

Projet CRY.ME

Spécifications cryptographiques

Les documents du projet CRY.ME ont été rédigés par CryptoExperts. Les dernières versions incluent quelques modifications apportées par l'ANSSI.

Table des matières

1	Fonctionnement général du produit	4
1.1	Introduction	4
1.2	Description du protocole	4
1.3	Acteurs	5
1.4	Moyens de communication	5
1.5	Architecture et fonctionnalités	6
1.5.1	Communication client - serveur	6
1.5.2	Communication client - yubikey	7
2	Résumé des clés	7
3	Protocoles	9
3.1	Enregistrement et connexion	10
3.2	Session Olm	13
3.3	Session Megolm	16
3.4	Stockage sécurisé	18
3.5	Authentification entre deux appareils	21
4	Mécanismes cryptographiques	24
4.1	Dérivation de clés	25
4.1.1	Fonction de dérivation de clé PBKDF2	25
4.1.2	Fonction de dérivation de clé HKDF	25
4.2	Codes d'authentification de messages	25
4.2.1	Code d'authentification de messages HMAC	25
4.3	Fonctions de hachage	25
4.3.1	Fonction de hachage SHA-3	25
4.3.2	Fonction de hachage SHA-1	25
4.3.3	Fonction de hachage de mot de passe bcrypt	26
4.4	Chiffrement symétrique (authentifié)	26
4.4.1	Chiffrement AES-CBC	26
4.4.2	Chiffrement AES-CTR	26
4.4.3	Chiffrement authentifié AES-GCM	26
4.5	Cryptographie asymétrique	26
4.5.1	Courbe elliptique Wei25519	26
4.5.2	Génération de clé Wei25519	27
4.5.3	Échange de clés EC Diffie-Hellman	27
4.5.4	Signature EC Schnorr	27
4.5.5	Génération de clé RSA	27
4.5.6	Signature RSA	27
4.6	Génération aléatoire	28
4.6.1	Génération des graines	28
4.6.2	Générateur linéaire congruentiel (LCG)	28
4.6.3	Générateur cryptographique	28

1 Fonctionnement général du produit

1.1 Introduction

Le logiciel **CRY.ME** est une application de messagerie instantanée. Il permet aux différents utilisateurs de communiquer entre eux de manière sécurisée via un mécanisme de chiffrement de bout en bout. Ces échanges peuvent se faire via des messages directs (entre deux utilisateurs) ou à travers des conversations de groupes.

Après s'être enregistré auprès d'un serveur accrédité, un utilisateur peut se connecter à son compte avec autant d'appareils Android (possédant le produit **CRY.ME**) qu'il le souhaite, en utilisant la clé YubiKey préalablement associée à son compte et dont il connaît le code PIN. Il peut alors utiliser ses différents appareils pour envoyer et recevoir des messages. A l'aide de l'annuaire, il peut commencer de nouvelles conversations avec un ou plusieurs utilisateurs du même serveur, en activant s'il le souhaite le mode sécurisé qui permet de garantir la confidentialité et l'intégrité des messages échangés. Si ce dernier est activé et si les utilisateurs se sont mutuellement vérifiés à l'aide d'une autre voie de communication (*e.g.*, oralement), le contenu des messages ne peut pas être compromis même si le serveur a été lui-même compromis.

Si un utilisateur se déconnecte de tous ses appareils, il ne pourra accéder aux messages chiffrés lors de sa prochaine reconnexion que s'il a préalablement mis en place une clé de restauration (*recovery key*).

1.2 Description du protocole

Le produit se base sur le protocole Matrix. Ce dernier est un protocole de communication décentralisée qui permet à différents serveurs indépendants de communiquer entre eux à travers le réseau Matrix. Un standard ouvert définit toutes les fonctionnalités et les règles nécessaires pour permettre une telle communication de manière sécurisée et cohérente. Cette spécification est disponible en ligne à l'adresse suivante :

<https://spec.matrix.org/v1.2/>.

Le protocole utilise le principe de *fédération*. Celui-ci ressemble au principe des e-mails avec, par exemple, des comptes **gmail** capables d'envoyer des messages à des comptes **outlook**. La figure 1 illustre le principe de fédération en donnant l'exemple de trois utilisateurs Alice, Bob et Charlie qui sont affectés à trois serveurs différents mais peuvent s'envoyer des messages entre eux si les serveurs correspondants implémentent le protocole Matrix. Lorsqu'un message est envoyé sur le réseau Matrix, il est copié sur tous les serveurs auxquels appartiennent les membres de la conversation, donc il n'y a pas de point de contrôle unique dans une conversation.

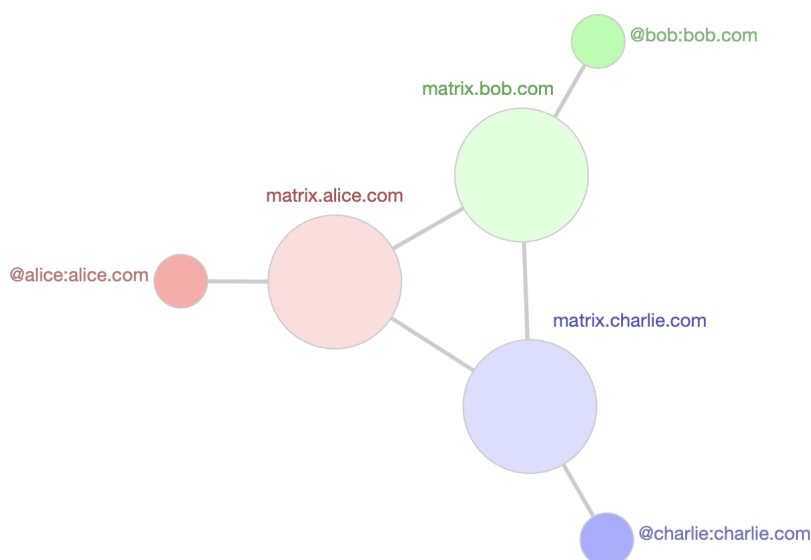


FIGURE 1 – Principe de fédération sur le réseau Matrix.

Afin de rendre ce principe disponible au grand public, l'équipe Matrix fournit le serveur **Synapse** qui implémente le protocole et dont le code source est disponible à l'adresse suivante sous licence Apache 2.0 :

<https://github.com/matrix-org/synapse>.

L'équipe Matrix fournit également le code open-source d'Element, un client qui implémente le protocole Matrix. L'application possède une implémentation compatible pour une utilisation web ainsi qu'une utilisation Android et iOS. Dans le cadre de ce projet, le code source utilisé est celui de l'application Android disponible à l'adresse suivante sous licence Apache 2.0 :

<https://github.com/vector-im/element-android>.

1.3 Acteurs

Dans le cadre de ce projet, l'application implémente un protocole de messagerie centralisé et l'option de fédération est désactivée. Dans ce contexte, les acteurs principaux sont les suivants :

- **matrix-crx** : un serveur Matrix qui utilise l'implémentation de **Synapse** avec des modifications liées au contexte du projet. **matrix-crx** est isolé des autres serveurs sur le réseau Matrix. Toutes les communications se font à travers le serveur, et ce dernier stocke l'historique des communications ainsi que les clés publiques de chiffrement ou signature au besoin, et les informations de compte de tous ses clients (notamment les noms utilisateurs, les valeurs hachées de leurs mots de passe ainsi que leurs certificats d'authentification avec la **yubikey**). **matrix-crx** est déployé sur un serveur OVH (instance b2-7) possédant un système avec une mémoire RAM de 7 Go, une capacité de stockage de 50 Go SSD et un CPU à 2 cœurs cadencés à plus de 2 GHz. Les caractéristiques techniques sont détaillées à l'adresse suivante :

<https://www.ovhcloud.com/fr/public-cloud/prices/>.

- **client** : l'application déployée sur les téléphones Android qui respecte le standard ouvert de Matrix pour les communications avec le serveur **matrix-crx**. Un utilisateur **user** se sert de l'application **client** pour s'authentifier auprès du serveur. Le **client** est déployé sur des téléphones Samsung Galaxy M12 disposant de connexions WiFi et NFC et de la dernière version d'Android, offrant également 4 Go de RAM et 64 Go de stockage. Les caractéristiques techniques sont détaillées à l'adresse suivante :

<https://www.samsung.com/fr/smartphones/galaxy-m/galaxy-m12-black-64gb-sm-m127fzkveuh/>.

- **user** : un utilisateur est un compte enregistré auprès du serveur **matrix-crx** qui s'y connecte en utilisant l'application **client**. Un utilisateur est identifié par son nom utilisateur sous le format `@user:matrix-crx` où le nom utilisateur **user** doit être unique. Un utilisateur s'authentifie de manière effective auprès du serveur en utilisant le **client** à travers son appareil, grâce à son mot de passe et sa clé **yubikey**.

Dans l'implémentation Matrix originale, un utilisateur s'authentifie auprès du serveur à travers son mot de passe uniquement. Dans le cadre de ce projet, un second moyen d'authentification basé sur l'utilisation d'une *token* est ajouté. Par conséquent, un acteur supplémentaire intervient dans le protocole :

- **yubikey** : l'utilisateur s'authentifie auprès du serveur **matrix-crx** avec son *token* d'authentification et son mot de passe. Son *token* est une clé YubiKey 5 NFC proposée par Yubico, équipée d'un composant sécurisé, de la technologie NFC et permettant une connexion rapide avec le client. Ses caractéristiques techniques sont détaillées à l'adresse suivante :

<https://www.yubico.com/fr/product/yubikey-5-nfc/>.

Une autre notion importante est celle d'**appareil** (ou *device*). Il s'agit d'un appareil virtuel créé par un **client** lors d'une connexion avec le serveur **matrix-crx** et qui matérialise une session active entre l'utilisateur et le serveur. Un utilisateur peut avoir plusieurs appareils connectés au serveur simultanément. Les appareils gèrent notamment les clés utilisées pour le chiffrement de bout en bout (chaque appareil obtient sa propre copie des clés de déchiffrement), mais ils aident également les utilisateurs à gérer leurs accès, par exemple en révoquant l'accès à des appareils particuliers. Lorsqu'un utilisateur utilise le **client** pour se connecter au serveur, il s'enregistre en tant que nouvel appareil. Un appareil sur un client Android a en général la durée de vie d'une connexion et est identifié par un identifiant unique pour l'utilisateur concerné.

1.4 Moyens de communication

Le protocole Matrix définit le standard pour tous les types de communications entre les serveurs et les clients. L'option de fédération étant désactivée dans ce projet, il n'existe pas de communications du type **serveur - serveur**.

- **client - matrix-crx** : tous les échanges et les communications passent par le serveur **matrix-crx**. Un client communique avec **matrix-crx** pour envoyer des informations qui lui sont destinées ou qui sont destinées à d'autres appareils sur d'autres clients. Ces échanges s'opèrent sur un canal sécurisé utilisant le protocole HTTPS avec TLS. Ce dernier assure l'*authentification* du serveur et du téléphone Android sur lequel est déployé le client, ainsi que l'*intégrité* et la *confidentialité* des échanges.
- **yubikey - client** : une clé physique **yubikey** sert de second moyen d'authentification d'un utilisateur auprès du serveur. Afin de l'utiliser au moment d'une connexion, un échange d'informations est nécessaire entre le client Android et cette clé avant d'envoyer les données au serveur. Cet échange s'effectue à travers une connexion sécurisée NFC entre la clé et le client.

1.5 Architecture et fonctionnalités

Le protocole Matrix propose de nombreuses fonctionnalités qui sont détaillées dans le standard ouvert ¹. Nous résumons dans la liste suivante les fonctionnalités principales exploitées dans le contexte de ce projet :

- gestion des comptes utilisateurs (enregistrement (*sign-up*), connexion (*sign-in*), déconnexion (*log-out*)), avec authentification en utilisant un mot de passe et un *token* d'authentification **yubikey**,
- gestion avancée des profils des utilisateurs (avatars, noms d'affichage, etc.),
- création et gestion des *salons* (ou *rooms*), dans lesquels se déroulent tous les échanges entre les utilisateurs. Un salon est une page de conversation entre deux ou plusieurs utilisateurs. Les échanges qui sont réalisés dans un salon passent tous à travers le serveur **matrix-crx**. Chaque salon possède un identifiant unique et les caractéristiques des salons (noms des salons, alias, sujets, interdictions, ...) peuvent être adaptées,
- envoi et réception de messages dans un salon avec chiffrement de bout en bout optionnel,
- gestion avancée des utilisateurs dans un salon (inviter, rejoindre, quitter, expulser, bannir) via un système de privilèges utilisateur basé sur les niveaux de puissance ou "*power levels*",
- possibilité de créer un *backup* chiffré des conversations depuis un appareil utilisateur afin de pouvoir déchiffrer les anciennes conversations de ses autres appareils. Ce mécanisme de *backup* chiffré est rendu possible par un mode d'authentification d'un appareil auprès des autres appareils connectés, ou par l'utilisation d'une clé de récupération (ou *recovery key*).

La fonctionnalité de réinitialisation du mot de passe n'est pas proposée dans le cadre de ce projet. Un utilisateur qui oublie son mot de passe perd l'accès à son compte et doit en créer un nouveau.

1.5.1 Communication client - serveur

Le serveur **matrix-crx** et les clients implémentent les APIs de Matrix pour synchroniser des objets JSON extensibles appelés *événements*. Les clients communiquent entre eux en synchronisant l'historique des communications avec le serveur **matrix-crx** à l'aide de l'*API client-serveur* ². Le serveur **matrix-crx** stocke l'historique des communications et les informations de compte de tous ses clients.

Un **client** commence par essayer d'établir une connexion sécurisée avec le serveur **matrix-crx** en utilisant HTTPS avec TLS. Une fois cette connexion établie, un utilisateur peut utiliser le client pour créer son compte ou s'y connecter, puis échanger avec le serveur et avec les autres clients.

Toutes les données échangées sont exprimées sous la forme d'*événements*. De manière générale, chaque action du client (par exemple, l'envoi d'un message) correspond à un *événement*. Chaque événement a un type qui est utilisé pour différencier les différents types de données ³. Les événements sont généralement envoyés dans un salon.

Les clients communiquent entre eux en envoyant des événements dans un *salon*. Par exemple, pour que le client A envoie un message au client B (où A et B sont tous les deux des clients de **matrix-crx**), le client A effectue un HTTPS PUT de l'événement JSON requis sur **matrix-crx** à l'aide de l'*API client - serveur*. **matrix-crx** ajoute cet événement à sa copie du graphe des événements du salon. Avant d'envoyer cet événement au client B, le serveur authentifie la demande, valide la signature de l'événement, autorise le contenu de l'événement puis l'ajoute à sa copie du graphe des événements du salon. Le client B reçoit alors le message du serveur via une requête GET.

Le serveur modélise l'historique des événements dans un salon sous la forme d'un graphe orienté acyclique partiellement ordonné nommé le *graphe d'événements*. L'ordre partiel des événements de ce graphe donne l'ordre chronologique des événements au sein du salon. Généralement, un événement dans le graphe a un seul parent :

1. <https://spec.matrix.org/v1.2/>
 2. <https://spec.matrix.org/v1.2/client-server-api/>
 3. <https://spec.matrix.org/v1.2/#events>

le message le plus récent dans le salon au moment où il a été envoyé. Chaque graphe d'événements a un seul événement racine sans parent.

1.5.2 Communication client - yubikey

Lorsqu'un utilisateur souhaite s'enregistrer auprès du serveur **matrix-crx**, il choisit un nom utilisateur et définit un mot de passe pour pouvoir s'authentifier auprès du serveur. Dans le cadre de ce projet, un deuxième moyen d'authentification est imposé en utilisant une clé **yubikey**. L'utilisateur se sert ainsi de sa clé yubikey pour se connecter en NFC au client, et transmettre le certificat de la clé au serveur à travers le client. Ce certificat correspond à une signature RSA dont la clé privée est stockée sur la **yubikey**. Le client offre à l'utilisateur la possibilité de générer une nouvelle paire de clés RSA et un nouveau certificat de signature à partir de l'algorithme de génération de clés intégré dans le client.

Lorsqu'un utilisateur ayant déjà un compte souhaite s'authentifier auprès du serveur, il se sert de deux facteurs d'authentification : son mot de passe et sa clé **yubikey**. Le serveur s'assure de la correction du mot de passe et envoie un *challenge* à faire signer par la **yubikey**. La signature est vérifiée en utilisant le certificat envoyé au moment de la création de compte.

2 Résumé des clés

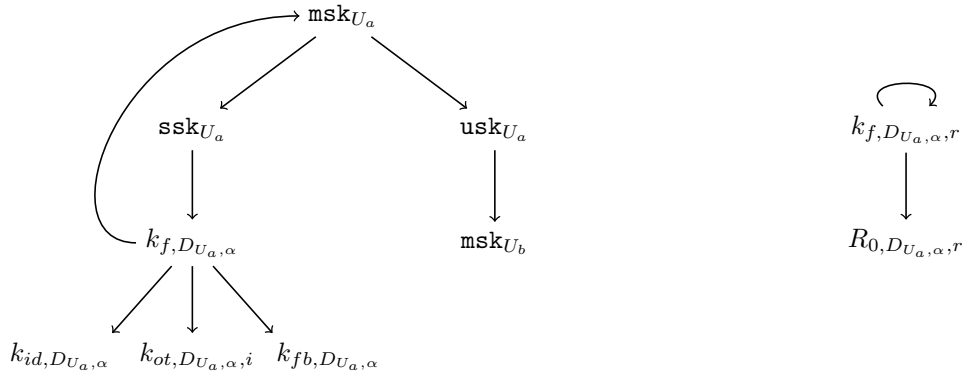
Le tableau ci-dessous décrit l'ensemble des clés intervenant dans les différents protocoles de la section 3. Pour les colonnes "Généré sur" et "Stocké sur", les étiquettes « C », « T » et « S » désignent respectivement le client, le *token* d'authentification (la clé **yubikey**) ainsi que le serveur. « S* » signifie que la clé correspondante est stockée de manière chiffrée sur le serveur. L'indice U_x désigne l'utilisateur x , $D_{U_x,y}$ désigne l'appareil y de l'utilisateur x , l'indice r dans le cas d'une conversation représente l'identifiant de la conversation, l'indice i de la clé *one-time* désigne la i ème clé *one-time* de l'appareil et les indices i et j des clés spécifiques aux protocoles Olm et Megolm sont détaillés avec les protocoles correspondants.

Clé(s)	Notations	Généré sur	Stocké sur	Protocole(s) concerné(s)	Durée de vie
Clés RSA					
Clé d'authentification des serveurs accrédités	$k_{\text{RSA},\text{server}}^{\text{pub/priv}}$	S	pub : C et S priv : S	1, 2	durée de vie illimitée
Clé du token d'authentification	$k_{\text{RSA},U_a}^{\text{pub/priv}}$	C ou T	pub : T et S priv : T	1, 2	durée de vie de U_a
Clés spécifiques à l'appareil d'un utilisateur					
Clé de signature	$k_{f,D_{U_a,\alpha}}^{\text{pub/priv}}$	C	pub : C et S priv : C	1, 2, 3.A, 3.B, 4.A, 4.B, 10, 13	durée de vie de $D_{U_a,\alpha}$
Clé d'identité	$k_{id,D_{U_a,\alpha}}^{\text{pub/priv}}$	C	pub : C et S priv : C	1, 2, 3.A, 3.B, 4.A, 4.B, 5, 6.A, 6.B	durée de vie de $D_{U_a,\alpha}$
Clé <i>one-time</i>	$k_{ot,D_{U_a,\alpha},i}^{\text{pub/priv}}$	C	pub : C et S priv : C	1, 2, 3.A, 3.B	durée de vie de $D_{U_a,\alpha}$
Clé <i>fallback</i>	$k_{fb,D_{U_a,\alpha}}^{\text{pub/priv}}$	C	pub : C et S priv : C	1, 2, 3.A	durée de vie de $D_{U_a,\alpha}$
Clés <i>cross-signing</i>					
Clé maître	$\text{msk}_{U_a}^{\text{pub/priv}}$	C	pub : C et S priv : C et S*	1, 8, 13	durée de vie de U_a

Clé <i>user-signing</i>	$usk_{U_a}^{\text{pub/priv}}$	C	pub : C et S priv : C et S*	1, 8	durée de vie de U_a
Clé <i>self-signing</i>	$ssk_{U_a}^{\text{pub/priv}}$	C	pub : C et S priv : C et S*	1, 8, 10	durée de vie de U_a
Clés spécifiques au protocole Olm					
Clé <i>ratchet</i>	$T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub/priv}}$	C	C	3.A, 3.B, 4.A, 4.B	supprimée à chaque nouveau message reçu
Clé racine	R_i	C	C	3.A, 3.B, 4.A, 4.B	supprimée à chaque nouveau message reçu
Clé de chaînage	$C_{i,j}$	C	C	3.A, 3.B, 4.A, 4.B	avancée à chaque nouveau message envoyé
Clé de message	$M_{i,j}$	C	C	3.A, 3.B, 4.A, 4.B	supprimée à la fin du protocole
Clé de chiffrement d'un message	$k_{i,j}$	C	C	3.A, 3.B, 4.A, 4.B	supprimée à la fin du protocole
Clés spécifiques au protocole Megolm					
Clé de signature pour un utilisateur de la conversation r	$k_{f,D_{U_a,\alpha},r}^{\text{pub/priv}}$	C	C	5, 6.A, 6.B	durée de vie du salon r
Clé <i>ratchet</i> symétrique pour un utilisateur de la conversation r	$R_{i,D_{U_a,\alpha},r}$	C	C et S*	5, 6.A, 6.B, 7.A	avancée à chaque nouveau message envoyé, supprimée au bout d'une semaine ou à la déconnexion de $D_{U_a,\alpha}$
Clé de chiffrement d'un message	k_i	C	C	6.A, 6.B	supprimée à la fin du protocole
Clé de chiffrement d'une pièce jointe	k^{AES}	C	C	7.A, 7.B	durée de vie de la pièce chiffrée
Clé d'intégrité d'une pièce jointe	k^{HMAC}	C	C	7.A, 7.B	durée de vie de la pièce chiffrée
Clés spécifiques au stockage sécurisé					
Clé <i>secret storage</i>	k_{sec,U_a}	C	C	8, 9, 10	durée de vie de U_a
Clé <i>recovery</i>	$k_{\text{rec},U_a}^{\text{pub/priv}}$	C	pub : C et S priv : C et S*	8, 11, 12	durée de vie de U_a
Clé de chiffrement d'une clé privée	k_i^{AES}	C	C	9, 10, 11, 12	supprimée à la fin du protocole

Clé d'intégrité d'une clé privée	k_i^{mac}	C	C	9, 10	supprimée à la fin du protocole
Clés spécifiques au protocole d'authentification entre deux appareils					
Clé éphémère pour une vérification entre deux appareils	$k_{\text{eph}, D_{U_a, \alpha}}^{\text{pub/priv}}$	C	C	13	supprimée à la fin du protocole
Clé d'intégrité de la clé publique de signature	$k_{f, D_{U_a, \alpha}}^{\text{HMAC}_{\text{pub}}}$	C	C	13	supprimée à la fin du protocole
Clé d'intégrité de la clé maître de <i>cross-signing</i>	$k_{\text{msk}_{U_a}}^{\text{HMAC}_{\text{pub}}}$	C	C	13	supprimée à la fin du protocole
Clé d'intégrité des identifiants des clés	$k_{\text{IDS}, D_{U_a, \alpha}}^{\text{HMAC}}$	C	C	13	supprimée à la fin du protocole

Les signatures entre les différentes clés peuvent être résumées dans l'arbre suivant où une flèche de la clé k_1 vers la clé k_2 indique que la clé privée k_1^{priv} signe la clé publique k_2^{pub} .



En résumé :

- La clé de signature $k_{f, D_{U_a, \alpha}}$ d'un appareil $D_{U_a, \alpha}$ signe la clé d'identité de cet appareil $k_{id, D_{U_a, \alpha}}$, les clés *one-time* de cet appareil $\{k_{ot, D_{U_a, \alpha}, i}\}_i$, la clé *fallback* de cet appareil $k_{fb, D_{U_a, \alpha}}$, ainsi que la clé maître (cross-signing) de l'utilisateur msk_{U_a} .
- La clé maître (cross-signing) msk_{U_a} d'un utilisateur signe les clés *user-signing* usk_{U_a} et *self-signing* ssk_{U_a} de cet utilisateur.
- La clé *self-signing* ssk_{U_a} d'un utilisateur signe les clés de signature des appareils de cet utilisateur $\{k_{f, D_{U_a, \alpha}}\}_\alpha$.
- La clé *user-signing* usk_{U_a} d'un utilisateur signe les clés maître des autres utilisateurs $\{\text{msk}_{U_b}\}_{U_b}$.
- La clé de signature d'un utilisateur U_a pour la conversation r , $k_{f, D_{U_a, \alpha}, r}$, s'autosigne et signe la clé *ratchet* symétrique initiale de cet utilisateur pour cette conversation $R_{0, D_{U_a, \alpha}, r}$.

3 Protocoles

Dans cette section, nous décrivons les principaux protocoles cryptographiques utilisés pour les communications de connexion, d'échange de messages et de stockage sécurisé.

Dans les protocoles décrits, l'échec de la vérification d'un *mac* ou d'une signature conduit à l'arrêt immédiat du protocole. Cette action possible n'est pas explicitement précisée pour des raisons de lisibilité.

3.1 Enregistrement et connexion

Dans cette section, nous décrivons les protocoles 1 et 2 qui servent à la création d'un compte utilisateur et la connexion d'un utilisateur à ce dernier. Le protocole 1 est exécuté lorsqu'un utilisateur souhaite créer un compte, cette exécution se termine par une première connexion automatique de l'utilisateur. Ensuite, pour chaque nouvelle connexion de l'utilisateur à son compte, le protocole 2 est exécuté.

Protocole 1 (création d'un compte utilisateur (*sign-up*)). Ce protocole est utilisé pour la création d'un compte utilisateur U_a à partir d'un de ses appareils $D_{U_a,\alpha}$ et d'un *token* token_{U_a} propre à l'utilisateur. Il suit les étapes suivantes :

1. l'utilisateur sélectionne le serveur par défaut ou renseigne l'adresse d'un serveur personnalisé.
2. le client envoie une requête au serveur renseigné afin de savoir si celui-ci est accrédité à communiquer avec lui. Ceci est vérifié si le serveur est en possession d'une clé privée RSA $k_{\text{RSA},\text{server}}^{\text{priv}}$, qui est obtenue par tous les serveurs accrédités depuis une autorité commune.
3. le serveur retourne son url publique dans un message m_{url} avec sa signature σ_{url} obtenue avec la clé privée $k_{\text{RSA},\text{server}}^{\text{priv}}$.
4. le client vérifie la signature σ_{url} avec la clé publique $k_{\text{RSA},\text{server}}^{\text{pub}}$ codée *en dur* dans l'application

$$\text{verifRSA}_{\text{server}}(k_{\text{RSA},\text{server}}^{\text{pub}}, m_{\text{url}}, \sigma_{\text{url}}).$$

Si la signature est valide et que l'adresse m_{url} correspond au serveur interrogé, le serveur est considéré de confiance et le protocole peut continuer.

5. le client envoie, via une connexion NFC (dont la sécurité ne fait pas partie du périmètre de cette évaluation), le code PIN_{U_a} du *token* saisi par l'utilisateur au *token* token_{U_a} qui le vérifie. Le protocole continue si le code est bien correct.
6. l'utilisateur U_a choisit alors s'il souhaite générer une nouvelle clé RSA en utilisant l'algorithme de génération de clés intégré au *token* (cas 1) ou s'il souhaite en générer une via son application (cas 2).

Cas 1. Dans le premier cas, une requête est envoyée au *token* token_{U_a} via la connexion NFC pour générer en interne la paire de clés :

$$(k_{\text{RSA},U_a}^{\text{pub}}, k_{\text{RSA},U_a}^{\text{priv}}).$$

Ensuite, à la demande du client, le *token* renvoie le certificat correspondant $\text{cert}_{\text{token}_{U_a}}$ via la connexion NFC.

Cas 2. Dans le second cas, l'appareil $D_{U_a,\alpha}$ génère une paire de clés RSA :

$$(k_{\text{RSA},U_a}^{\text{pub}}, k_{\text{RSA},U_a}^{\text{priv}}) \leftarrow \text{keyGenRSA}(2048)$$

et un certificat $\text{cert}_{\text{token}_{U_a}}$ et les charge dans le *token* token_{U_a} via la connexion NFC.

7. le client envoie le nom d'utilisateur ID_{U_a} , le mot de passe pwd_{U_a} et le certificat du *token* au serveur.
8. le serveur vérifie l'unicité du nom d'utilisateur et le format du certificat du *token* et les stocke. Il vérifie également le mot de passe qui, selon la politique de mot de passe, doit contenir 8 caractères alphanumériques (avec au moins une minuscule, une majuscule, un chiffre et un caractère spécial). Si ces critères sont vérifiés, le serveur hache le mot de passe

$$h_{\text{pwd}_{U_a}} \leftarrow \text{bcrypt}(\text{pwd}_{U_a})$$

et le stocke.

9. le serveur retourne OK au client ainsi que l'identifiant de session et l'identifiant d'un nouvel appareil ou une erreur (en cas de nom d'utilisateur déjà existant ou de mot de passe ne respectant pas la politique). L'appareil $D_{U_a,\alpha}$ est alors créé.
10. l'appareil $D_{U_a,\alpha}$ génère trois paires de clés Wei25519 :
 - une paire de clés maître servant à signer les deux clés ci-dessous,

$$(\text{msk}_{U_a}^{\text{pub}}, \text{msk}_{U_a}^{\text{priv}}) \leftarrow \text{keyGenWei}(),$$

- une paire de clés *user-signing* propre à U_a servant à signer les clés maître des autres utilisateurs

$$(\text{usk}_{U_a}^{\text{pub}}, \text{usk}_{U_a}^{\text{priv}}) \leftarrow \text{keyGenWei}(),$$

- une paire de clés *self-signing* servant à signer les clés des appareils de U_a

$$(\text{ssk}_{U_a}^{\text{pub}}, \text{ssk}_{U_a}^{\text{priv}}) \leftarrow \text{keyGenWei}().$$

et envoient leurs parties publiques sur le serveur. Ces trois paires de clés sont appelées des clés *cross-signing*.

- l'appareil $D_{U_a, \alpha}$ signe les clés publiques *user-signing* et *self-signing* avec la clé maître :

$$\begin{aligned} \sigma_{\text{usk}_{U_a}, \text{msk}_{U_a}} &\leftarrow \text{signECSchnorr}(\text{msk}_{U_a}^{\text{priv}}, \text{usk}_{U_a}^{\text{pub}}) \\ \sigma_{\text{ssk}_{U_a}, \text{msk}_{U_a}} &\leftarrow \text{signECSchnorr}(\text{msk}_{U_a}^{\text{priv}}, \text{ssk}_{U_a}^{\text{pub}}) \end{aligned}$$

et envoient les signatures au serveur.

- les parties privées des clés de l'utilisateur U_a sont finalement chiffrées et stockées sur le serveur en utilisant les protocoles 8, 9 et 10.
- l'appareil $D_{U_a, \alpha}$ génère ses clés de signatures :

$$(k_{f, D_{U_a, \alpha}}^{\text{pub}}, k_{f, D_{U_a, \alpha}}^{\text{priv}}) \leftarrow \text{keyGenWei}(),$$

et ses clés d'identité :

$$(k_{id, D_{U_a, \alpha}}^{\text{pub}}, k_{id, D_{U_a, \alpha}}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

et signe la partie publique de la clé d'identité avec la clé de signature :

$$\sigma_{id, D_{U_a, \alpha}} \leftarrow \text{signECSchnorr}(k_{f, D_{U_a, \alpha}}^{\text{priv}}, k_{id, D_{U_a, \alpha}}^{\text{pub}}).$$

L'appareil génère également une liste de cinquante clés *one-time* et signe les parties publiques avec sa clé de signature⁴ :

$$\begin{aligned} (k_{ot, D_{U_a, \alpha}, 0}^{\text{pub}}, k_{ot, D_{U_a, \alpha}, 0}^{\text{priv}}) &\leftarrow \text{keyGenWei}(), \\ \sigma_{ot, D_{U_a, \alpha}, 0} &\leftarrow \text{signECSchnorr}(k_{f, D_{U_a, \alpha}}^{\text{priv}}, k_{ot, D_{U_a, \alpha}, 0}^{\text{pub}}) \\ &\dots \\ (k_{ot, D_{U_a, \alpha}, 49}^{\text{pub}}, k_{ot, D_{U_a, \alpha}, 49}^{\text{priv}}) &\leftarrow \text{keyGenWei}() \\ \sigma_{ot, D_{U_a, \alpha}, 49} &\leftarrow \text{signECSchnorr}(k_{f, D_{U_a, \alpha}}^{\text{priv}}, k_{ot, D_{U_a, \alpha}, 49}^{\text{pub}}). \end{aligned}$$

Une paire de clé *fallback* est également générée par l'appareil au moment de la connexion et est signée avec la clé de signature :

$$\begin{aligned} (k_{fb, D_{U_a, \alpha}}^{\text{pub}}, k_{fb, D_{U_a, \alpha}}^{\text{priv}}) &\leftarrow \text{keyGenWei}() \\ \sigma_{fb, D_{U_a, \alpha}} &\leftarrow \text{signECSchnorr}(k_{f, D_{U_a, \alpha}}^{\text{priv}}, k_{fb, D_{U_a, \alpha}}^{\text{pub}}). \end{aligned}$$

La clé *fallback* est utilisée par le protocole 3.A en cas d'épuisement temporaire de clés *one-time*. La clé *fallback* a la même durée de vie que l'appareil $D_{U_a, \alpha}$.

- l'appareil $D_{U_a, \alpha}$ signe la partie publique de sa clé de signature avec la clé *self-signing* :

$$\sigma_{D_{U_a, \alpha}, \text{ssk}_{U_a}} \leftarrow \text{signECSchnorr}(\text{ssk}_{U_a}^{\text{priv}}, k_{f, D_{U_a, \alpha}}^{\text{pub}})$$

et la partie publique de la clé maître avec sa clé de signature :

$$\sigma_{\text{msk}_{U_a}, D_{U_a, \alpha}} \leftarrow \text{signECSchnorr}(k_{f, D_{U_a, \alpha}}^{\text{priv}}, \text{msk}_{U_a}^{\text{pub}}).$$

- l'appareil $D_{U_a, \alpha}$ envoie toutes les clés publiques générées et leur signature au serveur.

Protocole 2 (connexion d'un utilisateur (*sign-in*)). Le protocole 2 est exécuté à chaque connexion d'un utilisateur U_a à l'application (excepté au *sign-up*). L'appareil utilisé $D_{U_a, \alpha}$ est considéré comme un nouvel appareil à chaque connexion. Les étapes sont les suivantes :

- l'utilisateur sélectionne le serveur par défaut ou renseigne l'adresse d'un serveur personnalisé.
- le client envoie une requête au serveur renseigné afin de savoir si celui-ci est accrédité à communiquer avec lui. Ceci est vérifié si le serveur est en possession d'une clé privée RSA $k_{\text{RSA}, \text{server}}^{\text{priv}}$, qui est obtenue par tous les serveurs accrédités depuis une autorité commune.

4. Des clés *one-time* sont régulièrement re-générées par l'appareil à partir d'une requête asynchrone depuis la connexion et tout au long de la session pour avoir toujours cinquante clés disponibles.

- le serveur génère un challenge aléatoire de 32 bits (qui sera signé par token_{U_a} dans la suite du protocole) :

$$\text{challenge} \xleftarrow{\$} \{0,1\}^{32}.$$

Ce challenge est généré par un appel à `dev/urandom`.

- le serveur retourne son url publique dans un message m_{url} avec sa signature σ_{url} obtenue avec la clé privée $k_{\text{RSA,server}}^{\text{priv}}$. Il renvoie également le challenge généré.
- le client vérifie la signature σ_{url} avec la clé publique $k_{\text{RSA,server}}^{\text{pub}}$ codée *en dur* dans l'application

$$\text{verifRSA}_{\text{server}}(k_{\text{RSA,server}}^{\text{pub}}, m_{\text{url}}, \sigma_{\text{url}}).$$

Si la signature est valide et que l'adresse m_{url} correspond au serveur interrogé, le serveur est considéré de confiance et le protocole peut continuer.

- le client envoie, via une connexion NFC, le code PIN_{U_a} du *token* saisi par l'utilisateur au *token* token_{U_a} qui le vérifie. Le protocole continue si le code est bien correct.
- le client calcule le message m à signer à partir de l'heure courante et du challenge :

$$m \leftarrow \text{challenge} \parallel \text{timestamp}$$

où **timestamp** est le *temps Unix* en secondes (*i.e.*, le nombre de secondes écoulées depuis le 1er janvier 1970, 00:00:00 UTC) représenté sur 32 bits (m étant donc représenté comme un entier de 64 bits). Le client envoie m au *token* token_{U_a} .

- le *token* token_{U_a} signe le challenge modifié m avec sa clé RSA (interne ou générée par l'application) :

$$\sigma_m \leftarrow \text{signRSA}_{\text{token}}(k_{\text{RSA},U_a}^{\text{priv}}, m)$$

et retourne la signature σ_m à l'appareil.

- le client envoie au serveur le nom d'utilisateur ID_{U_a} , le mot de passe pwd_{U_a} , le message m et sa signature σ_m par le *token*.
- le serveur parse le message $m = \text{challenge} \parallel \text{timestamp}$, vérifie que la partie **challenge** correspond bien au challenge envoyé et vérifie que la partie **timestamp** correspond bien à la date actuelle $\text{time}_{\text{server}}$ avec un décalage maximum de 30 secondes (précisément le serveur vérifie $\text{timestamp} \in [\text{time}_{\text{server}} - 30s, \text{time}_{\text{server}}]$).
- le serveur vérifie le nom d'utilisateur, hache le mot de passe et le vérifie ($\text{bcrypt}(\text{pwd}_{U_a})$), extrait la clé publique du *token* du certificat enregistré et vérifie la signature :

$$\text{verifRSA}_{\text{token}}(k_{\text{RSA},U_a}^{\text{pub}}, m, \sigma_m).$$

Si les vérifications sont correctes, le protocole continue et le serveur renvoie le nouvel identifiant de session et un nouvel identifiant d'un appareil au client. L'appareil $D_{U_a,\alpha}$ est alors créé.

- l'appareil $D_{U_a,\alpha}$ génère ses clés de signatures :

$$(k_{f,D_{U_a,\alpha}}^{\text{pub}}, k_{f,D_{U_a,\alpha}}^{\text{priv}}) \leftarrow \text{keyGenWei}(),$$

et ses clés d'identité :

$$(k_{id,D_{U_a,\alpha}}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

et signe la partie publique de la clé d'identité avec la clé de signature :

$$\sigma_{id,D_{U_a,\alpha}} \leftarrow \text{signECSchnorr}(k_{f,D_{U_a,\alpha}}^{\text{priv}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}).$$

Des clés *one-time* sont générées régulièrement par l'appareil à partir d'une requête asynchrone depuis la connexion et tout au long de la session pour avoir toujours cinquante clés disponibles, et les parties publiques sont signées avec la clé de signature :

$$\begin{aligned} (k_{ot,D_{U_a,\alpha},0}^{\text{pub}}, k_{ot,D_{U_a,\alpha},0}^{\text{priv}}) &\leftarrow \text{keyGenWei}(), \\ \sigma_{ot,D_{U_a,\alpha},0} &\leftarrow \text{signECSchnorr}(k_{f,D_{U_a,\alpha}}^{\text{priv}}, k_{ot,D_{U_a,\alpha},0}^{\text{pub}}) \\ &\dots \\ (k_{ot,D_{U_a,\alpha},49}^{\text{pub}}, k_{ot,D_{U_a,\alpha},49}^{\text{priv}}) &\leftarrow \text{keyGenWei}() \\ \sigma_{ot,D_{U_a,\alpha},49} &\leftarrow \text{signECSchnorr}(k_{f,D_{U_a,\alpha}}^{\text{priv}}, k_{ot,D_{U_a,\alpha},49}^{\text{pub}}). \end{aligned}$$

Une paire de clé *fallback* est également générée par l'appareil au moment de la connexion et est signée avec la clé de signature :

$$\begin{aligned}(k_{fb,D_{U_a,\alpha}}^{\text{pub}}, k_{fb,D_{U_a,\alpha}}^{\text{priv}}) &\leftarrow \text{keyGenWei}() \\ \sigma_{fb,D_{U_a,\alpha}} &\leftarrow \text{signECSchnorr}(k_{fb,D_{U_a,\alpha}}^{\text{priv}}, k_{fb,D_{U_a,\alpha}}^{\text{pub}}).\end{aligned}$$

La clé *fallback* est utilisée par le protocole 3.A en cas d'épuisement temporaire de clés *one-time*. La clé *fallback* a la même durée de vie que l'appareil $D_{U_a,\alpha}$.

13. l'appareil $D_{U_a,\alpha}$ envoie toutes les clés publiques générées et leur signature au serveur.

3.2 Session 01m

Les conversations entre deux utilisateurs sont traitées par l'application comme des conversations de groupe dont les mécanismes sont décrits dans les protocoles 5, 6.A et 6.B. Néanmoins, ces conversations de groupe requièrent un échange de secrets qui s'effectue au moyen de sessions 01m entre deux appareils $D_{U_a,\alpha}$ et $D_{U_b,\beta}$. L'établissement de ces sessions et l'envoi et réception des messages correspondants sont décrits par les protocoles 3.A, 3.B, 4.A et 4.B. Notamment, le protocole 3.A est exécuté en premier par un appareil $D_{U_a,\alpha}$ qui souhaite établir une session 01m avec un appareil $D_{U_b,\beta}$. Le protocole 3.B est ensuite exécuté par $D_{U_b,\beta}$ pour traiter les premiers messages reçus par $D_{U_a,\alpha}$ avec le protocole 3.A. Dès lorsque $D_{U_b,\beta}$ envoie le premier message en retour, la session 01m est considérée établie et les deux appareils $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ exécutent par la suite les protocoles 4.A et 4.B pour l'envoi et la réception des messages.

Protocole 3.A (création de session 01m avec *outbound*). Nous décrivons l'initialisation de cette session par l'utilisateur U_a :

1. l'appareil $D_{U_a,\alpha}$ demande au serveur les données suivantes concernant un appareil $D_{U_b,\beta}$ de l'utilisateur U_b : sa clé publique de signature $k_{f,D_{U_b,\beta}}^{\text{pub}}$, sa clé publique d'identité $k_{id,D_{U_b,\beta}}^{\text{pub}}$ avec sa signature $\sigma_{id,D_{U_b,\beta}}$ et une de ses clés *one-time* $k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}$ avec sa signature $\sigma_{ot,D_{U_b,\beta},\ell}$.

Remarque. En cas d'épuisement de clés *one-time* sur le serveur pour $D_{U_b,\beta}$ (par exemple $D_{U_b,\beta}$ n'est pas en ligne ou n'est pas en mesure d'envoyer de nouvelles clés *one-time*), le serveur envoie la clé *fallback* $k_{fb,D_{U_b,\beta}}^{\text{pub}}$ avec sa signature $\sigma_{fb,D_{U_b,\beta}}$ à la place de la clé *one-time*.

2. l'appareil $D_{U_a,\alpha}$ vérifie les signatures des clés reçues :

$$\begin{aligned}\text{verifECSchnorr}(k_{f,D_{U_b,\beta}}^{\text{pub}}, k_{id,D_{U_b,\beta}}^{\text{pub}}, \sigma_{id,D_{U_b,\beta}}), \\ \text{verifECSchnorr}(k_{f,D_{U_b,\beta}}^{\text{pub}}, k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}, \sigma_{ot,D_{U_b,\beta},\ell}).\end{aligned}$$

3. l'appareil génère une nouvelle paire de clés *one-time* :

$$(k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}}, k_{ot,D_{U_a,\alpha},\ell'}^{\text{priv}}) \leftarrow \text{keyGenWei}().$$

4. l'appareil $D_{U_a,\alpha}$ dérive un secret partagé

$$S_{D_{U_a,\alpha},D_{U_b,\beta}} \leftarrow \text{ECDH}(k_{id,D_{U_a,\alpha}}^{\text{priv}}, k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}) || \text{ECDH}(k_{ot,D_{U_a,\alpha},\ell'}^{\text{priv}}, k_{id,D_{U_b,\beta}}^{\text{pub}}) || \text{ECDH}(k_{ot,D_{U_a,\alpha},\ell'}^{\text{priv}}, k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}).$$

5. l'appareil $D_{U_a,\alpha}$ calcule la clé racine R_0 et la première clé de chaînage $C_{0,0}$:

$$(R_0, C_{0,0}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \text{"OLM_ROOT"}, 64).$$

6. l'appareil $D_{U_a,\alpha}$ génère une clé *ratchet* Wei25519 :

$$(T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}, T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{priv}}) \leftarrow \text{keyGenWei}().$$

7. pour chaque message $\text{msg}_{0,j}$ ($j \geq 0$) à chiffrer avant que la connexion ne soit acceptée par le destinataire :

- (a) l'appareil $D_{U_a,\alpha}$ calcule une clé de message

$$M_{0,j} \leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{0,j}, \text{"x01"})$$

et une nouvelle clé de chaînage

$$C_{0,j+1} \leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{0,j}, \text{"x02"}).$$

(b) l'appareil $D_{U_a,\alpha}$ chiffre son message $\text{msg}_{0,j}$ comme suit :

$$\begin{aligned}
(k_{0,j}, \text{IV}_{0,j}^{\text{HKDF}}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, M_{0,j}, \text{"OLM_KEY"}, 48) \\
\text{session-id} &\leftarrow \text{SHA-3}(k_{id,D_{U_a,\alpha}}^{\text{pub}} \parallel k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}} \parallel k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}) \\
\text{IV}_{0,j}^{\text{AES}} &\leftarrow \text{session-id} \parallel \text{IV}_{0,j}^{\text{HKDF}} \\
c_{0,j} &\leftarrow \text{enc}^{\text{AES-CBC}}(k_{0,j}, \text{IV}_{0,j}^{\text{AES}}, \text{msg}_{0,j} \parallel \text{ID}_{U_a} \parallel \text{ID}_{D_{U_a,\alpha}} \parallel k_{f,D_{U_a,\alpha}}^{\text{pub}} \parallel \text{ID}_{U_b} \parallel k_{f,D_{U_b,\beta}}^{\text{pub}}) \\
\text{mac}_{0,j} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_{0,j}, \text{version} \parallel T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}} \parallel j \parallel \text{msg}_{0,j})|_{64 \text{ bits}}
\end{aligned}$$

où ID_{U_a} (respectivement ID_{U_b}) représente l'identifiant de l'utilisateur U_a (respectivement U_b), $\text{ID}_{D_{U_a,\alpha}}$ représente l'identifiant de l'appareil $D_{U_a,\alpha}$, et session-id représente l'identifiant de session.

(c) enfin, l'appareil $D_{U_a,\alpha}$ envoie :

$$(k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}, k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}, T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}, j, c_{0,j}, \text{mac}_{0,j}).$$

Protocole 3.B (création de session *01m* avec *inbound*). Ce protocole décrit la réception des premiers messages provenant de l'appareil $D_{U_a,\alpha}$ par l'appareil $D_{U_b,\beta}$. Il s'organise selon les étapes suivantes :

1. l'appareil $D_{U_b,\beta}$ reçoit de $D_{U_a,\alpha}$ le message

$$(k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}, k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}, T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}, j, c_{0,j}, \text{mac}_{0,j})$$

envoyé à la fin du protocole 3.A.

2. l'appareil $D_{U_b,\beta}$ demande au serveur les données suivantes concernant l'appareil $D_{U_a,\alpha}$ de l'utilisateur U_a duquel il a reçu un message du protocole 3.A : la clé publique de signature $k_{f,D_{U_a,\alpha}}^{\text{pub}}$ qui correspond à la clé publique d'identité $k_{id,D_{U_a,\alpha}}^{\text{pub}}$ qu'il a reçu dans le message, ainsi que sa signature $\sigma_{id,D_{U_a,\alpha}}$.

3. l'appareil $D_{U_b,\beta}$ vérifie la signature de la clé reçue :

$$\text{verifECSchnorr}(k_{f,D_{U_a,\alpha}}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}, \sigma_{id,D_{U_a,\alpha}}).$$

4. l'appareil $D_{U_b,\beta}$ dérive le secret partagé :

$$S_{D_{U_a,\alpha},D_{U_b,\beta}} \leftarrow \text{ECDH}(k_{ot,D_{U_b,\beta},\ell}^{\text{priv}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}) \parallel \text{ECDH}(k_{id,D_{U_b,\beta}}^{\text{priv}}, k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}}) \parallel \text{ECDH}(k_{ot,D_{U_b,\beta},\ell}^{\text{priv}}, k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}}),$$

où $k_{ot,D_{U_b,\beta},\ell}^{\text{priv}}$ est la partie privée de la clé *one time* de $D_{U_b,\beta}$ correspondant à la partie publique $k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}$ utilisée par $D_{U_a,\alpha}$ pour initialiser la session et reçue en étape 1.

5. l'appareil $D_{U_b,\beta}$ calcule la clé racine R_0 et la première clé de chaînage $C_{0,0}$:

$$(R_0, C_{0,0}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \text{"OLM_ROOT"}, 64).$$

6. pour chaque message $c_{0,j}$ ($j \geq 0$) à déchiffrer, envoyé avant que la conversation ne soit établie :

(a) si j est supérieur à l'indice x de la clé de chaînage $C_{0,x}$ courante, l'appareil $D_{U_b,\beta}$ calcule la clé de chaînage correspondant à l'indice j depuis la dernière clé de chaînage $C_{0,x}$ et les clés de messages correspondant aux clés de chaînage successives :

$$\begin{aligned}
\forall x < \ell \leq j, \quad C_{0,\ell} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{0,\ell-1}, \text{"x02"}) \\
M_{0,\ell} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{0,\ell}, \text{"x01"})
\end{aligned}$$

Les clés de message $M_{0,x}, \dots, M_{0,j-1}$ sont stockées en mémoire par $D_{U_b,\beta}$.

(b) l'appareil $D_{U_b,\beta}$ déchiffre et vérifie le message reçu :

$$\begin{aligned}
(k_{0,j}, \text{IV}_{0,j}^{\text{HKDF}}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, M_{0,j}, \text{"OLM_KEY"}, 48) \\
\text{session-id} &\leftarrow \text{SHA-3}(k_{id,D_{U_a,\alpha}}^{\text{pub}} \parallel k_{ot,D_{U_a,\alpha},\ell'}^{\text{pub}} \parallel k_{ot,D_{U_b,\beta},\ell}^{\text{pub}}) \\
\text{IV}_{0,j}^{\text{AES}} &\leftarrow \text{session-id} \parallel \text{IV}_{0,j}^{\text{HKDF}} \\
\text{msg}_{0,j} \parallel \text{ID}_{U_a} \parallel \text{ID}_{D_{U_a,\alpha}} \parallel k_{f,D_{U_a,\alpha}}^{\text{pub}'} \parallel \text{ID}_{U_b} \parallel k_{f,D_{U_b,\beta}}^{\text{pub}'} &\leftarrow \text{dec}^{\text{AES-CBC}}(k_{0,j}, \text{IV}_{0,j}^{\text{AES}}, c_{0,j}) \\
\text{verifMac}^{\text{HMAC-SHA-3}}(k_{0,j}, \text{version} \parallel T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}} \parallel j \parallel \text{msg}_{0,j}, \text{mac}_{0,j}) &|_{64 \text{ bits}}
\end{aligned}$$

- (c) si le déchiffrement réussit, l'appareil vérifie que ID_{U_b} correspond bien à son identifiant d'utilisateur. L'appareil vérifie également que la clé de signature $k_{f,D_{U_b,\beta}}^{\text{pub}'}$ correspond bien à sa clé de signature, et que la clé de signature $k_{f,D_{U_a,\alpha}}^{\text{pub}'}$ correspond bien à celle qu'il a récupérée du serveur à la première étape $k_{f,D_{U_a,\alpha}}^{\text{pub}}$, et que ID_{U_a} et $ID_{D_{U_a,\alpha}}$ correspondent aux identifiants d'utilisateur et d'appareil en possession des clés en question.
- (d) si toutes les vérifications réussissent, $D_{U_b,\beta}$ supprime $k_{ot,D_{U_b,\beta},\ell}^{\text{priv}}$ et enregistre $T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}$.

Protocole 4.A (envoi de messages à un utilisateur connu). Ce protocole est utilisé pour l'échange de clés *ratchet* dans le contexte de conversations de groupe. En particulier, il correspond à la procédure lorsque la communication entre deux appareils $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ est établie et que ces derniers souhaitent échanger de nouvelles clés correspondant à une nouvelle conversation de groupe par exemple. Lorsque que $D_{U_a,\alpha}$ envoie des messages à $D_{U_b,\beta}$, il suit les étapes suivantes :

1. $D_{U_a,\alpha}$ regarde s'il a une clé de chaînage *émetteur* $C_{i,j}$ pour l'indice i courant (cf. Protocole 4.B). Si ce n'est pas le cas,
 - (a) une nouvelle clé *ratchet* Wei25519 est générée :

$$(T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}, T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{priv}}) \leftarrow \text{keyGenWei}().$$

- (b) une nouvelle clé racine et une nouvelle clé de chaînage sont dérivées :

$$(R_i, C_{i,0}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(R_{i-1}, \text{ECDH}(T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{priv}}, T_{i-1,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}), \text{"OLM_RATCHET"}, 64).$$

2. une nouvelle clé de message est calculée ($j = 0$ si une nouvelle clé de chaînage vient d'être calculée)

$$M_{i,j} \leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{i,j}, \text{"\x01"})$$

et la clé de chaînage est avancée

$$C_{i,j+1} \leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{i,j}, \text{"\x02"}).$$

3. le message $\text{msg}_{i,j}$ est chiffré comme suit :

$$\begin{aligned} (k_{i,j}, \text{IV}_{i,j}^{\text{HKDF}}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, M_{i,j}, \text{"OLM_KEY"}, 48) \\ \text{IV}_i^{\text{AES}} &\leftarrow \text{session-id} || \text{IV}_i^{\text{HKDF}} \\ c_{i,j} &\leftarrow \text{enc}^{\text{AES-CBC}}(k_{i,j}, \text{IV}_i^{\text{AES}}, \text{msg}_{i,j} || \text{ID}_{U_a} || \text{ID}_{D_{U_a,\alpha}} || k_{f,D_{U_a,\alpha}}^{\text{pub}} || \text{ID}_{U_b} || k_{f,D_{U_b,\beta}}^{\text{pub}}) \\ \text{mac}_{i,j} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_{i,j}, \text{version} || T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}} || j || \text{msg}_{i,j})_{64 \text{ bits}} \end{aligned}$$

4. enfin, l'appareil $D_{U_a,\alpha}$ envoie

$$(j, c_{i,j}, \text{mac}_{i,j}, T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}),$$

où $T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}$ est sa clé *ratchet* publique et $k_{id,D_{U_a,\alpha}}^{\text{pub}}$ sa clé d'identité.

Protocole 4.B (réception de messages d'un utilisateur connu). Ce protocole fait suite au protocole 5 avec la réception par $D_{U_b,\beta}$ des messages envoyés par $D_{U_a,\alpha}$. A réception du message $c_{i,j}$, $D_{U_b,\beta}$ effectue les étapes suivantes :

1. l'appareil $D_{U_b,\beta}$ reçoit de $D_{U_a,\alpha}$ le message

$$(j, c_{i,j}, \text{mac}_{i,j}, T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}),$$

envoyé à la fin du protocole 4.A.

2. $D_{U_b,\beta}$ vérifie qu'il a bien une session 0lm établie avec l'appareil $D_{U_a,\alpha}$ de clé d'identité $k_{id,D_{U_a,\alpha}}^{\text{pub}}$.
3. $D_{U_b,\beta}$ regarde s'il a une clé de chaînage *récepteur* correspondant à $T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}$. Si ce n'est pas le cas, une nouvelle clé racine et une nouvelle clé de chaînage *récepteur* sont dérivées :

$$R_i || C_{i,0} \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(R_{i-1}, \text{ECDH}(T_{i-1,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{priv}}, T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}}), \text{"OLM_RATCHET"}, 64).$$

La clé chaînage *émetteur* courante est alors effacée.

4. si j est inférieur à l'indice x de la clé de chaînage $C_{i,x}$ courante, l'appareil $D_{U_b,\beta}$ récupère la clé de message $M_{i,j}$ préalablement calculée et stockée en mémoire.
5. si j est égal à l'indice x de la clé de chaînage $C_{i,x}$ courante, l'appareil $D_{U_b,\beta}$ calcule la clé de message correspondant à l'indice j :

$$M_{i,j} \leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{i,j}, "\text{x01}")$$

6. si j est supérieur à l'indice x de la clé de chaînage $C_{i,x}$ courante, l'appareil $D_{U_b,\beta}$ calcule la clé de chaînage correspondant à l'indice j depuis la dernière clé de chaînage $C_{i,x}$ et les clés de messages correspondant aux clés de chaînage successives :

$$\begin{aligned} \forall x < \ell \leq j, \quad C_{i,\ell} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{i,\ell-1}, "\text{x02}") \\ M_{i,\ell} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(C_{i,\ell}, "\text{x01}") \end{aligned}$$

Les clés de message $M_{i,x}, \dots, M_{i,j-1}$ sont stockées en mémoire par $D_{U_b,\beta}$.

7. enfin, le message $c_{i,j}$ est déchiffré comme suit :

$$\begin{aligned} (k_{i,j}, \text{IV}_{i,j}^{\text{HKDF}}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, M_{i,j}, \text{"OLM_KEY"}, 48) \\ \text{IV}_i^{\text{AES}} &\leftarrow \text{session-id} \parallel \text{IV}_i^{\text{HKDF}} \\ \text{msg}_{i,j} \parallel \text{ID}_{U_a} \parallel \text{ID}_{D_{U_a,\alpha}} \parallel k_{f,D_{U_a,\alpha}}^{\text{pub}'} \parallel \text{ID}_{U_b} \parallel k_{f,D_{U_b,\beta}}^{\text{pub}'} &\leftarrow \text{dec}^{\text{AES-CBC}}(k_{i,j}, \text{IV}_i^{\text{AES}}, c_{i,j}) \\ \text{verifMac}^{\text{HMAC-SHA-3}}(k_{i,j}, \text{version} \parallel T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\text{pub}} \parallel j \parallel \text{msg}_{i,j}, \text{mac}_{i,j}) &|_{64 \text{ bits}} \end{aligned}$$

8. si le déchiffrement réussit, l'appareil vérifie que ID_{U_b} correspond bien à son identifiant d'utilisateur. L'appareil vérifie également que la clé de signature $k_{f,D_{U_b,\beta}}^{\text{pub}'}$ correspond bien à sa clé de signature, et que la clé de signature $k_{f,D_{U_a,\alpha}}^{\text{pub}'}$ correspond bien à celle qu'il a récupéré au protocole 3.A pour l'appareil $D_{U_a,\alpha}$, et que ID_{U_a} et $\text{ID}_{D_{U_a,\alpha}}$ correspondent aux identifiants d'utilisateur et d'appareil en possession des clés en question.
9. la clé de chaînage $C_{i,j}$ est remplacée par

$$C_{i,j+1} = \text{computeMac}^{\text{HMAC-SHA-3}}(C_{i,j+1}, "\text{x02}").$$

Le cas échéant, l'ancienne clé racine R_{i-1} est remplacée par R_i .

3.3 Session Megolm

Les conversations entre deux ou plusieurs utilisateurs sont traitées par l'application comme des conversations de groupe dont les mécanismes sont décrits dans les protocoles 5, 6.A, 6.B, 7.A et 7.B. Chaque utilisateur dans le groupe commence par exécuter le protocole 5 afin de partager avec tous les autres utilisateurs sa clé de chiffrement *ratchet symétrique* pour que ces derniers puissent déchiffrer ses messages dans le groupe. Ce protocole est exécuté par chaque utilisateur à chaque nouveau changement de sa clé *ratchet symétrique*. Ensuite, un utilisateur qui souhaite envoyer un message dans la conversation exécute le protocole 6.A. A la réception d'un message, le protocole 6.B est déclenché. Enfin, le protocole 7.A est utilisé pour l'envoi d'une pièce jointe chiffrée, et le protocole 7.B est utilisé pour sa réception.

Protocole 5 (envoi du premier message dans un groupe). A la création d'une conversation de groupe *i.e.* d'un salon d'identifiant r , chaque participant (via son appareil $D_{U_a,\alpha}$) suit la séquence d'étapes suivantes pour l'envoi de son premier message :

1. l'appareil $D_{U_a,\alpha}$ initialise un compteur i à 0.
2. il génère une paire de clés de signature Wei25519 qui permettront de signer les messages de la conversation r :

$$(k_{f,D_{U_a,\alpha},r}^{\text{pub}}, k_{f,D_{U_a,\alpha},r}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

et envoie la partie publique ($k_{f,D_{U_a,\alpha},r}^{\text{pub}}$) au serveur.

3. l'appareil génère un *ratchet symétrique* de 1024 bits d'aléa $R_{0,D_{U_a,\alpha},r}$ sous forme de quatre blocs de 256 bits : $R_{0,j,D_{U_a,\alpha},r}$ pour $j \in \{0, 1, 2, 3\}$:

$$\begin{aligned} (R_{0,0,D_{U_a,\alpha},r}, R_{0,1,D_{U_a,\alpha},r}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, k_{id,D_{U_a,\alpha}}^{\text{priv}}, \text{"MEGOLM_ROOT"}, 64) \\ (R_{0,2,D_{U_a,\alpha},r}, R_{0,3,D_{U_a,\alpha},r}) &\leftarrow \text{DRBG}(512). \end{aligned}$$

4. toutes les données de session sont signées :

$$\begin{aligned} \text{data} &\leftarrow \text{version} || i || R_{0,0,D_{U_a,\alpha},r} || R_{0,1,D_{U_a,\alpha},r} || R_{0,2,D_{U_a,\alpha},r} || R_{0,3,D_{U_a,\alpha},r} || k_{f,D_{U_a,\alpha},r}^{\text{pub}} \\ \sigma_{\text{data}} &\leftarrow \text{signECSchnorr}(k_{f,D_{U_a,\alpha},r}^{\text{priv}}, \text{data}). \end{aligned}$$

5. les données **data** et leur signature σ_{data} sont envoyées à tous les participants de la conversation au moyen de sessions Olm (conversations un à un), explicitées dans les protocoles 3.A à 4.B.

La clé *ratchet symétrique* correspond désormais à une session Megolm unique associée à $D_{U_a,\alpha}$ dans le salon, d'identifiant $k_{f,D_{U_a,\alpha},r}^{\text{pub}}$. En effet, dans un salon d'identifiant r , il existe pour chaque appareil une session Megolm correspondant à sa clé de signature dans le salon et sa clé *ratchet symétrique*.

Protocole 6.A (envoi de messages dans une conversation de groupe). Lorsque l'appareil $D_{U_a,\alpha}$ souhaite envoyer un $i^{\text{ème}}$ message msg_i dans une conversation de groupe d'identifiant r en utilisant sa session Megolm créée avec le protocole 5, il suit les étapes suivantes :

1. il génère des clés et vecteur d'initialisation, et chiffre le message

$$\begin{aligned} \text{session-id} &\leftarrow k_{f,D_{U_a,\alpha},r}^{\text{pub}} \\ (k_i, \text{IV}_i^{\text{HKDF}}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, R_{i,D_{U_a,\alpha},r}, \text{"MEGOLM_KEYS"}, 48) \\ \text{IV}_i^{\text{AES}} &\leftarrow \text{session-id} || \text{IV}_i^{\text{HKDF}} \\ c_{\text{msg}_i} &\leftarrow \text{enc}^{\text{AES-CBC}}(k_i, \text{IV}_i^{\text{AES}}, \text{msg}_i) \end{aligned}$$

et l'authentifie

$$\text{mac}_i \leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_i, \text{version} || i || \text{msg}_i)_{64 \text{ bits}}.$$

2. le chiffré et les 64 premiers bits du mac sont signés

$$\sigma_i \leftarrow \text{signECSchnorr}(k_{f,D_{U_a,\alpha},r}^{\text{priv}}, c_{\text{msg}_i} || \text{mac}_i)_{64 \text{ bits}}$$

et le chiffré c_{msg_i} , le motif d'authentification mac_i , la signature σ_i , l'indice i , la clé publique $k_{f,D_{U_a,\alpha},r}^{\text{pub}}$, la clé publique d'identité de l'appareil $k_{id,D_{U_a,\alpha}}^{\text{pub}}$ ainsi que l'identifiant du salon en question ID_r sont tous envoyés dans un message

$$(c_{\text{msg}_i}, \text{mac}_i, \sigma_i, i, k_{f,D_{U_a,\alpha},r}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}, \text{ID}_r).$$

3. l'appareil met à jour son *ratchet symétrique*

$$\begin{aligned} R_{i+1,0,D_{U_a,\alpha},r} &\leftarrow \begin{cases} \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{24}(n-1),0,D_{U_a,\alpha},r}, \text{"x00"}) & \text{si } \exists n \text{ t.q. } i+1 = 2^{24}n \\ R_{i,0,D_{U_a,\alpha},r} & \text{sinon.} \end{cases} \\ R_{i+1,1,D_{U_a,\alpha},r} &\leftarrow \begin{cases} \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{24}(n-1),0,D_{U_a,\alpha},r}, \text{"x01"}) & \text{si } \exists n \text{ t.q. } i+1 = 2^{24}n \\ \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{16}(m-1),1,D_{U_a,\alpha},r}, \text{"x01"}) & \text{si } \exists m \text{ t.q. } i+1 = 2^{16}m \\ R_{i,1,D_{U_a,\alpha},r} & \text{sinon.} \end{cases} \\ R_{i+1,2,D_{U_a,\alpha},r} &\leftarrow \begin{cases} \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{24}(n-1),0,D_{U_a,\alpha},r}, \text{"x02"}) & \text{si } \exists n \text{ t.q. } i+1 = 2^{24}n \\ \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{16}(m-1),1,D_{U_a,\alpha},r}, \text{"x02"}) & \text{si } \exists m \text{ t.q. } i+1 = 2^{16}m \\ \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^8(p-1),2,D_{U_a,\alpha},r}, \text{"x02"}) & \text{si } \exists p \text{ t.q. } i+1 = 2^8p \\ R_{i,2,D_{U_a,\alpha},r} & \text{sinon.} \end{cases} \\ R_{i+1,3,D_{U_a,\alpha},r} &\leftarrow \begin{cases} \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{24}(n-1),0,D_{U_a,\alpha},r}, \text{"x03"}) & \text{si } \exists n \text{ t.q. } i+1 = 2^{24}n \\ \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^{16}(m-1),1,D_{U_a,\alpha},r}, \text{"x03"}) & \text{si } \exists m \text{ t.q. } i+1 = 2^{16}m \\ \text{computeMac}^{\text{HMAC-SHA-3}}(R_{2^8(p-1),2,D_{U_a,\alpha},r}, \text{"x03"}) & \text{si } \exists p \text{ t.q. } i+1 = 2^8p \\ \text{computeMac}^{\text{HMAC-SHA-3}}(R_{i,3,D_{U_a,\alpha},r}, \text{"x03"}) & \text{sinon.} \end{cases} \end{aligned}$$

4. toutes les semaines, l'application déclenche la génération d'un nouveau *ratchet symétrique*.

Protocole 6.B (réception de messages dans une conversation de groupe). Dans ce protocole, nous considérons la réception de messages de $D_{U_a,\alpha}$ par l'appareil $D_{U_b,\beta}$ dans la conversation de groupe r . Les *ratchets symétriques* ont déjà été échangés au moyen d'une session Olm. Les étapes sont donc les suivantes :

1. l'appareil $D_{U_b,\beta}$ reçoit de $D_{U_a,\alpha}$ le message

$$(c_{\text{msg}_i}, \text{mac}_i, \sigma_i, i, k_{f,D_{U_a,\alpha},r}^{\text{pub}}, k_{id,D_{U_a,\alpha}}^{\text{pub}}, \text{ID}_r).$$

envoyé à la fin du protocole 6.B.

- l'appareil vérifie qu'il a bien une session *megolm* établie dans le salon d'identifiant ID_r avec l'utilisateur de clé d'identité $k_{id,D_{U_a,\alpha}}^{pub}$.
- l'appareil $D_{U_b,\beta}$ vérifie la signature en utilisant la clé de signature publique $k_{f,D_{U_a,\alpha},r}^{pub}$:

$$\text{verifECSchnorr}(k_{f,D_{U_a,\alpha},r}^{pub}, c_{msg_i} || \text{mac}_i |_{64 \text{ bits}}, \sigma_i).$$

- si la signature est correcte, l'appareil $D_{U_b,\beta}$ avance le *ratchet symétrique* de $D_{U_a,\alpha}$ en fonction de l'indice i du message reçu, en suivant la procédure décrite dans le protocole 6.A.
- $D_{U_b,\beta}$ déchiffre le message

$$\begin{aligned} \text{session-id} &\leftarrow k_{f,D_{U_a,\alpha},r}^{pub} \\ (k_i, IV_i^{HKDF}) &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, R_{i,D_{U_a,\alpha},r}, \text{"MEGOLM_KEYS"}, 48) \\ IV_i^{AES} &\leftarrow \text{session-id} || IV_i^{HKDF} \\ \text{msg}_i &\leftarrow \text{dec}^{\text{AES-CBC}}(k_i, IV_i^{AES}, c_{msg_i}) \end{aligned}$$

et vérifie que le padding est correct.

- $D_{U_b,\beta}$ vérifie le motif d'authentification :

$$\text{verifMac}^{\text{HMAC-SHA-3}}(k_i, \text{version} || i || \text{msg}_i, \text{mac}_i) |_{64 \text{ bits}}.$$

Protocole 7.A (envoi d'une pièce jointe chiffrée). Pour envoyer une pièce jointe *pj* de manière sécurisée dans une conversation r , l'appareil $D_{U_a,\alpha}$ procède comme suit :

- l'appareil $D_{U_a,\alpha}$ génère une clé AES, un vecteur d'initialisation et une clé HMAC à partir du *ratchet symétrique* courant :

$$(k^{AES}, k^{HMAC}, IV^{AES}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, R_{i,D_{U_a,\alpha},r}, \text{"attachment"}, 80).$$

- la pièce jointe est chiffrée :

$$c_{pj} \leftarrow \text{enc}^{\text{AES-CTR}}(k^{AES}, IV^{AES}, \text{pj})$$

et un *mac* est calculé :

$$\text{mac}_{pj} \leftarrow \text{computeMac}^{\text{HMAC-SHA-1}}(k^{HMAC}, c_{pj}).$$

- la pièce jointe chiffrée c_{pj} est chargée sur le serveur et mise à disposition par un lien *url* public. La clé k^{AES} , la clé k^{HMAC} , le vecteur d'initialisation IV^{AES} , l'adresse *url* et le motif d'authentification mac_{pj} sont envoyés dans le message

$$(k^{AES}, k^{HMAC}, IV^{AES}, \text{url}, \text{mac}_{pj})$$

dans la session *Megolm* courante en utilisant le protocole 6.A.

Protocole 7.B (réception d'une pièce jointe chiffrée). Lorsqu'un appareil $D_{U_a,\alpha}$ envoie une pièce jointe chiffrée à un utilisateur U_b , son appareil courant $D_{U_b,\beta}$ reçoit, par le biais d'une session *Megolm*, et déchiffre en utilisant le protocole 6.B, le message

$$(k^{AES}, k^{HMAC}, IV^{AES}, \text{url}, \text{mac}_{pj})$$

envoyé à la fin du protocole 7.A. L'appareil $D_{U_b,\beta}$ suit donc les étapes suivantes :

- il télécharge la pièce jointe chiffrée c_{pj} et vérifie son motif d'authentification

$$\text{verifMac}^{\text{HMAC-SHA-1}}(k^{HMAC}, c_{pj}, \text{mac}_{pj}).$$

- il déchiffre ensuite la pièce jointe

$$\text{pj} \leftarrow \text{dec}^{\text{AES-CTR}}(k^{AES}, IV^{AES}, c_{pj}).$$

3.4 Stockage sécurisé

Dans cette section, nous décrivons les protocoles 8, 9, 10, 11 et 12 qui servent à réaliser le stockage sécurisé des clés privées chiffrées sur le serveur. Un utilisateur qui souhaite créer une sauvegarde de ses clés privées sur le serveur commence par exécuter par le biais de son appareil le protocole 8 afin de générer une clé symétrique nommée *secret storage* qui servira au chiffrement des clés de sauvegarde. Cette clé est générée en utilisant un mot de passe choisi par l'utilisateur. Ensuite, en utilisant la clé *secret storage*, l'utilisateur peut chiffrer ses clés privées et les stocker sur le serveur en utilisant le protocole 9, et peut également les récupérer et déchiffrer en utilisant le protocole 10. Pour la sauvegarde des messages chiffrés de conversation *Megolm*, l'utilisateur exécute le protocole 11 afin de créer une sauvegarde chiffrée des clés de session *Megolm* utilisées dans les protocoles 5 à 6.B et les stocker sur le serveur. L'utilisateur peut ensuite récupérer ces clés *Megolm* en utilisant le protocole 12.

Protocole 8 (génération et stockage d'une clé *secret storage*). La clé *secret storage* d'un utilisateur U_a lui permet de chiffrer des secrets qu'il pourra stocker sur le serveur. Nous décrivons sa génération ci-après :

1. l'appareil courant $D_{U_a,\alpha}$ de l'utilisateur U_a génère un sel S de 128 bits d'aléa cryptographique :

$$S \leftarrow \text{DRBG}(128).$$

2. l'utilisateur choisit une passphrase $\text{pass}_{U_a}^{\text{sec}}$ dont la résistance est évaluée par le module `zxcvbn-1.5.2` utilisé par l'application.
3. l'appareil génère la clé secrète *secret storage*

$$k_{\text{sec},U_a} \leftarrow \text{PBKDF2}^{\text{HMAC-SHA-3}}(\text{pass}_{U_a}^{\text{sec}}, S, 2^{20}, 32).$$

4. l'appareil $D_{U_a,\alpha}$ envoie au serveur le nombre d'itérations N et le sel S .
5. l'appareil vérifie s'il a accès aux clés privées *cross signing* (ce qui est le cas pour la toute première connexion de l'utilisateur au serveur avec le protocole 1 ou si l'appareil courant de l'utilisateur s'est authentifié auprès d'un ancien appareil avec le protocole 13 pour récupérer les anciennes clés). Si ce n'est pas le cas, l'appareil génère trois nouvelles paires de clés *cross signing* comme dans le protocole 1 :

$$(\text{msk}_{U_a}^{\text{pub}}, \text{msk}_{U_a}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

$$(\text{usk}_{U_a}^{\text{pub}}, \text{usk}_{U_a}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

$$(\text{ssk}_{U_a}^{\text{pub}}, \text{ssk}_{U_a}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

et signe les clés publiques *user-signing* et *self-signing* avec la clé maître :

$$\sigma_{\text{usk}_{U_a}, \text{msk}_{U_a}} \leftarrow \text{signECSchnorr}(\text{msk}_{U_a}^{\text{priv}}, \text{usk}_{U_a}^{\text{pub}})$$

$$\sigma_{\text{ssk}_{U_a}, \text{msk}_{U_a}} \leftarrow \text{signECSchnorr}(\text{msk}_{U_a}^{\text{priv}}, \text{ssk}_{U_a}^{\text{pub}}).$$

L'appareil $D_{U_a,\alpha}$ signe également la partie publique de sa clé de signature avec la clé *self-signing* :

$$\sigma_{D_{U_a,\alpha}, \text{ssk}_{U_a}} \leftarrow \text{signECSchnorr}(\text{ssk}_{U_a}^{\text{priv}}, k_{f,D_{U_a,\alpha}}^{\text{pub}})$$

et la partie publique de la clé maître avec sa clé de signature :

$$\sigma_{\text{msk}_{U_a}, D_{U_a,\alpha}} \leftarrow \text{signECSchnorr}(k_{f,D_{U_a,\alpha}}^{\text{priv}}, \text{msk}_{U_a}^{\text{pub}}).$$

Enfin, l'appareil $D_{U_a,\alpha}$ envoie toutes les clés publiques générées et les signatures au serveur. L'appareil utilise ensuite le protocole 9 pour stocker les parties privées de ces trois clés $\text{msk}_{U_a}^{\text{priv}}$, $\text{usk}_{U_a}^{\text{priv}}$ et $\text{ssk}_{U_a}^{\text{priv}}$ sur le serveur en utilisant k_{sec,U_a} . Le protocole 9 est exécuté pour chacune des clés secrète de manière indépendante.

6. l'appareil génère une nouvelle paire de clés *recovery* Wei25519 qui sera utilisée dans le protocole 11 pour stocker les clés secrètes des sessions *Megolm*

$$(k_{\text{rec},U_a}^{\text{priv}}, k_{\text{rec},U_a}^{\text{pub}}) \leftarrow \text{keyGenWei}().$$

L'appareil utilise ensuite le protocole 9 pour stocker la partie privée $k_{\text{rec},U_a}^{\text{priv}}$ sur le serveur en utilisant k_{sec,U_a} .

Protocole 9 (stockage sécurisé sur le serveur). L'appareil permet à chaque utilisateur U_a en utilisant sa clé *secret storage* k_{sec,U_a} générée avec le protocole 8 de stocker de manière sécurisée une clé privée backupK_i à la fois (*i.e.*, clés *cross-signing*, clés *recovery*) sur le serveur.

La procédure est la suivante :

1. l'appareil $D_{U_a,\alpha}$ génère deux clés de 256 bits chacune à partir de k_{sec,U_a}

$$(k_i^{\text{AES}}, k_i^{\text{mac}}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(\text{NULL}, k_{\text{sec},U_a}, \text{INFO}_i, 64)$$

où INFO_i est une chaîne connue qui dépend du type de la clé à stocker.

2. l'appareil génère aussi un vecteur d'initialisation IV_i^{AES} de 128 bits d'aléa cryptographique

$$\text{IV}_i^{\text{AES}} \leftarrow \text{DRBG}(128)$$

et fixe le 63^{ème} bit à 0 (ce bit fixé est présent dans les spécifications d'origine).

- l'appareil $D_{U_a, \alpha}$ chiffre et authentifie la clé backupK_i à stocker :

$$\begin{aligned} c_{\text{backupK}_i} &\leftarrow \text{enc}^{\text{AES-CTR}}(k_i^{\text{AES}}, \text{IV}_i^{\text{AES}}, \text{backupK}_i) \\ \text{mac}_{\text{backupK}_i} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_i^{\text{mac}}, c_{\text{backupK}_i}) \end{aligned}$$

et envoie c_{backupK} , IV^{AES} et $\text{mac}_{\text{backupK}}$ au serveur.

Protocole 10 (récupération des clés stockées sur le serveur). Pour récupérer une clé backupK_i stockée sur le serveur et chiffrée avec la clé *secret storage* en utilisant le protocole 9 de son utilisateur U_a , l'appareil $D_{U_a, \alpha}$ suit les étapes suivantes :

- l'appareil récupère du serveur la clé c_{backupK_i} qui a été chiffrée pendant une session précédente avec le protocole 8. Il récupère aussi le mac $\text{mac}_{\text{backupK}_i}$ et le vecteur d'initialisation IV_i^{AES} correspondant.
- l'utilisateur entre sa passphrase $\text{pass}_{U_a}^{\text{sec}}$ et $D_{U_a, \alpha}$ calcule sa clé *secret storage* :

$$k_{\text{sec}, U_a} \leftarrow \text{PBKDF2}^{\text{HMAC-SHA-3}}(\text{pass}_{U_a}^{\text{sec}}, S, 2^{20}, 32).$$

- $D_{U_a, \alpha}$ dérive les clés de chiffrement et d'authentification pour la clé chiffrée :

$$(k_i^{\text{AES}}, k_i^{\text{mac}}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, k_{\text{sec}, U_a}, \text{INFO}_i, 64)$$

où INFO_i est une chaîne connue qui dépend du type de la clé à récupérer.

- l'appareil $D_{U_a, \alpha}$ vérifie le $\text{mac}_{\text{backupK}_i}$

$$\text{verifMac}^{\text{HMAC-SHA-3}}(k_i^{\text{mac}}, c_{\text{backupK}_i}, \text{mac}_{\text{backupK}_i})$$

et déchiffre la clé :

$$\text{backupK}_i \leftarrow \text{dec}^{\text{AES-CTR}}(k_i^{\text{AES}}, \text{IV}_i^{\text{AES}}, c_{\text{backupK}_i}).$$

- Si $D_{U_a, \alpha}$ récupère et déchiffre pour la première fois les clés *cross-signing* du serveur, il signe sa clé publique de signature avec la clé *self-signing* de U_a

$$\sigma_{f, D_{U_a, \alpha}} \leftarrow \text{signECSchnorr}(\text{ssk}_{U_a}^{\text{priv}}, k_{f, D_{U_a, \alpha}}^{\text{pub}})$$

et il envoie cette clé publique ainsi que sa signature $\sigma_{f, D_{U_a, \alpha}}$ au serveur.

Remarque. Lorsque le protocole 10 est exécuté, l'appareil $D_{U_a, \alpha}$ demande au serveur toutes les clés c_{backupK_i} qui ont été chiffrées pendant des sessions précédentes avec le protocole 9. Ensuite, $D_{U_a, \alpha}$ exécute le protocole 10 pour chacune des clés récupérées.

Protocole 11 (stockage sécurisé de clés de session Megolm sur le serveur). L'application permet à chaque utilisateur U_a de stocker de manière sécurisée une clé secrète backupM_i de session Megolm sur le serveur en utilisant sa paire de clé *recovery* $(k_{\text{rec}, U_a}^{\text{priv}}, k_{\text{rec}, U_a}^{\text{pub}})$ générée dans le protocole 8. La procédure est la suivante :

- l'appareil $D_{U_a, \alpha}$ génère un secret partagé S :

$$S \leftarrow \text{ECDH}(k_{id, D_{U_a, \alpha}}^{\text{priv}}, k_{\text{rec}, U_a}^{\text{pub}}).$$

- une clé et un vecteur d'initialisation sont dérivés de ce secret partagé :

$$(k_i^{\text{AES}}, \text{IV}_i^{\text{AES}}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S, \emptyset, 64).$$

- l'appareil $D_{U_a, \alpha}$ chiffre et authentifie les clés backupM_i à stocker :

$$(c_{\text{backupM}_i}, \text{mac}_{\text{backupM}_i}) \leftarrow \text{encAE}^{\text{AES-GCM}}(k_i^{\text{AES}}, \text{IV}_i^{\text{AES}}, \text{backupM}_i)$$

et envoie c_{backupM_i} et $\text{mac}_{\text{backupM}_i} |_{32 \text{ bits}}$ au serveur.

Protocole 12 (récupération des clés de sessions Megolm stockées sur le serveur). Pour récupérer une clé de session megolm backupM_i stockée sur le serveur et chiffrée avec la clé *recovery* de son utilisateur U_a générée dans le protocole 8, l'appareil $D_{U_a, \alpha}$ suit les étapes suivantes :

- l'appareil récupère du serveur sa clé privée *recovery* $k_{\text{rec}, U_a}^{\text{priv}}$ en utilisant le protocole 10.
- L'appareil récupère aussi sa clé publique *recovery* $k_{\text{rec}, U_a}^{\text{pub}}$ stockée en clair sur le serveur. Il vérifie la cohérence entre la clé publique stockée et la clé privée déchiffrée

$$(k_{\text{rec}, U_a}^{\text{pub}} \stackrel{?}{=} \text{keyGenWeiPub}(k_{\text{rec}, U_a}^{\text{priv}})).$$

3. $D_{U_a,\alpha}$ récupère la clé publique *identité* $k_{id,D_{U_b,\beta}}^{\text{pub}}$ ayant servi au chiffrement du stockage sécurisé et le dernier backupM_i (avec son motif d'authentification $\text{mac}_{\text{backupM}_i}$) du serveur. Il recalcule le secret partagé :

$$S \leftarrow \text{ECDH}(k_{\text{rec},U_a}^{\text{priv}}, k_{id,D_{U_b,\beta}}^{\text{pub}}).$$

4. $D_{U_a,\alpha}$ dérive les clés et un IV :

$$(k_i^{\text{AES}}, \text{IV}_i^{\text{AES}}) \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S, \emptyset, 64).$$

5. l'appareil $D_{U_a,\alpha}$ vérifie le mac et déchiffre les clés :

$$\text{backupM}_i \leftarrow \text{decAE}^{\text{AES-GCM}}(k_i^{\text{AES}}, \text{IV}_i^{\text{AES}}, c_{\text{backupM}_i}, \text{mac}_{\text{backupM}_i} | 32 \text{ bits}).$$

Remarque. Lorsque le protocole 12 est exécuté, l'appareil $D_{U_a,\alpha}$ demande au serveur toutes les clés c_{backupM_i} qui ont été chiffrées pendant des sessions précédentes avec le protocole 11. Ensuite, $D_{U_a,\alpha}$ exécute le protocole 12 pour chacune des clés récupérées.

3.5 Authentification entre deux appareils

Dans cette section, nous décrivons le protocole 13 qui sert à deux utilisateurs U_a et U_b pour vérifier leurs appareils respectifs $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ l'un auprès de l'autre. Lorsque $U_a = U_b$, ce protocole sert à authentifier un nouvel appareil de l'utilisateur auprès d'un de ses anciens appareils encore actifs afin de pouvoir récupérer des clés des anciennes sessions lorsque le mécanisme du stockage sécurisé n'est pas mis en place. Lorsque $U_a \neq U_b$, ce protocole permet de confirmer les identités des appareils des deux utilisateurs $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ dans un salon afin d'éviter des attaques *man-in-the-middle* en se transmettant des données par d'autres voies de communication (e.g., oralement).

Protocole 13 (vérification entre deux appareils). Les étapes pour effectuer cela sont les suivantes :

1. si $U_a = U_b$ et que le mécanisme du stockage sécurisé a été mis en place auparavant avec le protocole 8, ce protocole n'est pas exécuté. En effet, pour qu'un nouvel appareil de l'utilisateur U_a s'authentifie dans ce cas auprès de ses anciens appareils, il suffit qu'il entre la passphrase utilisée pour générer la clé secrète *secret storage* et l'utilise dans le protocole 10 pour récupérer les clés privées stockées sur le serveur (y compris les parties privées des clés *cross-signing*).
2. si $U_a \neq U_b$ et que les deux utilisateurs utilisent des clés *cross-signing*, alors l'appareil $D_{U_a,\alpha}$ de l'utilisateur U_a peut directement faire confiance à l'appareil $D_{U_b,\beta}$ de l'utilisateur U_b s'il arrive à récupérer les signatures suivantes du serveur et les vérifier :

- (a) la signature $\sigma_{\text{usk}_{U_a}, \text{msk}_{U_a}}$ de la clé *user-signing* $\text{usk}_{U_a}^{\text{pub}}$ de U_a avec la clé maître msk_{U_a} de U_a

$$\text{verifECSchnorr}(\text{msk}_{U_a}^{\text{pub}}, \text{usk}_{U_a}^{\text{pub}}, \sigma_{\text{usk}_{U_a}, \text{msk}_{U_a}})$$

- (b) la signature $\sigma_{\text{msk}_{U_b}, \text{usk}_{U_a}}$ de la clé maître $\text{msk}_{U_b}^{\text{pub}}$ de U_b avec la clé *user-signing* usk_{U_a} de U_a

$$\text{verifECSchnorr}(\text{usk}_{U_a}^{\text{pub}}, \text{msk}_{U_b}^{\text{pub}}, \sigma_{\text{msk}_{U_b}, \text{usk}_{U_a}})$$

- (c) la signature $\sigma_{\text{ssk}_{U_b}, \text{msk}_{U_b}}$ de la clé *self-signing* $\text{ssk}_{U_b}^{\text{pub}}$ de U_b avec la clé maître msk_{U_b} de U_b

$$\text{verifECSchnorr}(\text{msk}_{U_b}^{\text{pub}}, \text{ssk}_{U_b}^{\text{pub}}, \sigma_{\text{ssk}_{U_b}, \text{msk}_{U_b}})$$

- (d) la signature $\sigma_{D_{U_b,\beta}, \text{ssk}_{U_b}}$ de la clé de signature $k_{f,D_{U_b,\beta}}^{\text{pub}}$ de $D_{U_b,\beta}$ de U_b avec la clé *self-signing* ssk_{U_b} de U_b

$$\text{verifECSchnorr}(\text{ssk}_{U_b}^{\text{pub}}, k_{f,D_{U_b,\beta}}^{\text{pub}}, \sigma_{D_{U_b,\beta}, \text{ssk}_{U_b}}).$$

L'appareil $D_{U_b,\beta}$ de l'utilisateur U_b peut également faire confiance à l'appareil $D_{U_a,\alpha}$ de l'utilisateur U_a s'il arrive à récupérer les signatures suivantes du serveur et les vérifier :

- (a) la signature $\sigma_{\text{usk}_{U_b}, \text{msk}_{U_b}}$ de la clé *user-signing* $\text{usk}_{U_b}^{\text{pub}}$ de U_b avec la clé maître msk_{U_b} de U_b

$$\text{verifECSchnorr}(\text{msk}_{U_b}^{\text{pub}}, \text{usk}_{U_b}^{\text{pub}}, \sigma_{\text{usk}_{U_b}, \text{msk}_{U_b}})$$

- (b) la signature $\sigma_{\text{msk}_{U_a}, \text{usk}_{U_b}}$ de la clé maître $\text{msk}_{U_a}^{\text{pub}}$ de U_a avec la clé *user-signing* usk_{U_b} de U_b

$$\text{verifECSchnorr}(\text{usk}_{U_b}^{\text{pub}}, \text{msk}_{U_a}^{\text{pub}}, \sigma_{\text{msk}_{U_a}, \text{usk}_{U_b}})$$

- (c) la signature $\sigma_{\text{ssk}_{U_a}, \text{msk}_{U_a}}$ de la clé *self-signing* $\text{ssk}_{U_a}^{\text{pub}}$ de U_a avec la clé maître msk_{U_a} de U_a

$$\text{verifECSchnorr}(\text{msk}_{U_a}^{\text{pub}}, \text{ssk}_{U_a}^{\text{pub}}, \sigma_{\text{ssk}_{U_a}, \text{msk}_{U_a}})$$

- (d) la signature $\sigma_{D_{U_a, \alpha}, \text{ssk}_{U_a}}$ de la clé de signature $k_{f, D_{U_a, \alpha}}^{\text{pub}}$ de $D_{U_a, \alpha}$ de U_a avec la clé *self-signing* ssk_{U_a} de U_a

$$\text{verifECSchnorr}(\text{ssk}_{U_a}^{\text{pub}}, k_{f, D_{U_a, \alpha}}^{\text{pub}}, \sigma_{D_{U_a, \alpha}, \text{ssk}_{U_a}}).$$

Si nous ne sommes dans aucun des cas dessus, les étapes suivantes sont alors exécutées :

1. U_a envoie une requête de vérification à U_b via son appareil $D_{U_a, \alpha}$ avec les méthodes qu'il souhaite utiliser (via le serveur).
2. si U_b accepte sur son appareil $D_{U_b, \beta}$, il répond avec les méthodes qu'il supporte parmi celles proposées par $D_{U_a, \alpha}$. U_a ou U_b peut donc ensuite sélectionner une de ces méthodes (ainsi qu'un protocole d'échange de clés et d'une méthode d'authentification).

En pratique, il existe deux méthodes supportées par le client : la vérification SAS (pour *Short Authentication String*) avec des *emojis*, et la vérification par scan de QR codes. La méthode de vérification par scan de QR codes est seulement possible lorsque les deux appareils $D_{U_a, \alpha}$ et $D_{U_b, \beta}$ utilisent des clés *cross-signing* vérifiées.

Si la méthode choisie est la vérification SAS (pour *Short Authentication String*) avec des *emojis*, alors les étapes sont les suivantes :

1. $D_{U_b, \beta}$ génère une paire de clés éphémères Wei25519

$$(k_{\text{eph}, D_{U_b, \beta}}^{\text{pub}}, k_{\text{eph}, D_{U_b, \beta}}^{\text{priv}}) \leftarrow \text{keyGenWei}().$$

2. $D_{U_b, \beta}$ calcule l'empreinte de la clé publique $k_{\text{eph}, D_{U_b, \beta}}^{\text{pub}}$:

$$h \leftarrow \text{SHA-1}(k_{\text{eph}, D_{U_b, \beta}}^{\text{pub}})$$

qu'il envoie à $D_{U_a, \alpha}$.

3. $D_{U_a, \alpha}$ génère une paire de clés éphémères Wei25519 :

$$(k_{\text{eph}, D_{U_a, \alpha}}^{\text{pub}}, k_{\text{eph}, D_{U_a, \alpha}}^{\text{priv}}) \leftarrow \text{keyGenWei}()$$

et envoie la partie publique à $D_{U_b, \beta}$.

4. à réception de la clé publique, $D_{U_b, \beta}$ envoie sa clé publique $k_{\text{eph}, D_{U_b, \beta}}^{\text{pub}}$ à $D_{U_a, \alpha}$.
5. $D_{U_a, \alpha}$ vérifie l'empreinte à partir de la clé reçue :

$$h \stackrel{?}{=} \text{SHA-1}(k_{\text{eph}, D_{U_b, \beta}}^{\text{pub}})$$

et calcule un secret partagé :

$$S_{D_{U_a, \alpha}, D_{U_b, \beta}} \leftarrow \text{ECDH}(k_{\text{eph}, D_{U_a, \alpha}}^{\text{priv}}, k_{\text{eph}, D_{U_b, \beta}}^{\text{pub}}).$$

6. $D_{U_b, \beta}$ calcule le même secret partagé :

$$S_{D_{U_a, \alpha}, D_{U_b, \beta}} \leftarrow \text{ECDH}(k_{\text{eph}, D_{U_b, \beta}}^{\text{priv}}, k_{\text{eph}, D_{U_a, \alpha}}^{\text{pub}}).$$

7. $D_{U_a, \alpha}$ et $D_{U_b, \beta}$ calculent un SAS à partir du secret partagé :

$$\text{SAS} \leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a, \alpha}, D_{U_b, \beta}}, \text{INFO}, 6)$$

où **INFO** est égale à la concaténation des informations suivantes :

- la chaîne de caractères "MATRIX_KEY_VERIFICATION_SAS"
- le nom d'utilisateur **user-name** $_{U_a}$
- l'identifiant de $D_{U_a, \alpha}$
- le nom d'utilisateur **user-name** $_{U_b}$
- l'identifiant de $D_{U_b, \beta}$
- l'identifiant de l'événement en cours s'il existe.

8. les octets du SAS sont ensuite utilisés pour afficher des *emojis* à l'utilisateur en utilisant la table de correspondance entre bits et *emojis* disponible au lien suivant <https://spec.matrix.org/v1.2/client-server-api/#sas-method-emoji>.

9. U_a et U_b communiquent (*e.g.*, oralement) pour déterminer si l’affichage concorde. Si c’est le cas, ils l’indiquent à leurs appareils qui procèdent aux calculs suivants :

- (a) $D_{U_a,\alpha}$ calcule un motif d’authentification pour sa clé de signature d’appareil en utilisant le secret partagé comme suit :

$$\begin{aligned} k_{k_{f,D_{U_a,\alpha}}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"ID}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}}, 32) \\ \text{mac}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_{k_{f,D_{U_a,\alpha}}}^{\text{HMAC}}, k_{f,D_{U_a,\alpha}}^{\text{pub}}) \end{aligned}$$

où $\text{ID}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}}$ est l’identifiant de la clé $k_{f,D_{U_a,\alpha}}^{\text{pub}}$. L’appareil calcule aussi un motif d’authentification pour sa clé maître de *cross-signing* si elle est vérifiée :

$$\begin{aligned} k_{\text{msk}_{U_a}^{\text{pub}}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"ID}_{\text{msk}_{U_a}^{\text{pub}}}, 32) \\ \text{mac}_{\text{msk}_{U_a}^{\text{pub}}} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_{\text{msk}_{U_a}^{\text{pub}}}^{\text{HMAC}}, \text{msk}_{U_a}^{\text{pub}}) \end{aligned}$$

où $\text{ID}_{\text{msk}_{U_a}^{\text{pub}}}$ est l’identifiant de la clé $\text{msk}_{U_a}^{\text{pub}}$. Ensuite, $D_{U_a,\alpha}$ calcule un motif d’authentification pour la liste des identifiants des clés pour lesquelles il a calculé des motifs d’authentification :

$$\begin{aligned} k_{\text{IDS},D_{U_a,\alpha}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"KEY_IDS"}, 32) \\ \text{mac}_{\text{IDS},D_{U_a,\alpha}} &\leftarrow \text{computeMac}^{\text{HMAC-SHA-3}}(k_{\text{IDS},D_{U_a,\alpha}}^{\text{HMAC}}, \text{"ID}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}}, \text{ID}_{\text{msk}_{U_a}^{\text{pub}}}) \end{aligned}$$

où "ID_{msk_{U_a}^{pub}}" est rajoutée dans la chaîne si $\text{mac}_{\text{msk}_{U_a}^{\text{pub}}}$ a été calculé. Dans les trois motifs, le champ INFO est égale à la concaténation des informations suivantes :

- la chaîne de caractères MATRIX_KEY_VERIFICATION_MAC
- le nom d’utilisateur de user-name_{U_a}
- l’identifiant de $D_{U_a,\alpha}$
- le nom d’utilisateur de user-name_{U_b}
- l’identifiant de $D_{U_b,\beta}$
- l’identifiant de l’événement en cours s’il existe.

Enfin, $D_{U_a,\alpha}$ envoie les trois motifs $\text{mac}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}}$, $\text{mac}_{\text{msk}_{U_a}^{\text{pub}}}$ (s’il existe), et $\text{mac}_{\text{IDS},D_{U_a,\alpha}}$ à $D_{U_b,\beta}$.

- (b) De la même manière, $D_{U_b,\beta}$ calcule ses trois motifs d’authentification $\text{mac}_{k_{f,D_{U_b,\beta}}}^{\text{pub}}$, $\text{mac}_{\text{msk}_{U_b}^{\text{pub}}}$ (s’il existe), $\text{mac}_{\text{IDS},D_{U_b,\beta}}$ et les envoie à $D_{U_a,\alpha}$.
- (c) $D_{U_a,\alpha}$ reçoit de $D_{U_b,\beta}$ les motifs d’authentification $\text{mac}_{k_{f,D_{U_b,\beta}}}^{\text{pub}}$, $\text{mac}_{\text{msk}_{U_b}^{\text{pub}}}$ (s’il existe), $\text{mac}_{\text{IDS},D_{U_b,\beta}}$ et les vérifie :

$$\begin{aligned} k_{k_{f,D_{U_b,\beta}}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"ID}_{k_{f,D_{U_b,\beta}}}^{\text{pub}}, 32) \\ k_{\text{msk}_{U_b}^{\text{pub}}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"ID}_{\text{msk}_{U_b}^{\text{pub}}}, 32) \\ k_{\text{IDS},D_{U_b,\beta}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"KEY_IDS"}, 32) \\ \text{verifMac}^{\text{HMAC-SHA-3}}(k_{k_{f,D_{U_b,\beta}}}^{\text{HMAC}}, k_{f,D_{U_b,\beta}}^{\text{pub}}, \text{mac}_{k_{f,D_{U_b,\beta}}}^{\text{pub}}) \\ \text{verifMac}^{\text{HMAC-SHA-3}}(k_{\text{msk}_{U_b}^{\text{pub}}}^{\text{HMAC}}, \text{msk}_{U_b}^{\text{pub}}, \text{mac}_{\text{msk}_{U_b}^{\text{pub}}}) \\ \text{verifMac}^{\text{HMAC-SHA-3}}(k_{\text{IDS},D_{U_b,\beta}}^{\text{HMAC}}, \text{"ID}_{k_{f,D_{U_b,\beta}}}^{\text{pub}}, \text{ID}_{\text{msk}_{U_b}^{\text{pub}}}, \text{mac}_{\text{IDS},D_{U_b,\beta}}) \end{aligned}$$

et valide l’identité de $D_{U_b,\beta}$ si toutes les vérifications réussissent. Si c’est le cas et que les deux appareils utilisent des clés *cross-signing*, alors $D_{U_a,\alpha}$ signe la partie publique de la clé $\text{msk}_{U_b}^{\text{pub}}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

$$\sigma_{\text{msk}_{U_b}, \text{usk}_{U_a}} \leftarrow \text{signECSchnorr}(\text{usk}_{U_a}^{\text{priv}}, \text{msk}_{U_b}^{\text{pub}}).$$

- (d) De la même manière, $D_{U_b,\beta}$ reçoit de $D_{U_a,\alpha}$ les motifs d’authentification $\text{mac}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}}$, $\text{mac}_{\text{msk}_{U_a}^{\text{pub}}}$ (s’il existe), $\text{mac}_{\text{IDS},D_{U_a,\alpha}}$ et les vérifie :

$$\begin{aligned} k_{k_{f,D_{U_a,\alpha}}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"ID}_{k_{f,D_{U_a,\alpha}}}^{\text{pub}}, 32) \\ k_{\text{msk}_{U_a}^{\text{pub}}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"ID}_{\text{msk}_{U_a}^{\text{pub}}}, 32) \\ k_{\text{IDS},D_{U_a,\alpha}}^{\text{HMAC}} &\leftarrow \text{HKDF}^{\text{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha}, D_{U_b,\beta}}, \text{"INFO"+"KEY_IDS"}, 32) \end{aligned}$$

$$\begin{aligned} & \text{verifMac}^{\text{HMAC-SHA-3}}(k_{f,D_{U_a,\alpha}}^{\text{pub}}, k_{f,D_{U_a,\alpha}}^{\text{pub}}, \text{mac}_{k_{f,D_{U_a,\alpha}}^{\text{pub}}}) \\ & \text{verifMac}^{\text{HMAC-SHA-3}}(k_{\text{msk}_{U_a}^{\text{pub}}}^{\text{HMAC}}, \text{msk}_{U_a}^{\text{pub}}, \text{mac}_{\text{msk}_{U_a}^{\text{pub}}}) \\ & \text{verifMac}^{\text{HMAC-SHA-3}}(k_{\text{IDS},D_{U_a,\alpha}}^{\text{HMAC}}, \text{"ID}_{\text{msk}_{U_a}^{\text{pub}}}\text{"}, \text{mac}_{\text{IDS},D_{U_a,\alpha}}) \end{aligned}$$

et valide l'identité de $D_{U_a,\alpha}$ si toutes les vérifications réussissent. Si c'est le cas et que les deux appareils utilisent des clés *cross-signing*, alors $D_{U_a,\alpha}$ signe la partie publique de la clé $\text{msk}_{U_a}^{\text{pub}}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

$$\sigma_{\text{msk}_{U_a}, \text{usk}_{U_b}} \leftarrow \text{signECSchnorr}(\text{usk}_{U_b}^{\text{priv}}, \text{msk}_{U_a}^{\text{pub}}).$$

Si la méthode choisie est l'utilisation de QR codes, alors les étapes sont les suivantes :

1. les appareils $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ commencent par construire leurs QR codes respectifs. Le QR code encode les informations suivantes :
 - (a) la chaîne de caractères "MATRIX",
 - (b) un octet représente le type de la vérification parmi trois possibilités :
 - \x00 : cet octet représente le cas où $U_a \neq U_b$, c'est-à-dire que la vérification s'effectue entre deux appareils de deux utilisateurs différents.
 - \x01 : cet octet représente le cas où $U_a = U_b$ et est encodé dans le QR code de l'ancien appareil de l'utilisateur qui utilise une clé maître *cross-signing* vérifiée, et qui doit valider l'identité d'un nouvel appareil.
 - \x02 : cet octet représente le cas où $U_a = U_b$ et est encodé dans le QR code du nouvel appareil de l'utilisateur qui doit vérifier la clé maître *cross-signing* auprès de l'ancien appareil vérifié.
- Donc, lorsque $U_a \neq U_b$, les deux appareils $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ encodent dans leur QR code l'octet \x00. Et lorsque $U_a = U_b$, $D_{U_a,\alpha}$ encode l'octet \x01 alors que $D_{U_b,\beta}$ encode l'octet \x02.
- (c) - lorsque $U_a \neq U_b$, les deux appareils $D_{U_a,\alpha}$ et $D_{U_b,\beta}$ encodent leurs clés maîtres de *cross-signing* publiques correspondants $\text{msk}_{U_a}^{\text{pub}}$ et $\text{msk}_{U_b}^{\text{pub}}$.
 - lorsque $U_a = U_b$, l'appareil $D_{U_a,\alpha}$ encode sa clé maître de *cross-signing* publique $\text{msk}_{U_a}^{\text{pub}}$, alors que $D_{U_b,\beta}$ sa clé de signature publique $k_{f,D_{U_b,\beta}}^{\text{pub}}$.
- (d) - lorsque $U_a \neq U_b$, l'appareil $D_{U_a,\alpha}$ (respectivement $D_{U_b,\beta}$) encode la clé maître potentielle $\text{msk}_{U_b}^{\text{pub}'}$ de l'appareil $D_{U_b,\beta}$ (respectivement $\text{msk}_{U_a}^{\text{pub}'}$ de $D_{U_a,\alpha}$) qu'il a récupérée du serveur auparavant.
 - lorsque $U_a = U_b$, l'appareil $D_{U_a,\alpha}$ (respectivement $D_{U_b,\beta}$) encode la clé de signature potentielle $k_{f,D_{U_b,\beta}}^{\text{pub}'}$ de l'appareil $D_{U_b,\beta}$ (respectivement la clé maître potentielle $\text{msk}_{U_a}^{\text{pub}'}$ de l'appareil $D_{U_a,\alpha}$) qu'il a récupérée du serveur auparavant.
- (e) Chaque appareil encode dans son QR code un secret (secret_a et secret_b) de 8 octets généré par un appel à DRBG.
2. $D_{U_a,\alpha}$ scanne le QR code affiché sur $D_{U_b,\beta}$ ($D_{U_b,\beta}$ peut à la place scanner le QR code affiché sur $D_{U_a,\alpha}$) et vérifie que
 - $\text{msk}_{U_b}^{\text{pub}'} = \text{msk}_{U_b}^{\text{pub}}$ et que $\text{msk}_{U_a}^{\text{pub}} = \text{msk}_{U_a}^{\text{pub}'}$ pour le cas $U_a \neq U_b$.
 - $\text{msk}_{U_a}^{\text{pub}} = \text{msk}_{U_a}^{\text{pub}'}$ et que $k_{f,D_{U_b,\beta}}^{\text{pub}'} = k_{f,D_{U_b,\beta}}^{\text{pub}}$ pour le cas $U_a = U_b$.
3. si la vérification fonctionne, $D_{U_a,\alpha}$ envoie le secret secret_b scanné à $D_{U_b,\beta}$, et signe la partie publique de la clé $\text{msk}_{U_b}^{\text{pub}}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

$$\sigma_{\text{msk}_{U_b}, \text{usk}_{U_a}} \leftarrow \text{signECSchnorr}(\text{usk}_{U_a}^{\text{priv}}, \text{msk}_{U_b}^{\text{pub}}).$$

4. $D_{U_b,\beta}$ vérifie que le secret correspond à celui qu'il a généré et confirme que $D_{U_a,\alpha}$ a bien fait la vérification des clés. Ensuite, $D_{U_b,\beta}$ signe la partie publique de la clé $\text{msk}_{U_a}^{\text{pub}}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

$$\sigma_{\text{msk}_{U_a}, \text{usk}_{U_b}} \leftarrow \text{signECSchnorr}(\text{usk}_{U_b}^{\text{priv}}, \text{msk}_{U_a}^{\text{pub}}).$$

4 Mécanismes cryptographiques

Nous listons ici les primitives cryptographiques utilisées dans les protocoles décrits dans la Section 3. Nous les classons par catégorie et pour chacune d'entre elles, nous référençons sa spécification lorsqu'elle existe ou nous décrivons le mécanisme le cas échéant.

4.1 Dérivation de clés

4.1.1 Fonction de dérivation de clé PBKDF2

La fonction de dérivation de clés PBKDF2 (*Password-Based Key Derivation Function 2*) est décrite dans les spécifications RFC8018 [?, Section 5.2]. Elle est utilisée avec la fonction pseudo-aléatoire HMAC-SHA-3 pour des blocs de sortie de 32 octets (*i.e.*, avec la fonction de hachage SHA-3-256) dans notre application et prend en entrée quatre arguments :

- un mot de passe sous forme d'une chaîne d'octets,
- un sel pour la fonction cryptographique sous forme d'une chaîne d'octets,
- un nombre d'itérations (entier positif),
- la taille `dkLen` de la clé souhaitée en octets,

et retourne une clé de `dkLen` octets. La primitive HMAC et la fonction de hachage SHA-3 associée sont décrites dans les sections 4.2.1 et 4.3.1.

4.1.2 Fonction de dérivation de clé HKDF

La primitive HKDF est une fonction de dérivation de clé basée sur une primitive HMAC. Elle est décrite dans les spécifications RFC5869 [?]. Dans ce projet, elle est utilisée avec la primitive HMAC-SHA-3 pour des blocs de sortie de 32 octets (*i.e.*, avec la fonction de hachage SHA-3-256). Elle prend en entrée quatre arguments :

- un sel optionnel (`salt` dans les spécifications),
- un secret (IKM dans les spécifications),
- une chaîne d'information qui peut être vide (`info` dans les spécifications),
- la taille `L` de la clé souhaitée en octets,

et retourne une clé de `L` octets. Cette primitive unique correspond à l'exécution successive des primitives HKDF-Extract puis HKDF-Expand dans les spécifications.

Dans ce document, quand le premier argument (le sel) est défini à 0, la valeur 0 est encodée sur un octet. Quand ce premier argument est vide NULL, le sel est absent.

4.2 Codes d'authentification de messages

4.2.1 Code d'authentification de messages HMAC

Le code d'authentification de messages HMAC est défini dans la documentation RFC2104 [?]. Il s'appuie sur une fonction de hachage H qui, dans le cas de cette application, correspond soit à la fonction de hachage SHA-1 ($x = 1$) soit à la fonction de hachage SHA-3 ($x = 3$), toutes deux décrites dans la section 4.3.

Le code d'authentification de messages HMAC est décliné en deux primitives dans ces spécifications :

- `computeMacHMAC-SHA-x` : prend en argument une clé K et un message `text` et retourne un `mac` de taille égale à la taille de la sortie de la fonction de hachage choisie,
- `verifMacHMAC-SHA-x` : prend en argument une clé K , un message `text` et un `mac` et retourne un booléen pour indiquer le résultat de la vérification.

La première primitive correspond au calcul du motif d'authentification décrit dans la RFC. La seconde primitive est équivalente à la première, suivie par un test comparant le `mac` présenté en entrée et celui recalculé par un appel à la première fonction.

Pour des raisons d'efficacité, les motifs d'authentification sont parfois tronqués. `computeMacHMAC-SHA-x(·)|t bits` signifie que le motif est tronqué à t bits de sortie. `verifMacHMAC-SHA-x|t bits` signifie que le motif recalculé est tronqué à t bits avant d'être vérifié.

4.3 Fonctions de hachage

4.3.1 Fonction de hachage SHA-3

Les spécifications de la fonction de hachage SHA-3 sont disponibles dans le standard FIPS202 [?]. La primitive SHA-3 prend en entrée un message et retourne une empreinte de 256 bits. Elle correspond à la primitive SHA-3-256 dans les spécifications.

4.3.2 Fonction de hachage SHA-1

Les spécifications de la fonction de hachage SHA-1 sont disponibles dans le standard FIPS180-4 [?]. La primitive SHA-1 prend en entrée un message de taille ℓ bits, avec $0 \leq \ell \leq 2^{64}$, et retourne une empreinte de 160 bits.

4.3.3 Fonction de hachage de mot de passe `bcrypt`

La fonction de hachage `bcrypt` (implémentée au niveau du serveur) est principalement dédiée au hachage de mots de passe. Elle prend en entrée un mot de passe à hacher et un sel de 128 bits généré aléatoirement (la génération du sel est effectuée en interne par le module python `bcrypt` qui utilise la méthode `urandom()` du système <https://docs.python.org/3/library/os.html>). La fonction prend également en entrée un coût numérique n pour réaliser 2^n tours pour la fonction de hachage ($n = 12$ dans notre contexte). Ses spécifications sont données dans l'article de référence [?].

4.4 Chiffrement symétrique (authentifié)

Le chiffrement par bloc **AES** (*Advanced Encryption Standard*) est standardisé depuis 2001 par le NIST (standard FIPS197 [?]). Dans le cadre de ce projet, nous utilisons exclusivement la version AES-256 (clés de 256 bits, bloc d'entrée-sortie de 128 bits, nombre de tours : 14). Il est appelé avec le mode d'opération CBC, le mode d'opération CTR et dans le chiffrement authentifié AES-GCM.

4.4.1 Chiffrement AES-CBC

Le mode de chiffrement CBC (*Cipher Block Chaining*) est spécifié dans le standard SP800-38a [?, Section 6.2]. Il est utilisé dans ce projet avec la primitive de chiffrement par bloc **AES** (spécifiée ci-dessus). La fonction `encAES-CBC` prend en arguments une clé de 256 bits, un vecteur d'initialisation de 128 bits et un message et retourne un chiffré. La fonction `decAES-CBC` prend en arguments une clé de 256 bits, un vecteur d'initialisation de 128 bits et un chiffré et retourne le message correspondant. Nous choisissons d'utiliser le *padding* PKCS#7 défini dans les spécifications RFC5652 [?, Section 6.3]. Il est appliqué à chaque message pour le chiffrement et vérifié au déchiffrement pour l'envoi du *feedback*.

4.4.2 Chiffrement AES-CTR

Le mode de chiffrement CTR (*Counter*) est spécifié dans le standard SP800-38a [?, Section 6.5]. Il est utilisé dans ce projet avec la primitive de chiffrement par bloc **AES** (spécifiée ci-dessus). La fonction `encAES-CTR` prend en arguments une clé de 256 bits, un vecteur d'initialisation de 128 bits et un message et retourne un chiffré. La fonction `decAES-CTR` prend en arguments une clé de 256 bits, un vecteur d'initialisation de 128 bits et un chiffré et retourne le message correspondant.

4.4.3 Chiffrement authentifié AES-GCM

Le chiffrement authentifié AES-GCM basé sur le chiffrement par bloc **AES** est décrit dans le standard SP800-38D [?, Section 7]. La primitive de chiffrement authentifié (`encAEAES-GCM`) prend en entrée une clé, un vecteur d'initialisation, un message et une donnée associée, et retourne un chiffré et un motif d'authentification. Dans ce projet, nous omettons le champ dédié à la donnée associée qui est considéré comme la chaîne vide. De manière similaire, la primitive de déchiffrement authentifié (`decAEAES-GCM`) prend en entrée une clé, un vecteur d'initialisation, un chiffré et un motif d'authentification. Elle retourne le message déchiffré ou une erreur si le motif d'authentification n'a pas été correctement vérifié.

4.5 Cryptographie asymétrique

4.5.1 Courbe elliptique Wei25519

Une courbe elliptique sur le corps à p éléments, avec p un nombre premier, est l'ensemble des points (x, y) qui vérifient l'équation de Weierstrass

$$y^2 = x^3 + ax + b$$

où a et b sont deux paramètres de la courbe tels que $4a^3 + 27b^2$ soit non nul, ensemble auquel un point à l'infini usuellement noté O est ajouté. La courbe Wei25519 est la courbe définie sur le corps à $p := 2^{255} - 19$ éléments avec les paramètres suivants :

$$\begin{aligned} a &:= 19298681539552699237261830834781317975544997444273427339909597334573241639236, \\ b &:= 55751746669818908907645289078257140818241103727901012315294400837956729358436. \end{aligned}$$

L'ordre de cette courbe (*i.e.* son nombre de points) est l'entier

$$n = h \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$$

où $h = 8$ est le *cofacteur* de la courbe.

Nous utilisons le point de base $G = (x_G, y_G)$ défini par

$$x_G := 1,$$

$$y_G := 18909347336615151728670064226692766338825600391524661473633862702332996581084.$$

Ce point est un générateur de la courbe selon la loi d'addition définie dans l'annexe A.1.1 du standard SP800-186 [?].

4.5.2 Génération de clé Wei25519

La primitive `keyGenWei` est une fonction qui ne prend aucune entrée et qui retourne un couple “clé privée-clé publique” défini de la manière suivante :

1. d'abord, un ensemble de 128 bits est généré aléatoirement.
2. puis ces bits sont interprétés comme l'écriture binaire d'un entier d .
3. si d vaut zéro ou si d est supérieur ou égal à n (l'ordre de la courbe), alors l'algorithme reprend du début.
4. la primitive calcule le point Q défini comme la multiplication scalaire du point G par l'entier d , où l'opération de multiplication scalaire est définie dans le standard SP800-186 [?, Annexe A.1.1].
5. la fonction retourne
 - d en tant que clé privée Wei25519,
 - Q en tant que clé publique Wei25519.

La clé publique Q est stockée sous la forme de deux tableaux d'octets concaténés représentant respectivement les coordonnées $x_Q \in \{0, \dots, n-1\}$ (premier tableau) et $y_Q \in \{0, \dots, n-1\}$ (deuxième tableau). La clé privée d est également stockée sous la forme d'un tableau d'octets. Pour les tableaux représentant d , x_Q et y_Q , le premier octet est l'octet de poids fort (*big-endian*).

La primitive `keyGenWeiPub`, quant à elle, prend en entrée une clé privée Wei25519 correspondant à un entier d inférieur à n et retourne la clé publique Q définie de la même manière que dans `keyGenWei` : $Q := d \cdot G$.

4.5.3 Échange de clés EC Diffie-Hellman

Le mécanisme d'échange de clés ECDH que nous utilisons est ECC CDH (*Elliptic Curve Cryptography Cofactor Diffie-Hellman*) décrit dans la section 5.7.1.2 du standard SP800-56a [?]. La primitive prend en arguments une clé privée ainsi qu'une clé publique, et retourne un secret partagé entre les propriétaires des clés. Dans ce projet, la courbe utilisée est la courbe Wei25519.

4.5.4 Signature EC Schnorr

Les spécifications de la signature EC Schnorr sont disponibles dans la section 4.2.3 du guide technique [?] du BSI. La fonction `signECSchnorr` prend en entrée une clé privée ainsi que le message à signer, et retourne la signature. La fonction `verifECSchnorr` quant à elle prend en entrée la clé publique de l'émetteur, le message à vérifier et la signature, et retourne un booléen indiquant si la signature est valide. Dans ce projet, la courbe utilisée est la courbe Wei25519 avec les paramètres et le point de base définis en Section 4.5.1. Pour la fonction de hachage, nous utilisons SHA-3-256 (voir Section 4.3.1).

4.5.5 Génération de clé RSA

La primitive `keyGenRSA` de génération d'une paire de clés de signature RSA utilisée dans le protocole 1 utilise la fonctionnalité décrite dans l'annexe B.2 du guide technique [?] de l'ANSSI. Pour le choix de l'exposant public e , nous le fixons à $e = 2^{16} + 1$. Nous fixons également le petit entier naturel B en entrée de la procédure à $B = 50$. Le test de primalité utilisé est celui de Miller-Rabin avec 6 itérations décrit dans l'annexe C.3.2 du standard FIPS186-4 [?].

4.5.6 Signature RSA

Le mécanisme de signature RSA est décrit dans les spécifications RFC8017 [?]. Il offre deux fonctionnalités :

- la signature d'un message qui prend en entrée une clé privée RSA, un message et renvoie une signature,
- la vérification d'une signature RSA qui prend en entrée la clé publique de l'émetteur, le message à vérifier et la signature. Il renvoie un booléen pour indiquer la validité de la signature.

Les mécanismes `signRSAtoken` et `verifRSAtoken` utilisés dans les protocoles 1 et 2 correspondent aux fonctionnalités RSASP1 et RSAVP1 respectivement décrites dans les spécifications [?]. Le mécanisme `verifRSAserver` utilisé dans les protocoles 1 et 2 correspond à la fonctionnalité RSASSA-PSS-VERIFY décrite dans les spécifications [?] (la fonctionnalité RSASSA-PSS-SIGN implémentée côté serveur est également décrite dans [?]).

4.6 Génération aléatoire

4.6.1 Génération des graines

Les graines utilisées pour instancier les générateurs de nombres pseudo-aléatoires décrits ci-après sont générées à partir de l'entropie disponible en local en utilisant la classe Java `SecureRandom`. Nous utilisons le constructeur `SecureRandom()` sans arguments qui construit un générateur de nombres aléatoires implémentant l'algorithme par défaut. L'algorithme par défaut est choisi en parcourant la liste des fournisseurs de sécurité enregistrés sur la plateforme, en commençant par le fournisseur préféré. La méthode `nextBytes(byte[] bytes)` est ensuite utilisée pour générer la suite de nombres aléatoires nécessaires.

4.6.2 Générateur linéaire congruentiel (LCG)

La primitive de signature `signECSchnorr` utilise le générateur linéaire congruentiel suivant pour la génération des nonces aléatoires. Une graine S_0 de 64 bits est initialisée par appel au générateur Java (cf. Section 4.6.1). Cette graine définit l'état initial du générateur LCG.

A chaque appel, le générateur utilise l'état courant S_i et

1. effectue le calcul suivant :

$$\begin{aligned} B_i &= (a \cdot S_i + c) \bmod 2^{64} \\ S_{i+1} &= (a \cdot B_i + c) \bmod 2^{64} \end{aligned}$$

où $a = 6364136223846793005$ et $c = 1442695040888963407$.

2. stocke S_{i+1} comme le nouvel état du générateur,
3. retourne B_i .

Dans le contexte d'une signature `signECSchnorr`, le générateur LCG est donc appelé quatre fois successivement afin d'obtenir un nonce de 256 bits.

4.6.3 Générateur cryptographique

Pour le générateur d'aléa cryptographique DRBG, nous utilisons le mécanisme Dual EC DRBG décrit dans le standard NIST SP800-90A [?, Section 10.3.1] avec les paramètres suivants :

- taille de la graine (*seedlen*) : 256 bits,
- sécurité (*security_strength*) : 128 bits,
- taille des blocs de sortie (*outlen*) : 232 bits.

Pour la fonction de hachage, nous utilisons SHA-3-256 (voir Section 4.3.1). Pour la courbe elliptique, nous utilisons Wei25519 (voir Section 4.5.1). Les points $P = (x_P, y_P)$ et $Q = (x_Q, y_Q)$ utilisés sont les suivants :

$$\begin{aligned} x_P &:= 29574854205587028187789910861250489067351937497348330655840054524429782953125 \\ y_P &:= 34787411194293159582962146500558888268755365180353843541161653519432463373063 \\ x_Q &:= 57863406254506098596402688243367836474675441954671054629065899898103707227695 \\ y_Q &:= 20008002610233705536182467594645266681062800035717294684676566971264333897253 \end{aligned}$$

Le générateur est initialisé par une graine de 256 bits générée par un appel au générateur Java (cf. Section 4.6.1). L'état du générateur est rafraîchi tous les 2^{32} blocs en utilisant la procédure de *reseeding* décrite dans le standard avec une chaîne aléatoire de 256 bits (*entropy_input*) générée par un appel au générateur Java.