Projet CRY.ME

Description de l'architecture du logiciel, détail des échanges protocolaires, 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

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.

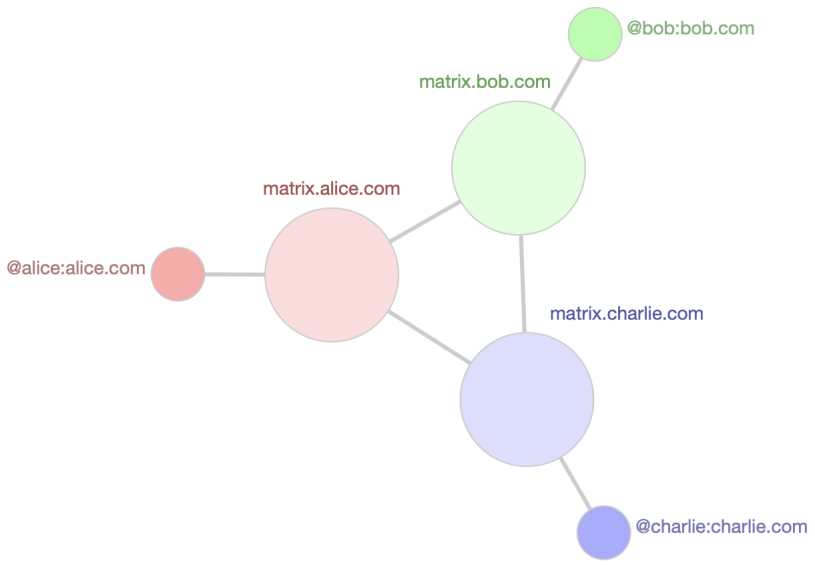


FIGURE 1 – Principe de fédération sur le réseau Matrix.

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'un *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

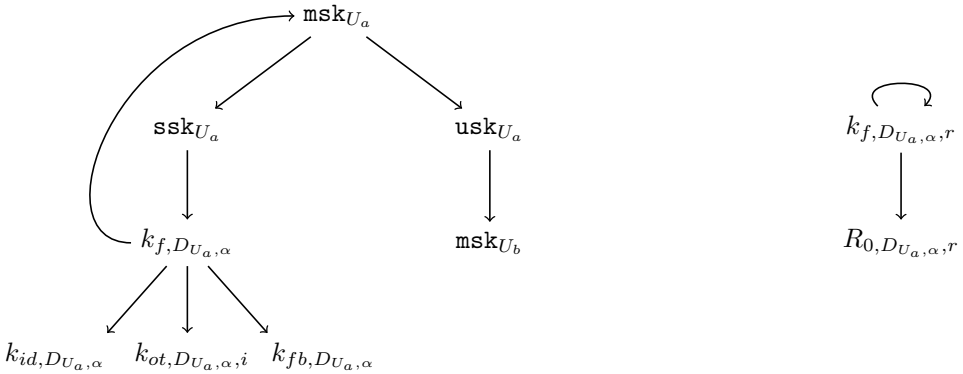
Le tableau ci-dessous décrit l'ensemble des clés intervenant dans les différents protocoles de la section 3. Pour les colonnes "Généré sur" et "Stocké sur", les étiquettes « C », « T » et « S » désignent respectivement le client, le *token* d'authentification (la clé **yubikey**) ainsi que le seveur. « 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	$k^{\texttt{pub/priv}}_{\texttt{RSA,server}}$	S	`pub` : C et S `priv` : S	1, 2	durée de vie illimitée
Clé du token d'authentification	$k^{\texttt{pub/priv}}_{\texttt{RSA},U_a}$	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	$k^{\texttt{pub/priv}}_{f,D_{U_a,\alpha}}$	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	$k^{\texttt{pub/priv}}_{id,D_{U_a,\alpha}}$	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 *one-time*	$k^{\texttt{pub/priv}}_{ot,D_{U_a,\alpha},i}$	C	`pub` : C et S `priv` : C	1, 2, 3.A, 3.B	durée de vie de $D_{U_a,\alpha}$
Clé *fallback*	$k^{\texttt{pub/priv}}_{fb,D_{U_a,\alpha}}$	C	`pub` : C et S `priv` : C	1, 2, 3.A	durée de vie de $D_{U_a,\alpha}$
Clés *cross-signing*					
Clé maître	$\texttt{msk}^{\texttt{pub/priv}}_{U_a}$	C	`pub` : C et S `priv` : C et S^*	1, 8, 13	durée de vie de U_a

Clé *user-signing*	$\texttt{usk}_{U_a}^{\texttt{pub/priv}}$	C	`pub` : C et S `priv` : C et S*	1, 8	durée de vie de U_a
Clé *self-signing*	$\texttt{ssk}_{U_a}^{\texttt{pub/priv}}$	C	`pub` : C et S `priv` : C et S*	1, 8, 10	durée de vie de U_a
Clés spécifiques au protocole `Olm`					
Clé *ratchet*	$T_{i,D_{U_a,\alpha},D_{U_b,\beta}}^{\texttt{pub/priv}}$	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	$k_{f,D_{U_a,\alpha},r}^{\texttt{pub/priv}}$	C	C	5, 6.A, 6.B	durée de vie du salon r
Clé *rachet* 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	$k^{\texttt{AES}}$	C	C	7.A, 7.B	durée de vie de la pièce chiffrée
Clé d'intégrité d'une pièce jointe	$k^{\texttt{HMAC}}$	C	C	7.A, 7.B	durée de vie de la pièce chiffrée
Clés spécifiques au stockage sécurisé					
Clé *secret storage*	$k_{\texttt{sec},U_a}$	C	C	8, 9, 10	durée de vie de U_a
Clé *recovery*	$k_{\texttt{rec},U_a}^{\texttt{pub/priv}}$	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	$k_i^{\texttt{AES}}$	C	C	9, 10, 11, 12	supprimée à la fin du protocole

Clé d'intégrité d'une clé privée	$k_i^{\texttt{mac}}$	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	$k^{\texttt{pub/priv}}_{\texttt{eph},D_{U_a,\alpha}}$	C	C	13	supprimée à la fin du protocole
Clé d'intégrité de la clé publique de signature	$k^{\texttt{HMAC}}_{k^{\texttt{pub}}_{f,D_{U_a,\alpha}}}$	C	C	13	supprimée à la fin du protocole
Clé d'intégrité de la clé maître de *cross-signing*	$k^{\texttt{HMAC}}_{\texttt{msk}^{\texttt{pub}}_{U_a}}$	C	C	13	supprimée à la fin du protocole
Clé d'intégrité des identifiants des clés	$k^{\texttt{HMAC}}_{\texttt{IDS},D_{U_a,\alpha}}$	C	C	13	supprimée à la fin du protocole

Les signatures entres 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 $k_1^{\texttt{priv}}$ signe la clé publique $k_2^{\texttt{pub}}$.



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 $\texttt{msk}_{U_a}$.

— La clé maître (cross-signing) $\texttt{msk}_{U_a}$ d'un utilisateur signe les clés *user-signing* $\texttt{usk}_{U_a}$ et *self-signing* $\texttt{ssk}_{U_a}$ de cet utilisateur.

— La clé *self-signing* $\texttt{ssk}_{U_a}$ d'un utilisateur signe les clés de signature des appareils de cet utilisateur $\{k_{f,D_{U_a,\alpha}}\}_\alpha$.

— La clé *user-signing* $\texttt{usk}_{U_a}$ d'un utilisateur signe les clés maître des autres utilisateurs $\{\texttt{msk}_{U_b}\}_{U_b}$.

— 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 *rachet* 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. Nous expliquons également les modifications réalisées par rapport à la spécification Matrix originale et les vulnérabilités introduites. Ces dernières sont détaillées dans la Section 5. Les spécifications des primitives cryptographiques utilisées sont référencées dans la Section 4. Nous utilisons la couleur orange dans les protocoles pour mettre en relief les composantes qui contiennent des vulnérabilités.

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* $\texttt{token}_{U_a}$ 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` $k^{\texttt{priv}}_{\texttt{RSA,server}}$, 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 $k^{\texttt{priv}}_{\texttt{RSA,server}}$.
4. le client vérifie la signature σ_{url} avec la clé publique $k^{\texttt{pub}}_{\texttt{RSA,server}}$ codée *en dur* dans l'application

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

   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 $\texttt{PIN}_{U_a}$ du *token* saisi par l'utilisateur au *token* $\texttt{token}_{U_a}$ 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* $\texttt{token}_{U_a}$ via la connexion NFC pour générer en interne la paire de clés :
   $$(k^{\texttt{pub}}_{\texttt{RSA},U_a}, k^{\texttt{priv}}_{\texttt{RSA},U_a}) \ .$$
   Ensuite, à la demande du client, le *token* renvoie le certificat correspondant $\texttt{cert}_{\texttt{token}_{U_a}}$ via la connexion NFC.

   Cas 2. Dans le second cas, l'appareil $D_{U_a,\alpha}$ génère une paire de clés RSA :
   $$(k^{\texttt{pub}}_{\texttt{RSA},U_a}, k^{\texttt{priv}}_{\texttt{RSA},U_a}) \leftarrow \texttt{keyGenRSA}(2048)$$
   et un certificat $\texttt{cert}_{\texttt{token}_{U_a}}$ et les charge dans le *token* $\texttt{token}_{U_a}$ via la connexion NFC.
7. le client envoie le nom d'utilisateur $\texttt{ID}_{U_a}$, le mot de passe $\texttt{pwd}_{U_a}$ 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
   $$h_{\texttt{pwd}_{U_a}} \leftarrow \texttt{bcrypt}(\texttt{pwd}_{U_a})$$
   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,
    $$(\texttt{msk}^{\texttt{pub}}_{U_a}, \texttt{msk}^{\texttt{priv}}_{U_a}) \leftarrow \texttt{keyGenWei}(),$$

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

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

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

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

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

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

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

et envoie les signatures au serveur.

12. 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.
13. l'appareil $D_{U_a,\alpha}$ génère ses clés de signatures :

$$(k^{\mathtt{pub}}_{f,D_{U_a,\alpha}}, k^{\mathtt{priv}}_{f,D_{U_a,\alpha}}) \leftarrow \mathtt{keyGenWei}(),$$

et ses clés d'identité :

$$(k^{\mathtt{pub}}_{id,D_{U_a,\alpha}}, k^{\mathtt{priv}}_{id,D_{U_a,\alpha}}) \leftarrow \mathtt{keyGenWei}()$$

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

$$\sigma_{id,D_{U_a,\alpha}} \leftarrow \mathtt{signECSchnorr}(k^{\mathtt{priv}}_{f,D_{U_a,\alpha}}, k^{\mathtt{pub}}_{id,D_{U_a,\alpha}}).$$

L'appareil génère également une liste de cinquante clés *one-time* et signe les parties publiques avec sa clé de signature [4] :

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

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

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

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 $D_{U_a,\alpha}$.

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

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

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

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

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

Application initiale. L'application initiale ne prenait pas en charge l'authentification à l'aide d'un *token*. Nous avons donc ajouté cette fonctionnalité et l'avons rendue obligatoire. Nous avons également ajouté une étape d'authentification du serveur qui permet de vérifier si le serveur a été accrédité par une autorité. Ce protocole est donc entièrement redéfini pour ce projet.

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.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes dans ce protocole :

- **Vulnérabilité 1** : non-conformité entre la politique de mot de passe décrite dans les spécifications et la politique de mot de passe implémentée au niveau du serveur.
- **Vulnérabilité 4** : utilisation de la primitive `SHA-1` obsolète pour la signature σ_{url}.
- **Vulnérabilité 14** : utilisation d'une clé RSA de 512 bits pour la clé privée commune aux serveurs accrédités.
- **Vulnérabilité 15** : génération faible de clés RSA.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (`keyGenWei`).
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature `signECSchnorr`.

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 :

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` $k^{\text{priv}}_{\text{RSA,server}}$, qui est obtenue par tous les serveurs accrédités depuis une autorité commune.
3. le serveur génère un challenge aléatoire de 32 bits (qui sera signé par $\texttt{token}_{U_a}$ dans la suite du protocole) :

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

   Ce challenge est généré par un appel à `dev/urandom`.
4. le serveur retourne son url publique dans un message m_{url} avec sa signature σ_{url} obtenue avec la clé privée $k^{\text{priv}}_{\text{RSA,server}}$. Il renvoie également le challenge généré.
5. le client vérifie la signature σ_{url} avec la clé publique $k^{\text{pub}}_{\text{RSA,server}}$ codée *en dur* dans l'application

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

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

   $$m \leftarrow \texttt{challenge} \,\|\, \texttt{timestamp}$$

   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* $\texttt{token}_{U_a}$.
8. le *token* $\texttt{token}_{U_a}$ signe le challenge modifié m avec sa clé RSA (interne ou générée par l'application) :

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

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

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

    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.

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

$$(k^{\texttt{pub}}_{f,D_{U_a,\alpha}}, k^{\texttt{priv}}_{f,D_{U_a,\alpha}}) \leftarrow \texttt{keyGenWei}(),$$

et ses clés d'identité :

$$(k^{\texttt{pub}}_{id,D_{U_a,\alpha}}, k^{\texttt{priv}}_{id,D_{U_a,\alpha}}) \leftarrow \texttt{keyGenWei}()$$

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

$$\sigma_{id,D_{U_a,\alpha}} \leftarrow \texttt{signECSchnorr}(k^{\texttt{priv}}_{f,D_{U_a,\alpha}}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}).$$

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 :

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

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

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

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.

Application initiale. L'application initiale ne prenait pas en compte l'authentification à l'aide d'un *token*. Nous avons donc ajouté cette fonctionnalité et l'avons rendue obligatoire. Nous avons également ajouté une étape d'authentification du serveur qui permet de vérifier si le serveur a été accrédité par une authorité. Ce protocole est donc entièrement redéfini pour ce projet.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes dans ce protocole :

- **Vulnérabilité 4** : utilisation de la primitive `SHA-1` obsolète pour la signature σ_{url}.
- **Vulnérabilité 14** : utilisation d'une clé RSA de 512 bits pour la clé privée commune aux serveurs accrédités.
- **Vulnérabilité 16** : la signature sans padding et le format du challenge permettent de forger la signature d'un challenge m à partir de quelques messages signés antérieurement.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (`keyGenWei`).
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature `signECSchnorr`.
- **Vulnérabilité 32** : la clé de management par défaut est connue mais piéger la Yubikey serait facilement détectable.
- **Vulnérabilité 31** : un attaquant peut récupérer les clés contenues dans la Yubikey dans le cas où le code PUK par défaut n'a pas été modifié.

3.2 Session `Olm`

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 `Olm` 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 `Olm` 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 `Olm` 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 `Olm` 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 $k^{\text{pub}}_{f,D_{U_b,\beta}}$, sa clé publique d'identité $k^{\text{pub}}_{id,D_{U_b,\beta}}$ avec sa signature $\sigma_{id,D_{U_b,\beta}}$ et une de ses clés *one-time* $k^{\text{pub}}_{ot,D_{U_b,\beta},\ell}$ 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 $k^{\text{pub}}_{fb,D_{U_b,\beta}}$ avec sa signature $\sigma_{fb,D_{U_b,\beta},\ell}$ à la place de la clé one-time. Cette clé n'est pas consommée après une utilisation, c'est-à-dire qu'elle peut être utilisée pour créer plusieurs sessions `Olm` différentes. Toutefois, les sessions créées avec la clé fallback pourraient être vulnérables aux attaques par "rejeu". Une fois que $D_{U_b,\beta}$ s'est reconnecté, et qu'il est raisonnablement certain qu'il a reçu tous les messages qui utilisaient l'ancienne clé fallback, par exemple après une heure depuis le premier message, il doit supprimer cette clé fallback, en générer une nouvelle et l'envoyer au serveur.*

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

$$\texttt{verifECSchnorr}(k^{\text{pub}}_{f,D_{U_b,\beta}}, k^{\text{pub}}_{id,D_{U_b,\beta}}, \sigma_{id,D_{U_b,\beta}}),$$
$$\texttt{verifECSchnorr}(k^{\text{pub}}_{f,D_{U_b,\beta}}, k^{\text{pub}}_{ot,D_{U_b,\beta},\ell}, \sigma_{ot,D_{U_b,\beta},\ell}).$$

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

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

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

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

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

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

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

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

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

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

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

   et une nouvelle clé de chaînage

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

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

$$\begin{aligned}
(k_{0,j}, \texttt{IV}^{\texttt{HKDF}}_{0,j}) &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, M_{0,j}, \text{“}\texttt{OLM_KEY}\text{”}, 48)\\
\text{session-id} &\leftarrow \texttt{SHA-3}(k^{\text{pub}}_{id,D_{U_a,\alpha}} \parallel k^{\text{pub}}_{ot,D_{U_a,\alpha},\ell'} \parallel k^{\text{pub}}_{ot,D_{U_b,\beta},\ell})\\
\texttt{IV}^{\texttt{AES}}_{0,j} &\leftarrow \text{session-id} || \texttt{IV}^{\texttt{HKDF}}_{0,j}\\
c_{0,j} &\leftarrow \texttt{enc}^{\texttt{AES-CBC}}(k_{0,j}, \texttt{IV}^{\texttt{AES}}_{0,j}, \texttt{msg}_{0,j} || \texttt{ID}_{U_a} || \texttt{ID}_{D_{U_a,\alpha}} || k^{\text{pub}}_{f,D_{U_a,\alpha}} || \texttt{ID}_{U_b} || k^{\text{pub}}_{f,D_{U_b,\beta}})\\
\texttt{mac}_{0,j} &\leftarrow \texttt{computeMac}^{\texttt{HMAC-SHA-3}}(k_{0,j}, \texttt{version} || T^{\text{pub}}_{0,D_{U_a,\alpha},D_{U_b,\beta}} || j || \texttt{msg}_{0,j})|_{64 \text{ bits}}
\end{aligned}$$

   où $\texttt{ID}_{U_a}$ (respectivement $\texttt{ID}_{U_b}$) représente l'identifiant de l'utilisateur U_a (respectivement U_b), $\texttt{ID}_{D_{U_a,\alpha}}$ 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 :

$$(k^{\text{pub}}_{ot,D_{U_b,\beta},\ell}, k^{\text{pub}}_{ot,D_{U_a,\alpha},\ell'}, k^{\text{pub}}_{id,D_{U_a,\alpha}}, T^{\text{pub}}_{0,D_{U_a,\alpha},D_{U_b,\beta}}, j, c_{0,j}, \texttt{mac}_{0,j}).$$

Application initiale. Les changements suivants ont été appliqués par rapport à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- nous avons modifié l'appel à la primitive $\texttt{HKDF}^{\texttt{HMAC-SHA-3}}$ pour ne générer qu'une clé unique pour le chiffrement et l'authentification, au lieu de deux clés différentes dans l'application initiale.
- nous utilisons des clés *one-time* Wei25519 pour remplacer les clés Ed25519 originales et des clés *ratchet* Wei25519 pour remplacer les clés Curve25519 de l'application initiale.
- la primitive de chiffrement par bloc `AES` est modifiée,
- le protocole *encrypt-then-mac* qui authentifie le chiffré est remplacé par le protocole *encrypt-and-mac* avec l'authentification du message en clair,
- le vecteur d'initialisation est modifié à l'issue de la fonction de dérivation.
- le protocole `ECDH` a été modifié.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes pour ce protocole :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 3** : même clé pour le chiffrement et l'authentification.
- **Vulnérabilité 6** : le vecteur d'initialisation ne contient que 32 bits de la sortie de la fonction de dérivation et est complété avec une troncature de 12 octets de l'identifiant de session.
- **Vulnérabilité 12** : algorithme `SHA-3` implémenté avec une capacité trop faible.
- **Vulnérabilité 17** : omission de vérifications dans le protocole `ECDH` qui permettent de monter des *subgroup confinement attacks*.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (`keyGenWei`).
- **Vulnérabilité 29** : Le motif d'intégrité `HMAC3` est tronqué à 64 bits.
- **Vulnérabilité 30** : Les messages peuvent être rejoués.

Protocole 3.B (création de session `Olm` 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

$$(k^{\texttt{pub}}_{ot,D_{U_b,\beta},\ell}, k^{\texttt{pub}}_{ot,D_{U_a,\alpha},\ell'}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}, T^{\texttt{pub}}_{0,D_{U_a,\alpha},D_{U_b,\beta}}, j, c_{0,j}, \texttt{mac}_{0,j})$$

   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 $k^{\texttt{pub}}_{f,D_{U_a,\alpha}}$ qui correspond à la clé publique d'identité $k^{\texttt{pub}}_{id,D_{U_a,\alpha}}$ 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 :

$$\texttt{verifECSchnorr}(k^{\texttt{pub}}_{f,D_{U_a,\alpha}}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}, \sigma_{id,D_{U_a,\alpha}}).$$

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

$$S_{D_{U_a,\alpha},D_{U_b,\beta}} \leftarrow \texttt{ECDH}(k^{\texttt{priv}}_{ot,D_{U_b,\beta},\ell}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}) || \texttt{ECDH}(k^{\texttt{priv}}_{id,D_{U_b,\beta}}, k^{\texttt{pub}}_{ot,D_{U_a,\alpha},\ell'}) || \texttt{ECDH}(k^{\texttt{priv}}_{ot,D_{U_b,\beta},\ell}, k^{\texttt{pub}}_{ot,D_{U_a,\alpha},\ell'}),$$

   où $k^{\texttt{priv}}_{ot,D_{U_b,\beta},\ell}$ est la partie privée de la clé *one time* de $D_{U_b,\beta}$ correspondant à la partie publique $k^{\texttt{pub}}_{ot,D_{U_b,\beta},\ell}$ 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 $C_{0,0}$:

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

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 :

$$\forall\ x < \ell \leq j, \qquad C_{0,\ell} \leftarrow \texttt{computeMac}^{\texttt{HMAC-SHA-3}}(C_{0,\ell-1}, \text{“}\backslash\texttt{x02}\text{”})$$
$$M_{0,\ell} \leftarrow \texttt{computeMac}^{\texttt{HMAC-SHA-3}}(C_{0,\ell}, \text{“}\backslash\texttt{x01}\text{”})$$

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

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

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

$$\mathtt{verifMac}^{\mathtt{HMAC\text{-}SHA\text{-}3}}(k_{0,j}, \mathtt{version}||T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\mathtt{pub}}||j||\mathtt{msg}_{0,j}, \mathtt{mac}_{0,j})|_{64 \text{ bits}}$$

(c) si le déchiffrement réussit, l'appareil vérifie que $\mathtt{ID}_{U_b}$ correspond bien à son identifiant d'utilisateur. L'appareil vérifie également que la clé de signature $k_{f,D_{U_b,\beta}}^{\mathtt{pub}'}$ correspond bien à sa clé de signature, et que la clé de signature $k_{f,D_{U_a,\alpha}}^{\mathtt{pub}'}$ correspond bien à celle qu'il a récupérée du serveur à la première étape $k_{f,D_{U_a,\alpha}}^{\mathtt{pub}}$, et que $\mathtt{ID}_{U_a}$ et $\mathtt{ID}_{D_{U_a,\alpha}}$ 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 $k_{ot,D_{U_b,\beta},\ell}^{\mathtt{priv}}$ et enregistre $T_{0,D_{U_a,\alpha},D_{U_b,\beta}}^{\mathtt{pub}}$.

Application initiale. Les changements suivants ont été appliqués par rapport à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- nous avons modifié l'appel à la primitive $\mathtt{HKDF}^{\mathtt{HMAC\text{-}SHA\text{-}3}}$ pour ne générer qu'une clé unique pour le chiffrement et l'authentification, au lieu de deux clés différentes dans l'application initiale.
- le protocole *encrypt-then-mac* qui authentifie le chiffré est remplacé par le protocole *encrypt-and-mac* avec l'authentification du message en clair,
- le vecteur d'initialisation est modifié à l'issue de la fonction de dérivation pour être remplacé par une concaténation de la troncature de l'identifiant de session sur 12 octets et le vecteur d'initialisation original sur 4 octets.
- le protocole `ECDH` a été modifié.
- la primitive `AES` a été modifiée.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes pour ce protocole :
- **Vulnérabilité 2** : la primitive `AES` est erronée, elle est implémentée avec un tour supplémentaire.
- **Vulnérabilité 3** : la même clé est utilisée pour le chiffrement et l'authentification.
- **Vulnérabilité 6** : le vecteur d'initialisation ne contient que 32 bits de la sortie de la fonction de dérivation et est complété avec une troncature de 12 octets de l'identifiant de session.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 17** : omission de vérifications dans le protocole `ECDH` qui permettent de monter des *subgroup confinement attacks*.
- **Vulnérabilité 29** : le motif d'intégrité `HMAC-SHA-3` est tronqué à 64 bits.
- **Vulnérabilité 30** : les messages peuvent être rejoués.

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 :

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

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

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

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

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

   et la clé de chaînage est avancée

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

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

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

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

$$(j, c_{i,j}, \texttt{mac}_{i,j}, T^{\texttt{pub}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}),$$

où $T^{\texttt{pub}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}$ est sa clé *ratchet* publique et $k^{\texttt{pub}}_{id,D_{U_a,\alpha}}$ sa clé d'identité.

Application initiale. Les changements suivants ont été appliqués par rapport à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- nous avons modifié l'appel à la primitive $\texttt{HKDF}^{\texttt{HMAC-SHA-3}}$ pour ne générer qu'une clé unique pour le chiffrement et l'authentification, au lieu de deux clés différentes dans l'application initiale.
- les clés *ratchet* Curve25519 de l'application initiale ont été remplacées par des clés Wei25519.
- le protocole *encrypt-then-mac* qui authentifie le chiffré est remplacé par le protocole *encrypt-and-mac* avec l'authentification du message en clair.
- le vecteur d'initialisation est modifié à l'issue de la fonction de dérivation pour être remplacé par une concaténation de la troncature de l'identifiant de session sur 12 octets et le vecteur d'initialisation original sur 4 octets.
- le protocole `ECDH` a été modifié.
- la primitive `AES` a été modifiée.
- dans le nouveau code de l'application, la fonction de dérivation de clé `HKDF` n'utilise que 32 bits dans la précédente clé racine, au lieu des 256 bits prévus dans l'application initiale.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes pour ce protocole :
- **Vulnérabilité 2** : la primitive `AES` est erronée, elle est implémentée avec un tour supplémentaire.
- **Vulnérabilité 3** : la même clé est utilisée pour le chiffrement et l'authentification.
- **Vulnérabilité 6** : le vecteur d'initialisation ne contient que 32 bits de la sortie de la fonction de dérivation et est complété avec une troncature de 12 octets de l'identifiant de session.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 17** : omission de vérifications dans le protocole `ECDH` qui permettent de monter des *subgroup confinement attacks*.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (`keyGenWei`).
- **Vulnérabilité 28** : bug de copie dans la dérivation de la nouvelle clé racine (seulement 32 bits de R_{i-1} sont utilisés)
- **Vulnérabilité 29** : le motif d'intégrité `HMAC-SHA-3` est tronqué à 64 bits.
- **Vulnérabilité 30** : les messages peuvent être rejoués.

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

$$(j, c_{i,j}, \texttt{mac}_{i,j}, T^{\texttt{pub}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}),$$

envoyé à la fin du protocole 4.A.

2. $D_{U_b,\beta}$ vérifie qu'il a bien une session `Olm` établie avec l'appareil $D_{U_a,\alpha}$ de clé d'identité $k^{\texttt{pub}}_{id,D_{U_a,\alpha}}$.
3. $D_{U_b,\beta}$ regarde s'il a une clé de chaînage *récepteur* correspondant à $T^{\texttt{pub}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}$. Si ce n'est pas le cas, une nouvelle clé racine et une nouvelle clé de chaînage *récepteur* sont dérivées :

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

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 :

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

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 :

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

   Les clés de message $M_{i,x}, \ldots M_{i,j-1}$ sont stockées en mémoire par $D_{U_b,\beta}$.
7. enfin, le message $c_{i,j}$ est déchiffré comme suit :

$$\begin{aligned} (k_{i,j}, \texttt{IV}_{i,j}^{\texttt{HKDF}}) &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, M_{i,j}, \text{“}\texttt{OLM\_KEY}\text{”}, 48) \\ \texttt{IV}_i^{\texttt{AES}} &\leftarrow \text{session-id}||\texttt{IV}_i^{\texttt{HKDF}} \\ \texttt{msg}_{i,j}||\texttt{ID}_{U_a}||\texttt{ID}_{D_{U_a,\alpha}}||k^{\texttt{pub}'}_{f,D_{U_a,\alpha}}||\texttt{ID}_{U_b}||k^{\texttt{pub}'}_{f,D_{U_b,\beta}} &\leftarrow \texttt{dec}^{\texttt{AES-CBC}}(k_{i,j}, \texttt{IV}_i^{\texttt{AES}}, c_{i,j}) \end{aligned}$$

$$\texttt{verifMac}^{\texttt{HMAC-SHA-3}}(k_{i,j}, \texttt{version}||T^{\texttt{pub}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}||j||\texttt{msg}_{i,j}, \texttt{mac}_{i,j})|_{64 \text{ bits}}$$

8. si le déchiffrement réussit, l'appareil vérifie que $\texttt{ID}_{U_b}$ correspond bien à son identifiant d'utilisateur. L'appareil vérifie également que la clé de signature $k^{\texttt{pub}'}_{f,D_{U_b,\beta}}$ correspond bien à sa clé de signature, et que la clé de signature $k^{\texttt{pub}'}_{f,D_{U_a,\alpha}}$ correspond bien à celle qu'il a récupéré au protocole 3.A pour l'appareil $D_{U_a,\alpha}$, et que $\texttt{ID}_{U_a}$ et $\texttt{ID}_{D_{U_a,\alpha}}$ 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

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

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

Application initiale. Les changements suivants ont été appliqués par rapport à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- nous avons modifié l'appel à la primitive $\texttt{HKDF}^{\texttt{HMAC-SHA-3}}$ pour ne générer qu'une clé unique pour le chiffrement et l'authentification, au lieu de deux clés différentes dans l'application initiale.
- le protocole *encrypt-then-mac* qui authentifie le chiffré est remplacé par le protocole *encrypt-and-mac* avec l'authentification du message en clair,
- le vecteur d'initialisation est modifié à l'issue de la fonction de dérivation,
- dans l'application initiale, la clé de chaînage n'est avancée que si le déchiffrement réussit. Dans notre implémentation, la clé de chaînage avance dans tous les cas.
- le protocole `ECDH` a été modifié.
- la primitive `AES` a été modifiée.
- dans le nouveau code de l'application, la fonction de dérivation de clé `HKDF` n'utilise que 32 bits dans la précédente clé racine, au lieu des 256 bits prévus dans l'application initiale.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes pour ce protocole :
- **Vulnérabilité 2** : la primitive `AES` est erronée, elle est implémentée avec un tour supplémentaire.
- **Vulnérabilité 3** : la même clé est utilisée pour le chiffrement et l'authentification.
- **Vulnérabilité 6** : le vecteur d'initialisation ne contient que 32 bits de la sortie de la fonction de dérivation et est complété avec une troncature de l'identifiant de session.
- **Vulnérabilité 10** : les clés racine et de chaînage sont avancées même si la vérification du motif d'authentification échoue.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 17** : omission de vérifications dans le protocole `ECDH` qui permettent de monter des *subgroup confinement attacks*.
- **Vulnérabilité 28** : bug de copie dans la dérivation de la nouvelle clé racine (seulement 32 bits de R_{i-1} sont utilisés)
- **Vulnérabilité 29** : le motif d'intégrité `HMAC-SHA-3` est tronqué à 64 bits.
- **Vulnérabilité 30** : les messages peuvent être rejoués.

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 :
$$(k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}, k^{\texttt{priv}}_{f,D_{U_a,\alpha},r}) \leftarrow \texttt{keyGenWei}()$$
et envoie la partie publique $(k^{\texttt{pub}}_{f,D_{U_a,\alpha},r})$ 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 $j \in \{0,1,2,3\}$:
$$\begin{aligned}(R_{0,0,D_{U_a,\alpha},r}, R_{0,1,D_{U_a,\alpha},r}) &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, k^{\texttt{priv}}_{id,D_{U_a,\alpha}}, \texttt{"MEGOLM\_ROOT"}, 64)\\ (R_{0,2,D_{U_a,\alpha},r}, R_{0,3,D_{U_a,\alpha},r}) &\leftarrow \texttt{DRBG}(512).\end{aligned}$$
4. toutes les données de session sont signées :
$$\begin{aligned}\texttt{data} &\leftarrow \texttt{version}||i||R_{0,0,D_{U_a,\alpha},r}||R_{0,1,D_{U_a,\alpha},r}||R_{0,2,D_{U_a,\alpha},r}||R_{0,3,D_{U_a,\alpha},r}||k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}\\ \sigma_{\texttt{data}} &\leftarrow \texttt{signECSchnorr}(k^{\texttt{priv}}_{f,D_{U_a,\alpha},r}, \texttt{data}).\end{aligned}$$
5. les données `data` et leur signature $\sigma_{\texttt{data}}$ 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 $k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}$. 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*.

Application initiale. Les changements suivants ont été implémentés par rapport à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- nous utilisons des clés de signature Wei25519 au lieu des clés Ed25519.
- le *ratchet symétrique* n'est pas généré en utilisant un générateur de nombres aléatoires cryptographique. Les deux premiers blocs de 256 bits sont dérivés à partir de la clé privée d'identité de l'appareil. Les deux derniers blocs sont générés à partir d'un `DRBG`.

Vulnérabilités. Les vulnérabilités suivantes ont été introduites dans ce protocole :
- **Vulnérabilité 8** : les 512 premiers bits du *ratchet symétrique* de groupe sont dérivés à partir de la clé de l'appareil. Après 2^{16} itérations (messages envoyés) par conversation, ces clés sont identiques pour toutes les conversations de groupe de l'appareil.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (`keyGenWei`).
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature `signECSchnorr`.
- **Vulnérabilité 22** : utilisation d'un générateur aléatoire cryptographique non recommandé.

Protocole 6.A (envoi de messages dans une conversation de groupe)**.** Lorsque l'appareil $D_{U_a,\alpha}$ souhaite envoyer un $i^{\texttt{ème}}$ message $\texttt{msg}_i$ 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

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

et l'authentifie

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

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

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

et le chiffré $c_{\texttt{msg}_i}$, le motif d'authentification $\texttt{mac}_i$, la signature σ_i, l'indice i, la clé publique $k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}$, la clé publique d'identité de l'appareil $k^{\texttt{pub}}_{id,D_{U_a,\alpha}}$ ainsi que l'identifiant du salon en question $\texttt{ID}_r$ sont tous envoyés dans un message

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

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

$$R_{i+1,0,D_{U_a,\alpha},r} \leftarrow \begin{cases} \texttt{computeMac}^{\texttt{HMAC-SHA-3}}(R_{2^{24}(n-1),0,D_{U_a,\alpha},r}, \text{“\textbackslash x00”}) & \texttt{si } \exists n \texttt{ t.q. } i+1 = 2^{24}n \\ R_{i,0,D_{U_a,\alpha},r} & \texttt{sinon.} \end{cases}$$

$$R_{i+1,1,D_{U_a,\alpha},r} \leftarrow \begin{cases} \texttt{computeMac}^{\texttt{HMAC-SHA-3}}(R_{2^{24}(n-1),0,D_{U_a,\alpha},r}, \text{“\textbackslash x01”}) & \texttt{si } \exists n \texttt{ t.q. } i+1 = 2^{24}n \\ \texttt{computeMac}^{\texttt{HMAC-SHA-3}}(R_{2^{16}(m-1),1,D_{U_a,\alpha},r}, \text{“\textbackslash x01”}) & \texttt{si } \exists m \texttt{ t.q. } i+1 = 2^{16}m \\ R_{i,1,D_{U_a,\alpha},r} & \texttt{sinon.} \end{cases}$$

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

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

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

Application initiale. Les changements suivants ont été réalisés par rapport à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- l'application initiale déclenche la génération d'un nouveau *ratchet symétrique* toutes les semaines et tous les 100 messages. Nous supprimons ce deuxième cas de génération automatique (mais conservons la limite hebdomadaire).
- une seule clé est dérivée du *ratchet symétrique* pour le chiffrement et l'authentification, contrairement à l'application initiale.
- la primitive de signature est modifiée,
- le protocole *encrypt-then-mac* qui authentifie le chiffré est remplacé par le protocole *encrypt-and-mac* avec l'authentification du message en clair.
- le vecteur d'initialisation est modifié à l'issue de la fonction de dérivation.
- la primitive `AES` a été modifiée.

Vulnérabilités.
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 3** : même clé est utilisée pour le chiffrement et l'authentification.
- **Vulnérabilité 6** : le vecteur d'initialisation ne contient que 32 bits de la sortie de la fonction de dérivation et est complété avec une troncature de 12 octets de l'identifiant de session.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature `signECSchnorr`.
- **Vulnérabilité 29** : le motif d'intégrité `HMAC-SHA-3` est tronqué à 64 bits.

- **Vulnérabilité 30** : les messages peuvent être rejoués.

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
$$(c_{\texttt{msg}_i}, \texttt{mac}_i, \sigma_i, i, k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}, k^{\texttt{pub}}_{id,D_{U_a,\alpha}}, \texttt{ID}_r).$$
envoyé à la fin du protocole 6.B.
2. l'appareil vérifie qu'il a bien une session *megolm* établie dans le salon d'identifiant $\texttt{ID}_r$ avec l'utilisateur de clé d'identité $k^{\texttt{pub}}_{id,D_{U_a,\alpha}}$.
3. l'appareil $D_{U_b,\beta}$ vérifie la signature en utilisant la clé de signature publique $k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}$:
$$\texttt{verifECSchnorr}(k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}, c_{\texttt{msg}_i}||\texttt{mac}_{i|64\text{ bits}}, \sigma_i).$$
4. 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.
5. $D_{U_b,\beta}$ déchiffre le message
$$\begin{aligned}
\text{session-id} &\leftarrow k^{\texttt{pub}}_{f,D_{U_a,\alpha},r}\\
(k_i, \texttt{IV}_i^{\texttt{HKDF}}) &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, R_{i,D_{U_a,\alpha},r}, \texttt{"MEGOLM_KEYS"}, 48)\\
\texttt{IV}_i^{\texttt{AES}} &\leftarrow \text{session-id}||\texttt{IV}_i^{\texttt{HKDF}}\\
\texttt{msg}_i &\leftarrow \texttt{dec}^{\texttt{AES-CBC}}(k_i, \texttt{IV}_i^{\texttt{AES}}, c_{\texttt{msg}_i})
\end{aligned}$$
et vérifie que le padding est correct.
6. $D_{U_b,\beta}$ vérifie le motif d'authentification :
$$\texttt{verifMac}^{\texttt{HMAC-SHA-3}}(k_i, \texttt{version}||i||\texttt{msg}_i, \texttt{mac}_i)|_{64\text{ bits}}.$$

Application initiale. Les changements suivants ont été apportés à l'application initiale :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- un message de *feedback* est ajouté pour informer l'émetteur en cas de padding incorrect.
- la primitive `AES` a été modifiée.
- une seule clé est dérivée du *ratchet symétrique* pour le chiffrement et l'authentification, contrairement à l'application initiale.
- le vecteur d'initialisation est modifié à l'issue de la fonction de dérivation.

Vulnérabilités.
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 3** : la même clé est utilisée pour le chiffrement et l'authentification.
- **Vulnérabilité 6** : le vecteur d'initialisation ne contient que 32 bits de la sortie de la fonction de dérivation et est complété avec une troncature de l'identifiant de session.
- **Vulnérabilité 9** : attaque par oracle de *padding* sur `AES-CBC`,
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 19** : le résultat de la vérification de signature est ignoré.
- **Vulnérabilité 29** : le motif d'intégrité `HMAC-SHA-3` est tronqué à 64 bits.
- **Vulnérabilité 30** : les messages peuvent être rejoués.

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 :

1. l'appareil $D_{U_a,\alpha}$ génère une clé `AES`, un vecteur d'initialisation et une clé `HMAC` à partir du *ratchet symétrique* courant :
$$(k^{\texttt{AES}}, k^{\texttt{HMAC}}, \texttt{IV}^{\texttt{AES}}) \leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, R_{i,D_{U_a,\alpha},r}, \texttt{"attachment"}, 80).$$
2. la pièce jointe est chiffrée :
$$c_{\texttt{pj}} \leftarrow \texttt{enc}^{\texttt{AES-CTR}}(k^{\texttt{AES}}, \texttt{IV}^{\texttt{AES}}, \texttt{pj})$$
et un `mac` est calculé :
$$\texttt{mac}_{\texttt{pj}} \leftarrow \texttt{computeMac}^{\texttt{HMAC-SHA-1}}(k^{\texttt{HMAC}}, c_{\texttt{pj}}).$$

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

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

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

Application initiale. Les changements suivants ont été opérés par rapport à l'application initiale :
- dans l'application initiale, la clé `AES` est générée aléatoirement par un générateur cryptographique.
- la primitive `AES` a été modifiée.
- dans l'application initiale le vecteur d'initialisation est constitué de 8 octets à zéro et 8 octets générés aléatoirement par un générateur cryptographique.
- l'application initiale ne calcule pas un motif d'authentification mais une empreinte avec la primitive `SHA-256`.

Vulnérabilités. Nous avons envisagé les vulnérabilités suivantes pour ce protocole :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 5** : dérivation des clés et du vecteur d'initialisation à partir du *ratchet symétrique* de la conversation qui n'est modifiée que tous les 2^{24} messages. Dans le code, $R_{i,D_{U_a,\alpha},r}$ est remplacé par $R_{i,0,D_{U_a,\alpha},r}$.
- **Vulnérabilité 11** : bug d'implémentation de la primitive `HMAC` permettant à l'attaquant de monter des *length extension attacks*.

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

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

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 $c_{\texttt{pj}}$ et vérifie son motif d'authentification

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

- il déchiffre ensuite la pièce jointe

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

Application initiale. Les changements suivants ont été opérés par rapport à l'application initiale :
- la primitive `AES` a été modifiée.
- l'application initiale ne vérifie pas un motif d'authentification mais une empreinte avec la primitive `SHA-256`.

Vulnérabilités. Nous avons envisagé les vulnérabilités suivantes pour ce protocole :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 11** : bug d'implémentation de la primitive `HMAC` permettant à l'attaquant de monter des *length attacks*.

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 :

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

2. l'utilisateur choisit une passphrase $\mathtt{pass}^{\mathtt{sec}}_{U_a}$ 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*

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

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 :

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

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

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

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

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

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

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

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

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

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 $\mathtt{msk}^{\mathtt{priv}}_{U_a}$, $\mathtt{usk}^{\mathtt{priv}}_{U_a}$ et $\mathtt{ssk}^{\mathtt{priv}}_{U_a}$ sur le serveur en utilisant $k_{\mathtt{sec},U_a}$. 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`

$$(k^{\mathtt{priv}}_{\mathtt{rec},U_a}, k^{\mathtt{pub}}_{\mathtt{rec},U_a}) \leftarrow \mathtt{keyGenWei}().$$

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

Application initiale. Nous avons procédé aux changements suivants :
- le générateur utilisé pour générer le sel cryptographique a été modifié.
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- la spécification originale prévoit un nombre d'itérations $N = 500.000$ que nous modifions par 2^{20} dans les spécifications et "$1 < 20$" (soit 1) au lieu de "$1 \ll 20$" dans le code de l'application,
- dans la primitive `PBKDF2`, un bug d'implémentation implique que la passphrase utilisateur est tronquée et n'utilise que $\mathtt{pass}^{\mathtt{sec}}_{U_a}|_t$ avec $t = 32$ bits au lieu de la passphrase complète.
- l'application initiale prévoyait une autre option pour la génération de la clé $k_{\mathtt{sec},U_a}$. Celle-ci pouvait être générée aléatoirement de manière uniforme sans l'utilisation de passphrase. L'utilisateur devait alors conserver cette clé qui ne pouvait pas être re-générée. Nous avons supprimé cette option.
- les clés *cross signing* Ed25519 ont été remplacées par des clés Wei25519 et la paire de clés *recovery* Curve25519 a été remplacée par une paire de clés Wei25519.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes :
- **Vulnérabilité 12** : algorithme `SHA-3` implémenté avec une capacité trop faible.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (`keyGenWei`).
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature `signECSchnorr`.
- **Vulnérabilité 22** : utilisation d'un générateur aléatoire cryptographique non recommandé.
- **Vulnérabilité 27** : un premier bug d'implémentation engendre une troncature de la passphrase pour n'utiliser que ses 32 bits de poids faibles et un second bug d'implémentation réduit le nombre d'itérations à 1.

Protocole 9 (stockage sécurisé sur le serveur)**.** L'appareil permet à chaque utilisateur U_a en utilisant sa clé *secret storage* $k_{\mathtt{sec},U_a}$ générée avec le protocole 8 de stocker de manière sécurisée une clé privée $\mathtt{backupK}_i$ 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 $k_{\mathtt{sec},U_a}$

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

   où $\mathtt{INFO}_i$ est une chaine connue qui dépend du type de la clé à stocker.

2. l'appareil génère aussi un vecteur d'initialisation $\mathtt{IV}_i^{\mathtt{AES}}$ de 128 bits d'aléa cryptographique

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

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

3. l'appareil $D_{U_a,\alpha}$ chiffre et authentifie la clé $\mathtt{backupK}_i$ à stocker :

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

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

Application initiale. Nous avons procédé aux changements suivants :
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- la primitive `AES` a été modifiée.
- le vecteur d'initialisation est généré à partir d'un Dual EC DRBG au lieu d'un PRNG cryptographique Java dans l'application initiale.

Vulnérabilités. Ce protocole implémente les vulnérabilités suivantes :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 22** : utilisation d'un générateur aléatoire cryptographique non recommandé.

Protocole 10 (récupération des clés stockées sur le serveur)**.** Pour récupérer une clé $\mathtt{backupK}_i$ 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 :

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

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

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

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

   où $\mathtt{INFO}_i$ est une chaine connue qui dépend du type de la clé à récupérer.

4. l'appareil $D_{U_a,\alpha}$ vérifie le $\mathtt{mac}_{\mathtt{backupK}_i}$

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

   et déchiffre la clé :

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

5. 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

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

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 $c_{\texttt{backupK}_i}$ 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 recupérées.*

Application initiale. Nous avons procédé aux changements suivants :
- pour la primitive `PBKDF2`, la spécification originale prévoit un nombre d'itérations $N = 500.000$ que nous modifions par 2^{20} dans les spécifications et "$1 < 20$" (soit 1) au lieu de "$1 \ll 20$" dans le code de l'application.
- dans la primitive `PBKDF2`, un bug d'implémentation implique que la passphrase utilisateur est tronquée et n'utilise que $\texttt{pass}_{U_a}^{\texttt{sec}}|_t$ avec $t = 32$ bits au lieu de la passphrase complète.
- la primitive `AES` a été modifiée.
- la primitive d'authentification utilisée dans l'application initiale était `HMAC-SHA-256`. Elle a été remplacée par `HMAC-SHA-3`.
- la primitive de signature a été modifiée.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature `signECSchnorr`.
- **Vulnérabilité 27** : un premier bug d'implémentation engendre une troncature de la passphrase pour n'utiliser que ses 32 bits de poids faibles et un second bug d'implémentation réduit le nombre d'itérations à 1.

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 $\texttt{backupM}_i$ de session `Megolm` sur le serveur en utilisant sa paire de clé *recovery* $(k_{\texttt{rec},U_a}^{\texttt{priv}}, k_{\texttt{rec},U_a}^{\texttt{pub}})$ générée dans le protocole 8. La procédure est la suivante :

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

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

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

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

3. l'appareil $D_{U_a,\alpha}$ chiffre et authentifie les clés $\texttt{backupM}_i$ à stocker :

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

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

Application initiale. Nous décrivons les changements opérés par rapport à l'application initiale :
- l'application initiale utilisait `AES-CBC` puis `HMAC-SHA-256` pour le chiffrement et l'authentification du `backup` chiffré. Nous avons remplacé ces appels par un appel au chiffrement authentifié `AES-GCM`,
- l'application initiale prévoyait la génération d'une paire de clés éphémères pour le calcul du secret partagé. Ces clés sont remplacées par les clés d'identité de l'appareil (persistantes) et peuvent donc être réutilisées,
- l'utilisation de la primitive `HMAC-SHA-256` a été remplacée par l'utilisation de la primitive `HMAC-SHA-3`.
- seuls 32 bits du motif d'authentification sont enregistrés sur le serveur.
- la primitive `AES` a été modifiée.
- le protocole `ECDH` a été modifié.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 7** : l'utilisation de la clé *identité* pour le calcul du secret partagé implique le chiffrement de plusieurs messages avec la même clé et le même vecteur d'initialisation.

- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 13** : seuls les 32 bits de poids faibles du motif d'authentification sont envoyés.
- **Vulnérabilité 17** : omission de vérifications dans le protocole `ECDH` qui permettent de monter des *subgroup confinement attacks*.

Protocole 12 (récupération des clés de sessions `Megolm` stockées sur le serveur)**.** Pour récupérer une clé de session *megolm* $\texttt{backupM}_i$ 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 :

1. l'appareil récupère du serveur sa clé privée *recovery* $k^{\texttt{priv}}_{\texttt{rec},U_a}$ en utilisant le protocole 10.
2. L'appareil récupère aussi sa clé publique *recovery* $k^{\texttt{pub}}_{\texttt{rec},U_a}$ stockée en clair sur le serveur. Il vérifie la cohérence entre la clé publique stockée et la clé privée déchiffrée
$$\left(k^{\texttt{pub}}_{\texttt{rec},U_a} \stackrel{?}{=} \texttt{keyGenWeiPub}(k^{\texttt{priv}}_{\texttt{rec},U_a})\right).$$
3. $D_{U_a,\alpha}$ récupère la clé publique *identité* $k^{\texttt{pub}}_{id,D_{U_b,\beta}}$ ayant servi au chiffrement du stockage sécurisé et le dernier $\texttt{backupM}_i$ (avec son motif d'authentification $\texttt{mac}_{\texttt{backupM}_i}$) du serveur. Il recalcule le secret partagé :
$$S \leftarrow \texttt{ECDH}(k^{\texttt{priv}}_{\texttt{rec},U_a}, k^{\texttt{pub}}_{id,D_{U_b,\beta}}).$$
4. $D_{U_a,\alpha}$ dérive les clés et un IV :
$$(k^{\texttt{AES}}_i, \texttt{IV}^{\texttt{AES}}_i) \leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S, \emptyset, 64).$$
5. l'appareil $D_{U_a,\alpha}$ vérifie le `mac` et déchiffre les clés :
$$\texttt{backupM}_i \leftarrow \texttt{decAE}^{\texttt{AES-GCM}}(k^{\texttt{AES}}_i, \texttt{IV}^{\texttt{AES}}_i, c_{\texttt{backupM}_i}, \texttt{mac}_{\texttt{backupM}_i}|_{32\text{ bits}}).$$

   Remarque. *Lorsque le protocole 12 est exécuté, l'appareil $D_{U_a,\alpha}$ demande au serveur toutes les clés $c_{\texttt{backupM}_i}$ 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 recupérées.*

Application initiale. Nous avons procédé aux changements suivants :
- la clé éphémère utilisée pour le stockage a été remplacée par la clé *identité* de l'appareil procédant au stockage.
- le protocole `ECDH` a été modifié.
- la primitive `AES` a été modifiée.
- la taille du motif d'authentification à vérifier a été réduite à 32 bits.
- l'utilisation de la primitive `HMAC-SHA-256` a été remplacée par l'utilisation de la primitive `HMAC-SHA-3`.
- la paire de clés *recovery* Curve25519 a été modifié par une paire de clés Wei25519 et la fonction de génération de la clé publique a été modifiée en fonction.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes :
- **Vulnérabilité 2** : primitive `AES` erronée, implémentée avec un tour supplémentaire.
- **Vulnérabilité 7** : l'utilisation de la clé *identité* pour le calcul du secret partagé permet de chiffrer plusieurs messages avec la même clé et le même vecteur d'initialisation.
- **Vulnérabilité 12** : l'algorithme `SHA-3` est implémenté avec une capacité trop faible.
- **Vulnérabilité 13** : seuls les 32 bits de poids faibles du motif d'authentification sont vérifiés.
- **Vulnérabilité 17** : omission de vérifications dans le protocole `ECDH` qui permettent de monter des *subgroup confinement attacks*.

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 $\sigma_{\mathtt{usk}_{U_a},\mathtt{msk}_{U_a}}$ de la clé *user-signing* $\mathtt{usk}^{\mathtt{pub}}_{U_a}$ de U_a avec la clé maître $\mathtt{msk}_{U_a}$ de U_a

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

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

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

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

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

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

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

   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 $\sigma_{\mathtt{usk}_{U_b},\mathtt{msk}_{U_b}}$ de la clé *user-signing* $\mathtt{usk}^{\mathtt{pub}}_{U_b}$ de U_b avec la clé maître $\mathtt{msk}_{U_b}$ de U_b

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

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

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

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

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

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

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

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

   $$(k^{\mathtt{pub}}_{\mathtt{eph},D_{U_b,\beta}}, k^{\mathtt{priv}}_{\mathtt{eph},D_{U_b,\beta}}) \leftarrow \mathtt{keyGenWei}().$$

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

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

qu'il envoie à $D_{U_a,\alpha}$.

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

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

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 $k^{\text{pub}}_{\text{eph},D_{U_b,\beta}}$ à $D_{U_a,\alpha}$.

5. $D_{U_a,\alpha}$ vérifie l'empreinte à partir de la clé reçue :

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

et calcule un secret partagé :

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

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

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

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

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

où `INFO` est égale à la concaténation des informations suivantes :
   - la chaîne de caractères `"MATRIX_KEY_VERIFICATION_SAS"`
   - le nom d'utilisateur $\texttt{user-name}_{U_a}$
   - l'identifiant de $D_{U_a,\alpha}$
   - le nom d'utilisateur $\texttt{user-name}_{U_b}$
   - 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 :

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

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

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

   où $\texttt{ID}_{\texttt{msk}^{\text{pub}}_{U_a}}$ est l'identifiant de la clé $\texttt{msk}^{\text{pub}}_{U_a}$. 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 :

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

   où "$\texttt{ID}_{\texttt{msk}^{\text{pub}}_{U_a}}$" est rajoutée dans la chaine si $\texttt{mac}_{\texttt{msk}^{\text{pub}}_{U_a}}$ 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 $\texttt{user-name}_{U_a}$
- l'identifiant de $D_{U_a,\alpha}$
- le nom d'utilisateur de $\texttt{user-name}_{U_b}$
- 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 $\texttt{mac}_{k^{\texttt{pub}}_{f,D_{U_a,\alpha}}}$, $\texttt{mac}_{\texttt{msk}^{\texttt{pub}}_{U_a}}$ (s'il existe), et $\texttt{mac}_{\texttt{IDS},D_{U_a,\alpha}}$ à $D_{U_b,\beta}$.

(b) De la même manière, $D_{U_b,\beta}$ calcule ses trois motifs d'authentification $\texttt{mac}_{k^{\texttt{pub}}_{f,D_{U_b,\beta}}}$, $\texttt{mac}_{\texttt{msk}^{\texttt{pub}}_{U_b}}$ (s'il existe), $\texttt{mac}_{\texttt{IDS},D_{U_b,\beta}}$ et les envoie à $D_{U_a,\alpha}$.

(c) $D_{U_a,\alpha}$ reçoit de $D_{U_b,\beta}$ les motifs d'authentification $\texttt{mac}_{k^{\texttt{pub}}_{f,D_{U_b,\beta}}}$, $\texttt{mac}_{\texttt{msk}^{\texttt{pub}}_{U_b}}$ (s'il existe), $\texttt{mac}_{\texttt{IDS},D_{U_b,\beta}}$ et les vérifie :

$$
\begin{aligned}
k^{\texttt{HMAC}}_{k^{\texttt{pub}}_{f,D_{U_b,\beta}}} &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \texttt{"INFO"+"ID}_{\texttt{k}^{\texttt{pub}}_{\texttt{f,D}_{\texttt{U}_\texttt{b}},\beta}}\texttt{"}, 32)\\
k^{\texttt{HMAC}}_{\texttt{msk}^{\texttt{pub}}_{U_b}} &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \texttt{"INFO"+"ID}_{\texttt{msk}^{\texttt{pub}}_{\texttt{U}_\texttt{b}}}\texttt{"}, 32)\\
k^{\texttt{HMAC}}_{\texttt{IDS},D_{U_b,\beta}} &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \texttt{"INFO"+"KEY\_IDS"}, 32)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{verifMac}^{\texttt{HMAC-SHA-3}}(k^{\texttt{HMAC}}_{k^{\texttt{pub}}_{f,D_{U_b,\beta}}}, k^{\texttt{pub}}_{f,D_{U_b,\beta}}, \texttt{mac}_{k^{\texttt{pub}}_{f,D_{U_b,\beta}}})\\
&\texttt{verifMac}^{\texttt{HMAC-SHA-3}}(k^{\texttt{HMAC}}_{\texttt{msk}^{\texttt{pub}}_{U_b}}, \texttt{msk}^{\texttt{pub}}_{U_b}, \texttt{mac}_{\texttt{msk}^{\texttt{pub}}_{U_b}})\\
&\texttt{verifMac}^{\texttt{HMAC-SHA-3}}(k^{\texttt{HMAC}}_{\texttt{IDS},D_{U_b,\beta}}, \texttt{"ID}_{\texttt{k}^{\texttt{pub}}_{\texttt{f,D}_{\texttt{U}_\texttt{b}},\alpha}}\texttt{"}, \texttt{ID}_{\texttt{msk}^{\texttt{pub}}_{\texttt{U}_\texttt{b}}}\texttt{"}, \texttt{mac}_{\texttt{IDS},D_{U_b,\beta}})
\end{aligned}
$$

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 $\texttt{msk}^{\texttt{pub}}_{U_b}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

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

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

$$
\begin{aligned}
k^{\texttt{HMAC}}_{k^{\texttt{pub}}_{f,D_{U_a,\alpha}}} &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \texttt{"INFO"+"ID}_{\texttt{k}^{\texttt{pub}}_{\texttt{f,D}_{\texttt{U}_\texttt{a}},\alpha}}\texttt{"}, 32)\\
k^{\texttt{HMAC}}_{\texttt{msk}^{\texttt{pub}}_{U_a}} &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \texttt{"INFO"+"ID}_{\texttt{msk}^{\texttt{pub}}_{\texttt{U}_\texttt{a}}}\texttt{"}, 32)\\
k^{\texttt{HMAC}}_{\texttt{IDS},D_{U_a,\alpha}} &\leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(0, S_{D_{U_a,\alpha},D_{U_b,\beta}}, \texttt{"INFO"+"KEY\_IDS"}, 32)
\end{aligned}
$$

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

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 $\texttt{msk}^{\texttt{pub}}_{U_a}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

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

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 chaine 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 $\mathtt{msk}^{\mathtt{pub}}_{U_a}$ et $\mathtt{msk}^{\mathtt{pub}}_{U_b}$.
- lorsque $U_a = U_b$, l'appareil $D_{U_a,\alpha}$ encode sa clé maître de *cross-signing* publique $\mathtt{msk}^{\mathtt{pub}}_{U_a}$, alors que $D_{U_b,\beta}$ sa clé de signature publique $k^{\mathtt{pub}}_{f,D_{U_b,\beta}}$.

(d)
- lorsque $U_a \neq U_b$, l'appareil $D_{U_a,\alpha}$ (respectivement $D_{U_b,\beta}$) encode la clé maître potentielle $\mathtt{msk}^{\mathtt{pub}'}_{U_b}$ de l'appareil $D_{U_b,\beta}$ (respectivement $\mathtt{msk}^{\mathtt{pub}'}_{U_a}$ 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 $k^{\mathtt{pub}'}_{f,D_{U_b,\beta}}$ de l'appareil $D_{U_b,\beta}$ (respectivement la clé maître potentielle $\mathtt{msk}^{\mathtt{pub}'}_{U_a}$ de l'appareil $D_{U_a,\alpha}$) qu'il a récupérée du serveur auparavant.

(e) Chaque appareil encode dans son QR code un secret ($\mathtt{secret}_a$ et $\mathtt{secret}_b$) 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
   - $\mathtt{msk}^{\mathtt{pub}'}_{U_b} = \mathtt{msk}^{\mathtt{pub}}_{U_b}$ et que $\mathtt{msk}^{\mathtt{pub}}_{U_a} = \mathtt{msk}^{\mathtt{pub}'}_{U_a}$ pour le cas $U_a \neq U_b$.
   - $\mathtt{msk}^{\mathtt{pub}}_{U_a} = \mathtt{msk}^{\mathtt{pub}'}_{U_a}$ et que $k^{\mathtt{pub}'}_{f,D_{U_b,\beta}} = k^{\mathtt{pub}}_{f,D_{U_b,\beta}}$ pour le cas $U_a = U_b$.
3. si la vérification fonctionne, $D_{U_a,\alpha}$ envoie le secret $\mathtt{secret}_b$ scanné à $D_{U_b,\beta}$, et signe la partie publique de la clé $\mathtt{msk}^{\mathtt{pub}}_{U_b}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

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

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 $\mathtt{msk}^{\mathtt{pub}}_{U_a}$ avec sa clé *user-signing* et envoie la signature au serveur (cette signature est uniquement effectuée lorsque $U_a \neq U_b$) :

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

Application initiale. Nous avons réalisé les changements suivants par rapport à l'application initiale :
- la fonction de hachage utilisée par les deux parties était SHA-256. Nous l'avons remplacée par SHA-1.
- l'utilisation de la primitive HMAC-SHA-256 a été remplacée par l'utilisation de la primitive HMAC-SHA-3.
- les clés éphémères Curve25519 ont été remplacées par des clés éphémères Wei25519.
- le protocole ECDH a été modifié.
- le secret $\mathtt{secret}_a$ (resp. $\mathtt{secret}_b$) est généré à partir d'un Dual EC DRBG au lieu d'un PRNG cryptographique Java dans l'application initiale.

Vulnérabilités. Nous avons introduit les vulnérabilités suivantes :
- **Vulnérabilité 4** : utilisation de la primitive SHA-1 obsolète pour le hachage.
- **Vulnérabilité 12** : algorithme SHA-3 implémenté avec une capacité trop faible.
- **Vulnérabilité 18** : génération de clés de signature de faible entropie (128 bits au lieu de 256 bits) dans la primitive (keyGenWei).
- **Vulnérabilité 20** et **Vulnérabilité 21** : génération de nonces à partir d'un générateur non cryptographique pour la signature signECSchnorr.

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 :

- $\texttt{computeMac}^{\texttt{HMAC-SHA-x}}$: 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,
- $\texttt{verifMac}^{\texttt{HMAC-SHA-x}}$: 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 raison d'efficacité, les motifs d'authentification sont parfois tronqués. $\texttt{computeMac}^{\texttt{HMAC-SHA-x}}(\cdot)|_{t \text{ bits}}$ signifie que le motif est tronqué à t bits de sortie. $\texttt{verifMac}^{\texttt{HMAC-SHA-x}}|_{t \text{ bits}}$ 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 $\texttt{enc}^{\texttt{AES-CBC}}$ prend en arguments une clé de 256 bits, un vecteur d'initialisation de 128 bits et un message et retourne un chiffré. La fonction $\texttt{dec}^{\texttt{AES-CBC}}$ 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* PCKS#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 [**?**, Sectin 6.5]. Il est utilisé dans ce projet avec la primitive de chiffrement par bloc `AES` (spécifiée ci-dessus). La fonction $\texttt{enc}^{\texttt{AES-CTR}}$ prend en arguments une clé de 256 bits, un vecteur d'initialisation de 128 bits et un message et retourne un chiffré. La fonction $\texttt{dec}^{\texttt{AES-CTR}}$ 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 ($\texttt{encAE}^{\texttt{AES-GCM}}$) 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 ($\texttt{decAE}^{\texttt{AES-GCM}}$) 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 :

$$a := 19298681539552699237261830834781317975544997444273427339909597334573241639236,$$
$$b := 55751746669818908907645289078257140818241103727901012315294400837956729358436.$$

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

$x_G := 1$,

$y_G := 1890934733661515172867006422669276633882560039152466147363386270233299658108 4$.

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 $x_Q \in \{0, \ldots, n-1\}$ (premier tableau) et $y_Q \in \{0, \ldots, n-1\}$ (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 $\texttt{signRSA}_{\texttt{token}}$ et $\texttt{verifRSA}_{\texttt{token}}$ utilisés dans les protocoles 1 et 2 correspondent aux fonctionnalités `RSASP1` et `RSAVP1` respectivement décrites dans les spécifications [**?**]. Le mécanisme $\texttt{verifRSA}_{\texttt{server}}$ 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 :

$$B_i = (a \cdot S_i + c) \mod 2^{64}$$
$$S_{i+1} = (a \cdot B_i + c) \mod 2^{64}$$

   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 :

$$x_P := 29574854205587028187789910861250489067351937497348330655840054524429782953125$$
$$y_P := 34787411194293159582962146500558888268755365180353843541161653519432463373063$$
$$x_Q := 57863406254506098596402688243367836474675441954671054629065899898103707227695$$
$$y_Q := 20008002610233705536182467594645266681062800035717294684676566971264333897253$$

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.

5 Vulnérabilités introduites

Nous décrivons les vulnérabilités introduites ci-après en les classant par catégorie. Les protocoles affectés et le niveau de difficulté sont renseignés pour chacune d'entre elles.

5.1 Conformité

Vulnérabilité 1 (Politique de mot de passe). Cette vulnérabilité concerne la politique de définition d'un mot de passe pour l'authentification d'un utilisateur auprès du serveur. Dans les spécifications, nous décrivons une politique de mot de passe exigeant 8 caractères alphanumériques (avec au moins une minuscule, une majuscule, un chiffre et un caractère spécial). Dans le code du serveur, nous vérifions uniquement le nombre de caractères, mais pas leur nature. Un utilisateur peut donc s'enregistrer avec un mot de passe contenant 8 lettres minuscules, malgré la politique décrite.

Protocole(s) concerné(s)	1 (*sign-up*)
Catégorie	conformité
Difficulté d'identification	moyen
Difficulté d'exploitation	moyen

Vulnérabilité 2 (**Implémentation AES non-conforme**)**.** Cette vulnérabilité porte sur l'implémentation de l'`AES`-256. Contrairement aux spécifications du standard, la primitive AES-256 est implémentée avec un tour de boucle supplémentaire. La vulnérabilité résulte d'un bug d'implémentation avec l'utilisation d'un symbole *inférieur ou égal* au lieu d'une inégalité stricte dans la boucle itérant sur les tours.

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 6.A, 6.B, 7.A, 7.B, 9, 10, 11, 12
Catégorie	conformité
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

Vulnérabilité 3 (**Même clé pour chiffrement et MAC**)**.** Pour chiffrer les messages dans les conversations privées et de groupe, nous utilisons la même clé pour le chiffrement et pour le calcul du `mac`, ce qui constitue une vulnérabilité. Cette vulnérabilité apparaît dans les spécifications qui seront transmises aux CESTIs et est implémentée dans le code source de l'application.

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 6.A, 6.B
Catégorie	conformité
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

Vulnérabilité 4 (**Utilisation de la fonction de hachage `SHA-1`**)**.** Dans le protocole d'authentification entre deux appareils (Protocole 13), les deux parties calculent l'empreinte d'une clé publique avec la primitive `SHA-1`. Lors de l'enregistrement (Protocole 1) ou de la connexion (Protocole 2) d'un utilisateur, l'application vérifie une signature de l'url du serveur utilisant RSA avec `SHA-1`.

La primitive `SHA-1` n'étant pas conforme au guide de l'ANSSI [**?**], son utilisation constitue une vulnérabilité.

Protocole(s) concerné(s)	1 (*sign-up*), 2 (*sign-in*), 13 (authentification entre deux appareils)
Catégorie	conformité
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

5.2 Chiffrement symétrique

Vulnérabilité 5 (**Collision de keystream**)**.** Au lieu de générer une nouvelle paire clé et vecteur d'initialisation pour le chiffrement d'une pièce jointe dans une salle r, nous proposons de générer cette paire à partir du *ratchet symétrique* $R_{i,0,D_{U_a,\alpha},r}$ courant de l'appareil $D_{U_a,\alpha}$ concerné (tout en indiquant dans la spécification l'utilisation du *ratchet symétrique* complet $R_{i,D_{U_a,\alpha},r}$). Le *ratchet symétrique* $R_{i,0,D_{U_a,\alpha},r}$ n'est modifié que tous les 2^{24} messages émis, ce qui génère plusieurs vulnérabilités en pratique :

- le couple clé et vecteur d'initialisation est utilisé à de multiples reprises, ce qui implique que le même *keystream* AES CTR est utilisé pour plusieurs pièces jointes. La confidentialité est donc cassée.
- combinée à la mauvaise utilisation du `HMAC-SHA-1` (Vulnérabilité 11), un attaquant peut forger des `mac` pour des pièces jointes sans connaître la clé utilisée. Ces pièces jointes peuvent donc être modifiées.

Protocole(s) concerné(s)	7.A (pièces jointes)
Catégorie	chiffrement et intégrité symétrique
Difficulté d'identification	difficile
Difficulté d'exploitation	moyen

Vulnérabilité 6 (**Mauvais IV**)**.** Dans le chiffrement des messages pour les conversations privées, nous introduisons une vulnérabilité au niveau du vecteur d'initialisation utilisé. Au lieu de générer un vecteur d'initialisation de 128 bits à partir de la primitive $\texttt{HKDF}^{\texttt{HMAC-SHA-3}}$ et de la clé de message comme préconisé dans les spécifications, nous fixons les 12 octets de poids fort du vecteur d'initialisation à la troncature de l'identifiant de la session, et seulement les quatre octets de poids faible à la sortie de la primitive $\texttt{HKDF}^{\texttt{HMAC-SHA-3}}$.

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 6.A, 6.B
Catégorie	chiffrement symétrique
Difficulté d'identification	facile
Difficulté d'exploitation	moyen

Vulnérabilité 7 (**Réutilisation d'une paire clé-IV**)**.** Dans le contexte du stockage sécurisé de clés sur le serveur, un secret partagé est calculé par l'utilisateur entre la clé publique *recovery* et une clé privée éphémère. Nous remplaçons la clé privée éphémère par la clé *identité* de l'appareil (persistante pour une session). Ainsi, la même clé et le même vecteur d'initialisation (tous deux dérivés de ce secret partagé) sont utilisés pour le stockage sécurisé de plusieurs clés. Cette vulnérabilité affecte le chiffrement authentifié (qui utilise plusieurs fois le même vecteur d'initialisation avec la même clé).

Protocole(s) concerné(s)	11 et 12 (stockage)
Catégorie	chiffrement et intégrité symétriques
Difficulté d'identification	moyen
Difficulté d'exploitation	impossible

Vulnérabilité 8 (**Perte d'entropie dans le ratchet symétrique**)**.** Dans l'application originale, un *ratchet symétrique* de 1024 bits est généré aléatoirement de manière uniforme par un appareil utilisateur pour chiffrer des messages dans une conversation. Ce *ratchet symétrique* est ensuite mis à jour à l'issue de chaque chiffrement avec un protocole basé sur la primitive `HMAC-SHA-256`.

Cette vulnérabilité consiste à modifier la génération de *ratchet symétrique* en dérivant les 512 premiers bits (*i.e.*, $R_{i,0,D_{U_a},\alpha,r}$ et $R_{i,1,D_{U_a},\alpha,r}$ dans le protocole 5) à partir de la clé *identité* de l'appareil. Suivant le protocole de mise à jour du *ratchet symétrique* de l'application originale, le *ratchet symétrique* ne dépend plus que de ces 512 bits après 2^{16} utilisations. Il est donc alors identique pour toutes les conversations de groupe de l'appareil, permettant aux destinataires d'un groupe de déchiffrer des messages de groupes auxquels il n'appartient pas.

Protocole(s) concerné(s)	5 (création groupe)
Catégorie	chiffrement et intégrité symétriques
Difficulté d'identification	moyen
Difficulté d'exploitation	moyen

Vulnérabilité 9 (**Padding oracle sur AES CBC**)**.** Par rapport à l'application originale, nous remplaçons le mode *encrypt-then-mac* basé sur l'utilisation des primitives `AES-CBC` et `HMAC` pour le chiffrement authentifié des messages par un mode *encrypt-and-mac* basé sur les mêmes primitives. Nous ajoutons également une procédure de *feedback* qui consiste à envoyer un message d'erreur à l'émetteur si le *padding* du message déchiffré est incorrect et ce avant la vérification du MAC. Cette vulnérabilité de type *padding oracle* permet à un attaquant de déchiffrer des messages sans la connaissance de la clé secrète.

Protocole(s) concerné(s)	6.B (réception groupe)
Catégorie	chiffrement symétrique
Difficulté d'identification	moyen
Difficulté d'exploitation	moyen

Vulnérabilité 10 (**State machine attack**)**.** Dans le contexte d'une conversation entre deux interlocuteurs (session Olm), à l'issue du déchiffrement d'un message, le destinataire met à jour la clé de chaînage si et seulement si le motif d'authentification a bien été vérifié. Cette vulnérabilité consiste à effectuer la mise à jour de la clé de chaînage dans tous les cas, même si la vérification du motif d'authentification échoue. Un attaquant peut donc forcer l'avancement de la chaîne.

Protocole(s) concerné(s)	4.B (réception)
Catégorie	chiffrement symétrique
Difficulté d'identification	difficile
Difficulté d'exploitation	difficile

5.3 Intégrité symétrique

Vulnérabilité 11 (**Attaque par extension de longueur sur MAC** `SHA-1`)**.** Les spécifications indiquent l'utilisation de la primitive `HMAC-SHA-1` pour calculer un motif d'authentification dans le contexte de l'envoi d'une pièce jointe chiffrée. Cette vulnérabilité consiste à remplacer, dans le code source de l'application, la primitive `HMAC-SHA-1` appelée avec une clé $k^{\texttt{HMAC}}$ et un chiffré c par un appel unique à la fonction de hachage :

$$\texttt{SHA-1}(k^{\texttt{HMAC}}||c).$$

Avec cette modification, un attaquant peut réaliser une *attaque par extension de longueur* [**?**, **?**] qui consiste à forger un `mac` d'un message pré-fixé par c. Dans le contexte du protocole d'envoi de pièces jointes chiffrées, cela permet notamment de générer un `mac` pour des pièces jointes sans connaître la clé $k^{\texttt{HMAC}}$ utilisée si celle-ci n'est pas unique (voir Vulnérabilité 5).

Protocole(s) concerné(s)	7.A, 7.B (pièces jointes)
Catégorie	conformité et intégrité symétrique
Difficulté d'identification	facile
Difficulté d'exploitation	moyen

Vulnérabilité 12 (**Faible capacité dans SHA-3**)**.** Nous avons remplacé l'utilisation de la primitive `HMAC-SHA-256` dans le code source de l'application par l'utilisation de la primitive `HMAC-SHA-3`. Nous avons introduit une vulnérabilité dans l'implémentation de la primitive `SHA-3` qui consiste à limiter la capacité à 16 bits, au lieu des paramètres recommandés. Cela permet de retrouver la clé utilisée par la primitive `HMAC-SHA-3` (si le message est connu) avec une complexité de $2^{2\cdot(\text{taille de la capacité})} = 2^{32}$.

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 5, 6.A, 6.B, 8, 9, 10, 11, 12, 13
Catégorie	conformité et intégrité symétrique
Difficulté d'identification	moyen
Difficulté d'exploitation	moyen

Vulnérabilité 13 (**Réduction de la taille du motif d'authentification 32 bits**)**.** Cette vulnérabilité consiste à tronquer le motif d'authentification utilisé dans le contexte du stockage sécurisé de clés de session `Megolm` sur le serveur à 32 bits.

Protocole(s) concerné(s)	11, 12 (stockage sécurisé `Megolm`)
Catégorie	intégrité symétrique
Difficulté d'identification	facile
Difficulté d'exploitation	facile

5.4 Chiffrement asymétrique

Vulnérabilité 14 (**Signatures RSA-512**)**.** Nous ajoutons une fonctionnalité à l'application qui demande au serveur (le serveur par défaut ou un serveur personnalisé renseigné par l'utilisateur au protocole 1 et 2) de valider qu'il est accrédité à communiquer avec le client. Ceci est vérifié si le serveur est en possession d'une clé privée `RSA`, qui est obtenue par tous les serveurs accrédités depuis une autorité commune. Au moment de la connexion au serveur, le serveur renvoie au client son url publique ainsi que sa signature avec la clé `RSA`. La clé publique de signature est codée *en dur* dans l'application, permettant au client de vérifier la signature. La vulnérabilité introduite consiste à utiliser une paire de clés RSA de seulement 512 bits pour la signature de l'url publique du serveur.

Protocole(s) concerné(s)	1 (*sign-up*), 2 (*sign-in*)
Catégorie	chiffrement asymétrique
Difficulté d'identification	moyen
Difficulté d'exploitation	difficile

Vulnérabilité 15 (**Génération faible de clé RSA**)**.** A l'enregistrement de l'utilisateur auprès du serveur, l'utilisateur peut choisir entre la génération d'une nouvelle paire de clés `RSA` en utilisant l'algorithme de génération de clés intégré au *token* ou la génération d'une nouvelle paire de clés `RSA` par l'application (qui est ensuite chargée dans le *token*). Si l'utilisateur choisit la seconde option, l'application doit générer une paire de clés `RSA` de 2048 bits selon la méthode de génération de nombres premiers par rejet améliorée, référencée dans la Section 4.5.5. La vulnérabilité introduite consiste à générer un premier nombre premier p selon cette méthode, puis à fixer le second nombre premier q au premier nombre premier suivant p. Cette vulnérabilité permet à un attaquant de factoriser le module `RSA` très facilement et ainsi de retrouver la clé privée.

Protocole(s) concerné(s)	1 (*sign-up*)
Catégorie	chiffrement asymétrique
Difficulté d'identification	moyen
Difficulté d'exploitation	facile

Vulnérabilité 16 (**Forge de signature sans padding**)**.** Étant donné l'absence de padding et le format des messages signés, il est possible d'outrepasser le *token* en y ayant préalablement eu accès (avec code PIN). Plus précisément, le scénario d'attaque considère un attaquant ayant connaissance du code PIN et ayant un accès temporaire au *token*. L'attaquant utilise cet accès temporaire pour collecter quelques signatures de messages de son choix. Sur la base de ces quelques signatures, l'attaquant pourra par la suite répondre à n'importe quel challenge du serveur (*i.e.* forger la signature correspondante) sans avoir accès au *token*.

La signature `RSA` calculée par le *token* n'utilise pas de padding. Pour un challenge $m =$ `challenge` $\|$ `timestamp`, forger la signature revient à déterminer $\sigma = m^d \bmod N$ où d et N sont l'exposant privé et le

module de $k^{\texttt{priv}}_{\texttt{RSA},U_a}$. L'attaque consiste à faire signer les message m_1, ..., m_ℓ correspondant aux ℓ premiers nombres premiers (2, 3, 5, 7, ...) et à stocker leurs signatures σ_1, ..., σ_ℓ. Lors d'une tentative ultérieure d'authentification, l'attaquant teste si m est m_ℓ-friable, *i.e.* si m peut s'écrire sous la forme

$$m = \prod_{i=1}^{\ell} m_i^{e_i} \ .$$

Si c'est le cas, il peut alors déduire la signature

$$\sigma_m = \prod_{i=1}^{\ell} \sigma_i^{e_i} \bmod N \ .$$

Notons que par définition du protocole, l'attaquant peut tester la friabilité des 30 entiers m, $m+1$, ..., $m+30$ puisque le serveur tolère un décalage de 30 secondes.

La probabilité de succès de l'attaque (*i.e.* la probabilité d'avoir m friable sur l'intervalle considéré) dépend du nombre ℓ de signatures préalablement collectées. Nous estimons cette probabilité de succès à 1% pour $\ell = 1000$, 13% pour $\ell = 5000$, et 21% pour $\ell = 10.000$.

Protocole(s) concerné(s)	2 (*sign-in*)
Catégorie	chiffrement asymétrique
Difficulté d'identification	facile
Difficulté d'exploitation	difficile

Vulnérabilité 17 (**Subgroup confinement attack**)**.** Dans notre implémentation de la primitive `ECDH`, nous omettons volontairement la multiplication par le co-facteur pour rendre possible une *subgroup confinement attack*. Ce type d'attaque consiste pour un attaquant interceptant les communications (*man-in-the-middle attack*) à multiplier les clés publiques échangées par un scalaire k correspondant à l'ordre du groupe (composite) divisé par le cofacteur (*i.e.* l'ordre d'un petit sous-groupe). Ce faisant, le secret partagé n'appartient plus au groupe initial mais seulement au petit sous-groupe identifié. L'attaquant peut donc le retrouver à partir d'une recherche exhaustive. De plus, pour rendre l'attaque possible, nous définissons le point de base G de la courbe Wei25519 comme un générateur de la courbe entière (*i.e.* d'ordre composite) et non uniquement du sous-groupe d'ordre premier (voir Section 4.5.1).

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 11, 12
Catégorie	chiffrement asymétrique
Difficulté d'identification	moyen
Difficulté d'exploitation	moyen

Vulnérabilité 18 (**Entropie clés ECC**)**.** La génération de clés `keyGenWei` requiert la génération aléatoire uniforme d'une clé privée de 256 bits. Cette vulnérabilité consiste à générer une clé privée de 128 bits (au lieu de 256 bits), ce qui permet une attaque générique en 2^{64} (méthode rho, *baby-step giant-step*, ...)

Protocole(s) concerné(s)	1, 2, 3.A, 4.A, 5, 8, 13
Catégorie	chiffrement asymétrique
Difficulté d'identification	facile
Difficulté d'exploitation	difficile

5.5 Signature électronique

Vulnérabilité 19 (**Signature non-vérifiée**)**.** Lors de la réception d'un message dans une conversation de groupe, nous proposons d'ignorer la vérification de signature, permettant ainsi à un utilisateur malveillant qui fait partie de la conversation de groupe de se faire passer pour un autre utilisateur et envoyer un message à son nom. En effet, chaque utilisateur dans la conversation possède les clés *ratchet symétriques* de tous les autres utilisateurs dans la conversation. Ainsi, un utilisateur malveillant peut chiffrer le message avec la clé ratchet d'un autre utilisateur et le signer avec n'importe quelle clé de signature. Puisque la vérification de signature est ignorée, le message sera ainsi bien reçu par les autres participants dans la conversation.

Protocole(s) concerné(s)	6.B (réception groupe)
Catégorie	signature électronique
Difficulté d'identification	moyen
Difficulté d'exploitation	facile

Vulnérabilité 20 (**Lattice attack sur signature EC Schnorr**)**.** Comme la plupart des schémas de signature basés sur les courbes elliptiques, l'algorithme de signature de Schnorr repose sur la génération aléatoire d'un nonce cryptographique, suivant une loi uniforme. L'utilisation d'un générateur congruentiel linéaire (LCG, voir Vulnérabilité 21) est en soit un problème car il s'agit d'un générateur non cryptographique qui dans notre contexte ne fournit que 64 bits d'entropie. Cependant, il ne nous semble pas évident que la structure des nonces ainsi générées soit exploitable par une attaque à base de réseaux Euclidiens (*lattice attack*), telle que l'attaque de Howgrave-Graham et Smart [**?**]. Nous introduisons une vulnérabilité permettant de monter ce type d'attaque : un bug d'implémentation utilise deux fois la même sortie du générateur LCG dans le nonce. Plus précisément, cette dernière est générée de la façon suivante :

$$\begin{aligned} B_i &= \mathsf{LCG}() \\ B_{i+1} &= \mathsf{LCG}() \\ B_{i+2} &= \mathsf{LCG}() \\ B_{i+3} &= \mathsf{LCG}() \\ \mathsf{nonce} &= B_{i+3} \parallel B_{i+2} \parallel B_{i+2} \parallel B_i \end{aligned}$$

Les deux blocs de 64 bits au milieu du nonce sont donc les mêmes, ce qui permet une adaptation de l'attaque par réseaux Euclidiens.

Protocole(s) concerné(s)	1, 2, 5, 6.A, 8, 10, 13
Catégorie	signature électronique et implémentation cryptographique
Difficulté d'identification	difficile
Difficulté d'exploitation	difficile

5.6 Génération des nombres aléatoires

Vulnérabilité 21 (**Générateur d'aléa non-cryptographique**)**.** Cette vulnérabilité consiste à utiliser un générateur congruentiel linéaire (LCG) non cryptographique pour la génération de nonces cryptographique pour l'algorithme de signature de Schnorr. Avec un tel générateur, chaque nombre pseudo-aléatoire est généré à partir d'une fonction affine du nombre précédent. Un générateur cryptographique doit normalement être utilisé pour ce type d'applications.

Protocole(s) concerné(s)	1, 2, 5, 6.A, 8, 10, 13
Catégorie	génération de nombres aléatoires
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

Vulnérabilité 22 (**Dual EC DRBG**)**.** Plusieurs protocoles impliquent l'utilisation d'un générateur d'aléa cryptographique pour la génération de clés secrètes. Cette vulnérabilité consiste à utiliser le générateur Dual EC DRBG dans ce contexte. Ce générateur ne doit pas être utilisé car il est de notoriété publique qu'il contient une *backdoor* introduite par la NSA. Il a en outre été rendu obsolète par les différentes organisations de standardisation.

Protocole(s) concerné(s)	5, 8, 9
Catégorie	génération de nombres aléatoires
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

5.7 Protocole cryptographique

Vulnérabilité 23 (**Application autorisant TLS v1.1**)**.** Les appareils clients et le serveur de notre application utilisent le protocole TLS pour leurs communications. Cette vulnérabilité consiste, pour l'application, à autoriser l'utilisation de versions de TLS qui ne sont plus recommandées (notamment TLS 1.1). Néanmoins, en pratique, ces versions de TLS ne sont pas acceptées par Android (dans sa dernière version) et cette vulnérabilité n'est donc pas directement exploitable sur le matériel identifié. Les spécifications à destination des CESTIs ne renseignent pas ces versions mais indiquent simplement l'utilisation de la dernière version de TLS par défaut.

Protocole(s) concerné(s)	1 (première connexion au serveur)
Catégorie	protocole cryptographique
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

Vulnérabilité 24 (Serveur autorisant TLS v1.1). Cette vulnérabilité est similaire à la vulnérabilité 23 mais celle-ci est implémentée au niveau du serveur de l'application. A son tour, le serveur autorise l'utilisation de versions de TLS qui ne sont plus recommandées (notamment TLS 1.1).

Protocole(s) concerné(s)	1 (première connexion au serveur)
Catégorie	protocole cryptographique
Difficulté d'identification	facile
Difficulté d'exploitation	impossible

Vulnérabilité 25 (Certificat auto-signé). L'application ne restreint pas les certificats utilisés par le serveur, leur permettant notamment d'être auto-signés. Cette vulnérabilité permet à un attaquant d'impersonnifier le serveur.

Protocole(s) concerné(s)	1 (première connexion au serveur)
Catégorie	protocole cryptographique
Difficulté d'identification	moyen
Difficulté d'exploitation	facile

Vulnérabilité 26 (Authentification statique). L'authentification de l'adresse du serveur par la signature σ_{url} est statique, *i.e.* la signature utilisée pour une adresse m_{url} est toujours la même, ce qui permet une attaque par rejeu. Par exemple, un attaquant ayant préalablement reçu la signature σ_{url} auprès du serveur peut impersonnifier le serveur en usurpant son adresse (spoofing) et en rejouant σ_{url}.

Protocole(s) concerné(s)	1 (première connexion au serveur)
Catégorie	protocole cryptographique
Difficulté d'identification	moyen
Difficulté d'exploitation	facile

5.8 Implémentation cryptographique

Vulnérabilité 27 (Failles d'implémentation dans PBKDF2). La clé privée $k_{\texttt{sec},U_a}$ (Protocole 8) est générée à partir d'une passphrase et de la primitive `PBKDF2`. Cette primitive prend également en entrée une PRF, un sel, un nombre d'itérations et une taille. Nous proposons deux erreurs d'implémentation (dans le code source de l'application) :

- seuls les 32 bits de poids faibles de la passphrase sont effectivement utilisés en entrée de la primitive `PBKDF2`,
- le nombre d'itérations est fixé à 2^{20} dans les spécifications et à $1 < 20\ (= 1)$ dans le code source de l'application.

Ces erreurs permettent à un attaquant de retrouver la clé $k_{\texttt{sec},U_a}$ avec une attaque par brute force sur toutes les passphrases possibles de 32 bits.

Protocole(s) concerné(s)	8, 10 (stockage et récupération des clés)
Catégorie	implémentation cryptographique
Difficulté d'identification	moyen
Difficulté d'exploitation	facile

Vulnérabilité 28 (Bug dérivation clé racine). Dans le cas d'une session `Olm` pour l'échange de clés, une clé racine R_i est maintenue par les deux appareils qui communiquent. Elle est initialement générée à partir d'un secret partagé (*i.e.*, R_0) puis elle est mise à jour avant l'envoi d'un message ou la réception d'un message par les deux parties à partir de la primitive $\texttt{HKDF}^{\texttt{HMAC-SHA-3}}$ appelée avec un secret partagé et la clé précédente R_{i-1}. Cette vulnérabilité consiste à n'utiliser que 32 bits de la clé R_{i-1} au lieu des 256 bits initiaux (bug de copie).

Protocole(s) concerné(s)	4.A et 4.B (sessions `Olm`)
Catégorie	implémentation cryptographique
Difficulté d'identification	difficile
Difficulté d'exploitation	difficile

5.9 Vulnérabilités en cascade

Nous décrivons ci-dessous quelques exemples de vulnérabilités exploitables *en cascade* afin d'atteindre des exploits de sécurité sur l'application. Cette liste n'est pas exhaustive et d'autres vulnérabilités décrites ci-dessus sont exploitables (seules ou en cascade) afin de casser la confidentialité et l'authenticité des messages échangés.

5.9.1 Impersonnification d'un utilisateur auprès du serveur

L'impersonnification d'un utilisateur auprès du serveur est rendue possible par

1. l'exploitation de la Vulnérabilité 16 pour outrepasser l'utilisation du *token*,
2. l'exploitation de la Vulnérabilité 1 pour deviner / faire une recherche exhaustive du mot de passe.

En exploitant ces deux vulnérabilités, un attaquant peut impersonnifier un utilisateur auprès du serveur, à condition d'avoir eu un accès préalable au *token* avec le code PIN (afin de collecter les signatures σ_1, ..., σ_m, voir Vulnérabilité 16), et que le mot de passe choisi par l'utilisateur soit suffisamment faible pour être deviné / recherché.

5.9.2 Impersonnification d'un utilisateur par un serveur malicieux

En cas de vérification des utilisateurs (et de leurs appareils) par le mécanisme des clés *cross-signing*, le serveur ne devrait pas pouvoir impersonnifier un utilisateur auprès d'un autre utilisateur.

Le serveur peut cependant impersonnifier un appareil "vérifié" ($D_{U_a,\alpha}$) ayant une session Olm en cours avec un autre appareil ($D_{U_b,\beta}$). Cet exploit consiste en les étapes suivantes :

1. le serveur envoie un nouveau message (Protocole 4.A) en tant que $D_{U_a,\alpha}$ à $D_{U_b,\beta}$ avec une nouvelle clé *ratchet* $T^{\texttt{pub}}_{\text{Eve}}$ (à la place de $T^{\texttt{pub}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}$). Le message envoyé est composé de $j := 0$, $c_{i,j}$, $\texttt{mac}_{i,j}$ (tous deux tirés aléatoirement ou choisis arbitrairement), la clé *ratchet* frauduleuse $T^{\texttt{pub}}_{\text{Eve}}$ et la clé d'identité $k^{\texttt{pub}}_{id,D_{U_a,\alpha}}$. Les nouvelles clés racine et de chaînage sous-jacentes seront

$$(\tilde{R}_i, \tilde{C}_{i,0}) \leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(R_{i-1}, \texttt{ECDH}(T^{\texttt{priv}}_{i,D_{U_a,\alpha},D_{U_b,\beta}}, T^{\texttt{pub}}_{\text{Eve}}), \texttt{"OLM\_RATCHET"}, 64).$$

2. à réception de ce message l'appareil $D_{U_b,\beta}$ (Protocole 4.B) dérive les nouvelles clés frauduleuses $(\tilde{R}_i, \tilde{C}_{i,0})$, derive la clé de message $\tilde{M}_{i,0}$ à partir de $\tilde{C}_{i,0}$, déchiffre le message envoyé $c_{i,0}$ et vérifie $\texttt{mac}_{i,0}$. La vérification échoue, le message est supprimé mais les nouvelles clés $(\tilde{R}_i, \tilde{C}_{i,0})$ sont conservées (Vulnérabilité 10),
3. lors de l'envoi d'un nouveau message, $D_{U_b,\beta}$ avance les clés de la façon suivante :

$$(\tilde{R}_{i+1}, \tilde{C}_{i+1,0}) \leftarrow \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(\tilde{R}_i, \texttt{ECDH}(T^{\texttt{priv}}_{i+1,D_{U_b,\beta},D_{U_a,\alpha}}, T^{\texttt{pub}}_{\text{Eve}}), \texttt{"OLM\_RATCHET"}, 64),$$

   dérive une clé $\tilde{M}_{i+1,0}$ et l'utilise pour chiffrer et authentifier le message envoyé.
4. à réception de ce nouveau message, le serveur utilise la Vulnérabilité 28 afin de faire une recherche exhaustive sur les 32 bits de $\tilde{R}_i$. Pour chaque valeur possible, le serveur peut alors calculer

$$(\tilde{R}_{i+1}, \tilde{C}_{i+1,0}) = \texttt{HKDF}^{\texttt{HMAC-SHA-3}}(\tilde{R}_i, \texttt{ECDH}(T^{\texttt{pub}}_{i+1,D_{U_b,\beta},D_{U_a,\alpha}}, T^{\texttt{priv}}_{\text{Eve}}), \texttt{"OLM\_RATCHET"}, 64).$$

   Une seule des valeurs testées donnera lieu à une paire $(\tilde{R}_{i+1}, \tilde{C}_{i+1,0})$ dont la clé de message sous-jacente $\tilde{M}_{i+1,0}$ permet de déchiffrer et de vérifier le message reçu. Le serveur retrouve ainsi la nouvelle paire $(\tilde{R}_{i+1}, \tilde{C}_{i+1,0})$ et la session peut continuer sans que $D_{U_b,\beta}$ s'aperçoive de la fraude.

Cet exploit peut être combiné avec la Vulnérabilité 25 : un serveur malicieux peut être mis en place avec un certificat auto-signé accepté par l'application.

5.9.3 Récupération de clés long termes

Les clés `backupK` (*i.e.*, clés *cross-signing*, clés *recovery*) peuvent être récupérées en exploitant la Vulnérabilité 27. Les bugs d'implémentation de cette vulnérabilité permettent une recherche exhaustive de la clé *secret storage* $k_{\texttt{sec},U_a}$ d'un utilisateur U_a ce qui permet de déchiffrer tout backup sécurisé $c_{\texttt{backupK}}$. Cet exploit peut être réalisé par

- un utilisateur ayant impersonnifié U_a auprès du serveur (Section 5.9.1),
- un serveur malicieux (qui peut ainsi déchiffrer toute clé `backupK` qui lui est transmise).

Notons qu'avec la connaissance des clés *cross-signing* de U_a et de sa clé *recovery*, un attaquant peut complètement impersonnifier U_a auprès des autres utilisateurs et déchiffrer ses anciennes conversations.

6 Vulnérabilités intrinsèques

Les vulnérabilités présentées dans cette section n'ont pas été introduites délibérément mais étaient présentes dans le code source original de l'application.

6.1 Conformité et Intégrité symétrique

Vulnérabilité 29 (**Réduction de la taille du motif d'authentification 64 bits**)**.** La sortie de la fonction `HMAC-SHA-3` a une taille insuffisante à savoir 64 bits. A ce jour, nous n'avons pas trouvé d'explications à ce choix dans le cadre du projet Matrix. L'attaque générique associée à cette vulnérabilité est coûteuse dans le cas où l'implémentation `SHA3` est conforme aux spécifications, ce qui n'est pas le cas ici.

Cette vulnérabilité, combinée avec la vulnérabilité 12, permet à un attaquant ayant un couple message/tag valide de forger un message différent ayant le même tag. La difficulté de l'attaque rejoint donc celle de la vulnérabilité 12.

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 6.A, 6.B (conversations)
Catégorie	conformité et intégrité symétrique
Difficulté d'identification	facile
Difficulté d'exploitation	moyen

6.2 Protocole cryptographique

Vulnérabilité 30 (**Attaque par rejeu**)**.** Il n'y a pas de protection anti-rejeu des messages dans le cadre des conversations OLM et MEGOLM. Il est donc possible pour un adversaire de rejouer les messages. Cette vulnérabilité est connue [5].

Protocole(s) concerné(s)	3.A, 3.B, 4.A, 4.B, 6.B, 6.A (conversations)
Catégorie	protocole cryptographique
Difficulté d'identification	difficile
Difficulté d'exploitation	facile

6.3 Yubikey

Dans le cas où l'utilisateur est mal formé ou ne suit pas les bonnes pratiques, il est possible d'exploiter les vulnérabilités suivantes. Dans le cadre d'un projet de certification, le changement obligatoire du code PUK aurait dû être précisé soit dans la cible de sécurité soit dans les guides d'utilisation du produit.

Vulnérabilité 31 (**PUK par défaut de la Yubikey**)**.** Le code PUK par défaut des Yubikey est égal à `12345678`. En effet, le code PIN de la Yubikey est un secret utilisateur qui permet d'accéder aux clés stockées dans cette dernière. L'utilisateur a le droit à un nombre limité de tentatives avant que la Yubikey soit verrouillée. Pour déverrouiller la Yubikey, l'utilisateur dispose d'un code PUK qui permet de changer le code PIN et de remettre à zéro le compteur d'essais indépendamment. Un attaquant peux exploiter cette faiblesse de deux manières différentes : (1) Il récupère la Yubikey de sa victime et change le code PIN. Ainsi, il accède aux clés de l'utilisateur légitime et il peut usurper son identité tant que l'utilisateur ne révoque pas cette Yubikey ; (2) Il peut exploiter la remise à zéro du compteur sans changer le code PIN. Dans ce cas, il peut récupérer le code PIN utilisateur de la Yubikey avec une recherche exhaustive en remettant à zéro le compteur et rendre la Yubikey à l'utilisateur.

Protocole(s) concerné(s)	2
Catégorie	Yubikey
Difficulté d'identification	facile
Difficulté d'exploitation	facile

Vulnérabilité 32 (**Clé par défaut de la Yubikey**)**.** La clé de management par défaut de la Yubikey est une clé 3DES dont la valeur est publique (`0102030405060708010203040506070801020304050607080`). Elle permet de recharger les clés dans les slots de la Yubikey et donc d'écraser les clés de l'utilisateur. Un attaquant qui voudrait exploiter cette faiblesse, à savoir piéger une Yubikey, n'irait pas bien loin puisque cette modification serait détectée lors de la prochaine utilisation de la Yubikey par son utilisateur légitime.

Protocole(s) concerné(s)	2
Catégorie	Yubikey
Difficulté d'identification	facile
Difficulté d'exploitation	facile

5. https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide#mmegolmv1aes-sha2