Vérification symbolique de protocoles cryptographiques en F*: application au sous-protocole TreeSync de MLS

Théophile Wallez

Vérification symbolique de protocoles cryptographiques en F*: application au sous-protocole TreeSync de MLS

Théophile Wallez

Inria Paris

Résumé

TreeSync est un sous-protocole d'authentification pour MLS, un protocole de messagerie de groupe sécurisé en cours de standardisation à l'IETF. Nous formalisons MLS en F*, et présentons un théorème de sécurité pour TreeSync à l'aide du cadriciel de cryptographie symbolique DY*. Afin de comprendre les hypothèses clés qui permettent d'exprimer un tel théorème, nous présentons en détail le fonctionnement interne de DY*. Au cours de notre analyse, nous avons proposé plusieurs changements à TreeSync qui ont été intégrés à MLS, et avons développé de nouvelles méthodes de preuves pour passer à l'échelle avec DY* afin de pouvoir s'attaquer à un gros protocole comme MLS, par exemple, un mécanisme permettant à la fois l'exécution concrète et symbolique du protocole, la génération automatique de parseurs et sérialiseurs binaires vérifiés, ou encore la génération d'invariants globaux du protocole à partir d'invariants locaux.

1 Introduction

Messagerie de groupe sécurisée. Que ce soit WhatsApp, Signal, Facebook Messenger ou Wire, toutes les applications de messagerie moderne exposent le chiffrement de bout-en-bout comme l'une de leurs fonctionnalités phare, ce qui confirme que la communication privée et sécurisée est maintenant un aspect important pour les utilisateurs. Contrairement aux connexions HTTPS de courte durée, les conversations peuvent s'étaler sur des années, donc les garanties de sécurité des messageries doivent prendre en compte la possibilité réaliste qu'un des appareils se fasse voler ou compromettre pendant la durée de vie de la conversation. Si un adversaire compromet un appareil, il pourra bien sûr lire les messages récents et envoyer de nouveaux messages, mais nous voulons protéger les messages envoyés dans le passé lointain (*confidentialité persistante*), ainsi que les messages envoyés ou reçus dans le futur après une période de guérison (*sécurité après compromission*).

Entre deux personnes, ces messageries utilisent le protocole Signal [22] qui possède toutes les propriétés de sécurité requises. Les conversations de groupes sont fondamentalement différentes : des participants peuvent entrer et partir de la conversation à n'importe quel moment, il faut donc qu'un participant enlevé d'un groupe ne puisse plus lire les messages envoyés après, propriété que l'on peut obtenir grâce à la sécurité après compromission. Les protocoles actuellement utilisés pour les conversations de groupe ayant la propriété de sécurité après compromission dépendent de n^2 canaux Signal (un pour chaque paire de participants), ce qui n'est pas efficace.

Messaging Layer Security (MLS). Le groupe de travail IETF MLS a pour but de concevoir un protocole de messagerie de groupe efficace ayant les propriétés de confidentialité persistante et de sécurité après compromission [6]. MLS peut se découper en trois sous-protocoles : TreeKEM se charge de calculer un secret partagé entre tous les membres d'un groupe à un instant donné, TreeDEM se charge de chiffrer les messages à partir du secret de groupe, et TreeSync se charge d'authentifier l'appartenance au groupe. L'authentification est un composant crucial : avant de dire que les communications sont confidentielles, il faut d'abord savoir entre qui elles pourraient être déclarées comme telles.

Preuve de protocole assistée par ordinateur. Nous avons pour objectif de faire une preuve formelle de sécurité pour MLS, c'est-à-dire une preuve vérifiée avec un assistant de preuve ou un outil dédié. Plusieurs méthodologies existent. Les preuves de sécurité *calculatoires*, en utilisant des outils comme Squirrel [5], EasyCrypt [7] ou CryptoVerif [12], utilisent des jeux cryptographiques pour borner la probabilité qu'un attaquant puisse casser un protocole cryptographique. Les preuves de sécurité *symboliques*, en utilisant des outils comme ProVerif [13], Tamarin [25] ou DY* [9], supposent que les primitives cryptographiques sont parfaites.

Nous choisissons ici d'utiliser le modèle symbolique, parce que les preuves sont moins complexes qu'avec le modèle calculatoire, ce qui permet d'étudier des protocoles gros et complexes comme MLS.

Contributions. Nous présentons un théorème de sécurité pour TreeSync prouvé dans le modèle symbolique en F* [31] à l'aide du cadriciel DY* [9], sous des hypothèses raisonnables sur TreeKEM et TreeDEM. Durant notre analyse, nous avons trouvé une attaque exploitant les interactions entre TreeSync et TreeDEM, et montré que les garanties de TreeSync n'étaient pas aussi fortes qu'initialement espérées. Nous avons proposé des modifications du protocole résolvant ces problèmes, qui sont depuis intégrées à MLS. Notre spécification est précise à l'octet près et est interopérable avec d'autres implémentations. La description complète du protocole et de sa preuve font l'objet d'un autre article [38].

Afin de bien comprendre la signification d'un tel théorème, nous expliquons en détail le fonctionnement interne de DY*, afin de saisir quelles sont les hypothèses clés qui font tenir l'édifice. Au cours de notre analyse, nous avons développé de nouveaux outils et méthodes de preuves afin de passer à l'échelle et accomplir la preuve d'un protocole de cette taille en DY*.

Sommaire. Nous commençons par une introduction à la preuve de protocole symbolique (§2) avant d'expliquer comment DY* permet de réaliser de telles preuves (§3), puis montrons les nouvelles méthodes de preuve que nous avons développées (§4) pour étudier le protocole TreeSync et décrivons l'impact qu'a eu notre travail sur MLS (§5), et concluons avec les travaux connexes (§6).

2 Analyse symbolique de protocoles

2.1 Modélisation de l'attaquant

L'attaquant. On se place dans le pire cas et on suppose que le monde entier cherche à casser notre protocole. Si le protocole est sécurisé malgré tous ces efforts, alors il sera aussi sécurisé dans un cas plus réaliste. Ainsi, on considère que l'attaquant représente tout ce qui est extérieur aux participants honnêtes du protocole, qu'il a un contrôle total du réseau et peut donc lire tous les messages envoyés, les modifier ou décider qu'ils ne soient pas livrés, envoyer des messages en se faisant passer pour d'autres personnes, et enfin faire des manipulations cryptographiques avec les données qu'il connaît.

Cryptographie symbolique. L'analyse symbolique de protocole, à la Dolev-Yao [18], abstrait les primitives cryptographiques en supposant qu'elles sont parfaites : par exemple, une fonction de hachage n'est pas inversible et est sans collision, ou encore le chiffrement symétrique d'un message à l'aide d'une clé ne peut se déchiffrer qu'à partir de la même clé.

Dans des outils d'analyse symbolique comme Tamarin [25], Proverif [13] ou DY* (§3.1), cela se modélise via des constructeurs et des règles de réduction : par exemple le chiffrement symétrique se modélise avec le constructeur $\mathsf{SymEnc}(\mathsf{cle}, \mathsf{msg})$ et se détruit avec la règle de

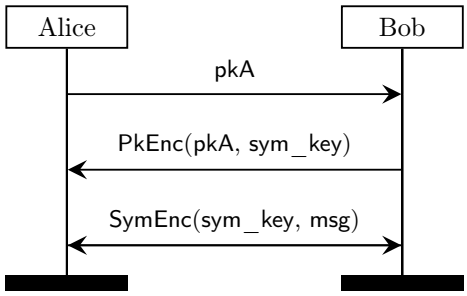

FIGURE 1 – Un premier essai pour MiniTLS. PkEnc(pk, msg) est le chiffrement de msg pour la clé publique de chiffrement pk, et SymEnc(key, msg) est le chiffrement de msg pour la clé symétrique key.

réduction SymDec(cle, SymEnc(cle, msg)) = msg. Sans autre règle de réduction, cela capture bien l'idée que SymEnc(cle, msg) ne peut se déchiffrer qu'avec la clé cle.

2.2 Cas d'étude : MiniTLS

Un protocole utilisé au quotidien est celui qui sécurise les connexions entre les navigateurs web et les sites web : il se nomme TLS. Essayons de construire un protocole ayant un cas d'usage similaire, que nous appellerons MiniTLS malgré les différences flagrantes qu'il a avec TLS : en effet, TLS dispose de nombreuses fonctionnalités dont nous ne nous soucierons pas ici ; MiniTLS est un protocole jouet qui n'a pas vocation à être utilisé dans la vie courante. Afin de rester dans la tradition, on appellera le navigateur web Alice et le site web Bob. Mallory essayera par la suite d'écouter la conversation.

Le chiffrement asymétrique étant en général moins efficace que le chiffrement symétrique, on l'utilise pour se mettre d'accord sur une clé symétrique (sym_key), qui est ensuite utilisée pour échanger les messages. Une première tentative basée sur ce principe est décrite Figure 1 : Alice envoie une clé publique de chiffrement à Bob, celui-ci génère sym_key, la chiffre avec la clé publique de chiffrement d'Alice et la lui envoie. Alice et Bob peuvent ensuite communiquer en chiffrant symétriquement avec sym_key.

Ce protocole n'est pas sécurisé. En effet, Mallory (un attaquant Dolev-Yao) peut effectuer une attaque dite de l'« homme du milieu » pour obtenir le message confidentiel, comme décrit dans la Figure 2 : Mallory intercepte pkA, envoie sa propre clé pkM à Bob. Ensuite Mallory déchiffre le message de Bob et récupère sym_key avant de le chiffrer pour Alice. Mallory peut donc lire tous les messages msg par la suite.

Pour réparer ce protocole et obtenir la version finale de MiniTLS décrite dans la Figure 3, on ajoute une signature de la part de Bob (que Mallory ne peut pas falsifier), qu'Alice vérifie par la suite avec la clé publique de vérification de signature de Bob.

Un problème subsiste : comment Alice peut-elle faire le lien entre l'identité de Bob et la clé de publique de vérification de signature de Bob ? Ce lien est crucial, sinon Alice pourrait utiliser la clé publique de vérification de signature de Mallory à la place de celle de Bob sans le savoir, et une attaque très similaire à celle décrite Figure 2 serait possible.

Ce problème ne se résout usuellement pas à l'aide de la cryptographie. Par exemple, HTTPS résout ce problème en le déléguant à un tiers de confiance [30, C.2] : les autorités de certifications sont responsables de faire le lien entre identité et clé publique, et fournissent des certificats que

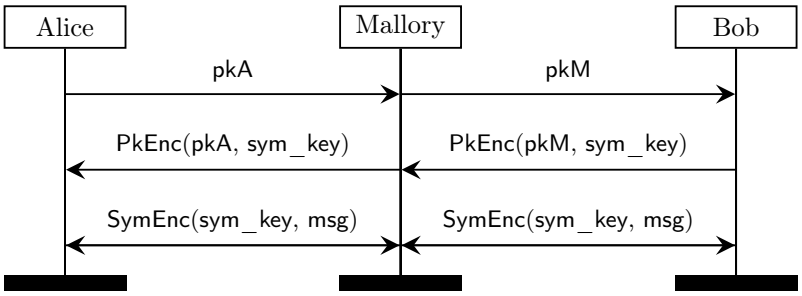

FIGURE 2 – Une attaque sur le premier essai de MiniTLS décrit dans la Figure 1. L'attaquant Dolev-Yao nommé « Mallory » obtient sym_key et peut déchiffrer tous les messages suivants.

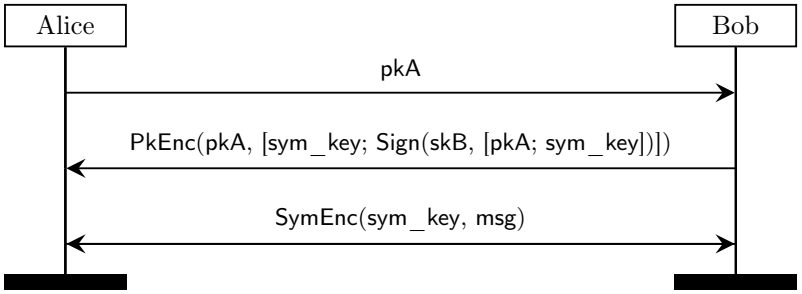

FIGURE 3 – Une version corrigée et sécurisée de MiniTLS. Une signature est rajoutée par rapport au premier essai décrit dans la Figure 1. Sign(sk, msg) est la signature de msg avec la clé de signature sk, et [a; b] représente une concaténation non-ambigue de a et b.

les navigateurs peuvent vérifier. Les navigateurs possèdent une liste d'autorités de certification dignes de confiance, et rejettent les certificats n'émanant pas de cette liste. Signal propose une autre solution à ce problème [23, 4.1] et les utilisateurs doivent vérifier leur identité via un autre canal. C'est un problème qu'on ne traitera pas dans cette étude de cas.

Sécurité de MiniTLS. Une première chose à remarquer avant d'énoncer un théorème de sécurité sur MiniTLS, est qu'Alice sait qui est Bob, mais Bob ne sait pas qui est Alice. Ainsi on ne peut pas garantir qu'un message envoyé par Bob est confidentiel entre Alice et Bob, on ne pourra que prouver qu'il est confidentiel entre Bob et les personnes connaissant la clé privée associée à pkA.

Le théorème de sécurité que nous allons énoncer est informel, et nous ne couvrirons que dans les grandes lignes sa preuve, informelle elle aussi. Le théorème sera plus tard formalisé précisément en F* à l'aide du cadriciel DY* (§3.4), et la preuve formelle sera similaire à la preuve informelle ci-dessous.

Au cours de la preuve, nous allons parfois dire qu'une donnée « ne peut être connue que de A et B », cela veut dire que par la suite nous vérifierons que cette donnée ne peut pas fuiter et être connue (par exemple) de C.

Théorème. *Pour chaque message msg transféré dans la dernière phase du protocole, si, du point de vue d'Alice, msg ne peut être connu que d'Alice et Bob, ou du point de vue de Bob, msg ne peut être connu que de Bob et des personnes connaissant la clé privée associée à pkA, alors*

FIGURE 4 – Protocole SigneTout.

le chiffrement de msg avec sym_key et l'envoi sur le réseau comme fait dans la dernière phase du protocole ne fait pas fuiter d'information.

Esquisse de preuve de la sécurité de msg. Il suffit de prouver que sym_key, du point de vue d'Alice, ne peut être connu que d'Alice et Bob, ou du point de vue de Bob, ne peut être connu que de Bob et des personnes connaissant la clé privée associée à pkA. En effet, pour des messages msg vérifiant les conditions ci-dessus, chiffrer msg avec sym_key et le diffuser sur le réseau implique que msg sera divulgué aux personnes connaissant sym_key, ce qui correspond aux personnes qui ont le droit de connaître msg donc cela ne fait pas fuiter d'information. □

Esquisse de preuve de la sécurité de sym_key. Prouvons maintenant la propriété pour sym_key.

Alice commence par générer une paire de clés de chiffrement asymétrique, la clé privée ne pouvant être connue que d'elle-même et la clé publique étant publique. Alice envoie la clé publique pkA sur le réseau, ce qui ne fait pas fuiter d'informations puisqu'elle est publique.

Bob reçoit pkA, une clé publique de chiffrement. Bob génère sym_key avec l'intention que cela ne puisse être connu que de lui-même et des personnes connaissant la clé privée associée à pkA. Bob effectue sa signature sur [pkA; sym_key], et chiffre avec la clé publique de chiffrement pkA un message contenant sym_key ainsi que la signature, avant de l'envoyer sur le réseau. Divulguer ce chiffrement au réseau de fait pas fuiter d'informations : seules les personnes connaissant la clé privée associée à pkA peuvent déchiffrer le message, et le contenu du message pouvant en effet être connu de ces personnes.

Alice reçoit le message, déchiffre et vérifie la signature. Puisque la clé de signature de Bob skB n'est connue que de Bob, Alice sait que c'est bien Bob qui a fait la signature et pas un attaquant. Lorsque Bob a fait la signature, la propriété suivante était vraie (c'est un invariant pour toutes les signatures du protocole) : sym_key peut être connue exactement par Bob et par les personnes connaissant la clé privée associée à pkA. Alice sait qu'elle seule connaît la clé privée associée à pkA, elle en déduit donc que sym_key n'est connu que d'elle-même et Bob. □

Un argument récurrent au cours de la preuve est qu'on ne chiffre un message que lorsque celui-ci est moins secret que la clé qui permet déchiffrer le message (que ce soit dans le cas symétrique ou asymétrique). De plus, la preuve repose sur un invariant sur les signatures : l'invariant est vrai lorsque Bob effectue la signature, plus tard quand Alice vérifie la signature elle sait que l'invariant est vrai. Ces techniques de preuves seront plus tard utilisées en F* (§3.2).

2.3 Modularité des protocoles

Nous étudions la sécurité de MLS en le décomposant en trois sous-protocoles (TreeSync, TreeKEM et TreeDEM). Cependant, la sécurité des protocoles cryptographiques ne se compose pas bien.

En effet, considérons le protocole SigneTout décrit Figure 4, qui signe n'importe quel message qu'on lui envoie. Si en parallèle de MiniTLS, nous exécutons le protocole SigneTout avec la même clé de signature, alors MiniTLS n'est plus sécurisé : en effet la signature dans le deuxième message de MiniTLS est obtenable par Mallory en utilisant le protocole SigneTout avec msg = [pkA; sym_key']. Ainsi, l'attaque de l'homme du milieu décrite dans la Figure 2 est à nouveau réalisable.

Une façon de composer les protocoles est de prouver qu'ils n'interfèrent pas entre eux : par exemple s'ils utilisent des clés de signature différentes, où s'ils signent des ensembles de messages disjoints. Cette dernière possibilité peut être réalisée en incluant un tag dans le message signé : par exemple MiniTLS signera le message ["MiniTLS"; pkA; sym_key] et SigneTout signera le message ["SignTout"; msg], rendant ainsi toute signature réalisée dans un des protocoles inutilisable dans l'autre protocole.

Nous avons trouvé une attaque exploitant l'interférence entre les signatures de TreeSync et TreeDEM, que nous avons corrigée en ajoutant un tag de façon systématique dans toutes les signatures de MLS (§5).

3 Preuve symbolique de protocoles cryptographiques avec DY*

Nous allons décrire ici le fonctionnement interne de DY* [9], un cadriciel de vérification de protocole contre un attaquant Dolev-Yao (§2.1), écrit en F* [31].

Les analyseurs de protocole symbolique comme ProVerif [13] ou Tamarin [25] sont complètement automatisés : à partir d'une description de protocole, ils prouvent la sécurité du protocole dans le modèle symbolique ou exhibent une attaque, ceci sans interaction supplémentaire de la part de l'utilisateur. Les preuves en DY* sont beaucoup plus manuelles, ce qui en contrepartie donne certains avantages : si les analyseurs automatiques brillent sur des protocoles avec quelques échanges de message (comme TLS [10]), DY* peut s'attaquer à des protocoles beaucoup plus gros, comme une preuve de Signal avec un nombre non borné de messages [9, §4], ou encore MLS (nombre non-borné de messages et de participants), l'objectif de cet article. De plus, la façon dont les preuves sont faites en DY* est compréhensible par un humain, et ressemble dans les grandes lignes à la preuve de MiniTLS (§2.2).

Le fonctionnement décrit par la suite est simplifié : DY* permet de modéliser un attaquant qui compromet dynamiquement l'état (secret) des participants du protocole au cours du temps, ainsi la plupart des prédicats de DY* sont paramétrés par le temps. Nous omettons ceci dans la description qui suit.

3.1 Bytes symboliques et pouvoir de l'attaquant

Constructeurs. La modélisation des primitives cryptographiques se fait à l'aide de constructeurs et de règles de réduction (§2.1). Nous modélisons les suites d'octets à l'aide d'un type inductif appelé bytes, où les constructeurs de bytes correspondent aux constructeurs des primitives cryptographiques.

En F*, cela ressemble à :

```
type bytes =
  | PublicVal: public_bytes → bytes // Donnée publique
  | Rand: len:nat → timestamp → label → bytes // Aléatoire
  | Concat: lhs:bytes → rhs:bytes → bytes // Concaténation
```

```
  | Hash: bytes → bytes // Fonction de hachage
  | Pk: sk:bytes → bytes // Clé publique de chiffrement à partir de clé privée de déchiffrement
  | PkEnc: pk:bytes → msg:bytes → bytes // Chiffrement à clé publique
  | SymEnc: key:bytes → msg:bytes → bytes // Chiffrement symétrique
  | Vk: sk:bytes → bytes // Clé publique de vérification à partir de clé privée de signature
  | Sign: sk:bytes → msg:bytes → bytes // Signature d'un message
  | ... // Insérer ici tous les autres constructeurs à considérer (MAC, KDF, DH, ...)

val length: bytes → nat // On dispose d'une fonction de longueur
```

Le type bytes modélise des suites d'octets que l'on peut concaténer, hacher, signer, chiffrer symétriquement et asymétriquement.

Les suites d'octets publiques sont représentées par le constructeur PublicVal, et les suites d'octets aléatoires fraîchement générées sont représentées par le constructeur Rand. Les suites d'octets aléatoires possèdent une longueur afin de pouvoir définir la fonction length dessus. La fraîcheur est modélisée par le timestamp, ce champ n'étant pas choisi par l'utilisateur de DY* : ce dernier ne peut construire des données aléatoires qu'à travers un effet (décrit dans §3.3) qui donne la garantie que tous les timestamps sont différents. Les personnes ayant le droit de connaître une donnée aléatoire doivent être choisies dès sa génération : c'est ce à quoi correspond le type label dans le constructeur Rand, qui sera décrit plus tard dans cette section.

En pratique, le type bytes est totalement opaque pour le code utilisant DY*. Ainsi les constructeurs sont cachés derrière des fonctions :

```
let concat (lhs:bytes) (rhs:bytes): bytes = Concat lhs rhs
let hash (msg:bytes): bytes = Hash bytes
// ...
```

Un utilisateur de DY* ne peut donc pas inspecter la façon dont un bytes a été construit, à part à travers les règles de réduction.

Règles de réduction. Les règles de réduction sont implémentées à l'aide de fonctions en F* qui opèrent sur le type bytes. Par exemple, un déchiffrement asymétrique réussit uniquement lorsque la suite d'octets est un chiffrement asymétrique avec une clé publique associée à la bonne clé privée.

```
let pk_dec (sk:bytes) (msg_chiffre:bytes): option bytes =
  match msg_chiffre with
  | PkEnc (Pk sk') msg_clair →
    if sk = sk' then Some msg_clair
    else None // Pas la bonne clé
  | _ → None // Pas un chiffré
```

De la même manière, on peut séparer une suite d'octets en deux uniquement lorsque la suite d'octets est une concaténation, et que l'indice auquel la séparation s'effectue se situe exactement entre les deux suites d'octets concaténées.

```
let split (i:nat) (msg:bytes): options bytes =
  match msg with
  | Concat lhs rhs →
    if i = length lhs then
      Some (lhs, rhs)
    else None // Pas le bon indice
  | _ → None // Pas une concaténation
```

Pouvoir de l'attaquant. On modélise le pouvoir de l'attaquant Dolev-Yao (§2.1) avec le prédicat attacker_knows : l'attaquant connaît les valeurs publiques, les messages envoyés sur le réseau, et peut appliquer les fonctions cryptographiques sur les valeurs déjà connues.

La modélisation de la connaissance des messages envoyés sur le réseau se fait via un effet (décrit dans §3.3). Comme cet effet modélise un état croissant (un journal d'évènements, stockant entre-autres les messages envoyés sur le réseau), nous pouvons utiliser les fonctionnalités de témoin [1] de F*, et ainsi définir le prédicat msg_sent_on_network comme témoin de l'existence du message dans le journal.

```
let attacker_knows (b:bytes): prop =
  (msg_sent_on_network b) ∨
  (∃ public_bytes. b == PublicVal public_bytes) ∨
  (∃ b1 b2. attacker_knows b1 ∧ attacker_knows b2 ∧ b == concat b1 b2) ∨
  (∃ pk msg. attacker_knows pk ∧ attacker_knows msg ∧ b == pk_enc pk msg) ∨
  (∃ sk msg. attacker_knows sk ∧ attacker_knows msg ∧ Some b == pk_dec sk msg) ∨
  ...
```

Ce prédicat est crucial dans la modélisation du pouvoir de l'attaquant, et fait partie de la base de confiance de DY*. On ne peut pas prouver que rien n'a été oublié lors de sa définition, mais on peut s'en convaincre, par exemple en instanciant la classe de type permettant de faire des opérations cryptographiques (§4.2) sur le type bytes raffiné avec la propriété attacker_knows. Si une propriété manquait dans le attacker_knows, alors cela se remarquerait lors de la définition de la fonction associé dans la classe de type.

Labels. De la même façon qu'il est courant de renforcer un théorème pour pouvoir le prouver par induction, le prédicat attacker_knows est trop peu précis pour faire des preuves de sécurité. Nous introduisons un treillis de labels, permettant de décrire l'ensemble des personnes connaissant une donnée. Les labels permettent de décrire des choses comme « cette donnée n'est connue que d'Alice et Bob », ou encore « cette donnée n'est connue que de Bob et des personnes qui connaissent la clé privée associée à une clé publique », comme nous l'avons fait dans la preuve de sécurité de MiniTLS (§2.2).

Les labels formant un treillis, on peut comparer deux labels et dire qu'un label est moins sécurisé qu'un autre : toute personne pouvant connaître les données associées au label plus sécurisé peut connaître les données associées au label moins sécurisé. Cela permet de modéliser le fait que quelqu'un, par exemple Alice, a le droit de connaître une certaine donnée, en disant que le label de la donnée est moins sécurisé que le label « connu que d'Alice ». Ce treillis n'est pas total, par exemple le label « connu que d'Alice » et le label « connu que de Bob » ne sont pas comparables.

```
// Opérations de treillis
val (<#): l1:label → l2:label → prop // Relation d'ordre : l1 est moins secret que l2
val union: label → label → label // Borne inférieure
val intersection: label → label → label // Borne supérieure

val public: label // Donnée connue par tout le monde : c'est un minimum global du treillis
val readers: list identity → label // Donnée connue que par la liste d'identitées
```

Pour chaque suite d'octets, on peut obtenir son label :

```
let get_label (b:bytes): label =
  match b with
  | PublicVal _ → public
```

```
  | Rand len lab → lab
  //...
  | Hash b → get_label b
  | Pk sk → public
  | PkEnc sk msg → public
  | SymEnc sk msg → public
  //...
```

Lien entre attaquant et label. On prouve le théorème suivant : si une donnée est associée a un label secret, alors l'attaquant ne peut pas connaître cette donnée.

```
val attacker_dont_know_secret_values:
  b:bytes → identities:list identity →
  Lemma (get_label b == readers identities ⟹ ¬(attacker_knows b))
```

Ce théorème se déduit à partir de deux théorèmes plus bas niveau. Le premier dit que si un attaquant connaît une donnée, alors le label associé à cette donnée est moins sécurisée que public, et le second dit que le label readers identities n'est pas moins sécurisé que le label public.

3.2 Propriété clé des messages circulant sur le réseau

Dans les analyseurs de protocole symbolique automatiques comme ProVerif ou Tamarin, l'analyseur dispose de la spécification complète du protocole, et connaît donc toutes les utilisations de la cryptographie faites par les personnes honnêtes : ils peuvent raisonner sur le protocole dans sa globalité. Par exemple, ils peuvent raisonner sur l'ensemble des signatures effectuées dans un protocole, et prouver qu'elles n'interfèrent pas entre elles. Ce n'est pas le cas dans un assistant de preuve générique comme F*, qui ne peut raisonner qu'à une échelle plus locale, sur un ensemble fixé de fonctions effectuant de la cryptographie. C'est en fait un problème similaire à la composition de la sécurité des protocoles (§2.3), si nous prouvons qu'une certaine fonction fait des opérations sécurisées avec une clé de signature, rien ne nous empêche d'écrire après-coup une autre fonction qui signe n'importe quel message.

Invariant clé des messages circulant sur le réseau. Nous imposons donc que les participants honnêtes d'un protocole obéissent à certaines règles. Par exemple, ils ne peuvent envoyer sur le réseau que des données publiques (c'est-à-dire, dont le label est moins sécurisé que public). De plus, on impose une certaine discipline sur les opérations cryptographiques exécutées : par exemple, on autorise à chiffrer un message avec une clé uniquement lorsque le message est moins secret que la clé. Cela empêche par exemple le cas où Alice chiffrerait une donnée connue que par Alice et Bob avec une clé connue de Charlie, sinon Charlie pourrait déchiffrer ces données confidentielles et apprendre des choses qu'il n'est pas censé savoir. Cet argument est utilisé plusieurs fois dans la preuve de sécurité de MiniTLS (§2.2).

Toutes ces règles de discipline que les participants honnêtes doivent respecter sont encodées dans un prédicat dit de *validité*, et on impose aux participants honnêtes de n'envoyer sur le réseau que des messages publics et valides. En contrepartie, lorsque un participant reçoit un message sur le réseau, ce dernier sait que ce message est valide et est publique.

Validité des suites d'octets. Le prédicat de validité est au cœur des preuves de sécurité en DY*. Il doit prendre en compte les utilisations honnêtes de la cryptographie (comme brièvement décrit ci-dessus), mais aussi les utilisations malhonnêtes. En effet, puisqu'un attaquant peut envoyer des messages sur le réseau, si nous voulons la garantie que les messages reçus sur le réseau sont valides, il faut donc que les messages construits par l'attaquant soient valides.

Voici par exemple la règle de validité dans le cas du chiffrement symétrique :

```
let is_valid ... (b:bytes): prop =
  match b with
  ...
  | SymEnc key msg →
    is_valid key ∧ is_valid msg ∧ ( // key et msg sont récursivement valides
      ((get_label msg) <# (get_label key)) ∨ // cas honnête : msg est moins secret que key
      // cas malhonnête : b est construit par l'attaquant, qui connaît donc key et msg
      ((get_label key) <# public ∧ (get_label msg) <# public)
    )
  ...
```

Concernant les signatures, il n'y a pas de façon évidente de restreindre leur utilisation de façon générique comme nous le faisons avec les chiffrements. Chaque protocole définit un prédicat de signature, qui doit être vrai pour toutes les suites d'octets signées. Ce prédicat de signature fait partie d'une liste de prédicats globaux, spécifique au protocole. Ces prédicats sont au cœur de la preuve, c'est à partir d'eux que découlent toutes les propriétés de sécurité que l'on peut prouver, comme nous l'avons vu avec MiniTLS (§2.2).

```
let is_valid (preds:global_predicates) (b:bytes): prop =
  match b with
  ...
  | Sign sk msg →
    // sign sk msg est valide lorsque
    is_valid sk ∧ is_valid msg ∧ ( // sk et msg sont récursivement valides
      (preds.sign_pred (Vk sk) msg) ∨ // cas honnête : le prédicat est vrai
      // cas malhonnête : b est construit par l'attaquant, qui connaît donc sk et msg
      ((get_label sk) <# public ∧ (get_label msg) <# public)
    )
  ...
```

Quand une signature est valide, cela permet de savoir que soit la clé privée de signature associée à la clé publique de vérification est connue de l'attaquant, soit le prédicat de signature est vrai sur le message signé.

```
val verify_lemma:
  preds:global_predicates →
  verification_key:bytes → msg:bytes → signature:bytes →
  Lemma
  (requires
    // si tous les arguments sont valides
    is_valid preds verification_key ∧ is_valid preds msg ∧ is_valid preds signature ∧
    // et que la signature est valide
    verify verification_key msg signature
  )
  (ensures
    ( // alors soit elle a été construite par un participant honnête et le prédicat de signature est vrai
      preds.sign_pred verification_key msg
    ) ∨ ( // soit la signature a été construite par l'attaquant
      (get_label msg <# public) ∧
      // (get_signkey_label récupère le label de la clé de signature associée à verification_key)
      (get_signkey_label verification_key) <# public
    )
```

```
)
```

3.3 Modélisation des effets de bord avec un effet de journal

Les protocoles cryptographiques ont besoin de faire des opérations impures : envoi et réception de messages sur le réseau, stockage d'état, génération de données aléatoires.

Cette impureté est encodée dans un journal d'effets de bord, qui grandit au fur et à mesure que les participants du protocole ont des effets de bord. Ainsi, les messages envoyés sur le réseau sont ajoutés au journal, et les messages sont par la suite réceptionnés en regardant parmi les messages envoyés dans le journal. D'une façon similaire, un participant qui veut stocker son état l'ajoute au journal, et pourra par la suite récupérer son état en le cherchant dans le journal. Chaque génération de données aléatoires est ajoutée dans le journal afin de permettre de garantir leur fraîcheur, comme discuté dans la description du constructeur Rand du type bytes (§3.1).

De la même façon que les protocoles définissent des invariants de signature que les participants honnêtes se doivent de respecter (§3.2), les protocoles peuvent définir des invariants sur le journal. Par exemple on peut imposer un certain prédicat à être vrai sur tous les états que l'on stocke, en contrepartie lorsque l'on récupère un état, on sait que le prédicat est vrai.

Ce journal d'effets de bord est modifié avec une monade d'état monotone encodée comme un effet F* [29]. Comme le journal ne fait que grandir, si à un instant donné on a la propriété « un certain effet de bord existe dans le journal », alors cette propriété restera vraie lorsque le journal grandira. Cela permet d'utiliser les fonctionnalités de témoin de F* [1] et de définir des prédicats comme msg_sent_on_network (§3.1).

3.4 Application au protocole MiniTLS

Nous utilisons DY* pour prouver la sécurité de MiniTLS (§2.2), et allons présenter une petite partie de la preuve de sécurité, à savoir les garanties qui sont offertes par la signature.

Le message à signer est représenté par le type respond_handshake_tbs. Un parseur et sérialiseur, prouvés inverses l'un de l'autre, sont automatiquement générés à l'aide de la bibliothèque Comparse (§4.3).

```
type respond_handshake_tbs = {
  pk_a: bytes;
  sym_key: bytes;
}

// génère le parseur et sérialiseur automatiquement avec un métaprogramme de Comparse
%splice (gen_parser (`respond_handshake_tbs))
```

Le code de vérification de signature s'écrit alors de la façon suivante :

```
let verify_respond_handshake (verif_key:bytes) (pk_a:bytes) (sym_key:bytes) (signature:bytes): bool =
  let tbs = {
    pk_a = pk_a;
    sym_key = sym_key;
  } in
  verify verif_key (serialize tbs) signature
```

Nous écrivons ensuite l'invariant de signature : on ne signe que des sérialisations de respond_handshake_tbs, de plus son champ sym_key n'est connu que des personnes connaissant

la clé privée de déchiffrement associée à pk_a ainsi que des personnes connaissant la clé privée de signature associée à clé publique de vérification de signature vk.

```
let sign_pred (vk:bytes) (msg:bytes) =
  match parse msg with
  | None → ⊥
  | Some tbs → get_label tbs.sym_key == union (get_sk_label tbs.pk_a) (get_signkey_label vk)
```

On peut alors prouver un lemme de sécurité donnant les garanties de la vérification de signature : si la signature est bien authentique et provient de Bob, et que toutes les suites d'octets que l'on considère vérifient les invariants des messages qui passent sur le réseau (§3.2), alors on peut en déduire le label de sym_key.

```
val verify_respond_handshake_lemma:
  a:identity → b:identity →
  verif_key:bytes → pk_a:bytes → sym_key:bytes → signature:bytes →
  Lemma
  (requires
    verify_respond_handshake verif_key pk_a sym_key signature ∧ // la signature est authentique
    // verif_key provient de Bob et toutes les suites d'octets obéissent aux invariants du réseau (§3.2)
    get_signkey_label verif_key == readers [b] ∧ is_valid preds verif_key ∧
    is_valid preds pk_a ∧ is_valid preds sym_key ∧ is_valid preds signature
  )
  // alors on en déduit le label de sym_key
  (ensures get_label sym_key == union (get_sk_label pk_a) (readers [b]))
```

En combinant de tels lemmes de sécurité pour toutes les sous-fonctions du protocole, et en modélisant l'exécution complète du protocole avec ses effets de bord (§3.3), nous obtenons une preuve de sécurité pour MiniTLS avec DY*.

4 Techniques de preuve modulaires pour DY*

Maintenant que nous avons vu les outils fournis par DY* pour exprimer des théorèmes de sécurité sur un protocole cryptographique, nous pouvons nous intéresser à la façon précise dont nous allons spécifier le protocole, énoncer le théorème, et effectuer les preuves. Pour cela, il y a de multiples choix techniques à effectuer, et chacun de ces choix aura un impact sur la façon dont seront structurées les preuves. L'ensemble de ces choix techniques constitue donc une méthode qui permet de s'attaquer à la preuve de sécurité d'un protocole. Même si n'importe quelle méthode permet de prouver des petits protocoles avec suffisamment de travail, il convient d'adopter une méthode adéquate afin de s'attaquer à un protocole conséquent comme MLS.

4.1 Idiomes utilisés dans les exemples DY*

Les exemples fournis avec DY* ont tous une façon idiomatique d'être définis, qui fonctionne bien sur des protocoles de taille similaire à MiniTLS, et que nous documentons dans cette section. Il nous était difficile d'envisager d'utiliser ces idiomes pour un protocole aussi gros et complexe que MLS, nous présenterons alors d'autres techniques que nous avons utilisées pour prouver TreeSync (§5), un sous-protocole de MLS dédié à l'authentification.

Exécution symbolique et exécution concrète. DY* est une bibliothèque de cryptographie symbolique pour faire des preuves de sécurité, qui n'a donc pas vocation à s'exécuter sur des

suites d'octets concrètes. Cependant elle supporte l'exécution concrète des fonctions cryptographiques d'une façon détournée : la définition du type des suites d'octets et l'implémentation des opérations dessus sont cachées derrière un fichier interface, ce qui fait qu'on peut choisir, à la compilation, d'utiliser une implémentation symbolique des suites d'octets, ou d'utiliser une implémentation concrète qui encapsule HACL* [39], une bibliothèque d'implémentations de primitives cryptographiques vérifiées avec F*. Cette façon de faire implique que ce choix entre exécution concrète et symbolique se fait à la compilation, et implique que les théorèmes vrais seulement dans le monde symbolique (par exemple l'injectivité des fonctions de hachage) doivent être admis dans le cas concret. De plus, nous n'avons pas le choix sur les primitives cryptographiques choisies. Nous résolvons ce problème en utilisant des classes de types (§4.2), que nous instancions séparément avec DY* comme bibliothèque de cryptographie symbolique et HACL* comme bibliothèque de cryptographie concrète.

Parseurs et sérialiseurs pour les messages et l'état. Les messages envoyés sur le réseau ainsi que l'état stocké ont le type bytes, une première étape consiste donc à définir des convertisseurs entre bytes et des types plus haut niveau, communément appelés parseurs et sérialiseurs. Les parseurs et sérialiseurs sont écrits à la main, avec des preuves manuelles d'inversion (e.g., sérialiser puis parser donne l'objet de départ). Ceci est possible parce qu'il y a peu de structures en jeu (en général, un type pour le message et un type pour l'état), et parce que le protocole est un modèle qui n'a pas vocation à être interopérable : le format des messages peut être choisi de façon à simplifier les preuves. La spécification du protocole MLS [6] possède plus de cinquante structures binaires, qui doivent toutes être parsés et sérialisés d'une façon précise. Nous résolvons ce problème en développant une nouvelle bibliothèque de parseurs et sérialiseurs binaires vérifiés en F*, appelé Comparse (§4.3).

Style intrinsèque. Afin de respecter la discipline que les participants honnêtes doivent suivre (§3.2), une interface à la bibliothèque cryptographique raffinée est mise à disposition par DY*. Dans cette interface, avant de chiffrer un message avec une clé symétrique, il faut prouver que le message est moins secret que la clé. Ou encore, avant d'effectuer une signature, il faut prouver que l'invariant de signature est vrai sur le contenu signé. Cela a pour conséquence que la définition du protocole est mélangée avec les preuves de sécurité, ce qui pose plusieurs problèmes : cela rend la définition du protocole moins lisible (et donc auditable) ; ces obligations de preuve n'ont pas lieu d'être dans une exécution concrète du protocole ; et cela fait que toutes les preuves de sécurité d'une fonction doivent êtres faites simultanément. Nous utilisons un style extrinsèque qui consiste à totalement séparer code et preuves, ce qui permet de mieux modulariser les preuves et garder une définition du protocole lisible et indépendante des preuves (§4.4).

Des prédicats de sécurité globaux. Les différents prédicats de sécurité décrivant les invariants, comme le prédicat de signature (§3.2) ou les prédicats d'état (§3.3), sont définis de façon globale et sont référencés explicitement tout le long de la preuve de sécurité. Cela ne fonctionne pas pour prouver les protocoles de façon modulaire comme nous voulons le faire sur MLS : si un sous-protocole est prouvé en DY* avec ses prédicats définis de façon globale, et que par la suite nous voulons prouver un nouveau sous-protocole qui s'exécutera en parallèle, dans ce cas il faudrait modifier les prédicats globaux, ce qui risquerait certainement de casser les preuves du premier sous-protocole. Nous résolvons ce problème en définissant des prédicats globaux à partir de multiples prédicats locaux (§4.5).

4.2 Classes de types pour les suites d'octets

Exécution concrète et symbolique. Nous paramétrons le protocole par une classe de type sur les suites d'octets, d'une façon similaire à ce qu'on pourrait faire avec une Section en Coq, ou un foncteur en OCaml. Cette classe de type est construite en deux étages. Une première classe de type bytes_like permet de faire des opérations non-cryptographiques sur les suites d'octets, comme la concaténation ou le calcul de longueur. Une seconde classe de type crypto, dépendant d'une instance de bytes_like permet d'effectuer des opérations cryptographiques sur les suites d'octets.

Conséquences techniques. D'une part, à partir d'une unique description du protocole, l'instance de la classe de type permet de choisir entre exécution concrète (pour réellement exécuter le protocole) et exécution symbolique (pour faire des preuves de sécurité), d'autre part, plusieurs instances concrètes permettent de choisir les primitives cryptographiques utilisées par le protocole. De plus, les preuves de sécurité se font sur l'instance symbolique de la classe de type, on peut donc utiliser des propriétés vraies seulement dans les preuves symboliques, comme l'injectivité des fonctions de hachages, sans avoir à les admettre sur les instances concrètes.

Importance de l'exécution concrète. Pouvoir exécuter l'instance concrète du protocole permet de tester que la spécification du protocole est interopérable avec d'autres implémentations, et ainsi s'assurer que les preuves sont faites sur la spécification précise du protocole, jusqu'à la façon dont les données haut-niveau sont sérialisées en suites d'octets, et pas seulement sur un modèle approximatif. Ainsi les théorèmes de sécurité ne sont pas déconnectés de la réalité : l'intéropérabilité de la spécification du protocole renforce la confiance dans le fait que celle-ci est une implémentation correcte du standard étudié. Cette façon de faire a par exemple permis de trouver l'attaque exploitant l'ambiguité des signatures dans MLS (§5), qui ne peut se trouver en faisant la preuve que lorsque celle-ci raisonne sur la façon dont les contenus des différentes signatures sont sérialisés en suites d'octets.

4.3 Comparse : bibliothèque pour parseurs et sérialiseurs vérifiés

Nous avons développé une nouvelle bibliothèque appelée « Comparse » qui permet d'obtenir des parseurs et sérialiseurs binaires vérifiés, interopérables, adaptés à la preuve symbolique, le tout de façon automatique.

Fonctionnalités de Comparse. Les parseurs et sérialiseurs produits par Comparse sont prouvés inverses l'un de l'autre. Comparse fournit des parseurs et sérialiseurs pour des types de base, et un métaprogramme écrit en Meta-F* [24] permet de générer des parseurs et sérialiseurs pour des types algébriques à partir de leur définition, lorsqu'ils sont construits à partir de types déjà connus de Comparse. De plus, des annotations permettent de contrôler la façon dont le type est sérialisé afin d'être conforme à un format binaire précis. Nous arrivons ainsi à définir automatiquement des parseurs et sérialiseurs vérifiés pour de nombreux types, dont les plus de cinquante décrits dans la définition du protocole MLS [6].

Suites d'octets concrètes et symboliques. La classe de type bytes_like axiomatise de façon minimale les suites d'octets : existence d'une suite d'octets vide, d'une longueur, d'une concaténation, d'une séparation, et de conversions depuis et vers les entiers de taille bornée, ainsi que de lemmes de compatibilité des différentes opérations. Par exemple, on ne suppose pas l'existence de lemme sur l'associativité de la concaténation, ni même de l'unicité de la suite d'octets de taille nulle ! En effet, ces propriétés ne sont pas vérifiées sur les bytes symboliques de DY* (§3.1).

Propriétés de sécurité. Des théorèmes sur les combinateurs et les types de base permettent de prouver des théorèmes utiles à la preuve symbolique, par exemple si un type algébrique ne contient que des données secrètes pour un label donné, alors sa sérialisation est elle aussi secrète pour ce label, et inversement.

Fonctionnement interne de Comparse. Les parseurs et sérialiseurs sont générés à l'aide d'un type intermédiaire qui se prête mieux à la composition, sur lequel nous disposons de quelques instances pour des types de base, ainsi que de quelques combinateurs. Un premier combinateur crée une instance pour une paire dépendante, un second combinateur crée une instance pour un type isomorphe à un autre type qui possède une instance. Ainsi, si un type est isomorphe à un embriquement de paire dépendantes de type de base (ce qui est le cas des types algébriques) alors il est possible de définir un parseur et sérialiseur pour ce type.

4.4 Utilisation du style extrinsèque

Dans la preuve de TreeSync, nous séparons la définition du protocole, les preuves de théorèmes vrais quelle que soit l'instance des suites d'octets, les preuves de théorèmes vrais sur l'instance symbolique des suites d'octets, et la gestion de l'état interne du protocole. Ce style a été montré dans la preuve de MiniTLS en DY* (§3.4).

Ce style est plus verbeux, mais a des avantages : la définition du protocole n'est pas entrelacée avec des obligations de preuve ce qui permet de préserver sa lisibilité, de plus les différentes propriétés de sécurité peuvent être prouvées séparément, augmentant ainsi la lisibilité des preuves.

4.5 Prédicats globaux à partir de prédicats locaux

Prédicats globaux dans DY*. Nous avons vu (§3.2 et §3.3) que les preuves en DY* reposent sur des prédicats globaux, par exemple sur les signatures ou sur l'état. Ces prédicats doivent être prouvés lors de la signature ou lorsque l'on met à jour l'état, en contrepartie on obtient que le prédicat est vrai lorsque l'on vérifie une signature ou que l'on récupère l'état. Étudions plus spécifiquement les signatures ; la gestion de l'état possède un problème similaire qui se traite aussi de façon similaire.

Le prédicat de signature est un prédicat sur le message signé, ainsi que la clé publique de vérification de signature utilisée (comme vu dans §3.2). Une difficulté arrive lorsqu'il existe plusieurs types de signatures dans le protocole. Le prédicat de signature doit disposer d'un moyen de différencier les différents types de signatures (comme discuté dans §2.3), que ce soit en raisonnant sur la clé de vérification ou sur le message signé. Le prédicat global de signature doit donc procéder en deux étapes : (1) déduire de la clé de vérification ou du message le type de signature dont il s'agit, (2) appeler le prédicat local correspondant à ce type de signature.

Stabilité des preuves. Une preuve de sécurité qui exploite la définition du prédicat global est fragile : si le prédicat global est modifié (par exemple pour inclure un nouveau prédicat local), alors les preuves qui en dépendent risquent de casser. De plus, de multiples prédicats doivent être définis lors d'une preuve avec DY*, ce qui invite à résoudre ce problème une fois pour toute.

Nous introduisons une façon générique de construire un prédicat global à partir d'une liste de prédicats locaux, dont nous présentons ici une version simplifiée. Par la suite, les preuves de sécurité ne dépendront pas directement du prédicat global (et de la façon dont il a été défini), elles dépendront seulement du fait qu'un quelconque prédicat global contient un certain prédicat local.

Construction d'un prédicat global. Étant donné un type de donnée data, un type de catégorie category et une fonction qui déduit la catégorie associé à une donnée, nous construisons un prédicat global à partir d'une liste de catégories et de prédicats locaux. Ce prédicat global est prouvé correct, c'est-à-dire qu'il est équivalent à chacun des prédicats locaux sur les données ayant la catégorie correspondante.

```
val data: Type // le type sur lequel nous voulons construire un prédicat
val category: Type // le type décrivant la catégorie à laquelle appartient une donnée
val get_category: data → option category // une fonction de désambiguation

val mk_global_pred: list (category & (data → prop)) → (data → prop)

let global_pred_has_local_pred (global_pred:data → prop) (cat:category) (local_pred:data → prop) =
  ∀(x:data). get_category x == Some cat ⟹ (global_pred x ⟺ local_pred x)

val mk_global_pred_correct:
  local_preds:list (category & (data → prop)) → cat:category → local_pred:(data → prop) →
  Lemma
  (requires
    no_repeats (map fst local_preds) ∧ // les categories sont disjointes
    mem (cat, local_pred) local_preds // cat and local_pred sont dans la liste local_preds
  )
  (ensures global_pred_has_local_pred (mk_global_pred local_preds) cat local_pred)
```

Par la suite, les preuves n'ont pas besoin de savoir comment est construit le prédicat global : elles peuvent être paramétrées par ce prédicat global et seulement savoir qu'il contient un prédicat local à l'aide de la propriété global_pred_has_local_pred. Une fois les preuves faites, on peut alors modifier le prédicat global sans que les preuves existantes cassent, puisqu'elles ne dépendent que du fait que le prédicat global contient un prédicat local donné.

5 Impact sur la standardisation du protocole MLS

Nous présentons ici brièvement une version simplifiée du protocole TreeSync et de son théorème de sécurité, ainsi que les améliorations que nous avons apporté à ce protocole. La description complète du protocole TreeSync et de sa preuve font l'objet d'un autre article [38], et la formalisation complète est disponible sur GitHub [20].

TreeSync et TreeKEM. TreeKEM et TreeSync travaillent sur un même arbre : un arbre binaire complet où les participants sont dans les feuilles (voir Figure 5). Chaque nœud interne contient une paire de clés de chiffrement asymétrique, avec la propriété suivante : la clé privée d'un nœud n'est connue que des participants dans son sous-arbre (propriété assurée par TreeKEM). La clé privée de la racine est alors un secret qui n'est connu que du groupe. Lorsqu'un nouveau participant rejoint le groupe, ce dernier reçoit un arbre rempli de clés publiques, et lorsqu'il chiffre un message pour une de ces clés publiques, s'attend à ce que ce message ne soit déchiffrable seulement par les participants dans le sous-arbre associé. Pour établir cette propriété, et éviter des attaques similaires à celle sur la première version de MiniTLS (§2.2), on peut se reposer sur les garanties d'authenticité offertes par TreeSync.

Théorème de sécurité pour TreeSync. Nous prouvons, à l'aide de DY*, que pour chaque nœud de l'arbre, nous pouvons trouver un participant dans le sous-arbre enraciné en ce nœud (celui qui l'a modifié en dernier) dont la signature authentifie ce sous-arbre tel qu'il était au

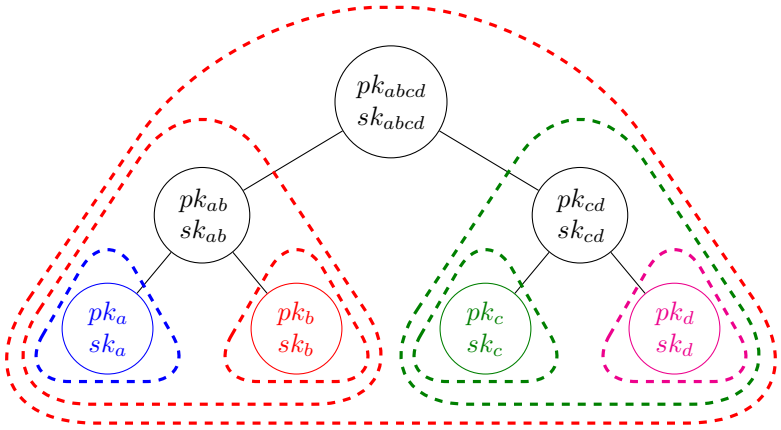

FIGURE 5 – Un arbre de clés TreeKEM, authentifié par TreeSync, où chaque sous-arbre est entouré de la couleur de la feuille qui l'authentifie. TreeKEM s'assure du fait qu'une clé privée sk_n n'est connue que des feuilles du sous-arbre enraciné en n, par exemple sk_{ab} n'est connu que de a et b. TreeSync s'assure du fait que chaque sous-arbre est authentifié par une de ses feuilles, par exemple le nœud ab est authentifié par b (en rouge). Une feuille peut authentifier plusieurs sous-arbres en même temps, par exemple b authentifie trois sous-arbres : celui enraciné en b, en ab et en $abcd$.

moment de la signature. Cette signature permet entre-autres d'initialiser la propriété de TreeKEM grâce à un invariant de signature. La preuve est complexe pour deux raisons : d'une part, il est possible d'ajouter des participants dans les feuilles d'un sous-arbre sans pour autant ré-authentifier celui-ci, il faut donc retrouver l'arbre tel qu'il était au moment de son authentification ; d'autre part, une seule signature authentifie de multiples sous-arbres (tous ceux dont la racine a été modifiée par le participant) et cette signature reste vérifiable même après que certains de ces sous-arbres ont été modifiés par d'autres personnes.

Attaque sur les signatures. Nous avons trouvé une attaque similaire à celle entre MiniTLS en SigneTout (§2.3), dû au fait que TreeSync et TreeDEM utilisent les mêmes clés de signature. Nous avons modifié le protocole MLS pour ajouter un tag de façon systématique dans les signatures, ce qui permet de les distinguer [32].

Renforcement des invariants. Les invariants des arbres manipulés dans MLS doivent être vrais lors de la création du groupe et être préservés par toutes les modifications du groupe. Mais ils doivent aussi être vérifiés au moment de rejoindre un groupe, sinon nous ne pouvons pas les utiliser lors des preuves de sécurité. Au cours de notre analyse de TreeSync, nous avons remarqué que les invariants vérifiés au moment de rejoindre un groupe n'étaient pas assez forts pour être préservés par les modifications du groupe. Nous les avons renforcés pour que ce soient de vrais invariants de MLS [35, 36].

Renforcement du théorème de sécurité. Le théorème de sécurité de TreeSync tel que décrit au début de cette section n'était initialement pas vrai : le contenu couvert par la signature était légèrement malléable, ce qui aurait rendu le théorème d'authentification plus compliqué à écrire. Nous avons renforcé la signature [37, 33, 34] pour pouvoir obtenir ce théorème de sécurité à la fois puissant et simple à énoncer : la signature des participants couvre exactement tous les sous-arbres modifiés par ce participant, là où auparavant elle ne couvrait qu'une partie de ces sous-arbres qu'il aurait fallu caractériser.

6 Travaux connexes

Analyse de la sécurité de MLS. J. Alwen et al. [2] étudie la sécurité de TreeKEM du brouillon 7 de MLS contre un adversaire passif. J. Alwen et al. [3] étudie modulairement la sécurité du brouillon 11 de MLS contre un adversaire actif. Ils ne trouvent pas l'attaque exploitant l'ambiguité des signatures, que nous avons trouvé en faisant les preuves jusqu'aux manipulations de suites d'octets. C. Brzuska et al. [14] analyse les dérivations de clé du brouillon 11 de MLS. C. Cremers et al. [15] étudie la sécurité de MLS dans un contexte multi-groupe.

J. Alwen et al. [4] étudie la sécurité de TreeKEM contre un adversaire interne au groupe, et analyse les signatures du brouillon 11 de MLS. Cependant, ils étudient TreeSync et TreeKEM comme un protocole monolithique.

Tous ces travaux se basent sur des preuves à la main, papier et crayon. Au fur et à mesure que MLS grandit, ces preuves manuelles deviennent complexes, difficiles à vérifier et à maintenir. Nous utilisons des outils de vérification assistée par ordinateur afin de faire des preuves sur une implémentation de MLS interopérable.

Une analyse de la confidentialité persistante de TreeKEM en Tamarin a été réalisée dans [17], sans considérer la sécurité après compromission ni l'authentification. K. Bhargavan et al. [8] utilise F* pour analyser TreeKEM du brouillon 7 de MLS et trouve une attaque sur l'authentification, mais ne prouve pas l'authentification de façon indépendante.

Preuve assistée par ordinateur de protocoles cryptographiques. Certains outils de vérification de protocole se basent sur le modèle symbolique, qui traite la cryptographie de façon abstraite. Certains outils sont automatiques, comme ProVerif [13] et Tamarin [25], d'autres outils se basent sur un assistant de preuve général et permettent des preuves manuelles, comme DY* [9] qui se repose sur F* [31], ou les travaux de L. Paulson [27] qui se reposent sur Isabelle [26]. Des outils comme CryptoVerif [12], EasyCrypt [7], et Squirrel [5], se basent sur le modèle calculatoire, ce qui permet d'avoir une modélisation plus précise de la cryptographie mais permet moins d'automatisation. Les deux types d'outils ont été utilisés pour des protocoles de la vie courante, comme Signal [21, 9] et TLS 1.3 [10, 16].

Finalement, beaucoup de travaux vérifient la correction fonctionnelle d'implémentations de protocoles comme Signal [28], Noise [19], et TLS [11].

7 Conclusion

Nous avons produit une spécification formelle et interopérable de la version actuelle de MLS, ainsi qu'une preuve assistée par ordinateur de la sécurité de son sous-protocole TreeSync à l'aide du cadriciel DY*. Au cours de ce travail nous avons corrigé divers bogues et amélioré la conception de MLS, et nous avons développé de nouveaux outils pour faire des preuves de sécurité avec DY*.

8 Remerciements

Je remercie Lucas Franceschino pour ses remarques sur le premier brouillon de cet article, ainsi que (par ordre alphabétique) Adrien Koutsos, Antonin Reitz, Aymeric Fromherz, Jonathan Protzenko, Sylvain Wallez, et les rapporteurs anonymes pour leur relecture de cet article et leurs nombreuses suggestions.

Références

[1] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness : foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL) :65 :1–65 :30, 2018.

[2] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In *CRYPTO*, volume 12170 of *Lecture Notes in Computer Science*, pages 248–277. Springer, 2020.

[3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1463–1483, 2021.

[4] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Paper 2020/1327, 2020.

[5] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *IEEE Symposium on Security and Privacy (S&P)*, pages 537–554. IEEE, 2021.

[6] R. Barnes, J. Millican B. Beurdouche, R. Robert, E. Omara, and K. Cohn-Gordon. The messaging layer security protocol. IETF Internet Draft, September 2022. version 16.

[7] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO*, pages 71–90, 2011.

[8] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging : Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[9] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY* : A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[10] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P)*, pages 483–502. IEEE, 2017.

[11] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy, (S&P)*, pages 445–459, 2013.

[12] Bruno Blanchet. CryptoVerif : Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[13] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2) :1–135, 2016.

[14] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2535–2553. IEEE, 2022.

[15] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging : Why cross-group effects matter. In *USENIX Security Symposium*, pages 1847–1864. USENIX Association, 2021.

[16] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1773–1788, 2017.

[17] Cas Cremers, Charlie Jacomme, and Philip Lukert. Subterm-based proof techniques for improving the automation and scope of security protocol analysis. Cryptology ePrint Archive, Paper 2022/1130, 2022. https://eprint.iacr.org/2022/1130.

[18] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on*

information theory, 29(2) :198–208, 1983.

[19] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise* : A library of verified high-performance secure channel protocol implementations. In *IEEE Symposium on Security and Privacy (S&P)*, pages 107–124, 2022.

[20] Inria-Prosecco. TreeSync : Supplementary material, 2022. https://github.com/Inria-Prosecco/treesync.

[21] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations : A symbolic and computational approach. In *IEEE European symposium on security and privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[22] Moxie Marlinspike and Trevor Perrin. Signal protocol, 2016. https://signal.org/docs.

[23] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol, 2016. https://signal.org/docs/specifications/x3dh/.

[24] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F* : Proof automation with SMT, tactics, and metaprograms. In Luís Caires, editor, *Programming Languages and Systems - European Symposium on Programming, ESOP, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 30–59. Springer, 2019.

[25] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The Tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[26] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[27] Lawrence Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6, 12 2000.

[28] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1256–1274. IEEE, 2019.

[29] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and proving with indexed effects, 2021. https://www.fstar-lang.org/papers/indexedeffects/.

[30] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[31] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[32] Théophile Wallez. Unambiguous signatures. MLS protocol, pull-request #526, 2021. https://github.com/mlswg/mls-protocol/pull/526.

[33] Théophile Wallez. Improve parent hash guarantees. MLS protocol, pull-request #713, 2022. https://github.com/mlswg/mls-protocol/pull/713.

[34] Théophile Wallez. Include leaf index in LeafNodeTBS for better parent-hash guarantees. MLS protocol, pull-request #731, 2022. https://github.com/mlswg/mls-protocol/pull/731.

[35] Théophile Wallez. MLS protocol, comment on pull-request #713, 2022. https://github.com/mlswg/mls-protocol/pull/713#issuecomment-1146371281.

[36] Théophile Wallez. MLS protocol, comment on pull-request #713, 2022. https://github.com/mlswg/mls-protocol/pull/713#issuecomment-1146599668.

[37] Théophile Wallez. Stronger parent hashes for authenticated identities. MLS protocol pull-request

#527, 2022. https://github.com/mlswg/mls-protocol/pull/527.

[38] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync : Authenticated group management for messaging layer security. Cryptology ePrint Archive, Paper 2022/1732, 2022. https://eprint.iacr.org/2022/1732.

[39] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL* : A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017.