks: Lessons Learned and Study of TLS 1.3 Benefits. *In 15th International*

Conference on Risks and Security of Internet and Systems (LNCS 12528), CRiSIS 2020, Paris, France, pages 87–104, novembre 2020.

[LMD05] Corrado Leita, Ken Mermoud et Marc Dacier : Scriptgen: an Automated Script Generation Tool for Honeyd. *In 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 12–pp. IEEE, 2005.

[LNR21] Olivier Levillain, Sébastien Naud et Aina Toky Rasoamanana : Work-in-Progress: Towards a Platform to Compare Binary Parser Generators. *In 35. IEEE Security and Privacy Workshops, SPW (LangSec) 2021, San Jose, CA, USA*, mai 2021.

[LOW22] Jonas Lingg, Mateus de Oliveira Oliveira et Petra Wolf : Learning from Positive and Negative Examples: New Proof for Binary Alphabets. *arXiv preprint arXiv:2206.10025*, 2022.

[LP20] Gaëtan Leurent et Thomas Peyrin : SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. *In* Srdjan Capkun et Franziska Roesner, éditeurs : *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1839–1856. USENIX Association, 2020.

[LY96] David Lee et Mihalis Yannakakis : Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[Lyo98] Gordon Fyodor Lyon : Remote OS detection via TCP/IP stack fingerprinting. *Phrack Magazine*, 8(54), 1998.

[Lá93] Lovász László : Random Walks on Graphs: A Survey, Combinatorics, Paul Erdos is Eighty. *Bolyai Soc. Math. Stud.*, 2, 1993.

[Man01] James Manger : A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS# 1 v2.0. *In Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings 21*, pages 230–238. Springer, 2001.

[MAP+22] Edi Muškardin, Bernhard Aichernig, Ingo Pill, Andrea Pferscher et Martin Tappler : AALpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 18:1–10, 03 2022.

[MDK14] Bodo Möller, Thai Duong et Krzysztof Kotowicz : This POODLE bites: exploiting the SSL 3.0 fallback. *Security Advisory*, 21:34–58, 2014.

[Mei18] Karl Meinke : Learning-based testing: recent progress and future prospects. *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 53–73, 2018.

[MNH+22] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky et Jörg Schwenk : TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries. *In 31st USENIX Security Symposium (USENIX Security 22)*, pages 215–232, 2022.

[MS13] Karl Meinke et Muddassar A Sindhu : LBTest: a learning-based testing tool for reactive systems. *In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 447–454. IEEE, 2013.

[MSA+19] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk et Yuval Shavitt : Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities. *In* Nadia Heninger et Patrick Traynor, éditeurs : *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1029–1046. USENIX Association, 2019.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers et David Basin : The TAMARIN prover for the symbolic analysis of security protocols. *In International conference on computer aided verification*, pages 696–701. Springer, 2013.

[MSCR18] Chris McMahon Stone, Tom Chothia et Joeri de Ruiter : Extending Automated Protocol State Learning for the 802.11 4-way Handshake. *In European Symposium on Research in Computer Security*, pages 325–345. Springer, 2018.

[MSTV+22] Chris McMahon Stone, Sam L Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet et Tom Chothia : The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning. *In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2265–2278, 2022.

[MSW+14] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel et Erik Tews : Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. *In 23rd USENIX Security Symposium (USENIX Security 14)*, pages 733–748, 2014.

[PA20] Andrea Pferscher et Bernhard K Aichernig : Learning abstracted non-deterministic finite state machines. *In IFIP International Conference on Testing Software and Systems*, pages 52–69. Springer, 2020.

[PA21] Andrea Pferscher et Bernhard K Aichernig : Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning. *In International Symposium on Formal Methods*, pages 524–542. Springer, 2021.

[PA22] Andrea Pferscher et Bernhard K Aichernig : Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning. *In NASA Formal Methods Symposium*, pages 373–392. Springer, 2022.

[PABS13] Warawoot Pacharoen, Toshiaki Aoki, Pattarasinee Bhattarakosol et Athasit Surarerks : Active learning of nondeterministic finite state machines. *Mathematical Problems in Engineering*, 2013, 2013.

[PLJZ22] Yan Pan, Wei Lin, Liang Jiao et Yuefei Zhu : Model-Based Grey-Box Fuzzing of Network Protocols. *Security and Communication Networks*, 2022, 2022.

[PS07] Erik Poll et Aleksy Schubert : Verifying an implementation of SSH. 2007.

[PS11] Erik Poll et AA Schubert : Rigorous specifications of the SSH Transport Layer. 2011.

[PVY99] Doron Peled, Moshe Y Vardi et Mihalis Yannakakis : Black box checking. *In Formal Methods for Protocol Engineering and Distributed Systems*, pages 225–240. Springer, 1999.

[PY04] Kenneth G Paterson et Arnold Yau : Padding oracle attacks on the ISO CBC mode encryption standard. *In Cryptographers' Track at the RSA Conference*, pages 305–323. Springer, 2004.

[RAdR19] Abdullah Rasool, Greg Alpár et Joeri de Ruiter : State machine inference of QUIC. *CoRR*, abs/1903.04384, 2019.

[RD10] Juliano Rizzo et Thai Duong : Practical padding oracle attacks. *In 4th USENIX Workshop on Offensive Technologies (WOOT 10)*, 2010.

[RD11] Juliano Rizzo et Thai Duong : Here come the XOR ninjas. *Unpublished manuscript*, 2011.

[Res18] Eric Rescorla : The transport layer security (TLS) protocol version 1.3. Rapport technique, 2018.

[RGG+19] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong et Yuval Yarom : The 9 lives of Bleichenbacher's CAT: new cache attacks on TLS implementations. *In 2019 IEEE Symposium on Security and Privacy (SP)*, pages 435–452. IEEE, 2019.

[RL11] Tristan Richardson et John Levine : The remote framebuffer protocol. *RFC 6143*, 2011.

[RLD22] Aina Toky Rasoamanana, Olivier Levillain et Hervé Debar : Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks. *In European Symposium on Research in Computer Security*, pages 637–657. Springer, 2022.

[RLM+18] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang et Pavol Cerný : DroidStar: callback typestates for Android classes. *In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1160–1170, 2018.

[RMSM09] Harald Raffelt, Maik Merten, Bernhard Steffen et Tiziana Margaria : Dynamic testing via automata learning. *International journal on software tools for technology transfer*, 11(4):307–324, 2009.

[RPS18] Eyal Ronen, Kenneth G Paterson et Adi Shamir : Pseudo constant time implementations of TLS are only pseudo secure. *In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1397–1414, 2018.

[RS89] Ronald L Rivest et Robert E Schapire : Inference of finite automata using homing sequences. *In Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 411–420, 1989.

[RSB05] Harald Raffelt, Bernhard Steffen et Therese Berg : Learnlib: A Library for Automata Learning and Experimentation. *In Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, 2005.

[SA98] CORE SDI SA : An Attack on CRC-32 Integrity Checks of Encrypted Channels using CBC and CFB Modes. 1998.

[SG09] Muzammil Shahbaz et Roland Groz : Inferring Mealy Machines. *In FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 de *Lecture Notes in Computer Science*, pages 207–222, 2009.

[SG14] Muzammil Shahbaz et Roland Groz : Analysis and testing of black-box component-based systems by inferring partial models. *Software Testing, Verification and Reliability*, 24(4):253–288, 2014.

[SL11] Guoqiang Shu et David Lee : A Formal Methodology for Network Protocol Fingerprinting. *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1813–1825, 2011.

[SL+16] Ashutosh Satapathy, Jenila Livingston *et al.* : A Comprehensive Survey on SSL/TLS and their Vulnerabilities. *International Journal of Computer Applications*, 153(5):31–38, 2016.

[SMJ00] Matthew Smart, G. Robert Malan et Farnam Jahanian : Defeating TCP/IP Stack Fingerprinting. *In* Steven M. Bellovin et Greg Rose, éditeurs : *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*. USENIX Association, 2000.

[Som16] Juraj Somorovsky : Systematic Fuzzing and Testing of TLS Libraries. *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[Sta14] OASIS Standard : MQTT version 3.1.1. *URL http://docs.oasis-open.org/mqtt/mqtt/v3*, 1:29, 2014.

[Sta19] OASIS Standard : MQTT Version 5.0. *URL https://docs.oasis-open.org/mqtt/mqtt/v5.0*, 22:2020, 2019.

[SY22] Zhan Shu et Guanhua Yan : IoTInfer: Automated Blackbox Fuzz Testing of IoT Network Protocols Guided by Finite State Machine Inference. *IEEE Internet of Things Journal*, 9(22):22737–22751, 2022.

[SZS+22] Markus Sosnowski, Johannes Zirngibl, Patrick Sattler, Georg Carle, Claas Grohnfeldt, Michele Russo et Daniele Sgandurra : Active TLS Stack Fingerprinting: Characterizing TLS Server Deployments at Scale. *arXiv preprint arXiv:2206.13230*, 2022.

[SZSC23] Markus Sosnowski, Johannes Zirngibl, Patrick Sattler et Georg Carle : DissecTLS: A Scalable Active Scanner for TLS Server Configurations, Capabilities, and TLS Fingerprinting. *In Passive and Active Measurement: 24th International Conference, PAM 2023, Virtual Event, March 21–23, 2023, Proceedings*, pages 110–126. Springer, 2023.

[Vaa17] Frits Vaandrager : Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[VAS+17] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman et Nadia Heninger : Measuring small subgroup attacks against Diffie-Hellman. *In 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[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, April 28–May 2, 2002 Proceedings 21*, pages 534–545. Springer, 2002.

[VB17] L Velvindron et M Baushke : RFC 8270: Increase the Secure Shell Minimum Recommended Diffie-Hellman Modulus Size to 2048 Bits, 2017.

[vdLV18] Wesley van der Lee et Sicco Verwer : Vulnerability Detection on Mobile Applications Using State Machine Inference. *In 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 1–10. IEEE, 2018.

[VGRW22] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot et Thorsten Wissmann : A new approach for active automata learning based on

apartness. *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243. Springer, 2022.

[VOW96] Paul C Van Oorschot et Michael J Wiener : On Diffie-Hellman key agreement with short exponents. *In Advances in Cryptology—EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings 15*, pages 332–343. Springer, 1996.

[vTdRP18] Jules van Thoor, Joeri de Ruiter et Erik Poll : Learning state machines of TLS 1.3 implementations. Bachelor thesis. Radboud University, 2018.

[Win] Microsoft Windows : Server Message Block (SMB) Protocol.

[WJT+21] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X Liu *et al.* : MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. *In 30th USENIX Security Symposium (USENIX Security 21)*, pages 4205–4222, 2021.

[Xie03] Tao Xie : Software Component Protocol Inference. *Univ. of Washington Dep. of Comp. Sc. and Eng., Seattle, WA, General Examination Report*, 2003.

[XWQH21] Zhiwu Xu, Cheng Wen, Shengchao Qin et Mengda He : Extracting Automata from Neural Networks using Active Learning. *PeerJ Computer Science*, 7:e436, 2021.

[YL] Ed T Ylonen et C Lonvick : RFC 4252: The secure shell (SSH) Authentication Protocol.

[Ylo06a] T Ylonen : RFC 4254: The secure shell (SSH) connection protocol, 2006.

[Ylo06b] Tatu Ylonen : RFC 4251: The Secure Shell (SSH) Protocol Architecture, 2006.

[Ylo06c] Tatu Ylonen : RFC 4253: The secure shell (SSH) transport layer protocol, 2006.

[YR15] Tarun YADAV et Arvind Mallari RAO : Technical aspects of cyber kill chain. *In International symposium on security in computing and communication*, pages 438–452. Springer, 2015.

[YS19] Tarun YADAV et Koustav SADHUKHAN : Identification of bugs and vulnerabilities in TLS implementation for windows operating system using state machine learning. *In International Symposium on Security in Computing and Communication*, pages 348–362. Springer, 2019.

Appendices

Appendix A

Results of the Benchmarks for the Equivalence Query

		DB	**DBBased** with validation	**DBBased**
Learning	Total Duration	57 139	56 164	32 767
	Nb states	48	48	48
	Nb hypotheses	7	7	7
Searching cex	Duration	31 274	18 616	17 290
	Nb query	609 440	207 521	207 521
	Nb submited query	116 007	50 852	50 852
	Nb letter	3 587 985	1 394 904	1 394 904
	Nb submited letter	1 017 682	465 256	465 256
Hypotheses Validation	Duration	22 068	33 788	11 839
	Nb query	394 000	180 769	152 033
	Nb submited query	74 527	49 648	34 867
	Nb letter	3 005 200	1 582 145	1 356 737
	Nb submited letter	776 372	499 488	371 035
Building hypotheses	Duration	3 797	3 760	3 638
	Nb query	9 600	9 599	9 599
	Nb submited query	7 011	7 646	7 646
	Nb letter	82 550	82 849	82 849
	Nb submited letter	63 469	67 795	67 795

Table A.1: `Net::SSH` client v7.1.0 (transport and authentication), size of the vocabulary = 8. Duration is in seconde.

		DB	**DBBased** with validation	**DBBased**
Learning	Total Duration	5464	5462	4922
	Nb states	18	18	18
	Nb hypotheses	4	4	4
Searching cex	Duration	3145	2681	2699
	Nb query	24754	13056	13056
	Nb submited query	7396	4680	4680
	Nb letter	129492	73592	73592
	Nb submited letter	46166	30547	30547
hypotheses Validation	Duration	1836	2246	1687
	Nb query	22720	13609	12469
	Nb submited query	3388	3125	2538
	Nb letter	126630	83194	75654
	Nb submited letter	22348	21247	17084
Building hypotheses	Duration	483	535	536
	Nb query	1102	1193	1193
	Nb submited query	731	789	789
	Nb letter	6016	6866	6866
	Nb submited letter	4146	4722	4722

Table A.2: `wolfSSH` server v1.4.12 (transport), size of the vocabulary = 5. Duration is in seconde.

		DB	**DBBased** with validation	**DBBased**
Learning	Total Duration	355989	354233	285519
	Nb states	74	74	74
	Nb hypotheses	5	5	5
Searching cex	Duration	67715	50145	50085
	Nb query	361106	212778	212778
	Nb submited query	123924	82752	82752
	Nb letter	2507839	1538319	1538319
	Nb submited letter	1007894	665915	665915
hypotheses Validation	Duration	280652	296123	227485
	Nb query	972000	725388	616380
	Nb submited query	355808	319366	262602
	Nb letter	6698907	5276742	4444494
	Nb submited letter	2644334	2426576	1969022
Building hypotheses	Duration	7622	7965	7949
	Nb query	10851	11234	11234
	Nb submited query	8678	8949	8949
	Nb letter	67746	71298	71298
	Nb submited letter	55127	57961	57961

Table A.3: `wolfSSH` server v1.4.12 (transport and authentication), size of the vocabulary = 10. Duration is in seconde.

		DB	**DBBased** with validation	**DBBased**
Learning	Total Duration	10574	10508	9216
	Nb states	26	26	26
	Nb hypotheses	7	7	7
Searching cex	Duration	2704	2321	2327
	Nb query	263657	27419	27419
	Nb submited query	28782	6736	6736
	Nb letter	1290118	197068	197068
	Nb submited letter	184985	55257	55257
hypotheses Validation	Duration	6344	6527	5221
	Nb query	213952	42334	40478
	Nb submited query	75013	18764	17829
	Nb letter	1490392	379782	363382
	Nb submited letter	759610	184289	175540
Building hypotheses	Duration	1526	1660	1668
	Nb query	5279	5418	5418
	Nb submited query	3855	4271	4271
	Nb letter	45129	47026	47026
	Nb submited letter	34446	37655	37655

Table A.4: `ssh2` client v1.11.0 (transport and authentication), size of the vocabulary = 8. Duration is in seconde.

Appendix B

Authentication Bypasses

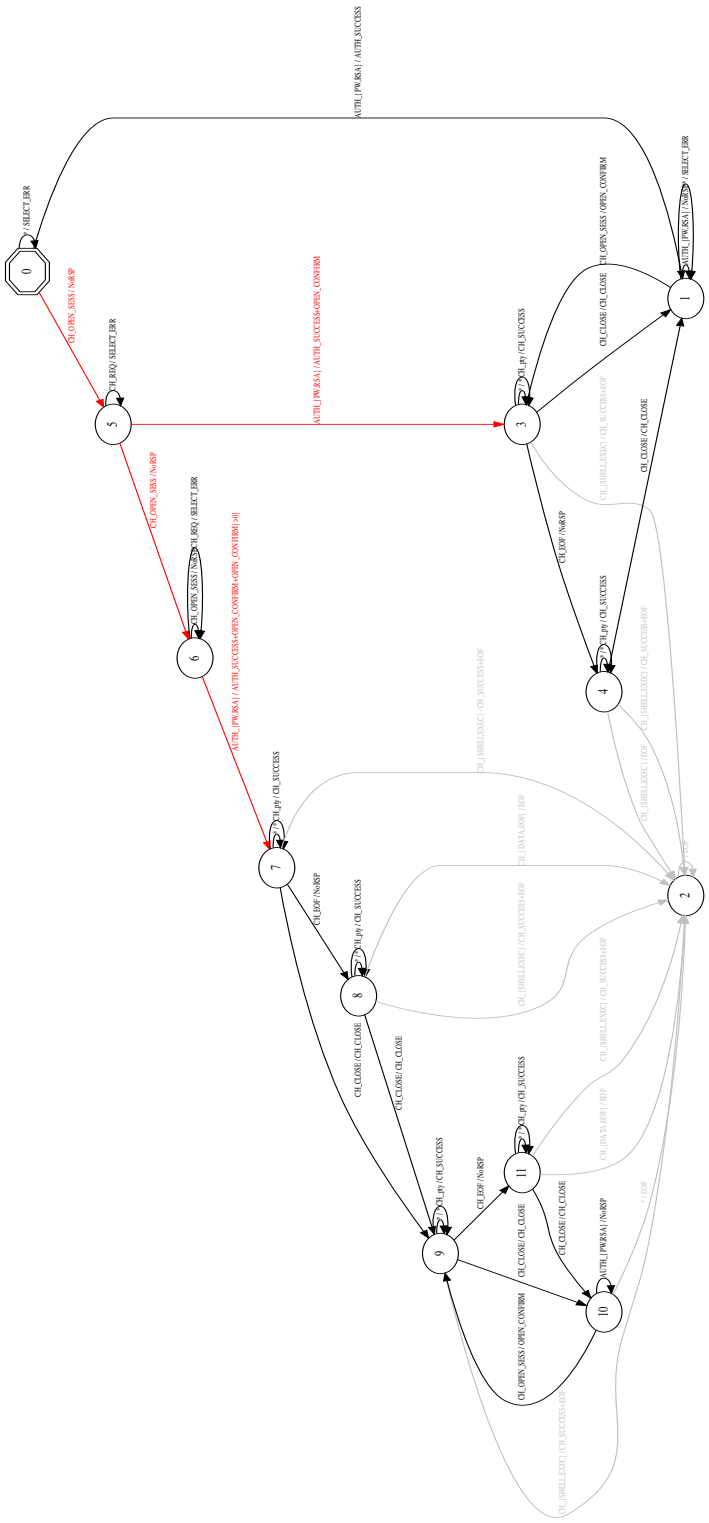


Figure B.1: CVE-2018-7749, a server authentication bypass in `AsyncSSH` version before 1.12.1.

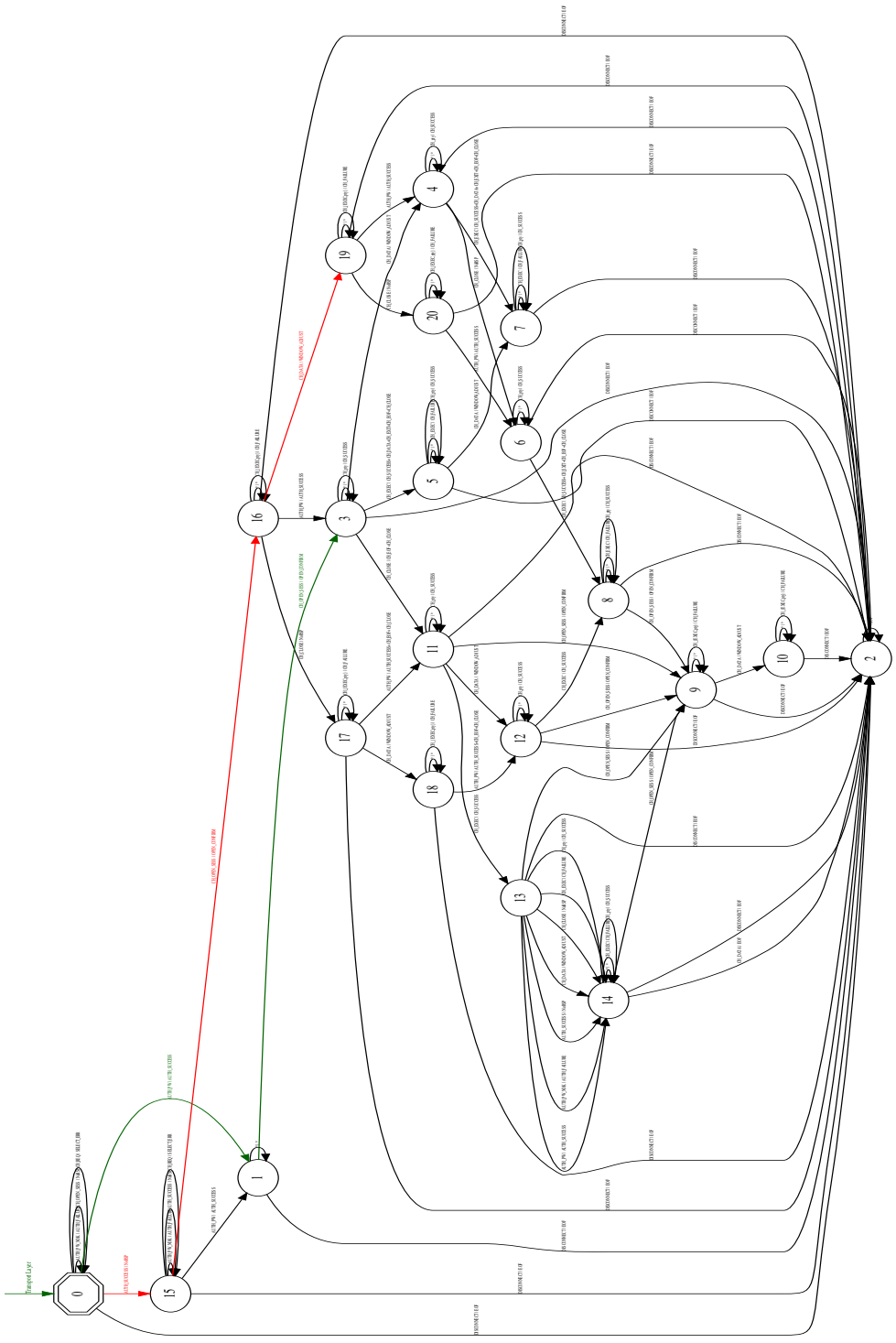


Figure B.2: CVE-2018-10933, a server authentication bypass in `libssh` before versions 0.7.6 and 0.8.4.

DOCTORAL SCHOOL

Titre : Dérivation et Analyse des Implémentations de Protocoles Cryptographiques

Mots clés : TLS, SSH, Sécurité Logicielle, Sécurité des Communications, Inférence d'Automates, L*

Résumé : TLS et SSH sont deux protocoles de sécurité très répandu et étudiés par la communauté de la recherche. Dans cette thèse, nous nous concentrons sur une classe spécifique de vulnérabilités affectant les implémentations TLS et SSH, tels que les problèmes de machine à états. Ces vulnérabilités sont dues par des différences d'interprétation de la norme et correspondent à des écarts par rapport aux spécifications, par exemple l'acceptation de messages non valides ou l'acceptation de messages valides hors séquence.
Nous développons une méthodologie généralisée et systématique pour déduire les machines d'état des protocoles tels que TLS et SSH à partir de stimuli et d'observations, et pour étudier leur évolution au fil des révisions. Nous utilisons l'algorithme L* pour calculer les machines d'état correspondant à différents scénarios d'exécution.
Nous reproduisons plusieurs vulnérabilités connues (déni de service, contournement d'authentification) et en découvrons de nouvelles. Nous montrons également que l'inférence des machines à états est suffisamment efficace et pratique dans de nombreux cas pour être intégrée dans un pipeline d'intégration continue, afin d'aider à trouver de nouvelles vulnérabilités ou déviations introduites au cours du développement.
Grâce à notre approche systématique en boîte noire, nous étudions plus de 600 versions différentes d'implémentations de serveurs et de clients dans divers scénarios (versions de protocoles, options). En utilisant les machines d'état résultantes, nous proposons un algorithme robuste pour identifier les piles TLS et SSH. Il s'agit de la première application de cette approche sur un périmètre aussi large, en termes de nombre de piles TLS et SSH, de révisions ou de scénarios étudiés.

Title : Derivation and Analysis of Cryptographic Protocol Implementations

Keywords : TLS, SSH, Software Security, Network Security, State Machine Inference, L*

Abstract : TLS and SSH are two well-known and thoroughly studied security protocols. In this thesis, we focus on a specific class of vulnerabilities affecting both protocols implementations, state machine errors. These vulnerabilities are caused by differences in interpreting the standard and correspond to deviations from the specifications, e.g., accepting invalid messages, or accepting valid messages out of sequence.
We develop a generalized and systematic methodology to infer the protocol state machines such as the major TLS and SSH stacks from stimuli and observations, and to study their evolution across revisions. We use the L* algorithm to compute state machines corresponding to different execution scenarios.
We reproduce several known vulnerabilities (denial of service, authentication bypasses), and uncover new ones. We also show that state machine inference is efficient and practical enough in many cases for integration within a continuous integration pipeline, to help find new vulnerabilities or deviations introduced during development.
With our systematic black-box approach, we study over 600 different versions of server and client implementations in various scenarios (protocol versions, options). Using the resulting state machines, we propose a robust algorithm to fingerprint TLS and SSH stacks. To the best of our knowledge, this is the first application of this approach on such a broad perimeter, in terms of number of TLS and SSH stacks, revisions, or execution scenarios studied.

Institut Polytechnique de Paris
91120 Palaiseau, France